

Copyright is owned by the Author of the thesis. Permission is given for a copy to be downloaded by an individual for the purpose of research and private study only. The thesis may not be reproduced elsewhere without the permission of the Author.

# A Modelling Language for Rich Internet Applications

A thesis presented in partial fulfillment of the  
requirements for the degree of

Doctor of Philosophy  
in  
Computer Science

at Massey University, Turitea,  
New Zealand.

Jevon Michael Wright

2011

Copyright © 2011 by Jevon Wright

Copyright is owned by the author of this thesis. Permission is given for a copy to be downloaded by an individual for the purpose of research and private study only. This thesis may not be reproduced elsewhere without the permission of the author.

# Abstract

This thesis presents the Internet Application Modelling Language (IAML), a modelling language to support the model-driven development of Rich Internet Applications (RIAs). This definition includes a visual syntax to support the graphical development of IAML model instances, and the underlying metamodel satisfies the metamodeling and viewpoint architectures of the Model Driven Architecture.

While there are many existing modelling languages for web applications, none of these languages were found to be expressive enough to describe fundamental RIA concepts such as client-side events and user interaction. This thesis therefore presents IAML as a new language that reuses existing standards where appropriate. IAML is supported by a proof-of-concept CASE tool within the Eclipse framework, and released under an open source license to encourage industry use. This reference implementation successfully integrates a number of different model-driven technologies to demonstrate the expressiveness of the modelling language.

The IAML metamodel supports many features not found in other web application modelling languages, such as Event-Condition-Action rules; the expression of reusable patterns through [Wires](#); and a metamodel core based on first-order logic. Through the implementation of the RIA benchmarking application *Ticket 2.0*, the concepts behind the design of IAML have been shown to simplify the development of real-world RIAs when compared to conventional web application frameworks.





# Acknowledgments

This thesis would not have been possible without the support of many important people.

In particular, to my supervisors Jens Dietrich and Giovanni Moretti – thank you for all of your advice, encouragement and inspiration throughout my undergrad degree and my doctoral studies. I am gracious for the patience and support that you have both given me.

To my office mates at Massey that have helped me throughout these years – Graham Jenson, Fahim Abbasi, Yuliya Bozhko and others – thank you for your feedback, ideas and your constructive criticisms; and the innumerable breaks for caffeine, whether in coffee, chocolate or cola form.

To the support staff at Massey – including Patrick Rynhart, Dilantha Punchihewa, and Michele Wagner – thank you for helping me out with the stress and effort necessary to deal with the administrative side of research.

To all my bandmates – Aaron Badman, Aaron Shirriffs, Ian Luxmoore, Josh Williamson, Matt Carkeek, Steve Starr, Wayne Bryant – thank you for supporting me throughout my busy schedules, and for giving me a noisy outlet to express my creativity far away from the thesis.

To all of my family, especially my mum and dad – thank you for your never-ending and reliable support and patience throughout all of my decisions, and encouraging my independence.

And finally to my best friend and fiancé, Krystle Chester, you have helped me and supported me through more than I could have imagined, and completing this thesis would not have been possible without your love, patience and mews. <3



# Contents

<b>Abstract</b>	<b>i</b>
<b>Acknowledgments</b>	<b>iii</b>
<b>Contents</b>	<b>v</b>
<b>List of Tables</b>	<b>vii</b>
<b>List of Figures</b>	<b>ix</b>
<b>List of Listings</b>	<b>xi</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Web Applications . . . . .	1
1.2 Modelling . . . . .	3
1.3 Research Questions . . . . .	3
1.4 Research Design . . . . .	4
1.5 Research Method . . . . .	6
1.6 Thesis Outline . . . . .	7
<b>2 Background</b>	<b>9</b>
2.1 Rich Internet Applications . . . . .	9
2.1.1 Classifications of Rich Internet Applications . . . . .	11
2.1.2 Interactivity . . . . .	11
2.1.3 Enterprise Software . . . . .	12
2.1.4 Security Risks of Rich Internet Applications . . . . .	12
2.2 Frameworks . . . . .	13
2.3 Features of Rich Internet Applications . . . . .	14
2.3.1 Feature Categories . . . . .	14
2.3.2 Detailed Modelling Requirements . . . . .	17
2.4 Existing Web Application Modelling Languages . . . . .	20
2.4.1 WebML . . . . .	20
2.4.2 UML . . . . .	21
2.4.3 UWE . . . . .	22
2.4.4 WebDSL . . . . .	23
2.4.5 Web Information Systems . . . . .	24
2.4.6 Older Languages . . . . .	24

2.4.7	Evaluation . . . . .	24
2.5	Benchmarking Application . . . . .	24
2.6	Metrics . . . . .	26
2.6.1	Web Application Metrics . . . . .	26
2.6.2	Metamodelling Metrics . . . . .	28
2.6.3	System Metrics . . . . .	29
2.7	Software Engineering Principles within the Development of Modelling Languages . .	30
2.7.1	Modelling Language Design Approaches . . . . .	30
2.7.2	Software Process Models . . . . .	31
2.7.3	Test Driven Development . . . . .	33
2.7.4	Open Source . . . . .	33
2.8	Conclusion . . . . .	33
<b>3</b>	<b>Modelling</b>	<b>35</b>
3.1	Modelling . . . . .	35
3.1.1	Metamodels . . . . .	36
3.1.2	Domain-Specific Languages . . . . .	36
3.1.3	Model-Driven Development . . . . .	38
3.1.4	Model-Driven Engineering . . . . .	38
3.1.5	Model Driven Architecture . . . . .	39
3.1.6	Model Spaces . . . . .	43
3.1.7	Model Transformations . . . . .	44
3.1.8	Query/Views/Transformations Language . . . . .	46
3.1.9	The Syntax and Semantics of Metamodels . . . . .	48
3.1.10	A Formal Definition of Metamodels . . . . .	49
3.1.11	The Bigger Picture of Modelling Approaches . . . . .	50
3.1.12	Metamodelling Environments . . . . .	51
3.2	Model Completion . . . . .	51
3.2.1	Documented Conventions . . . . .	52
3.2.2	Model Completion Concepts . . . . .	52
3.2.3	Model Completion Semantics . . . . .	53
3.3	Model Instance Verification . . . . .	54
3.3.1	Validation vs. Verification . . . . .	55
3.3.2	A Formal Definitions of Constraints . . . . .	55
3.3.3	Discussion . . . . .	57
3.4	Visual Modelling . . . . .	58
3.4.1	Visual Metaphors . . . . .	58
3.4.2	Visual Metaphors in Existing Models . . . . .	59
3.4.3	Visual Modelling Software . . . . .	59
3.4.4	Visual Notation . . . . .	59
3.5	Conclusion . . . . .	61

<b>4</b>	<b>Rich Internet Application Modelling Concepts</b>	<b>63</b>
4.1	Managing Complexity through Structural Decomposition . . . . .	63
4.1.1	Modularity . . . . .	63
4.1.2	Hierarchical Modelling . . . . .	64
4.1.3	Aspects . . . . .	65
4.1.4	Discussion . . . . .	66
4.2	Events . . . . .	66
4.2.1	UML . . . . .	66
4.2.2	AORML . . . . .	67
4.2.3	URML . . . . .	67
4.2.4	Kaitiaki . . . . .	68
4.2.5	Event Algebra . . . . .	68
4.2.6	DOM Events . . . . .	69
4.3	Operation Modelling . . . . .	70
4.3.1	UML Activity Diagrams . . . . .	70
4.3.2	Business Process Modelling Notation . . . . .	71
4.3.3	Predicate Modelling . . . . .	71
4.4	Type Systems . . . . .	72
4.4.1	Primitive Types . . . . .	72
4.4.2	Derived Types . . . . .	73
4.4.3	Type Systems in the Model Driven Architecture . . . . .	74
4.5	Domain Modelling . . . . .	75
4.5.1	ER Diagrams . . . . .	75
4.5.2	UML Class Diagrams . . . . .	76
4.5.3	XML Schema . . . . .	76
4.5.4	EMF Ecore . . . . .	77
4.6	Design Patterns for Data Access . . . . .	77
4.6.1	Database Broker . . . . .	77
4.6.2	Iterator . . . . .	77
4.7	Lifecycle Modelling . . . . .	79
4.7.1	Implementation-Level Examples . . . . .	79
4.8	Users and Access Control . . . . .	80
4.8.1	Access Control Lists . . . . .	80
4.8.2	Discretionary Access Control . . . . .	82
4.8.3	Mandatory Access Control . . . . .	82
4.8.4	Role-based Access Control . . . . .	83
4.9	Describing Reusable Patterns within a Metamodel . . . . .	83
4.9.1	Independence of the Reusable Pattern Metamodel . . . . .	83
4.9.2	Implementation of the Reusable Pattern Metamodel . . . . .	83
4.10	Conclusion . . . . .	84
<b>5</b>	<b>The Internet Application Modelling Language</b>	<b>85</b>
5.1	Applying Software Engineering Principles . . . . .	85
5.1.1	Requirements Planning . . . . .	85
5.1.2	Modelling Language Development Approach . . . . .	86

5.1.3	Software Process Model . . . . .	87
5.1.4	Evaluation . . . . .	88
5.1.5	Design Goals . . . . .	88
5.2	Metamodel Design Principles . . . . .	89
5.2.1	Addressing Security Risks through Modelling Language Design . . . . .	89
5.2.2	Language Design Principles . . . . .	89
5.2.3	Hierarchical Modelling Approach . . . . .	91
5.2.4	Guidelines for Metamodel Refactoring . . . . .	92
5.3	Language Overview . . . . .	93
5.3.1	Package Overview . . . . .	95
5.4	Metamodel Core . . . . .	96
5.4.1	Logic Model . . . . .	97
5.4.2	Function Model . . . . .	100
5.4.3	Event-Condition-Action Model . . . . .	101
5.4.4	An example decomposition of a Simple Condition . . . . .	103
5.4.5	Wires Model . . . . .	105
5.4.6	Constructs Model . . . . .	105
5.5	Type System . . . . .	106
5.5.1	Primitive Types . . . . .	107
5.5.2	Additional Built-in Primitive Types . . . . .	109
5.5.3	Value Instances . . . . .	110
5.5.4	Client-side Input Validation . . . . .	111
5.6	Domain Modelling . . . . .	112
5.6.1	Domain Types . . . . .	112
5.6.2	Design Decisions on Modelling Domain Types . . . . .	114
5.6.3	Domain Iterators . . . . .	116
5.6.4	Domain Instances . . . . .	119
5.6.5	Design Decisions on Modelling Domain Instances . . . . .	120
5.7	Events . . . . .	122
5.7.1	onChange . . . . .	122
5.7.2	onInput . . . . .	122
5.7.3	onAccess . . . . .	123
5.7.4	onInit . . . . .	123
5.7.5	onClick . . . . .	123
5.7.6	onSent . . . . .	123
5.7.7	onFailure . . . . .	124
5.7.8	onIterate . . . . .	124
5.7.9	Client-side Events . . . . .	124
5.8	Operations . . . . .	125
5.8.1	Modelling Complex Behaviour . . . . .	125
5.8.2	Activity Operations . . . . .	128
5.8.3	Activity Predicates . . . . .	130
5.8.4	Replacing the Default Behaviour of Model Elements . . . . .	131
5.9	Wires . . . . .	131

5.9.1	Sync Wire . . . . .	132
5.9.2	Set Wire . . . . .	133
5.9.3	Detail Wire . . . . .	133
5.9.4	Autocomplete Wire . . . . .	134
5.10	Users and Access Control . . . . .	136
5.10.1	Roles and Permissions . . . . .	136
5.10.2	Login Handlers . . . . .	137
5.10.3	Access Control Handlers . . . . .	138
5.10.4	OpenID . . . . .	139
5.11	Scopes . . . . .	140
5.11.1	Internet Application . . . . .	140
5.11.2	Session . . . . .	141
5.11.3	Failure Handlers . . . . .	141
5.11.4	Gates . . . . .	142
5.11.5	Query Parameter . . . . .	143
5.11.6	Discussion . . . . .	143
5.12	Messaging . . . . .	143
5.12.1	Email . . . . .	144
5.12.2	Additional Messaging Types . . . . .	144
5.13	User Interface Modelling . . . . .	145
5.13.1	Existing Approaches in Modelling User Interface Types . . . . .	145
5.13.2	Visible Thing . . . . .	146
5.13.3	Sample Representations . . . . .	150
5.13.4	Frame . . . . .	150
5.14	Internet Application . . . . .	150
5.15	Visual Modelling Metaphors . . . . .	151
5.15.1	Overview Layer . . . . .	151
5.15.2	Navigation Layer . . . . .	152
5.15.3	User Interface Layer . . . . .	152
5.15.4	Domain Modelling Layer . . . . .	152
5.15.5	Operation Modelling Layer . . . . .	152
5.16	Unification of Type Instantiation and Classification . . . . .	153
5.16.1	Guidelines for Applying Instance Unification in Metamodels . . . . .	155
5.16.2	Applying Instance Unification to the IAML Metamodel . . . . .	156
5.17	Conclusion . . . . .	157
<b>6</b>	<b>Implementation Technologies</b>	<b>159</b>
6.1	Common Comparison Criteria . . . . .	159
6.1.1	Execution Environment Comparison Properties . . . . .	159
6.1.2	Open Source Comparison Criteria . . . . .	160
6.2	Metamodelling Environments . . . . .	161
6.2.1	Eclipse Modeling Framework . . . . .	162
6.2.2	ArgoUML . . . . .	163
6.2.3	Whitehorse . . . . .	164
6.2.4	Marama . . . . .	166



6.2.5	Summary . . . . .	167
6.3	Graphical Modelling . . . . .	167
6.3.1	Graphical Editor Framework . . . . .	169
6.3.2	Graphical Modeling Framework . . . . .	169
6.3.3	ArgoUML . . . . .	170
6.3.4	Whitehorse . . . . .	170
6.3.5	Marama . . . . .	171
6.3.6	Summary . . . . .	172
6.4	Code Generation . . . . .	173
6.4.1	XSLT . . . . .	173
6.4.2	Java Emitter Templates . . . . .	174
6.4.3	openArchitectureWare . . . . .	175
6.4.4	Xpand . . . . .	175
6.4.5	Summary . . . . .	176
6.5	Model Completion . . . . .	177
6.5.1	Jena . . . . .	178
6.5.2	Jess . . . . .	179
6.5.3	Drools . . . . .	180
6.5.4	Summary . . . . .	180
6.6	Model Instance Verification . . . . .	181
6.6.1	Drools . . . . .	182
6.6.2	Jena . . . . .	184
6.6.3	OWL 2 Full . . . . .	185
6.6.4	CrocoPat . . . . .	185
6.6.5	Alloy . . . . .	187
6.6.6	openArchitectureWare . . . . .	188
6.6.7	OCL . . . . .	189
6.6.8	NuSMV . . . . .	190
6.6.9	Summary . . . . .	191
6.7	Conclusion . . . . .	192
<b>7</b>	<b>Proof-of-Concept Implementation</b>	<b>195</b>
7.1	Introduction . . . . .	195
7.1.1	Implementation License . . . . .	195
7.2	<i>IAML Metamodel</i> . . . . .	197
7.3	<i>IAML Model</i> . . . . .	197
7.3.1	<i>Model Instance</i> . . . . .	197
7.3.2	<i>Model Migration</i> . . . . .	198
7.4	<i>Graphical Editor</i> . . . . .	198
7.4.1	<i>Model Edit</i> . . . . .	199
7.4.2	<i>Diagram Editors</i> . . . . .	199
7.4.3	<i>Diagram Definitions</i> . . . . .	201
7.4.4	<i>Diagram Extensions</i> . . . . .	205
7.4.5	<i>Diagram Actions</i> . . . . .	209
7.5	<i>Model Completion</i> . . . . .	210

7.5.1	<i>Completion</i>	211
7.5.2	<i>Rule Set</i>	211
7.5.3	<i>Drools</i>	214
7.6	<i>Code Generation</i>	214
7.6.1	<i>openArchitectureWare</i>	214
7.6.2	<i>Templates</i>	215
7.6.3	<i>Runtime Library</i>	216
7.6.4	<i>Platform Configuration</i>	217
7.6.5	<i>Output Formatter</i>	217
7.6.6	<i>Generator</i>	218
7.7	<i>Model Verification</i>	219
7.7.1	Model Instance Verification with Checks	220
7.7.2	Model Instance Verification with OCL in the EMF Validation Framework	221
7.7.3	Model Instance Verification with CrocoPat	221
7.7.4	Model Instance Verification with NuSMV	222
7.8	<i>Model Actions</i>	222
7.9	<i>Tests</i>	223
7.9.1	Model-driven Code Coverage	224
7.10	<i>Conclusion</i>	225
<b>8</b>	<b>Evaluation</b>	<b>227</b>
8.1	Feature Comparison	227
8.2	Modelling Requirements	229
8.3	Benchmarking Application Implementation	231
8.3.1	Implementation in Symphony	232
8.3.2	Implementation in IAML	233
8.3.3	Re-implementation in Symphony	252
8.3.4	System Metrics Evaluation	254
8.4	Metamodelling Metrics	255
8.5	Visual Model Evaluation	257
8.5.1	Notation Information Capacity	258
8.5.2	Cognitive Dimensions Evaluation	259
8.6	<i>Conclusion</i>	259
<b>9</b>	<b>Conclusions and Future Research</b>	<b>261</b>
9.1	Research Contributions and Conclusions	261
9.1.1	Requirements for Modelling Rich Internet Applications	261
9.1.2	A Benchmarking Application for Rich Internet Applications	261
9.1.3	A Modelling Language for Rich Internet Applications	262
9.1.4	Model Completion	263
9.1.5	Model Instance Verification	264
9.1.6	Evaluation of Model-Driven Technologies	264
9.1.7	Other Contributions	265
9.2	Answers to Research Questions	265
9.3	Future Research	267

9.3.1	Further Evaluation of Existing Web Modelling Languages . . . . .	267
9.3.2	Modelling Full RIAs . . . . .	268
9.3.3	Improved Graphical Editor View Mappings . . . . .	268
9.3.4	Integrating Textual Expression Languages in the Visual Editor . . . . .	269
9.3.5	Extraction of Reusable Components . . . . .	269
9.3.6	Code Generation Templates for Additional Platforms . . . . .	270
9.3.7	Incremental Transformations . . . . .	270
9.3.8	Extensibility of the Proof-of-Concept Implementation of IAML . . . . .	270
9.3.9	User Evaluations on the Proof-of-Concept Implementation of IAML . . . . .	271
9.3.10	Identifying Metamodel Refactoring Patterns . . . . .	271
9.4	Summary . . . . .	273
<b>A</b>	<b>Use Cases</b>	<b>275</b>
A.1	Actors . . . . .	275
A.2	List of Use Cases . . . . .	276
UC-01	View Data . . . . .	277
UC-02	Update Data . . . . .	277
UC-03	Pagination . . . . .	278
UC-04	User Action Auditing . . . . .	278
UC-05	Debug Mode . . . . .	279
UC-06	Server Transaction Support . . . . .	279
UC-07	Local Data Storage . . . . .	280
UC-08	Server Data Access . . . . .	281
UC-09	Persistent Client Data . . . . .	281
UC-10	Temporary Server Data . . . . .	282
UC-11	Uploading Files . . . . .	283
UC-12	Restore Server Session . . . . .	284
UC-13	User Authorisation . . . . .	285
UC-14	Password Reset . . . . .	286
UC-15	Session Support . . . . .	286
UC-16	Account Registration . . . . .	287
UC-17	Automatic User Authorisation . . . . .	288
UC-18	Static Views (HTML) . . . . .	289
UC-19	Asynchronous Form Validation . . . . .	290
UC-20	Client Form Validation . . . . .	291
UC-21	Server Form Validation . . . . .	292
UC-22	Multiple Browser Support . . . . .	293
UC-23	Mobile Phone Support . . . . .	293
UC-24	Remote Data Source . . . . .	294
UC-25	Active Remote Data Source . . . . .	294
UC-26	Data Feeds . . . . .	295
UC-27	Web Service . . . . .	295
UC-28	Back/Forwards Button Control . . . . .	296
UC-29	Opening New Windows . . . . .	296
UC-30	Client-Side Application . . . . .	297

UC-31	Communication with Software . . . . .	298
UC-32	Mobile Phone Communication . . . . .	299
UC-33	E-mailing Users . . . . .	300
UC-34	E-mail Unsubscription . . . . .	301
UC-35	Persistent Errors . . . . .	302
UC-36	User Content Access Control . . . . .	302
UC-37	Private User Content Access Control . . . . .	303
UC-38	User Collaboration . . . . .	304
UC-39	Interactive Map . . . . .	305
UC-40	Drag and Drop . . . . .	305
UC-41	Client Timer Support . . . . .	306
UC-42	Server Timer Support . . . . .	307
UC-43	Page Caching . . . . .	307
UC-44	Offline Application Support . . . . .	308
UC-45	Loading Time Support . . . . .	308
UC-46	Flash MP3 Support . . . . .	309
UC-47	Flash Communication Support . . . . .	310
UC-48	Internationalisation Support . . . . .	310
UC-49	Logout Control . . . . .	311
UC-50	Single Sign-In Solutions . . . . .	311
UC-51	User Redirection . . . . .	312
UC-52	Keyboard Shortcuts . . . . .	313
UC-53	Undo/Redo Support . . . . .	313
UC-54	Browser-Based Chat . . . . .	314
UC-55	Pop-up Window Support . . . . .	315
UC-56	Incompatible Client Warning . . . . .	315
UC-57	Dynamic Objects . . . . .	316
UC-58	Store Data in Local Database . . . . .	317
UC-59	Store Resources Locally . . . . .	318
UC-60	Multiple Client Threads . . . . .	319
UC-61	Multiple Server Threads . . . . .	320
UC-62	Communication with Plugins . . . . .	321
UC-63	Scheduled Events . . . . .	321
UC-64	Custom API Publishing . . . . .	322
UC-65	Runtime Interface Updates . . . . .	322
UC-66	Out-of-Order Events . . . . .	323
UC-67	Backwards-Compatible Scripting . . . . .	323
UC-68	Spellchecking . . . . .	324
UC-69	Autocomplete . . . . .	324
<b>B</b>	<b>Documentation By Example: Sync Wire</b>	<b>325</b>
<b>C</b>	<b>OpenBRR Evaluations of Model-driven Technologies</b>	<b>331</b>
<b>D</b>	<b>Model Checking using NuSMV: Implementing the <i>Infinitely Redirects</i> Constraint</b>	<b>335</b>

<b>E</b>	<b>Description of the Attached Media</b>	<b>339</b>
E.1	Media Contents . . . . .	339
E.2	IAML Tutorial . . . . .	340
E.2.1	Creating a New Model Instance . . . . .	340
E.2.2	Editing the Model Instance . . . . .	340
E.2.3	Generating the Application . . . . .	343
E.2.4	Configure the Web Server . . . . .	343
E.2.5	Access the Generated Application . . . . .	343
<b>F</b>	<b>XMI Representation of Ticketiaml</b>	<b>347</b>
<b>G</b>	<b>Language-specific Metrics of the Ticket 2.0 Implementations</b>	<b>357</b>
<b>H</b>	<b>Cognitive Dimensions Evaluation of IAML</b>	<b>361</b>
<b>I</b>	<b>Model Element Reference</b>	<b>367</b>
I.1	Access Control Handler . . . . .	368
I.2	Accessible . . . . .	369
I.3	Action . . . . .	369
I.4	Action Edge Source . . . . .	370
I.5	Activity Node . . . . .	370
I.6	Activity Operation . . . . .	371
I.7	Activity Parameter . . . . .	373
I.8	Activity Predicate . . . . .	374
I.9	Arithmetic . . . . .	375
I.10	Autocomplete Wire . . . . .	376
I.11	Boolean Property . . . . .	377
I.12	Builtin Operation . . . . .	377
I.13	Builtin Property . . . . .	379
I.14	Button . . . . .	380
I.15	Can Be Synced . . . . .	381
I.16	Cancel Node . . . . .	382
I.17	Cast Node . . . . .	383
I.18	Changeable . . . . .	383
I.19	Complex Term . . . . .	384
I.20	Condition Edge Destination . . . . .	385
I.21	Constraint Edge . . . . .	385
I.22	Constraint Edge Destination . . . . .	386
I.23	Constraint Edges Source . . . . .	386
I.24	Contains Functions . . . . .	386
I.25	Contains Operations . . . . .	387
I.26	Contains Values . . . . .	387
I.27	Contains Wires . . . . .	388
I.28	Data Flow Edge . . . . .	389
I.29	Data Flow Edge Destination . . . . .	389
I.30	Data Flow Edges Source . . . . .	390

I.31	Decision Node . . . . .	390
I.32	Detail Wire . . . . .	391
I.33	Domain Attribute . . . . .	392
I.34	Domain Attribute Instance . . . . .	392
I.35	Domain Feature . . . . .	393
I.36	Domain Feature Instance . . . . .	394
I.37	Domain Instance . . . . .	394
I.38	Domain Iterator . . . . .	395
I.39	Domain Source . . . . .	400
I.40	Domain Type . . . . .	401
I.41	ECA Rule . . . . .	402
I.42	EXSD Data Type . . . . .	403
I.43	Email . . . . .	404
I.44	Event . . . . .	405
I.45	Execution Edge . . . . .	407
I.46	Execution Edge Destination . . . . .	407
I.47	Execution Edges Source . . . . .	408
I.48	Extends Edge . . . . .	408
I.49	Extends Edge Destination . . . . .	409
I.50	Extends Edges Source . . . . .	409
I.51	External Value . . . . .	410
I.52	Finish Node . . . . .	410
I.53	Frame . . . . .	411
I.54	Function . . . . .	412
I.55	Gate . . . . .	413
I.56	Generated Element . . . . .	414
I.57	Generates Elements . . . . .	415
I.58	Input Form . . . . .	416
I.59	Input Text Field . . . . .	417
I.60	Internet Application . . . . .	418
I.61	Iterator List . . . . .	419
I.62	Join Node . . . . .	419
I.63	Label . . . . .	420
I.64	Login Handler . . . . .	421
I.65	Map . . . . .	425
I.66	Map Point . . . . .	425
I.67	Message . . . . .	426
I.68	Named Element . . . . .	426
I.69	Operation . . . . .	427
I.70	Operation Call Node . . . . .	427
I.71	Parameter . . . . .	427
I.72	Parameter Edge Destination . . . . .	430
I.73	Parameter Edges Source . . . . .	430
I.74	Parameter Value . . . . .	431

I.75	Permission . . . . .	431
I.76	Predicate . . . . .	431
I.77	Provides Edge . . . . .	432
I.78	Provides Edge Destination . . . . .	432
I.79	Provides Edges Source . . . . .	433
I.80	Query Parameter . . . . .	433
I.81	Requires Edge . . . . .	433
I.82	Requires Edge Destination . . . . .	434
I.83	Requires Edges Source . . . . .	435
I.84	Role . . . . .	435
I.85	Schema Edge . . . . .	436
I.86	Scope . . . . .	436
I.87	Select Edge . . . . .	438
I.88	Session . . . . .	439
I.89	Set Node . . . . .	440
I.90	Set Wire . . . . .	440
I.91	Simple Condition . . . . .	444
I.92	Split Node . . . . .	445
I.93	Start Node . . . . .	446
I.94	Sync Wire . . . . .	446
I.95	Temporary Variable . . . . .	449
I.96	Value . . . . .	449
I.97	Visible Thing . . . . .	451
I.98	Wire . . . . .	452
I.99	Wire Destination . . . . .	452
I.100	Wire Source . . . . .	453
I.101	Wireable . . . . .	453
I.102	XQuery Function . . . . .	454
I.103	XQuery Predicate . . . . .	454

<b>Bibliography</b>	<b>455</b>
---------------------	------------

# List of Tables

1.1	A brief history of Web application technologies . . . . .	2
2.1	Fundamental modelling requirements of Rich Internet Applications (1) . . . . .	18
2.2	Fundamental modelling requirements of Rich Internet Applications (2) . . . . .	19
2.3	Existing modelling language support for the general feature categories of modelling Rich Internet Applications . . . . .	25
2.4	Selected metamodeling metrics for metamodels within the Eclipse Modeling Frame- work . . . . .	28
2.5	Overall system metrics . . . . .	29
2.6	Language system metrics . . . . .	29
3.1	A list of popular metamodels and their meta-metamodels . . . . .	40
3.2	The relationships between different modelling approaches . . . . .	51
3.3	Information encoding capacity of different visual variables . . . . .	60
4.1	Selected event algebra operators for composing complex events . . . . .	69
4.2	JUnit lifecycle methods and annotations . . . . .	79
4.3	Possible lifecycle layers of Rich Internet Applications and their potential use . . . .	81
4.4	Example of an Access Matrix . . . . .	82
5.1	Additional Requirements for <i>Full RIAs</i> . . . . .	86
5.2	Associations between first-order logic elements and IAML elements . . . . .	98
5.3	A selection of permitted, conditional and prohibited datatype casts in IAML . . . . .	109
5.4	Sample visual representations of interface modelling elements in IAML . . . . .	149
5.5	Visual Metaphors for a Rich Internet Application modelling language . . . . .	152
6.1	Requirement comparisons between existing metamodeling environments . . . . .	167
6.2	Requirement comparisons between existing graphical modelling environments . . . .	172
6.3	Requirement comparisons between existing code generation environments . . . . .	176
6.4	Requirement comparisons between existing model completion rule engines . . . . .	181
6.5	Requirement comparisons between existing model verification environments (1) . . .	192
6.6	Requirement comparisons between existing model verification environments (2) . . .	193
6.7	A summary of the suitability of verification languages for addressing each verification approach category . . . . .	193
7.1	Generated IAML diagram editors and their associated OSGi bundle IDs . . . . .	201
7.2	Shape styles for the visual representation of IAML model elements . . . . .	203



7.3	Background colours for the visual representation of IAML model elements . . . . .	204
7.4	Summary of the rules used for model completion on IAML model instances . . . . .	213
7.5	The set of platform-specific configuration properties used by the IAML <i>Code Generator</i> component . . . . .	218
8.1	Reuse of existing metamodels in the IAML metamodel . . . . .	228
8.2	A comparison of IAML against existing modelling language support for the general feature categories of modelling Rich Internet Applications . . . . .	229
8.3	Evaluation of the modelling requirements of <i>Basic RIAs</i> against IAML (1) . . . . .	230
8.4	Evaluation of the modelling requirements of <i>Basic RIAs</i> against IAML (2) . . . . .	231
8.5	Legend to the modelling requirements evaluation of IAML . . . . .	231
8.6	Comparing two Ticket 2.0 implementations using system metrics: overall development effort . . . . .	254
8.7	Comparing two Ticket 2.0 implementations using system metrics: manual effort . . . . .	254
8.8	Using metamodel metrics to evaluate the IAML metamodel against other similar metamodels implemented using the Eclipse Modeling Framework . . . . .	256
8.9	Selected metamodeling metrics for metamodels within the Eclipse Modeling Framework . . . . .	256
8.10	Evaluation of the information encoding capacity of IAML visual notation . . . . .	259
C.1	Evaluations of model-driven technologies against OpenBRR ratings (1) . . . . .	332
C.2	Evaluations of model-driven technologies against OpenBRR ratings (2) . . . . .	333
C.3	Evaluations of model-driven technologies against OpenBRR ratings (3) . . . . .	334
G.1	File-based metrics of the development of Ticketsf within Symfony 1.0.12 . . . . .	358
G.2	File-based metrics of the development of Ticketsf-mini within Symfony 1.4.8 . . . . .	359
G.3	File-based metrics of the development of Ticketiaml within IAML 0.6 . . . . .	360

# List of Figures

1.1	The general methodology of design science research . . . . .	5
2.1	A reasonable architecture for Rich Internet Applications . . . . .	10
2.2	A sample WebML <i>hypertext model</i> . . . . .	21
2.3	A sample UWE <i>navigation structure model</i> . . . . .	22
2.4	The conceptual structure of the <i>Ticket 2.0</i> benchmarking application . . . . .	27
3.1	The Metamodelling Architecture of MDA . . . . .	39
3.2	The Viewpoint Architecture of MDA . . . . .	41
3.3	The layers of UML under the MDA . . . . .	42
3.4	Using both architectures of the MDA to describe the model-driven development of RIAs	44
3.5	Dealing with different Modelling Spaces . . . . .	45
3.6	The Model Transformation Pattern in MDA . . . . .	46
3.7	Using change models for round-trip engineering . . . . .	47
3.8	The model completion process within model-driven development . . . . .	52
3.9	Definition of the rule program <i>C</i> for the <i>default checkbox rule</i> convention . . . . .	54
3.10	Model completion <i>C(A)</i> . . . . .	54
4.1	Managing complexity through horizontal partitioning . . . . .	64
4.2	Managing complexity through hierarchical modelling . . . . .	64
4.3	Managing complexity through aspect-oriented modelling . . . . .	65
4.4	Event-Condition-Actions in UML activity diagrams . . . . .	67
4.5	An AORML Interaction Pattern diagram . . . . .	67
4.6	Modelling Reaction Rules in URML . . . . .	68
4.7	Visual Event Handler definitions in Kaitiaki . . . . .	68
4.8	Operation modelling using UML activity diagrams . . . . .	70
4.9	Operation modelling of an online shopping business process using BPMN . . . . .	71
4.10	The built-in datatype hierarchy of XML Schema Datatypes . . . . .	73
4.11	Representing XML Schema Datatypes within the metamodelling architecture of MDA using modelling spaces . . . . .	74
4.12	A sample ER Diagram and Data Object Table . . . . .	75
4.13	A sample UML class diagram . . . . .	76
4.14	The Database Broker design pattern represented as a UML sequence diagram . . . . .	78
4.15	The Iterator design pattern using parameterised types . . . . .	78
4.16	Component lifecycle events in OSGi . . . . .	81

5.1	The hybrid modelling language process model used in the development of IAML . . .	87
5.2	Hierarchical Modelling in IAML . . . . .	91
5.3	An overview of some of the important concepts of modelling RIAs using IAML . . .	94
5.4	The generated implementation of the IAML overview model instance . . . . .	94
5.5	Dependencies between packages of related model elements within IAML . . . . .	96
5.6	Layered architecture diagram illustrating the layered design of packages within the IAML metamodel . . . . .	97
5.7	A partial syntax of first-order logic . . . . .	98
5.8	UML class diagram for the <i>Core</i> Package: Logic Model . . . . .	99
5.9	UML class diagram for the <i>Core</i> Package: Function Model . . . . .	100
5.10	UML class diagram for the <i>Core</i> Package: Event-Condition-Action Model . . . . .	102
5.11	Ticketiaml: An <i>ECA Rule</i> connecting an <i>Event</i> to an <i>Operation</i> . . . . .	103
5.12	The <i>Simple Condition</i> decomposition of a single predicate using variables and domain types as terms, represented in terms of UML syntax . . . . .	104
5.13	The model developer view of the <i>Simple Condition</i> decomposition . . . . .	104
5.14	VisualAge for Smalltalk: Using Connections . . . . .	105
5.15	UML class diagram for the <i>Core</i> Package: Wires Model . . . . .	105
5.16	UML class diagram for the <i>Core</i> Package: Constructs Model . . . . .	106
5.17	Adapting the metamodeling architecture of MDA to the use of primitive type systems	107
5.18	UML class diagram for the <i>Domain</i> package of IAML . . . . .	112
5.19	Ticketiaml: The defined <i>Domain Attributes</i> of the <i>Domain Type Event</i> . . . . .	113
5.20	If domain types are defined using metamodeling, the resulting models are <i>incompatible</i> in terms of the MDA metamodeling architecture . . . . .	115
5.21	Defining instances using references instead of instantiation is MDA-compatible . . .	116
5.22	UML class diagram for the <i>Domain Instances</i> package of IAML . . . . .	118
5.23	Ticketiaml: Selecting an instance of the <i>Event Domain Type</i> through a <i>Domain Iterator</i> connected to a <i>Domain Source</i> . . . . .	119
5.24	Ticketiaml: Creating a new instance of an <i>Event</i> using a <i>Domain Iterator</i> . . . . .	121
5.25	Representing complex behaviour within IAML using two <i>ECA Rules</i> . . . . .	126
5.26	UML class diagram for the <i>Activity</i> Package: Operations Model . . . . .	128
5.27	The same complex behaviour of Figure 5.25 expressed with an <i>Activity Operation</i> .	129
5.28	UML class diagram for the <i>Activity</i> Package: Execution Model . . . . .	130
5.29	UML class diagram for the <i>Activity</i> Package: Data Flow Model . . . . .	131
5.30	UML class diagram for the <i>Activity</i> Package: Predicates Model . . . . .	132
5.31	UML class diagram for the <i>Wires</i> package of IAML . . . . .	132
5.32	Ticketiaml: Connecting a <i>Detail Wire</i> to a <i>Domain Iterator</i> . . . . .	134
5.33	An instance of autocomplete in Gmail . . . . .	135
5.34	Autocomplete implemented in IAML using an <i>Autocomplete Wire</i> . . . . .	135
5.35	The user interface for an <i>Autocomplete Wire</i> , as generated by the proof-of-concept implementation of IAML . . . . .	136
5.36	UML class diagram for the <i>Users</i> package of IAML . . . . .	137
5.37	UML class diagram for the <i>Access Control</i> Package . . . . .	138
5.38	Ticketiaml: Protecting access to a <i>Scope</i> via an <i>Login Handler</i> and <i>Access Control Handler</i> . . . . .	139

5.39	UML class diagram for the <i>Scopes</i> package of IAML . . . . .	141
5.40	Ticketiaml: Protecting access to a <i>Scope</i> via an entry <i>Gate</i> . . . . .	142
5.41	UML class diagram for the <i>Messaging</i> package of IAML . . . . .	144
5.42	Instantiation of user interface elements in WebML . . . . .	145
5.43	Instantiation of user interface elements in UWE . . . . .	146
5.44	UML class diagram for the <i>Visual</i> package of IAML . . . . .	147
5.45	Ticketiaml: Designing the user interface for the <i>Browse Events</i> page using <i>Visible Things</i>	151
5.46	The consequences of unifying instance types within the metamodeling architecture of the MDA . . . . .	153
5.47	A proposed refactoring of the IAML metamodel to unify the instantiation of <i>Domain Instances</i> and <i>Visible Things</i> . . . . .	154
6.1	Using ArgoUML to define a UML class diagram . . . . .	165
6.2	Defining a metamodel instance using Marama . . . . .	166
6.3	The GMF Framework Wizard in Eclipse . . . . .	170
6.4	The <i>Shape Designer</i> of Marama showing the shape design and corresponding concrete view . . . . .	171
7.1	Overall UML component diagram of the Proof-of-concept Implementation of IAML	196
7.2	UML component diagram of the <i>IAML Model</i> Component Decomposition . . . . .	197
7.3	UML component diagram of the <i>Graphical Editor</i> Component Decomposition . . .	199
7.4	Implementation of a graphical editor for IAML model instances using the Graphical Modeling Framework . . . . .	200
7.5	Illustrating generated and overridden elements in the IAML editor . . . . .	208
7.6	UML component diagram of the <i>Model Completion</i> Component Decomposition . . .	210
7.7	UML component diagram of the <i>Code Generation</i> Component Decomposition . . .	215
7.8	UML component diagram of the <i>Model Verification</i> Component Decomposition . . .	219
7.9	Visualisation of model instance constraint violations within a diagram editor . . . .	220
7.10	Illustrating code coverage of code generation templates . . . . .	225
8.1	A screenshot of the <i>Browse Events</i> page implemented in Ticketiaml . . . . .	251
8.2	A screenshot of the <i>View Event</i> page implemented in Ticketiaml . . . . .	251
8.3	A screenshot of the <i>Browse Events</i> page implemented in Ticketsf-mini . . . . .	253
8.4	A screenshot of the <i>View Event</i> page implemented in Ticketsf-mini . . . . .	253
8.5	Comparison of the number of classes ( <i>NoC</i> ) and abstract classes ( <i>NoAC</i> ) of the IAML metamodel against other similar metamodels . . . . .	257
8.6	Comparison of the total number of references ( <i>TNoR</i> ) and attributes ( <i>TNoA</i> ) of the IAML metamodel against other similar metamodels . . . . .	258
9.1	Two approaches in defining a <i>Complex Term</i> that references a <i>Value</i> contained within a <i>Changeable</i> as a <i>Parameter</i> . . . . .	269
B.1	Ticketiaml: The completed contents of the <i>edit event Domain Iterator</i> . . . . .	327
B.2	Ticketiaml: The completed contents of the <i>Event-typed Domain Instance</i> . . . . .	327
B.3	Ticketiaml: The completed contents of the <i>Edit Event Input Form</i> . . . . .	328

B.4	Ticketiaml: The completed contents of the <i>title</i> <b>Input Text Field</b> , within the <i>Edit Event</i> <b>Input Form</b> , to implement synchronisation . . . . .	328
B.5	A screenshot of the <i>Edit Event</i> page implemented in Ticketiaml . . . . .	329
E.1	Creating a new IAML model instance through the Eclipse wizard menu (1) . . . . .	341
E.2	Creating a new IAML model instance through the Eclipse wizard menu (2) . . . . .	341
E.3	The IAML visual editor for a new IAML model instance . . . . .	342
E.4	The default contents of the “Home” <b>Frame</b> within a new IAML model instance . . . . .	342
E.5	The tool palette of the IAML visual editor . . . . .	342
E.6	The IAML visual editor representing a <b>Frame</b> containing two <b>Input Text Fields</b> connected by a <b>Sync Wire</b> . . . . .	344
E.7	Generating the source code for an IAML model instance (1) . . . . .	344
E.8	Generating the source code for an IAML model instance (2) . . . . .	344
E.9	Screenshot of the generated tutorial application accessed through Mozilla Firefox . . . . .	345

# List of Listings

1	Part of a web application implemented in WebDSL . . . . .	23
2	Implementation of the <i>acyclical class inheritance</i> constraint of UML class diagrams in OCL . . . . .	49
3	The IDL definition of the MouseEvent interface . . . . .	69
4	EAnnotation documentation in Ecore . . . . .	164
5	Internet Application code generation using a JET template . . . . .	174
6	Internet Application code generation using an OAW Xpand Template . . . . .	175
7	Example rule implementation in Jena . . . . .	178
8	Example rule implementation in Jess . . . . .	179
9	Example rule implementation in Drools . . . . .	180
10	Implementation of <i>infinitely redirects</i> in Drools . . . . .	183
11	Implementation of <i>infinitely redirects</i> in Jena . . . . .	184
12	Implementation of <i>infinitely redirects</i> in OWL 2 Full . . . . .	186
13	Implementation of <i>infinitely redirects</i> in CrocoPat . . . . .	187
14	Implementation of <i>infinitely redirects</i> in Alloy . . . . .	188
15	Implementation of <i>infinitely redirects</i> in openArchitectureWare: Xtend . . . . .	188
16	Implementation of <i>infinitely redirects</i> in openArchitectureWare: Checks . . . . .	189
17	Implementation of <i>infinitely redirects</i> in OCL . . . . .	189
18	Implementation of <i>infinitely redirects</i> in LTL . . . . .	191
19	One Drools rule used in the model completion implementation of the <a href="#">Sync Wire</a> model element . . . . .	212
20	Implementation of the <code>runFrameEvents</code> code generation template in Xpand . . . . .	216
21	Implementation of the <code>containingScope</code> model extension in Xtend . . . . .	216
22	Implementation of a Checks constraint for <a href="#">Sync Wires</a> . . . . .	220
23	Implementation of an OCL constraint for <a href="#">Values</a> in the EMF Verification Framework . . . . .	221
24	Configuration details to host generated applications through the Apache HTTP Server . . . . .	344



# Chapter 1

## Introduction

The invention and introduction of the Internet in 1969, and more significantly the World Wide Web in 1990, has had a massive and permanent influence on our lives [175]. While originally designed as a purely informational medium, the web has evolved into an application medium supporting complex software systems accessible through different devices. This evolution culminated in the development of *Rich Internet Applications* (RIAs) that aim to unify traditional desktop applications with the distributed nature of the web [40].

### 1.1 Web Applications

As discussed by Berners-Lee and Fischetti [30] and illustrated in Table 1.1, the World Wide Web was originally envisaged as a medium to share documents, but the publication of dynamic content soon followed. These dynamic server-side applications became the first *web applications*, which reversed the conventional architecture of software applications at the time. Conventional software applications were generally designed to be installed and executed on a single machine. Web applications are instead designed to be executed remotely on a *web server*, and the local *client* provides an interface to the application within a *web browser*.

By considering web browsers as thin-client interfaces to services operating remotely over a network, web applications could support significant benefits in the delivery and operation of software. For example, processing power could now be centralised, reducing the performance requirements of clients; updates could be applied directly to the server, lowering the total cost of ownership of the software [216]; and new interface standards such as HTML [333] would guarantee that the software could be accessed by a wider audience using a variety of devices.

As the popularity of the World Wide Web increased, new concepts and technologies soon followed, such as HTML [333], XML [342] and CSS [344]. Many of these web technologies were standardised by the World Wide Web Consortium (W3C) after its foundation in 1994. Over time, web applications became more distributed and independent, and *service-enabled web applications* [217] could be developed using technologies such as WSDL [335], REST [96] and RSS [290]. These technologies enabled web applications to aggregate content from many remote data sources as *mashups* [371].

However, these conventional web applications suffered from poor responsiveness; could not be used offline; and their interfaces were not as responsive as desktop applications [40]. This was because all interaction had to adhere to request-response cycles over slow network connections. The device-independent accessibility of web applications was also hindered by continuing *browser wars* [373],



Year	Era	Intended Use	Web Technologies	Web Browsers
1990	World Wide Web	Document sharing	HTML Tags, HTTP 0.9	WorldWideWeb
1991				
1992		Server-side scripting	CGI	Mosaic
1993				Netscape Navigator
1994	Web Applications	Interactivity	Java Applets, HTML 2.0	Internet Explorer
1995				Opera
1996		Client-side scripting	CSS, ActiveX, Javascript	
1997				
1998	Service-Enabled Web Applications	Web services	HTML 4.0	Mozilla
1999			ASP, CSS 2, XML, WML	
2000			RSS, HTML 4.01, JSP	
2001			PHP 4, XHTML, REST	
2002			WSDL 1.1	Mozilla Firefox
2003			SOAP 1.2	Safari
2004		Semantic markup	RDF, OWL	
2005			OpenID, Javascript 1.6	Flock, Opera Mini
2006	Rich Internet Applications	Mashups, AJAX	XMLHttpRequest	
2007			WSDL 2.0	Safari for iPhone, Prism
2008		Push events		Google Chrome
2009				Android
2010		Cloud computing	OWL 2, OCCI	
2011			OAuth	
2011	...	...	HTML 5, CSS 3, ...	...

Table 1.1: A brief history of Web application technologies

where different web browsers would often fail to comply with the specifications of established web application technology standards [320].

In the early 2000s, the use of client-side scripting to dynamically modify the web application (known as *dynamic HTML*, or DHTML) became popular, and in 2005 the term AJAX – standing for Asynchronous Javascript and XML – was introduced by Garrett [121] to describe the common combination of several new web browser technologies such as XHTML [346], DOM [336], and the *XMLHttpRequest* client-side API [348]. Through these technologies, richer applications attempted to reduce the impact of slow network connections and improve the interactivity of their interfaces. For example, the input validation of a text field could now occur asynchronously through background network requests, rather than solely through a request-response cycle to the server.

The term “Rich Internet Application” was first introduced in a Macromedia white paper in 2002 [6], describing the unification of traditional desktop and Web applications in an attempt to leverage the advantages and overcome the drawbacks of these architectures [40]. Rather than simply providing new types of user interface elements, RIAs aim to improve all of the aspects of interactivity, accessibility and reliability in a web application.

The range of technologies used to implement RIAs continued to expand, and RIA development currently requires the integration of dozens of different technologies and concepts [246]. Mikko-nen and Taivalsaari [233] have argued that the complex and tangled structure of RIAs resemble the “spaghetti programs” of the 1960s and 1970s. Disciplines such as *Web Engineering* have been established to improve the quality of Web applications [175, pg. 3], but the development effort necessary to combine all of these technologies together continues to be an issue.

## 1.2 Modelling

One successful approach to reducing the complexity of developing large software systems is through abstraction of the underlying system into *models* [102, 302, 305]. These models may serve as documentation to the underlying system, but these abstractions may rapidly become outdated as the underlying system changes. Model-driven development (MDD) encourages the use of models as first-class citizens in the development process, where the system may be automatically generated from its model [305]. Modelling is particularly beneficial for rapidly-evolving domains, as a modelled system is less sensitive to the changes in its underlying platforms.

If a system is modelled effectively against a well-defined modelling language, the model may also be used to predict the behaviour of the system, and at a significantly lower cost than the implementation of the modelled system [305]. For example, model instance verification tools may evaluate the design and behaviour of a system model against design constraints; to verify that system requirements are satisfied; or to identify potential security flaws.

Previous studies have shown promising results for the model-driven development of web applications [302] through existing web application modelling languages such as WebML [51], UWE [190] and UML [254]. For example, Acerbis et al. found through a case study that applications modelled in WebML only needed a third of the development effort compared to using conventional development techniques [1]. However, these existing modelling languages were designed to support conventional web applications, and it is not clear whether these languages can support essential RIA concepts such as client-side events and user interaction.

In order to obtain the benefits of applying model-driven techniques to web application development – such as improved productivity, security and documentation – this thesis will investigate the development of a modelling language for RIAs, and evaluate the impact of using this language on RIA development productivity. It is unclear whether the fundamental concepts of RIAs can be “bolted on” to an existing language, or whether a language will need to be designed from scratch with these concepts in mind.

## 1.3 Research Questions

This thesis will therefore look at the application of modelling for the development of Rich Internet Applications, and leads to the thesis statement:

*The development of a modelling language for Rich Internet Applications will address many of the challenges currently faced in web development; in particular, improving the security of the modelled applications, and increasing the productivity of their development and maintenance.*

This thesis statement can be broken down into six key research questions, which will each be addressed throughout the course of this thesis, and summarised later in Section 9.2:

1. What unique challenges do Rich Internet Applications introduce into the process of their development and maintenance?
2. Are there shortcomings in using existing modelling languages to model Rich Internet Applications?

3. If existing modelling languages cannot be used to develop Rich Internet Applications, can a modelling language be developed to address these challenges, either from the extension of an existing language or the development of a new language?
4. Are there techniques and technologies that may be used to improve the maintainability of a modelled Rich Internet Application?
5. Can it be shown that an Rich Internet Application modelling language increases the security of the underlying system?
6. Can it be shown that such a modelling language improves the development process of these web applications, in terms of speed, simplicity and consistency?

## 1.4 Research Design

*Design science research* is a rapidly evolving field which “involves the design of novel or innovative artifacts” such as models and methods to improve and understand the behaviour of Information Systems [326]. Vaishnavi and Kuechler [326] argue that design can be considered an appropriate technique for conducting research, and propose a method for performing design science research effort, reproduced here in Figure 1.1.

Design science research, by definition, changes the state-of-the-world through the introduction of novel artefacts [326]. As an important contribution of this thesis is in the design and development of a modelling language as an artefact, a design science research approach may describe the overall research framework. The overall structure of this thesis therefore follows these proposed steps:

1. The **Awareness of Problem** step is performed in Chapter 2, where recent activity in the development of RIAs is discussed, and existing web application modelling languages are evaluated.
2. The **Suggestion** step is “an essentially creative step wherein new functionality is envisioned based on a novel configuration of either existing or new and existing elements” [326]. In this thesis, this step is performed in Chapters 3–5, where existing research into modelling is discussed in Chapter 3; existing model-based design concepts are described in Chapter 4; and a new modelling language for RIAs is proposed in Chapter 5 based on new and existing elements.
3. The **Development** step is performed in Chapters 6 and 7, where potential implementation technologies are discussed and evaluated, and subsequently combined into a proof-of-concept implementation of the proposed modelling language. Vaishnavi and Kuechler argue that the novelty in the artefact is performed primarily in its design, rather than its development, however this proof-of-concept development is critical to validate the proposed design.
4. The **Evaluation** step is performed in Chapter 8, where the modelling language and its proof-of-concept implementation are evaluated according to a number of evaluation criteria. Unexpected evaluation results or anomalous behaviour can be used to suggest avenues for future research after additional library research has been performed.
5. Finally, the **Conclusion** step is performed in Chapter 9, where knowledge gained in the design effort is consolidated.

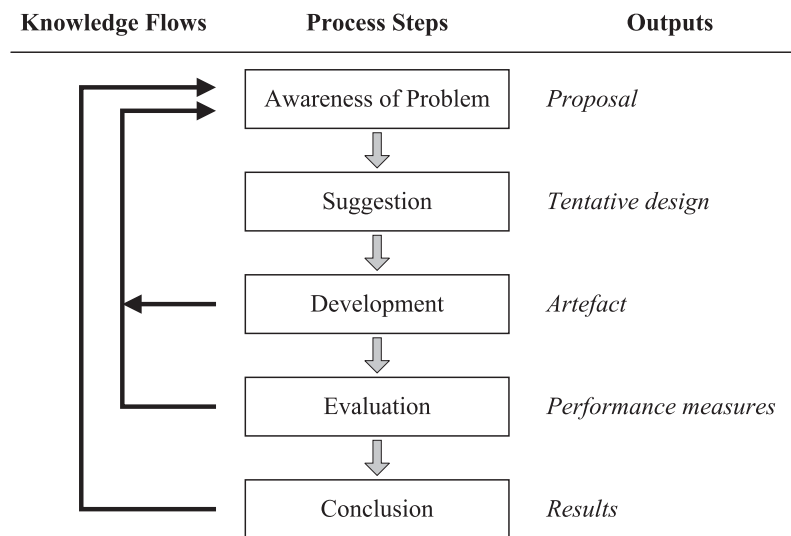


Figure 1.1: The general methodology of design science research, adapted from Vaishnavi and Kuechler [326]

The design of a piece of given research is also often described in terms of qualitative or quantitative research, as summarised by Creswell [64]. The research design is a major consideration in the process of designing the research method and selecting the types of research questions which will be asked, but these designs are not mutually exclusive. This research may be closely characterised as a qualitative research problem [239]:

1. *The concept is “immature” due to a conspicuous lack of theory and previous research.* While there is a large body of research, and to a lesser extent theory, into the related topics of Internet applications and *hypermedia* [275, 276], the body of work for RIAs is significantly less.
2. *A notion that the available theory may be inaccurate, inappropriate, incorrect, or biased.* The only available theory on RIAs are derived from the component topics of web applications and hypermedia, and each of these topics fail to consider fundamental concepts from the other.
3. *A need exists to explore and describe the phenomena and to develop theory.* By proposing a modelling language for RIAs, this thesis will attempt to identify the core concepts and constructs of RIAs. These proposals will help in developing theory for RIAs as a whole, although this is outside the scope of this thesis.
4. *The nature of the phenomenon may not be suited to quantitative measures.* The development of modelling languages is difficult and expensive, and their success is strongly linked to the success of their implementation [102]. Language evaluation also difficult and expensive, as it is difficult to provide enough real-world examples for conclusive evaluations.

However, it is important to note that while this thesis is primarily qualitative, some of the arguments presented will be supported with the results of quantitative experiments. This includes the performance evaluations of model completion by Wright and Dietrich [369], and through the measurement of system metrics of different benchmark application implementations in Section 8.3.

## 1.5 Research Method

To answer the research questions and address the statement of this thesis, a research method consisting of the following steps is proposed, and discussed in detail throughout the body of this thesis. The outcomes of several of these steps, each performed during the research process, were peer-reviewed and externally published:

1. Identify the requirements and features of Rich Internet Applications; identify the requirements and existing work into developing other modelling languages, including strengths and weaknesses and existing evaluations; and outline existing future work in this area. From this research, the decision whether to extend an existing language, or develop a new language from scratch, can be made. These requirements have been published by Wright and Dietrich [368].
2. Investigate and propose a number of methods in which to evaluate the proposed modelling language; this could include methods such as quality criteria, language metrics, expressibility checklists, user evaluations, and so on. In this thesis, one such method resulted in the development of the benchmarking application *Ticket 2.0*, which has been published by Wright and Dietrich [367].
3. Investigate methods to reduce the effort necessary for a model developer to develop and maintain a model-driven implementation, possibly through the use of frameworks, knowledge inference or design conventions. Some of these results have been published by Wright and Dietrich [369].
4. Perform an extensive amount of prototyping and exploratory design work to adapt existing metamodels, propose new modelling constructs, and experiment with visual syntax. This will include evaluations of existing approaches to modelling fundamental RIA concepts such as events, access control, and user interfaces.
5. Define a modelling language for the development of RIAs through syntax and semantics definitions. This modelling language will be designed in a bottom-up fashion from a strong meta-model *core* supported by formal definitions, through to additional layers supported by this core.
6. Develop a proof-of-concept CASE tool to support this modelling language with a reference implementation, and implement this software by following an evolutionary software process model. Agile development processes will also be followed to improve feedback and evaluation, and to adapt to new technologies and concepts that emerge from the development community.
7. To ensure the simplicity and effectiveness of this modelling language, frequently evaluate the implementation against the final requirements and intended design, and refactor or remove meta-model elements from the language as appropriate.
8. Once the implementation of this modelling language is complete, evaluate the design implementation against the previously defined evaluation criteria to determine the strengths and weaknesses of the language.
9. Investigate existing formal verification methods and technologies, and prototype each approach within the proof-of-concept modelling language implementation, with the intent of developing a suite of model verification tools to evaluate different design and correctness properties against modelled web applications. Where appropriate, successful prototypes will be integrated directly into the implementation environment itself.

## 1.6 Thesis Outline

The remainder of this thesis follows the order of the research method proposed in the previous section:

**Chapter 2** discusses the state of the art and related work in the field of modelling RIAs. To identify the detailed requirements of RIAs, a suite of 69 use cases are described in **Appendix A**. The chapter concludes with a short discussion on software engineering methods, one of which will be used in the proof-of-concept implementation.

**Chapter 3** provides an in-depth investigation into the many aspects and concepts involved in system modelling, such as domain-specific languages, model-driven development, visual modelling and model verification. The *model completion* concept is introduced and defined, which is used later to simplify the end-user development effort of the proposed language. The chapter concludes with a discussion on visual modelling techniques and best practices.

**Chapter 4** provides an in-depth discussion into all of the fundamental RIA concepts that will be considered as part of the proposed modelling language. For each concept, a variety of existing technologies used to implement the concept are investigated, and their relevant benefits and drawbacks are discussed.

**Chapter 5** details the design, approach and rationale behind the proposed modelling language, named the *Internet Application Modelling Language* (IAML). Modelling constructs are proposed for each fundamental RIA concept, along with a brief discussion on their rationale and intended behaviour. The syntax of the metamodel, intent of each model completion rule, and full functional definitions of each metamodel element, are provided in **Appendix I**.

**Chapter 6** details the evaluation process used to select the implementation technologies that form the basis of the proof-of-concept implementation in the Eclipse framework. This includes an evaluation of the quality of open-source components using OpenBRR in **Appendix C**.

**Chapter 7** provides a detailed decomposition of the proof-of-concept implementation in terms of the components necessary to implement the final system. This proof-of-concept implementation is supplied on the media attached to this thesis, as described in **Appendix E**.

**Chapter 8** evaluates the proof-of-concept implementation of IAML against the evaluation criteria discussed in the Background chapter. This includes an IAML implementation of the benchmarking application *Ticket 2.0* represented using XMI in **Appendix F**, and metric-based source code comparisons with a Symfony implementation of the same application in **Appendix G**.

**Chapter 9** concludes the thesis with a discussion on the overall research contributions and outlines possible areas of future research.



## Chapter 2

# Background

In this chapter, the key concepts and topics behind this research project will be discussed in further detail. The current state of the art in web application modelling research will be presented, along with a suite of requirements and evaluation techniques that have been developed to evaluate these modelling languages. This includes the definition of 69 use cases of RIAs, reproduced here in Appendix A.

### 2.1 Rich Internet Applications

Kappel et al. define a *web application*, introduced in the previous chapter, as “a software system based on technologies and standards of the World Wide Web Consortium (W3C) that provides Web specific resources such as content and services through a user interface, the Web browser” [175, pg. 2], and this definition is used in this thesis. The underlying architecture of the web application considerably influences its quality and reliability [48], and many potential architectures exist as discussed by Eichinger [88]; this thesis will assume that all web applications will follow the general application architecture illustrated in Figure 2.1.

As discussed by Nussbaumer and Gaedke [246], web application development involves the particular combination of components and technologies within an architecture, and may include:

1. *Presentation technologies* such as various HTML dialects [333, 343, 345, 346] or CSS [344];
2. *Server-side languages* such as C/C++, PHP, Java, Ruby, Perl, Python, or Haskell;
3. *Client/server communication protocols* such as SMTP, HTTP, cookies, or sessions [246];
4. *Configuration file formats* such as XML [342], YML [274], JSON [65] or flat text files;
5. *Application platforms* such as Flash [69], Silverlight [360], J2ME [323] or Java Applets [71];
6. *Web service languages* such as SOAP [340], REST [96], XML-RPC [363], or WSDL [335]; and
7. *Database query languages* such as versions of SQL (MySQL, PostgreSQL, NoSQL and others), OQL, or LINQ.

One aspect of the complexity of web application development is due to the combinatorial nature of these technologies and their interaction. For example, different *content markup languages*, such as HTML or XHTML, are each implemented differently within different web browsers, and may interact differently when a CSS stylesheet is applied. The *Acid* series of web browser tests were created in an



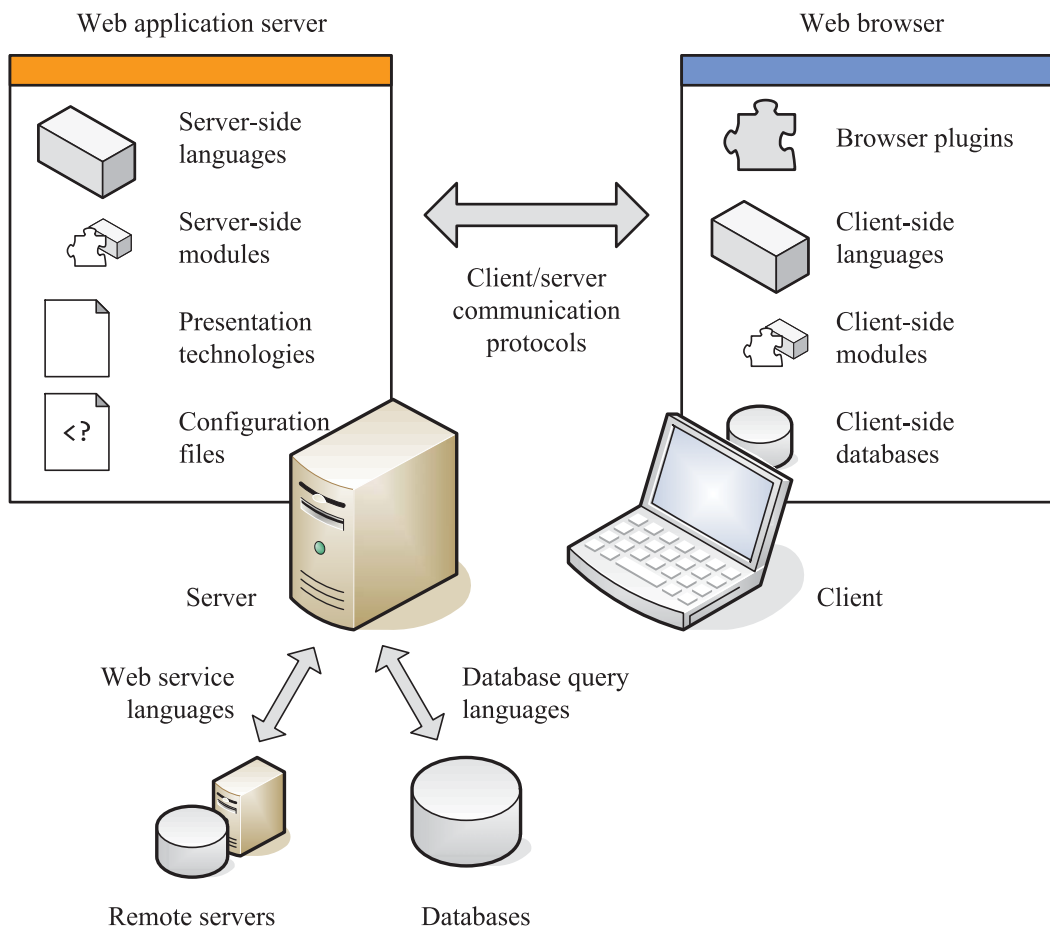


Figure 2.1: A reasonable architecture for Rich Internet Applications

effort to improve the standards-compliance of web browsers [43, 320]; but many web browsers still fail to fully implement these standards correctly, hindering web development [373].

Rich Internet Applications introduce new client-side interactivity technologies into this mix, as illustrated in Figure 2.1. In particular, the web application may incorporate some of the following client-side technologies, increasing the complexity of interactions between the technologies used in its implementation<sup>1</sup>:

1. *Client-side scripting languages* such as Javascript [86, 99] or VBScript [183];
2. *Client-side modules* such as CommonJS [200]; and
3. *Client-side databases* such as Google Gears [129] or HTML 5 [343].

A particular RIA may also be classified as an AJAX application, which Garrett defines as the combination of XHTML, CSS, DOM, XML, Javascript and the *XMLHttpRequest* API [121]. Not all RIAs will use all of these different implementation technologies<sup>2</sup>, and the specific technologies chosen will depend on the requirements of the application.

<sup>1</sup>For example, Flash is commonly used to play video content, but is not available on the iPhone platform [356]; however, older browsers cannot support the new video features of HTML 5 [343], so some form of backwards compatibility must be supported by the RIA.

<sup>2</sup>As an example later in this thesis, Section 8.3.1 discusses the implementation of a standard RIA which required the use of 16 different technologies.

### 2.1.1 Classifications of Rich Internet Applications

While the individual technologies and platforms selected for the implementation of a particular RIA may differ, the architectures of these applications may be classified into four categories, as described by Bozzon et al. [40]:

1. *Scripting-based*, in which client-side logic is implemented via scripting languages, and interfaces are based on presentation technologies such as HTML and CSS;
2. *Plugin-based*, where browser plugins integrate application platforms into the browser itself, such as Flash, Silverlight or Java Applets;
3. *Browser-based*, where client-side interactivity is natively supported by the browser in a browser-specific way, such as ActiveX [120] and XUL [39]; and
4. *Web-based desktop technologies*, where software applications are downloaded and executed over the Internet, but executed using a different platform, such as Java Web Start [219] and Adobe Air [3].

Since this work by Bozzon et al. was published, a fifth category has emerged, which will be proposed in this thesis as *desktop-based web applications*. These applications appear to be native applications to the end-user and are executed outside of the users' web browser, but are actually standard RIAs within a browser wrapper [282] through technologies such as Mozilla Prism [242] and Fluid [78]. However, these categories are not mutually exclusive; it is possible (and in many cases necessary) to have a RIA that covers multiple categories. For example, a client-side application (*scripting-based*) could also need to interact with the Google Gears plugin [129] (*plugin-based*) in order to emulate offline functionality.

Some related research in describing Rich Internet Applications adapts research from interactive *hypermedia* applications, which refers to the mixture of hypertext and multimedia [275, 276], although hypermedia usually refers to standalone proprietary user interfaces [21]. Web applications can be considered low-level hypermedia applications [202]; however web applications adds unique challenges to system requirements – such as security, performance, usability, navigability and internationalisation [124] – that existing surveys tend to neglect.

### 2.1.2 Interactivity

The term “interactivity” can be a controversial term, as discussed by Gane and Beer [119]. Interactivity does not refer to a list of requirements of an interface, but rather is a variable measure of the ways in which a system supports or requires interaction. Almost all types of media are interactive; Manovich [218] argues that even sculpture and architecture are interactive media, because they demand the viewer to move to appreciate the structure.

Consequently, it is not possible to compare the interactivity of two different media types, but the term rather refers to *differences* in the way a system is presented. The consensus seems to be that increased forms of interactivity are beneficial, and provide more usable and effective user interfaces [277]. However, it is widely accepted that conventional web applications have *less* interactivity than Rich Internet Applications [40, 276, 368].

### 2.1.3 Enterprise Software

Instances of software can be categorised according to their challenges, development complexities and intended domain [175, pg. 5–7], and certain applications may be categorised as *enterprise software*. There is no precise definition of an “enterprise application”, however they often possess common characteristics and design goals.

Some of the characteristics of enterprise software, in terms of functional and non-functional requirements, are discussed by Fowler [101]. These characteristics include needing to persistently store a lot of data; supporting concurrent data access; handling many user interface screens; expressing complex business logic; integrating with other enterprise applications; and supporting the reuse of data in semantically diverse ways.

Enterprise applications must also possess reasonable performance characteristics, and this often dictates the architectural decisions of an enterprise system. This non-functional requirement is often measured using metrics such as scalability, latency and throughput. To improve the success rate of developing enterprise applications, certain application design patterns should be used [101, pg. 6–13].

### 2.1.4 Security Risks of Rich Internet Applications

As Wimmer et al. [362, pg. 265] discuss, web applications are expected to be secure, and present security characteristics that demand comprehensive security techniques compared to conventional software applications. The ISO/IEC 9126 software quality model defines *security* as a functional characteristic, in “the capability of the software product to protect information and data so that unauthorised persons or systems cannot read or modify them and authorised persons or systems are not denied access to them” [166, pg. 14].

In order to identify the most common security risks that web application developers need to consider, the *Open Web Application Security Project* (OWASP) maintains a list of the top ten security vulnerabilities for web applications, ranked according to subjective risk factors [319]. As of 2010, the top ten security risks were:

1. *Injection*, such as permitting SQL, PHP or remote code injection;
2. *Cross-site scripting* (XSS), permitting user sessions and cookies to be read or hijacked;
3. *Broken authentication and session management*, such as incorrectly-implemented sessions or authentication routines;
4. *Insecure direct object references*, such as directly exposing files or databases;
5. *Cross-site request forgery* (CSRF) through cookies, such as by forging HTTP requests;
6. *Security misconfiguration*, such as not keeping software up-to-date, or using insecure defaults;
7. *Insecure cryptographic storage*, such as not encrypting or hashing sensitive data (e.g. passwords or credit card details);
8. *Failure to restrict URL access*, such as incorrectly configured access control checks;
9. *Insufficient transport layer protection*; and
10. *Unvalidated redirects and forwards*, which can forward users to insecure pages.

As a class of web application, RIAs inherit the security risks of web applications; for example, the injection of Javascript in order to read or hijack the state of a client-side application could be enabled through cross-site scripting. At the time of writing, there was no known database of common vulnerabilities for RIAs in particular.

Alternatively, one may measure the security of running software applications using metrics. For example, the ISO/IEC 9126 standard on software quality defines security using internal metrics such as *access auditability*, *data corruption prevention* and *data encryption* [167]. All of these metrics are measured through the audit of an implemented system against the identified functional requirements of the system.

## 2.2 Frameworks

A practical solution to bridging the conceptual gap in web development is through the use of a framework, which is “a reusable, ‘semi-complete’ application that can be specialised to produce custom applications” [93, pg. 32]. By taking a pragmatic approach, frameworks are popular in industry due to rapid development time and stability. The distinctions between frameworks, programming languages and modelling languages are discussed later in Section 3.1.11.

The Ruby on Rails framework [321] has arguably been one of the most influential frameworks for the development of web applications, which inspired the Symfony for PHP [274] framework. Other popular server-side frameworks include Struts for JSP [10]; Seaside for Smalltalk [90]; Spring for Java [172]; and Spring.NET for ASP.Net [312].

The common tasks faced by the client-side scripting aspect of RIAs has also been addressed through a number of client-side frameworks, such as jQuery [208] and the Prototype Javascript Framework *prototype.js* [280]. Recently, some web application frameworks such as Google Web Toolkit [147] and Symfony [274] have combined both server-side and client-side functionality, to provide the first *Rich Internet Application frameworks*.

Frameworks solve a number of problems in complex software development. They often provide or generate common infrastructural source code<sup>3</sup> specific to a particular domain; for example, Symfony can automatically generate object-relational mapping source code from a defined Propel schema [274]. They can also provide an abstraction layer upon a number of different technologies; for example, PDO for PHP [207] provides database abstraction to ten different relational database managers.

Finally, client-side frameworks can abstract away differences in browser implementations; for example, the Prototype Javascript Framework provides a common *Ajax* component, which performs identically across different browsers [280]. In these ways, frameworks can adapt to the fragmentation of different technologies, which is especially pronounced in the field of RIA development.

Extending a framework to work in different problem domains is a significant issue; in many cases, additional complexity must be introduced into the framework, or the framework may need to be re-architected. The runtime overhead of the necessary libraries that support a framework is another concern, as illustrated later in Section 8.3.1.

The distinction between a framework and a programming language is generally well-defined. While both aspects provide a level of abstraction, a programming language usually has its own syntax, semantics and implementation into a lower target language (e.g. through compilation) [299]; whereas

---

<sup>3</sup>This is often called *boilerplate code*, referring to how the code can often be included in new contexts without requiring significant change.

a framework reuses (or extends) the syntax and implementation of existing programming languages. Similarly, the distinction between a framework and a modelling language is also generally defined in the same way; that is, a modelling language has its own syntax, semantics and optional implementation, whereas a framework reuses an existing programming language.

A framework is therefore ideal for providing a domain-specific abstraction within an existing general-purpose software development environment, whereas a modelling language is ideal for providing the same abstraction in a platform-independent manner. In both cases, however, a framework can be used as part of the implementation of either a programming language or a modelling language; although Kelly and Pohjonen note that many modelling languages suffer the problem of adhering too closely to a particular framework [179].

## 2.3 Features of Rich Internet Applications

In order to effectively evaluate existing approaches in modelling RIAs, comprehensive sets of evaluation criteria need to be defined. In this research, two sets of evaluation criteria were defined. The first set of criteria define a general list of feature categories that can be evaluated subjectively against a modelling approach, as defined by Wright and Dietrich in 2008 [368]. The second set of criteria focuses solely on the functionality of modelling approaches to define a comprehensive list of modelling requirements, as defined by Wright and Dietrich in 2008 [367].

### 2.3.1 Feature Categories

By investigating seven of the most popular Rich Internet Applications at the time of writing<sup>4</sup>, new features enabled by the introduction of these technologies could be identified. These features were summarised into the thirteen overall feature categories discussed by Wright and Dietrich [368], which will now be briefly summarised in this section. This process was not designed to conclusively capture all of the features of RIAs, but rather sets the stage for the comprehensive requirements analysis described later.

#### Events

Web applications are inherently event-driven, as web servers respond to incoming request events from clients. Events are particularly important to RIAs as they are more sensitive to interactivity and asynchronous events, and events can either occur locally (on the client) or remotely (on the server). Remote events permit a client to communicate with the server and with other clients, and the server can communicate directly to the client (e.g. through a pushlet).

Since events are such a fundamental part of the web, it may be useful to promote events as *first-class citizens*<sup>5</sup> in a RIA modelling language. A RIA modelling language should be able to describe events, which may in turn conditionally execute defined actions.

<sup>4</sup>These seven web applications were also used to investigate the use cases of modelling RIAs. As listed in Section A.2, these web applications were Gmail, Google Calendar, Google Reader, Google Docs, Last.fm, Google Pages, and Facebook.

<sup>5</sup>A *first-class citizen* represents an element that can be represented both directly as an element *and* indirectly by a variable or expression, as opposed to a second-class citizen, which can only be used directly as an element [316].

## Browser Interaction

The web browser is the client-side user interface of a web application [175], and consequently must be considered when modelling RIAs. A RIA modelling language should be able to model navigation concepts, such as the “back” button or out-of-order navigation; storing data on the client through cookies or offline databases; opening additional windows; identifying available scripting and plugins support; and being able to identify the user agent, and being able to respond appropriately [368].

## Lifecycle Management

The events and conditions surrounding the creation, termination, and state changes of a software element can be expressed as its *lifecycle*. Lifecycles are more than just firing events, and can add contextual information to fired events, or group similar events in a well-defined way. While lifecycles are not necessary to implement an RIA, the concept can simplify the development of certain types of web application. For example, a JSP servlet may trigger events when the servlet is initialised or destroyed (the `init` and `destroy` methods) [159]; a browser window may trigger events when it is loaded or unloaded (the `onload` and `onunload` handlers) [333]; and a web server may trigger events when a session<sup>6</sup> is initialised or destroyed.

## Users

It is important to identify different web application users, identified using particular credentials with respect to a certain access policy. *Users* are therefore an important part of web applications, with the term *user* referring to both people and non-human entities<sup>7</sup>. Most web applications deal with the creation, management and authentication of user accounts, so a RIA modelling language should also promote users to first-class citizens. Users may also interact together in a collaborative way, such as through the collaborative editing interface of Google Docs.

## Access Control

Correctly implemented access control in a web application should be an inherent property of a designed application. Sensitive or private data should not be unnecessarily disclosed, and should be unavailable to users without appropriate permissions, but available to users that do possess the necessary permissions.

Access control<sup>8</sup> concerns include both authentication and authorisation, two distinct processes that are sometimes mistaken [228]. Authentication involves validating the identity of a certain party, often in order to permit them access. Authorisation, on the other hand, is the definition of which rights and permissions a certain party will have once access is granted. Open standards for distributed authentication and authorisation on the web have recently been announced, such as the OpenID [281] and OAuth [145] standards, respectively.

---

<sup>6</sup>In this thesis, a *session* is defined as “a sequence of related HTTP requests between a specific user and a server within a specific time window” [246, pg. 114].

<sup>7</sup>For example, software that utilises a web service can also be considered a *user*.

<sup>8</sup>Wright and Dietrich [368] instead refer to the *access control* feature category as *security*.



## Databases

The vast majority of web applications are concerned with the integration with databases, in essence becoming *data-intensive web applications* [52, 51]. Abstraction of data access to a persistent domain model can increase the database-independence, reliability and development productivity of the application [274], and a wide variety of existing domain models – such as UML class diagrams [254] and ER diagrams [54, 278] – already exist. A RIA modelling approach should also support the common use cases of users uploading content, and describing offline functionality provided by offline toolkits.

## Messaging

Messaging encompasses the entire concept of sending contextual data from one device to another. In web applications, e-mail was once the most popular and accessible form of messaging, but a rapid increase in communication mediums have introduced many new ways of communicating. In RIAs, this includes domains such as sending e-mails; sending short text messages to mobile phones; invoking and responding to web services; using RSS feeds and public APIs; and using OpenID authentication.

## User Interfaces

RIAs provide an extended suite of user interface elements to web browsers, such as maps, calendars and autocompletion [367]. A modelling language for RIAs should support a wide suite of existing and new user interface elements. The integration of the presentation and underlying models of a RIA tend to remain strongly correlated [11], often following best practice architectures such as the Model-View-Controller (MVC) design pattern [195]. A RIA modelling approach should therefore support these best practices, yet simultaneously not be restricted to a single architecture.

## Standards Support

Innovations on the web occur very quickly and frequently [246, pg. 111], and the most popular innovations codified and shared through standards, such as the W3C standardisation of HTML [333] and DOM [336, 334], and the ECMA standardisation of Javascript [99]. To improve acceptance by the web development community, a RIA modelling approach should embrace new innovations where possible, but overwhelmingly adhere to existing standards.

## Platform Independence

As discussed earlier, a web application may be implemented on any number and combination of technologies. Since a model is an abstraction, a RIA modelling approach should not be limited to a single combination; the model should be *platform-independent* to permit deployment on any number of platforms. A platform-independent model also allows for incorrectly implemented standards on certain platforms to be ignored, as workarounds can be deployed automatically. Platform independence also adheres to the viewpoint modelling architecture advocated by the OMG, as discussed later in Section 3.1.5.

## Use of Metamodels

A modelling approach should consider itself as part of a larger model-driven development community. The Model Driven Architecture (MDA), for example, discusses the interaction between models,

metamodels and the real world [80] in order to improve the quality and interchangeability of different model instances through standardisation [243]. A full discussion on modelling is provided in Chapter 3.

### Verification

It is possible to describe a model instance that is valid according to the syntax of a modelling language, but unusable according to informal standards outside of the language definition. For example, deadlocks may be expressed in most procedural programming languages, even though this is generally not desirable in software applications [149]. It is preferable to identify these errors as early as possible, as *defect amplification* dramatically increases the relative cost of correcting errors once they have been made [278, pg. 197–205].

*Verification* allows the analysis of such model instances to identify these problems, and this is discussed in greater detail in Section 3.3. Verification is already used against conventional web applications for scenarios such as detecting broken links or unreachable code [315], but there is little research in using verification against RIAs. For example, some RIAs allow multiple users to work concurrently on a single document; a verification tool to locate potential deadlock and resource starvation code may be beneficial. Consequently, a RIA modelling language should investigate integrating these checks into its implementation.

### Software Support

Finally, a RIA modelling language should be supported by software tools, such as a dedicated CASE tool for designing model instances, code generators, or analysis tools [300]. Proof-of-concept implementations of a language can be used as a reference implementation for further work, and can increase language acceptance by the development community [102]. For model-driven approaches, Kent [180] argues that a software implementation is essential in order to maximise the benefits of using models. However, a single software implementation can be at odds with keeping the language platform-independent; to an extent, platform- or technology-dependent implementation concerns should be prevented from modifying the design of the modelling language [179].

#### 2.3.2 Detailed Modelling Requirements

While these feature categories are useful for quickly evaluating a RIA modelling approach, their subjective nature is not useful in performing a detailed evaluation, or to specifically identify missing functionality. Subsequently, the functionality of seven popular Rich Internet Applications were investigated in detail, as discussed in Appendix A. Each application was investigated in order to identify the individual features that are a result of the RIA domain, rather than from business or design decisions. For each requirement, a *use case* was filled out, describing the actors, sequence, pre- and post-conditions, exceptions, related use cases and other interesting comments [278]. From this survey, 69 use cases of RIA features were identified; this suite of use cases is reproduced in Appendix A.



#	Requirement	Example	Basic RIA
<b>Data</b>			
D1	Static Pages	Gmail: Static help pages	✓
D2	View Data	Gmail: View an e-mail	✓
D3	Update Data	Gmail: Create an e-mail	✓
D4	Pagination	Gmail: Display e-mails in pages	✓
D5	Provide Data Feed	Last.fm: Provide RSS feed of recommendations	✓
D6	Use Web Services	Calendar: Use external iCal feed	✓
D7	Offline Data	Reader: Download new feeds before going offline	
D8	Offline Resources	Reader: Download resources before going offline	
D9	Web Service Provider	Facebook: Provide Facebook application using API	✓
D10	Uploading Files	Gmail: Adding attachments	✓
D11	Access Server Data	Gmail: Download new message headers	✓
D12	Local Variables/Data	Docs: Download document source to client	✓
D13	Cookies	Gmail: Recall last input language	✓
<b>Events</b>			
E1	Scheduled Events	Calendar: Event reminders on client and server	✓
E2	Client Timer Support	Gmail: Check server for new e-mails	✓
E3	Server Timer Support	Gmail: Check POP3 servers for new e-mails	✓
E4	Async Form Validation	Last.fm: Check in entered event artist data	✓
E5	Client Form Validation	Gmail: Warn user if subject is missing	✓
E6	Server Form Validation	Gmail: Sending an e-mail to an invalid address	✓
E7	User Collaboration	Docs: Two users can work on the same document	
E8	Browser-Based Chat	Gmail: Google chat	✓
E9	Out-Of-Order Events	Docs: Dealing with edit events with multiple users	
E10	Server Transactions	Gmail: Purchasing more storage space	
<b>Users and Access Control</b>			
S1	User Authorisation	Gmail: Sign in	✓
S2	Session Support	Gmail: Sign in	✓
S3	User Logout	Gmail: Sign out	✓
S4	Automatic User Auth	Gmail: Log in automatically	✓
S5	User Access Control	Calendar: Only certain users can access a calendar	✓
S6	Group Access Control	Calendar: Shared calendars secured to certain groups	✓
S7	Access Control Levels	Calendar: Read/write/change sharing permissions	✓
S8	Single Sign-In Solutions	Google Services; OpenID	✓
S9	Personalisation	Calendar: Display a custom timetable format	✓

Table 2.1: Fundamental modelling requirements of Rich Internet Applications (1), adapted from Wright and Dietrich [367]

#	Requirement	Example	Basic RIA
<b>User Agents</b>			
A1	Browser Identification	Gmail: Redirect user if user agent fails requirements	✓
A2	User Redirection	Gmail: Redirect to e-mail web links	✓
A3	Multiple Browser Support	Gmail: Load different interfaces depending on agent	✓
A4	Multiple Outputs	Calendar: Provide a feed in iCal, XML, HTML	✓
A5	Client-Side Application	Gmail: Webmail application	✓
A6	Load Additional Scripting	Gmail: <i>Contacts</i> menu loads another script	
A7	Back Button Control	Gmail: A user cannot go back once logged out	✓
A8	Plugin Support	Gmail: Play MP3 attachment	
A9	Plugin Communication	Last.fm: Clicking on a track updates the Flash player	
A10	Navigation Control	Gmail: Update URL fragment identifier	✓
<b>Interaction</b>			
T1	E-Mailing Users	Gmail: Can send e-mails	✓
T2	E-Mail Unsubscription	Facebook: User can unsubscribe from all e-mails	✓
T3	Mobile Phone Comm.	Calendar: Can send text message reminders	✓
T4	Internationalisation Support	Last.fm: Different locales	
T5	Multiple Domain Support	Last.fm: Different domains display different locales	
<b>User Interface</b>			
U1	Presentation	Calendar: Displaying a particular user interface	
U2	Client-Side Scripting	Gmail: Home page displaying available space	✓
U3	Drag And Drop	Calendar: Can drag and drop events	✓
U4	Loading Time Support	Gmail: Switch to HTML view after 30 seconds	✓
U5	Keyboard Shortcuts	Calendar: Can browse using keyboard	✓
U6	Opening New Windows	Pages: Open links in new windows	✓
U7	Pop-Up Dialog Boxes	Gmail: Can compose an e-mail in a new window	✓
U8	Runtime Interface Updates	Gmail: Update <i>Unread Mails</i> in real time	✓
U9	Static Views (HTML)	Gmail: Provide a static HTML view	
U10	Modal Dialogs	Pages: Inserting an image shows a modal dialog	✓
U11	Use External Components	Facebook: Transitions with <i>script.aculo.us</i>	
U12	Provide External Libraries	Gmail, Calendar: A consistent calendar input box	✓

Table 2.2: Fundamental modelling requirements of Rich Internet Applications (2), adapted from Wright and Dietrich [367]

These use cases were then summarised into a suite of 59 individual functionality requirements, as described by Wright and Dietrich [367] and republished here in Tables 2.1 and 2.2<sup>9</sup>. Some of these requirements may be the result of business decisions – such as *Internationalisation* and *Use Web Services* – but they still represent functional requirements due to the technical nature of their implementation.

For each requirement, a real-world example of its usage is included, with respect to the seven RIA applications selected in the previous appendix. This suite of detailed requirements may then be used to perform a detailed evaluation of a modelling language in terms of its modelling functionality, without having to rely on subjective measurements.

## 2.4 Existing Web Application Modelling Languages

Web application modelling has been extensively researched, and many reviews and surveys of existing approaches exist. Selmi et al. [307] provides an excellent survey of some of the fundamental problems with existing approaches. Other reviews concern themselves with evaluating the functional requirements of languages [11, 140], and consistently find that existing languages are inadequate due to deficiencies in expressibility, usability or implementation support. A discussion on the suitability of model-driven approaches is discussed later in Section 3.1.

In this section, a range of existing RIA modelling approaches will be briefly discussed. This evaluation had been performed earlier by Wright and Dietrich [368], which identified that these existing languages tend to either be abandoned or poorly implemented. This section will focus on new web application modelling language research that has emerged since this evaluation was published, rather than revisiting these older languages.

### 2.4.1 WebML

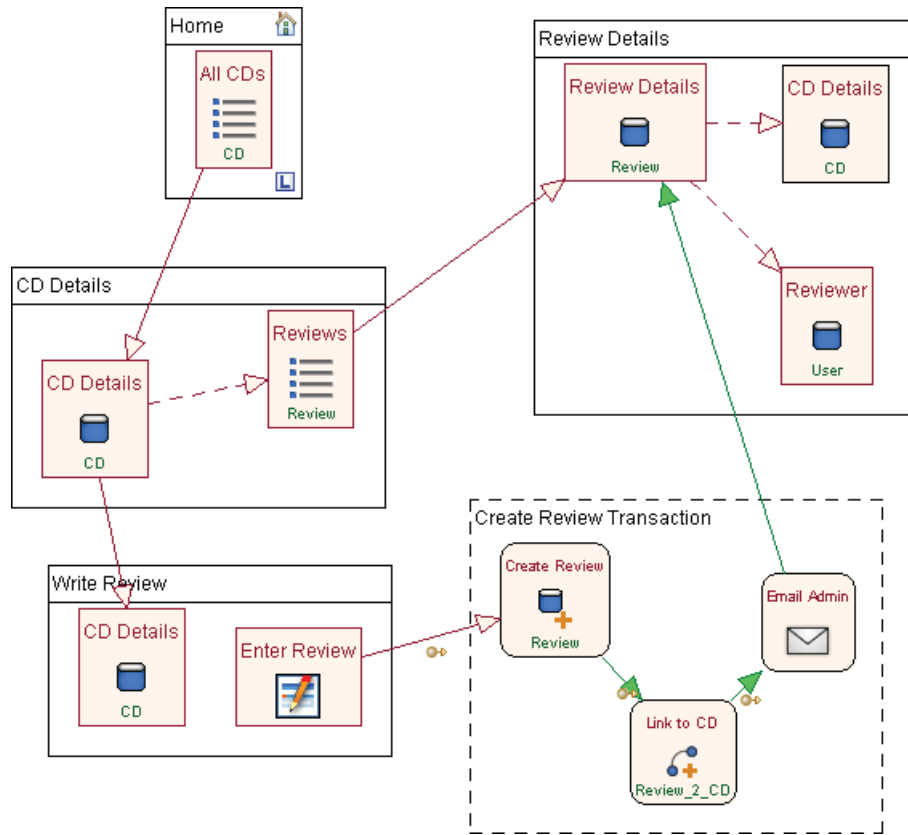
WebML is a well-researched and commercialised approach to modelling data-intensive web applications [51]. Web applications are described using five models, of which the *hypertext model* (the combination of the *composition* and *navigation* models) is the most important. In Figure 2.2, a simple web application for publishing CD reviews is illustrated. These models are combined together using code generators and custom XSLT templates [341] to generate the final Java-based application, which can then be published onto a web server.

The language is well-supported with a CASE tool named *WebRatio*, which is still actively developed but at the time of writing, has not been extended to implement new modelling approaches, such as recent client/server extensions to the model [105]. This CASE tool is commercialised and closed-source which hinders extensibility, although there is some support for plugins. With regards to supporting RIAs, it appears that WebML is focused on supporting conventional data-driven web applications, and instead attempts to “bolt on” RIA support to the WebML model, instead of using RIA concepts fundamentally.

Wright and Dietrich [368] found that WebML was the most complete modelling language for web applications, yet lacked support for many of the fundamental concepts of RIAs, such as object lifecycles, user interface modelling, and controlling the browser. Following the publication of this review, the WebML authors agreed that the language lacked these important features [105].

---

<sup>9</sup>As discussed later in Section 5.1.1, these requirements have been separated into two profiles – *Basic RIAs* and *Full RIAs* – to simplify the implementation of a modelling language for RIAs against these requirements.

Figure 2.2: A sample WebML *hypertext model*

### 2.4.2 UML

UML is a general-purpose modelling language for the analysis, design and implementation of software systems and other similar processes [254], which can include web applications and RIAs. However, this flexibility impacts on its suitability as a RIA modelling language. UML model instances are often verbose and extremely informal, with a lot of domain knowledge assumed. Common web concepts such as sessions, timed events and e-mails are difficult to describe with standard UML, and extensions are usually required.

As UML does not have a complete set of defined formal semantics and implementation rules [113], it is not possible to take an arbitrary UML model instance and translate it into a functional piece of software. This reduces the suitability of using UML for modelling RIAs, as the semantics of UML would need to be extended – which is likely to spawn platform-dependent UML model instances that cannot integrate with other UML model instances. Some authors argue that UML should not capture code-level semantics, such as Fu et al. [113]; however, this viewpoint suggests UML is therefore strictly for documentation-purposes only, negating many of the benefits of model-driven approaches.

The OMG recommends extending UML through the use of three key extension mechanisms; stereotypes, tagged values, and constraints [44]. In particular, a *stereotype* is defined as “a kind of *Class* that extends *Classes* through *Extensions*” which are applied to an existing metaclass<sup>10</sup> [254, pg. 670], allowing domain-specific terminology to be provided against a base UML model. The formal combination of a particular set of extensions can be published as a single *UML Profile* [114].

<sup>10</sup>In this thesis, the UML term *metaclass* [254, pg. 23] will be defined as the *defining metamodel class for a given instance of a metamodel element*.

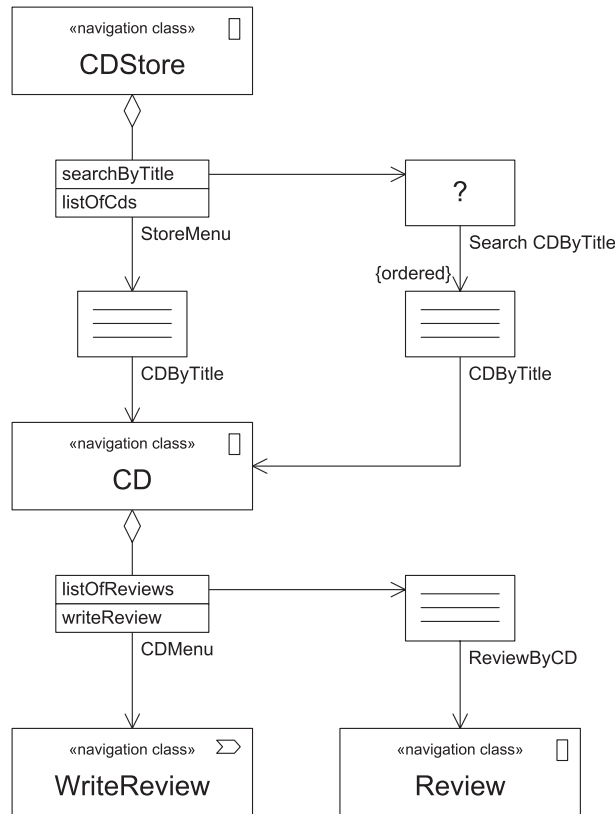


Figure 2.3: A sample UWE *navigation structure model*, adapted from Koch [190]

In practice, only the stereotypes of a UML Profile are supported by existing CASE tools [298], and constraints defined by UML Profiles are poorly supported.

UML’s extensibility is entirely additive; you cannot remove existing UML notations or create new operations, without creating an entirely new UML derivative language [44]. Bruck and Hussey argue that DSLs provide more precision and less complexity, and recommend extending UML only if the majority of your concepts easily map onto existing UML concepts [44].

Nevertheless, one UML extension, UWE – discussed in the next section – has emerged as a promising candidate for modelling web applications. Another extension of UML by Conallen [59] also attempts to extend UML to describe web applications, and can translate the model into a functional web application, but lacks significant functionality. A third extension, WUML [176], attempts to add models to describe web concepts, such as events and user profiles; WUML’s intent appears to focus on documentation and extension purposes, rather than systems development.

### 2.4.3 UWE

UML-based Web Engineering [192] is a web application modelling language extension to UML, following the UML extensibility requirements [44], which has a heavy focus on using modelling standards. UWE uses more types of models than WebML<sup>11</sup>, but advocates using automatic and semi-automatic tools to assist the developer in constructing these models<sup>12</sup>.

<sup>11</sup>At the time of writing, UWE uses at least eight models [190]: requirements, content, architecture, navigation, process, business logic, architecture and integration models; whereas WebML uses at least three models [51, 50]: data, hypertext, and presentation models.

<sup>12</sup>For example, through the use of model transformations in QVT and ATL, as discussed later in Section 3.1.8.

```

define page editUser (u : User) {
  title { "Edit User: " output(u.name) }
  section {
    header { "Edit User: " output(u.name) }
    form {
      par { "Name: " input(u.name) }
      par { "Password: " input(u.password) }
      par { action("Save Changes", saveUser()) }
    }
    action saveUser() {
      u.persist(); return viewUser(u);
    }
    navigate(home()) { "return to home page" }
  }
}

```

Listing 1: Part of a web application implemented in WebDSL, adapted from Groenewegen et al. [134]

Compared to WebML, the model instances are often more verbose but the individual model elements are simpler; this means the models are easier to understand, but less efficient to develop large applications. One of these models is a *navigation structure model* to model the navigational aspect of web applications [191], and the navigation of the same CD reviewing application is illustrated in Figure 2.3. Recent extensions to UWE have introduced RIA concepts, such as autocomplete and input validation, to the UWE metamodel [193]; however these extensions are simply functionality extensions for conventional web applications, and do not support fundamental RIA concepts such as events and browser control.

Wright and Dietrich [368] found that UWE had no support for common web concepts such as e-mails and browser identification. A significant extension to UWE is therefore necessary to support basic web concepts, which would then need to be extended again to support RIA concepts; but these extension would be vastly simplified due to its adherence to model-driven standards. The language's openness, automatic developer processes, clean models, and standards-based approach is promising for future extensibility.

#### 2.4.4 WebDSL

WebDSL is a textual domain-specific language for implementing data-driven web applications with a rich data model, and has limited support for user interface modelling [134]. It can support a wide range of access control models – including most of the access control models discussed later in Section 4.8 – although these policies have to be individually implemented [135]. A short excerpt of a WebDSL model instance is illustrated in Listing 1; as discussed later in Section 3.4, a modelling language does not require a graphical syntax in order to be effective.

Web applications defined in WebDSL may then be generated into source code through the model transformation language *Stratego/XT*. Language extensions also extend the code generator by providing plugins upon the base language. At the time of writing, WebDSL is still fairly recent and lacks supports for RIA concepts such as client-side scripts and browser control, making it unsuitable in modelling RIAs.

### 2.4.5 Web Information Systems

Rather than providing a visual model, work has been done on describing web applications from a more formal perspective, with roots in algebra and graph theory. Schewe [297] describes *Web Information Systems* (WIS) as a triplet of issues: content, navigation and presentation, with each issue described using strict notation. Like most existing approaches, WIS lacks modelling support for most web concepts, and cannot handle the interactivity of RIAs.

The argued benefit for using formal models is that it simplifies formal verification of the model; but the line between formal and informal models continues to blur, as modelling languages (and instances) are increasingly using formal syntax and semantics in their own definition [188]. By themselves, formal models are difficult for developers to understand or utilise, so any formal modelling approach requires tool support in order to make any impact [49].

### 2.4.6 Older Languages

There is a vast selection of other modelling approaches to web applications, and each of these have been similarly investigated as to the feature requirements of modelling RIAs. Wright and Dietrich [368] investigates the languages W2000 [17], OOWS [271], OOHDM [288] and Araneus [225] and finds that these languages are particularly unsuitable for modelling RIAs. All of these older languages have been found to lack support for many of the fundamental concepts in web applications [276, 307], and many are currently out-of-date or poorly maintained. These approaches will therefore not be considered in any further detail in this thesis.

### 2.4.7 Evaluation

Wright and Dietrich [368] critically evaluated the most recent versions of existing web application modelling languages against the general feature categories of RIAs introduced earlier in Section 2.3.1, with respect to a subjective ranking. The evaluation results published in this paper are included here in Table 2.3. This review found that the majority of approaches have limited existing support for standard web application concepts that should be supported already, such as sessions, events, messaging and access control.

These languages are therefore unsuitable as candidates for an extension to support RIAs, as significant effort must first be expended to address *conventional* web applications. Existing approaches do not directly support events or interactive functionality, but instead provide this functionality through the use of user interface libraries. Fundamental web concepts such as users and access control were often ignored, and very few approaches directly investigated the use of verification techniques.

Along with the existing reviews discussed earlier at the beginning of this section, these evaluation results strongly suggest that no existing modelling language satisfies the requirements defined by Wright and Dietrich for a RIA modelling language [368]. In particular, there is a significant conceptual gap between the high-level web system concepts and the low-level technologies required to support its implementation, and this size of this gap is increased by the arrival of new RIA technologies.

## 2.5 Benchmarking Application

While the basic feature evaluation discussed in the previous section is useful to obtain a brief overview of a RIA modelling language, a more detailed functionality-based evaluation is necessary in order to



Feature Category	WebML	UWE	W2000	OOWS	OOHDM	Araneus
Events	Some	-	Poor	-	-	-
Browser Interaction	Poor	-	-	-	-	-
Lifecycle Management	Poor	Good	Poor	-	-	-
Users	Good	Poor	Poor	Poor	-	-
Access Control	Some	Some	Poor	-	-	-
Database Support	Good	Some	Poor	Poor	Poor	Poor
Messaging	Good	Poor	Some	-	-	-
UI Modelling	Poor	Some	Some	Poor	Some	Poor
Platform Independence	Excellent	Excellent	Good	Excellent	Good	Some
Standards Support	Poor	Excellent	Excellent	Some	Poor	-
Use of Metamodels	Poor	Excellent	Excellent	Poor	-	-
Verification	Some	Some	-	-	-	Poor
Software Support	Good	Some	Poor	Some	-	Some

Rating	Concept Support
-	No support at all
Poor	Very limited support, difficult to implement
Some	Some support, some aspects cannot be implemented
Good	Most aspects can be implemented with ease
Excellent	Ideal implementation of the concept

Table 2.3: Existing modelling language support for the general feature categories of modelling Rich Internet Applications, adapted from Wright and Dietrich [368]

evaluate the expressiveness of these different approaches with greater accuracy. Importantly, these evaluations may be compared objectively, and will also highlight any specific functionality deficiencies of a particular approach.

Functional *benchmarking applications* may be useful in identifying and validating the expressiveness of different technologies, and have been used in many computing domains such as business rules and server-side performance [126, 357]. These applications allow different technologies to implement an agreed-upon concept, and the final implementation in each technology can be evaluated against metrics such as size, performance or development cost.

Wright and Dietrich [367] propose a new benchmarking application named *Ticket 2.0* as a functional benchmarking application for the evaluation of RIA modelling languages. The 59 detailed functionality requirements introduced earlier in Section 2.3.2 were each included as functionality within the requirements for a RIA, within the domain of a social networking-enabled, event ticketing application. As described in the paper:

“Its business goal is to provide a rich interface for users to browse upcoming events and book tickets using a credit card. They may interact with other users on the site through friends lists and chat rooms on the event detail pages themselves, permitting open discussions and user interaction. It also aims to provide a unified interface for event managers, allowing them to schedule upcoming events and track their progress.” [367, pg. 111]



The conceptual structure of the application is illustrated in Figure 2.4, and at the time of writing, has been implemented using two different implementation technologies<sup>13</sup>. For the full description of the requirements, and a mapping from the defined requirements to the features of the application itself, the interested reader is referred to the published paper. *Ticket 2.0* may also be used to compare other forms of RIAs, such as conventional web application frameworks and general-purpose languages.

## 2.6 Metrics

Measurement is as essential part of software engineering, as discussed by Pressman [278]. Park et al. [270] argue that software measurement allows the characterisation, evaluation, prediction or improvement of the process. This concept of measurement is defined formally by the IEEE through a *metric* [161], which is “a quantitative measure of the degree to which a system, component, or process possesses a given attribute.”

In the software engineering domain, these measurements may therefore be used to get some sense of whether the requirements are consistent, testable and implemented; whether the design is of high quality; to understand the complexity of the system; and whether the development activity can be improved [94, pg. 3–21]. Software metrics, in particular, can be combined in a well-defined way in order to aid analysis and assessment of a software system through a *software quality model* [94, pg. 338–344], such as the ISO 9126 model [166].

Individual metrics are defined for a particular domain, and often cannot be evaluated across different domains. For example, web application metrics such as “number of pages” cannot be directly compared to implementation metrics as “lines of code” or “number of comments”. In this thesis, four domains of metrics will be discussed: domain-specific metrics for a particular web application (*web application metrics*); metrics for the modelling approach used (*metamodelling metrics*); and metrics for the development process itself (*system metrics*).

A number of other domain-specific metrics were considered, but have not been included in this thesis. For example, Selic [305] introduces the model-driven metrics of *compilation time* for translating a model, and the *turnaround time* for applying incremental changes. Other metrics, such as code coverage [244, pg. 417] and execution time were also considered, however these metrics are more appropriate for highlighting deficiencies in a single implementation, and not for evaluating different implementations of the same system.

### 2.6.1 Web Application Metrics

*Web application metrics* are metrics evaluated against a particular implementation of a web application, and there is a significant body of existing research into defining and using these metrics. These metrics are often used to estimate the effort required to develop different applications, but they can also often be used to compare different web applications for complexity. They cannot be used to compare different implementations of the *same* design, but are instead used to compare the complexity of different web application designs.

By reviewing a wide selection of research into web application metrics, Mendes et al. [224] found that most existing metrics are concerned with estimating the development effort for a given web application (*cost estimation*). As an example, Cowderoy [62] defines size metrics such as number of

<sup>13</sup>As discussed later in Section 8.3, *Ticket 2.0* has been implemented using the Symfony framework as *Ticketsf* and *Ticketsf-mini*, and using the modelling language proposed in this thesis as *Ticketiaml*.

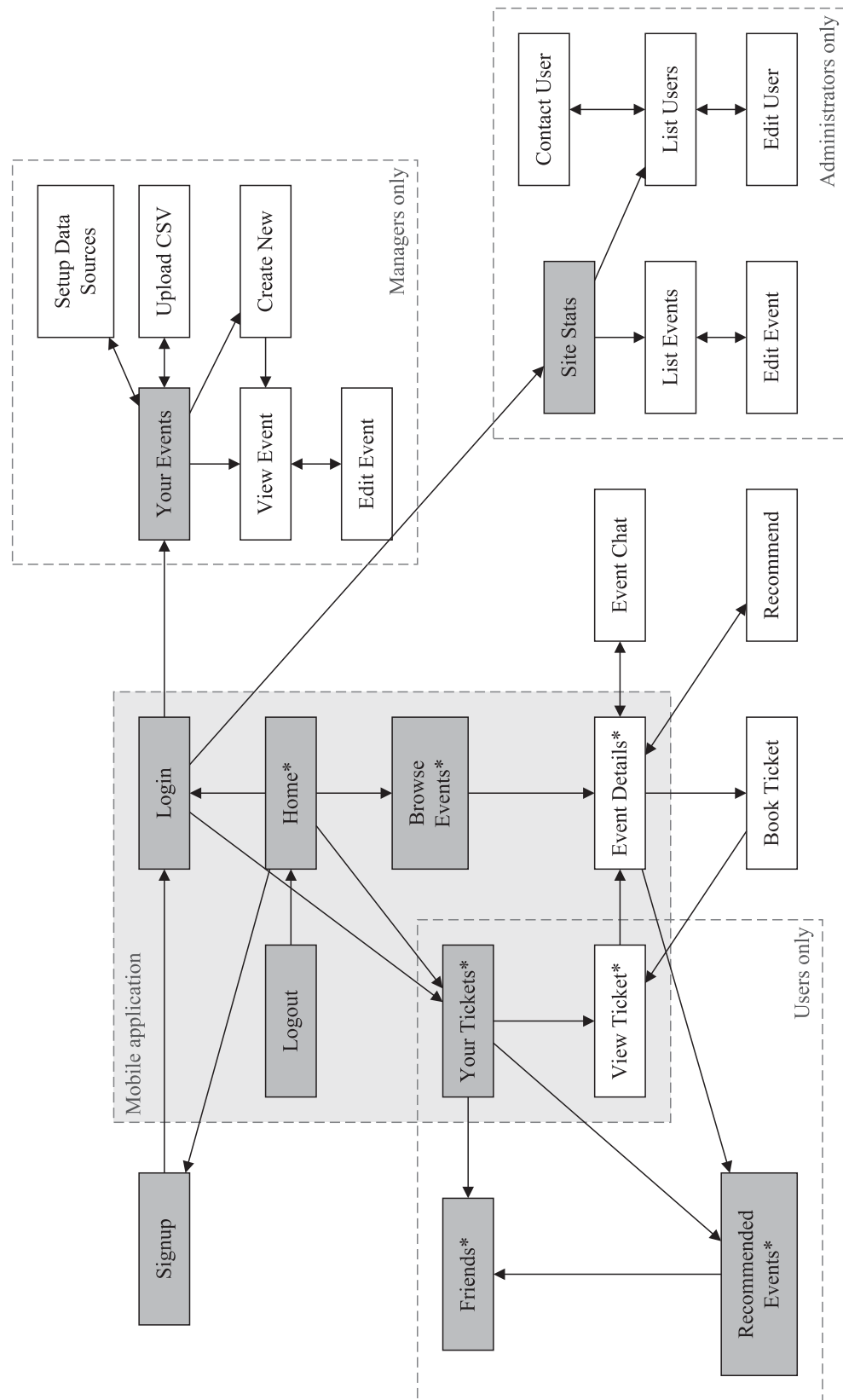


Figure 2.4: The conceptual structure of the *Ticket 2.0* benchmarking application, adapted from Wright and Dietrich [367]

Metric [235]	[331]	Description
<i>NoC</i>	<i>TNP</i>	Number of packages, i.e. EPackages.
<i>NoAC</i>	<i>TNC</i>	Number of classes, i.e. EClasses.
<i>TNoR</i>		Number of abstract classes, i.e. EClasses that are specified as abstract.
<i>TNoA</i>	<i>TNA</i>	Total number of references, i.e. EReferences.
<i>NoD</i>		Total number of attributes, i.e. EAttributes.
<i>NoE</i>		Number of primitive datatypes, i.e. EDataTypes.
<i>Nav</i>		Number of enumerations, i.e. EEnums.
<i>Cont</i>		Navigability: The proportion of references with a defined eOpposite.
<i>Dat</i>		Containment: The proportion of containment references to all references.
		Data quantity: The proportion of attributes to overall structural features.

Table 2.4: Selected metamodeling metrics for metamodels within the Eclipse Modeling Framework, adapted from Monperrus et al. [235] and Vépa et al. [331]

web pages; amount of text; number of images; number of features off-the-shelf; and include metrics dedicated to reuse or outsourcing.

Many metrics have been proposed for object-oriented software systems; for example, the MOOD, MOOSE, EMOOSE and QMOOD metrics are all different collections of object-oriented software metrics, as discussed by Baroni and Abreu [19]. Baroni [18] describes a comprehensive collection of software metrics, including these object-oriented metrics, that can be applied to a system. As a web application may be implemented using any conceptual methodology – including object-oriented, procedural and functional – these metrics cannot be consistently applied to web applications.

## 2.6.2 Metamodeling Metrics

*Metamodeling metrics* can be useful to compare different metamodeling approaches intending to model the same domain. These metrics can either be concerned with individual elements of the meta-model, or the metamodel as a whole. For example, Rossi and Brinkkemper [289] proposed a set of seventeen non-empirical – i.e., not derived from usability studies – complexity measurements for modelling languages, such as a metric for average complexity ( $\bar{C}(M_T)$ ), and a metric for total conceptual complexity ( $C'(M_T)$ ). Siau and Cao [309] applied these metrics to the UML 1.4 specification [249], arguing that these metrics illustrate that as a whole, the UML specification is very complex compared to other object-oriented modelling techniques.

Monperrus et al. [235] investigated applying similar metrics to modelling languages implemented in the Eclipse Modeling Framework<sup>14</sup>, and some of these proposed metrics are listed here in Table 2.4. Vépa et al. [331] also look at a selection of metamodel metrics, in order to measure and compare different model repositories, and derive their metrics through a series of ATL transformations<sup>15</sup>; these metrics are also listed here in Table 2.4. The productivity of a modelling language could also be measured in terms of a function points per staff month metric; Mernik et al. [226] used this metric to measure the productivity of different programming languages.

In certain metamodeling implementations where metamodels are structurally represented as a graph<sup>16</sup>, one could also compare metamodels instances against standard graph metrics. For example,

<sup>14</sup>The Eclipse Modeling Framework (EMF) is described in further detail later in this thesis, in Section 6.2.1.

<sup>15</sup>ATL transformations are described in further detail later in this thesis, in Section 3.1.8.

<sup>16</sup>This includes metamodels defined using the Model Driven Architecture and the Eclipse Modeling Framework, as de-

Metric	Description
<i>Tasks</i>	Tasks implemented
<i>Time</i>	Development time, in whole weeks
<i>NDev</i>	Number of developers [305]
<i>FC</i>	Number of file changes
<i>Rev</i>	Number of SVN revisions
<i>DTech</i>	Number of different programming technologies used [261]
<i>DMedia</i>	Number of different media types used [261]

Table 2.5: Overall system metrics, adapted from Selic [305] and Olsina et al. [261]

Metric	Description
<i>Files</i>	Number of files
<i>Size</i>	Size of all files, in bytes
<i>NCLOC</i>	Physical lines of code [269, 94]
<i>ALOC</i>	Average NCLOC per file

Table 2.6: Language system metrics, adapted from Park [269]

the *degree* of a given class would refer to the number of attributes and references directly by the class, and *distinct types* would refer to the distinct number of metamodel types used (e.g. classes, attributes, references, or enumerations). A suite of some metamodel graph metrics have already been proposed by Wright and Dietrich [369] to evaluate the complexity of EMF model instances.

### 2.6.3 System Metrics

There are a limited number of *system metrics* that can be used to compare two diverse implementations of the same project, possibly because the development artefacts depend heavily on the staff and technologies selected for a given implementation. Metrics that are useful within one approach (for example, “number of plugins” in a Symfony project) may have no reciprocal metrics in another approach (for example, “number of visual models” in a MDD project).

In this research, it seems that only two common domains exist between different implementations of a given RIA. *Overall system metrics* may be used to evaluate the system size and development effort, as illustrated in Table 2.5; a number of these metrics are derived from existing work by Olsina et al. [261] and Selic [305]. Similarly, *language system metrics* may be used to evaluate the physical representations of the development artefacts, as discussed by Park [269].

With regards to both of these domains of metrics, frameworks allow three different types of development effort to be independently evaluated. As discussed earlier in Section 2.2, these approaches can automatically generate *boilerplate code*, or encapsulate common functionality in runtime libraries. Consequently, system metrics should be applied to each of these different versions of a system:

- The *manual effort*, representing only content that has been developed independently of any boilerplate code or external libraries;
- The *generated application*, representing both the custom code and the generated boilerplate code provided by the framework; and

- The *complete application*, representing all custom code, generated code, and included libraries.

By splitting up system metrics into these three different versions, the level of complexity hidden or managed by the framework can be quantified. For example, external libraries can be expected to have a large number of files (*Files*); and the size of the generated application (*Size, NCLOC*<sup>17</sup>) should be significantly greater than the manual effort.

## 2.7 Software Engineering Principles within the Development of Modelling Languages

There are a number of approaches in which a modelling language may be designed, and in how such a modelling language may be developed into a proof-of-concept implementation. A full discussion on the applicability of software engineering principles to the development of modelling languages is well outside the scope of this thesis. This section will instead briefly introduce four important aspects considered in the development of a RIA modelling language in Chapter 5.

### 2.7.1 Modelling Language Design Approaches

If a modelling language is being developed with respect to an existing modelling language, three methods in developing a new modelling language are commonly used [44]: extending the language; restricting the language; or abstracting the language into a higher level of abstraction. Each of these approaches have a resulting impact on the complexity or expressivity of the language, and on the complexity of modelling a system using the language.

#### Extension

*Extension* is the officially-supported extension approach of UML, through the use of stereotypes, tagged values, and constraints [44]. The development of a framework for a programming language falls under this category as well, because the existing libraries are essentially being extended, adding complexity in the implementation. The extension of a common industry-accepted language, such as UML or Java, improves the acceptance of the new approach because the concepts are already defined and widely understood in industry [179]. The improved expressibility simplifies development of the resulting model instances, but the complexity of the language itself increases. Kelly and Pohjonen [179] also argue that it is difficult to adapt a general-purpose language to a particular domain.

#### Restriction

*Restriction* – called a *heavyweight extension* by Bruck and Hussey [44] – is the approach of removing language elements and concepts that are deemed unnecessary, thereby reducing the complexity of the language, but also making the restricted language incompatible with the original base language. For example, removing all UML modelling language elements except for those used in modelling class diagrams would lead to a smaller language. However, a well-designed modelling language is one that cannot be restricted any further; the restriction of a modelling language necessitates a reduction in functionality.

---

<sup>17</sup>In this thesis, the *NCLOC* metric represents the physical lines of code that are not comments, as defined by Fenton and Pfleeger [94, pg. 247].

## Abstraction

The *abstraction* of a modelled system into a higher-level model is the most popular approach in model-driven approaches, and the development of a DSL follows this approach [102]. Model instances at the higher level of abstraction are intended to be later transformed into lower-level model instances of the abstracted existing language [33]. By abstracting a model, the resulting language can become more expressive, less complex, and easier to use; however, these gains only occur if the abstracted language can be translated into the lower-level language. Developers of new abstractions must also be careful not to create an abstraction that is too generic or too specific [179].

## UML Profiles

As discussed earlier in Section 2.4.2, UML's extensibility is entirely through extension; restriction is not supported. By defining new stereotypes, tagged values and visual stereotypes, UML Profiles also permits limited abstraction. However, since UML does not specify translation semantics, the translation from these abstracted levels into lower-level models must be provided separately. Bruck and Hussey argue that UML should only be extended if the majority of your concepts easily map onto existing UML concepts [44].

Despite its popularity in industry, there seems to be no reason why WebML was developed as a separate DSL, rather than an extension of UML (like UWE). Perhaps it was because UML and WebML were being developed at roughly the same time; or perhaps the WebML developers thought that web application concepts could not easily map onto existing UML concepts. Ceri et al. discuss how to translate WebML models into UML models [51], but this integration appears to be one-way.

## Discussion

The actual approach used for a model-driven project should depend on the project's requirements. If a system is already fully described in an existing language, then restriction should be used; or if an existing language almost describes the system, then extension may be the best approach. If the system does not easily map onto any existing language, then abstraction may be the best approach.

Other than the overall architecture used in the development of a new language, there are also practical concerns. Kelly and Pohjonen [179] investigate 76 cases of modelling language development, in which they identify fifteen common problems in the development of modelling languages. For example, they find that ignoring or avoiding prototyping and language evolution is a common mistake; evolution is inevitable, especially when the language is intended for real-world usage. The interested reader is referred to Kelly and Pohjonen [179] for a detailed discussion on these real-world problems.

### 2.7.2 Software Process Models

To satisfy good software engineering principles as discussed by Pressman [278], software development should be performed according to a process model adapted to the individual project. As an important goal of this thesis is in the development of a proof-of-concept implementation of the designed language, this section will discuss two of these software process models. Many other process models have been discussed in literature – such as prototyping, rapid application development (RAD), component-based and formal methods [278] – but a full discussion is outside the scope of this thesis.

## Linear Sequential Model

The *linear sequential model* is sometimes called the *classic life cycle* or the *waterfall model*, and suggests a systematic, sequential approach to software development [278, pg. 28–30]. Using this classical model in the development of a software system will theoretically resolve into a complete solution, but assumes that all requirements can be captured at the beginning of the process, and the final architecture can be completed before any development begins.

However, the model cannot adapt to changing requirements; it is difficult to capture all of the requirements with such detail at the start of a project; and a working version cannot be produced until the very end of the process, delaying any sort of feedback cycle. These disadvantages generally outweigh the benefits for all but the simplest of software projects [278, pg. 30].

## Evolutionary Model

The *evolutionary model* covers an ever-expanding list of different models, such as the agile model, SCRUM, the spiral model, and others [278, pg. 34–42]. They refer to an approach in which developers produce increasingly more complex versions of the final system, with each version a deployable artefact. Evolutionary models encourage the rapid production of working, but incomplete, solutions, that evolve over time into the final version.

An evolutionary approach guarantees that a working copy of an early solution is always available, allowing for rapid and early feedback. It also ensures that even if the project exhausts its available resources, it still has at least partially-implemented working deliverables. As Kelly and Pohjonen discuss, the evolutionary model may be better suited towards the implementation of modelling languages due to their natural evolutionary properties [179].

It is important to note that the terms *iterative* and *evolutionary*, while similar, have two distinct meanings; that is, all evolutionary models are iterative, but not all iterative models are evolutionary. The iterative model refers to a process where a series of smaller sequential tasks are done repeatedly. For example, the linear sequential model can be executed iteratively, but only on the final iteration can the software be deployed [278].

## Discussion

Evolutionary development is particularly suited to model-driven approaches: Kelly and Pohjonen argue that modelling language evolution is inevitable, and should be embraced rather than avoided [179]; similarly, Fowler argues that the development of domain-specific languages are well suited towards evolutionary development [102]. UML has itself been developed in an evolutionary fashion, and at the time of writing is continuing to evolve [188].

It is important to also consider the impact of a particular software process model in a research environment such as this thesis. Purely evolutionary approaches are not suitable for research environments, as the design and analysis of a research problem is often more valuable than providing an implementation. However, purely sequential approaches are also not suitable for research environments, as the validation of a partial implementation is often more valuable than providing a complete implementation that cannot be validated. Within a research environment, it seems that a balanced approach between the sequential and evolutionary approaches may be beneficial.



### 2.7.3 Test Driven Development

*Test-driven development* (TDD) is not a software process model; rather, it is a development style that aims to improve the quality of software. As Beck [23] discusses, the goal of TDD is to develop “clean code that works”: in particular, it is a predictable way to develop, encourages frequent releases and can support changing requirements. TDD is therefore not very useful for designing modelling languages, but would rather be a good development style to use to produce the proof-of-concept implementation of the modelling language, and works well with an evolutionary software process model.

Outside of an implementation of a modelling language, TDD may also be independently used by the model developer to improve the quality of their developed model instances; capturing the functional requirements of the modelled system through test cases can enable the evaluation of quality metrics against the resulting implementation [166]. Kent [180] argues that tooling to support *model-driven testing* is essential to maximise the benefits of model-driven development. Model instance verification, as described earlier in Section 2.3.1, may also be introduced as part of a model-driven testing framework.

### 2.7.4 Open Source

In the software development world, *open source* refers to software where the source code is available and distributed under a particular *open source license* [231]. Mockus et al. [234, pg. 311] argue that open source software can be of a high quality and functionality, as access to the source code of a software project can be used to identify bugs and develop third-party extensions. In particular, studies have found that open-source projects have fewer testable defects than similar proprietary software [234]. Development under an open-source license can also improve the functionality of the developed system, as software projects under compatible licenses can be integrated together.

In this thesis, open source licenses will only be briefly mentioned, and a full discussion on the preferable range and combinations of licenses will not be discussed. There may be legal issues in a model-driven architecture similar to those in a code compilation scenario. For example, a GPL-licensed compiler [106] cannot enforce the GPL on the code that it compiles, *except* in the case that portions of the compiler – for example, runtime libraries – are also included in the compiled output<sup>18</sup>.

In particular, the implementation of a model development environment for web applications may consist of a number of components, and each may be under a separate open source license. A full discussion on the legal repercussions of combining different open source licenses is well outside the scope of this thesis, but it is important to keep licensing restrictions in mind.

## 2.8 Conclusion

This chapter has discussed the new challenges that Rich Internet Applications have imposed on conventional web development processes, and that no existing modelling language for web applications can support these challenges. The extension of an existing general-purpose programming language in terms of a framework is not desirable, in order to gain the benefits of a platform-independent abstraction to RIAs.

---

<sup>18</sup>The interested reader is referred to the *GPL FAQ* [108] – “Can I use GPL-covered editors [...] to develop non-free programs?” – and the *GCC Runtime Library Exception* [107].



Consequently this thesis proposes the development from scratch of a new language for modelling Rich Internet Applications, but reusing concepts and semantics from established languages; and one that can be manipulated using a visual representation. This language will be used to implement the benchmarking application *Ticket 2.0* according to some of the development approaches discussed in this chapter, and will be evaluated using a selection of metrics, feature comparisons, and evaluating the initial language requirements.

## Chapter 3

# Modelling

The general area of modelling within software engineering is a rich area of existing research. This chapter will investigate and briefly summarise many of the techniques, approaches and standards in the model-driven area. This is necessary to clarify the scope of this research, to understand the interactions and relationships of this research within the larger scope of model-driven approaches, and to improve the interoperability of the proposed modelling language with other technologies.

### 3.1 Modelling

A model of a system can represent almost anything, and be represented by almost anything; for example, source code can be considered a model, yet the grammar of the source code can also be considered a model. The most consistent and agreed-upon definition is that *a model is a simplified abstraction of reality* [143]. Models may coexist in different modelling spaces and themselves be abstractions, or instances, of other models [80].

However, the idea that a model is an abstraction of reality does not necessarily mean that a given model is useful. Selic [305] argues that for a model to be useful and effective, it must sufficiently possess the following five key characteristics:

1. **Abstraction:** “A model is always a reduced representation of the system that it represents.” If the model is actually more detailed than the original concept, then it is not a model.
2. **Understandability:** “It isn’t sufficient just to abstract away detail.” A model must still contain concepts that are understandable.
3. **Accuracy:** “A model must be a true-to-life representation of the modeled system’s features of interest.”
4. **Predictiveness:** “You should be able to use a model to correctly predict the modelled system’s interesting but non-obvious properties, either through experimentation or some type of formal analysis.”
5. **Inexpensive:** “It must be significantly cheaper to construct and analyze [a model] than the modelled system.”

Discussions on the suitability of model-driven approaches are found widely in existing literature, with good overviews provided by Meservy and Fenstermacher [227], and Djurić et al. [80]. Gitzel et

al. [124] discuss the use of MDA and its natural advantages to address the unique problems introduced by web applications, such as improving usability and future maintenance.

Within the world of modelling, there are many different techniques and standards, and the relationships between them are often misunderstood or ill-defined. This section will describe each of these approaches and their relationships to other model-driven approaches.

### 3.1.1 Metamodels

If a model is a simplified abstraction of reality, a model developer still needs to define what the abstraction means, and how the reality may be mapped to a model. This represents the *modelling language* for a collection of models, and this language describes the syntax and semantics of the abstracted *model instances*<sup>1</sup>. These model instances are said to be *valid* if their representation conforms to the syntax, and their meaning conforms to the semantics defined in the modelling language. The syntax of a modelling language is often represented formally using grammars such as EBNF, while the semantics of a modelling language is usually represented informally using plain text [148].

The most widely-known modelling language used in software engineering is the Unified Modelling Language (UML) [254], as described earlier in Section 2.4.2. This *general-purpose* modelling language aims to support the analysis, design and implementation of systems and processes, although it is mainly used in software engineering. For this language, both its syntax and semantics are defined using a mixture of formal constraints, visual rules and structured English language.

The distinction between a modelling language and a *metamodel* is still under some debate [198, pg. 11]. For example, Favre defines a metamodel as “a model of the modelling language” [92, pg. 14]; the Meta Object Facility (MOF) specification defines a metamodel as “a model used to model modelling itself” [258, pg. 31]; Henderson-Sellers considers the two terms interchangeable [151, pg. 302]; and the UML specification is simultaneously used as both a modelling language and a metamodel [80, 122, pg. 94]. A detailed exploration of this research space is outside the scope of this thesis, and such a distinction would not impact on the language definition or implementation.

Therefore, in this thesis the two terms are used interchangeably; that is, a modelling language may also be termed a metamodel. The term *model instance* is also preferred over the term *model* to reduce confusion; likewise, the term *metamodel* is preferred over *modelling language*, to distinguish the approach from a *domain-specific language*, as discussed in the following section.

### 3.1.2 Domain-Specific Languages

A domain-specific language (DSL) takes the abstraction concept of modelling and represents it in terms of a human-editable language. Fowler defines a domain-specific language as “a computer programming language of limited expressiveness focused on a particular domain” [102, pg. 27], and this definition is used in this thesis. He argues that the boundaries between a DSL and a general-purpose language are blurred, but certain elements of a language may be used to categorise the language.

Fowler puts a heavy emphasis on the limited expressiveness of a language as an indicator of a DSL. For example, he argues that the powerful statistical language R [162] has enough general-purpose functionality to be categorised as a general-purpose language, and not a DSL. In particular, a domain-specific language should not exhibit Turing completeness. Similarly, Fowler argues that a

<sup>1</sup>One logic-based definition of the relationships between *modelling languages* and *model instances* is discussed later in Section 3.1.10.

DSL should be designed to be human-editable; for example, if the language is serialised using XML, and is intended to be edited in this representation, then the language cannot be considered domain-specific.

### The Benefits and Drawbacks of Domain-Specific Languages

Fowler [102, pg. 33–39] discusses some of the potential benefits and drawbacks of using a domain-specific language, in order to help inform the design choices of a system architect. Many of these benefits are similar to the benefits of using models in the development process. These benefits include:

1. **Improving development productivity:** As discussed earlier, abstractions can simplify the development of complex software applications. A DSL provides an expressive way to manipulate instances of these abstractions, reducing defects and improving productivity.
2. **Communication with domain experts:** The abstraction of a DSL allows domain experts to interact directly with a tailored abstraction for their domain.
3. **Change in execution context:** As discussed later in Section 3.1.7, general-purpose languages often can only be evaluated once the instances have been compiled into a lower-level platform. DSLs on the other hand can permit evaluation either at compile time or at runtime, allowing a shift in the execution context of a particular approach.
4. **Alternative computational model:** A domain-specific language may allow a developer to use alternative paradigm models. For example, DSLs can be used to introduce functional or object-oriented code into a procedural language.

There are also drawbacks that need to be considered when using a domain-specific language:

1. **Language cacophony:** This is the concern that “languages are hard to learn, so using many languages will be much more complicated than using a single one” [102, pg. 37]. This viewpoint is particularly noteworthy in the field of RIA development as discussed earlier in Section 2.1, where a RIA must be implemented across many different languages resulting in the conceptual gap of development.
2. **Cost of building:** The effort necessary to design, implement, test and maintain a DSL cannot be ignored, and not every abstraction will improve productivity to the degree necessary to justify the investment.
3. **Ghetto language:** Similarly to *feature creep* as discussed by McConnell [222, pg. 319–344], an incrementally-developed DSL may gradually gain features and functionality until it reaches a point of general-purposeness, at which point the development investment necessary to maintain the system outweighs the value of the DSL. Fowler argues that the only defense to feature creep is to firmly define the scope of the DSL.
4. **Blinkered abstraction:** A danger when using a certain abstraction is that of trying to implement concepts outside of the original abstraction scope, and not permitting changes to the original abstraction. This manifests itself when the time spent trying to implement a certain piece of functionality exceeds the effort that would have been necessary to adapt the abstraction to absorb the new behaviour.

## Modelling Languages and Domain-Specific Languages

Modelling languages and domain-specific languages are two similar concepts; however, the former represents a method of defining an instance of an abstraction, and the latter represents limited expressiveness through restriction on the types of abstractions used. The terms are not mutually exclusive; adapting the definition of a DSL provided by Fowler [102, pg. 27], a *domain-specific modelling language* (DSML) therefore represents a modelling language of limited expressiveness focused on a particular domain. As discussed earlier, this distinction is made clearer in this thesis by only referring to modelling languages as metamodels.

### 3.1.3 Model-Driven Development

*Model-Driven Development* (MDD) does not refer to a defined approach, but rather the idea that models should be embraced as part of systems development. Selic [305] asserts that the key of model-driven development is not the use of particular technologies or standards, but rather the essentials of model automation (including model transformation and model verification), and modelling standards; using these essentials, a model-driven process can be tailored for a specific system development.

System development can benefit from model-driven development in a number of ways, as models can be used in a number of potential applications as summarised by Czarnecki and Helsen [66]:

- Generating lower-level models, and eventually code, from higher-level models;
- Mapping and synchronising among models at the same level, or different levels, of abstraction;
- Creating query-based views of a system;
- Model evolution tasks such as model refactoring; and
- Reverse engineering of higher-level models from lower-level models or code.

Model-driven development also encourages a system to integrate into the ecosystem of existing models, to increase interoperability and model reuse. Finally, another important application of model-driven development is the ability to verify models against consistency and correctness properties; this has become increasingly important in integrating models with the semantic web and related reasoning applications [305, pg. 21]<sup>2</sup>.

### 3.1.4 Model-Driven Engineering

*Model-Driven Engineering* (MDE) is closely related to MDD, and some authors consider the terms equivalent to the point of interchangeability; for example, Koch appears to use the terms MDE and MDD interchangeably in definitions of UWE [189, 190]. In this thesis, the distinction between the two terms will be taken from the definition by Schmidt [300]: MDE follows the same concepts as MDD, but proposes using specific technical approaches for the applications of MDD, whereas MDD is technology-agnostic.

These technical approaches are still platform-independent however, and are not dependent on any particular implementations of each technology. For example, MDE advocates using DSLs as opposed

---

<sup>2</sup>The specific details of verifying and validating model instances against certain properties is discussed in further detail later in Section 3.3.

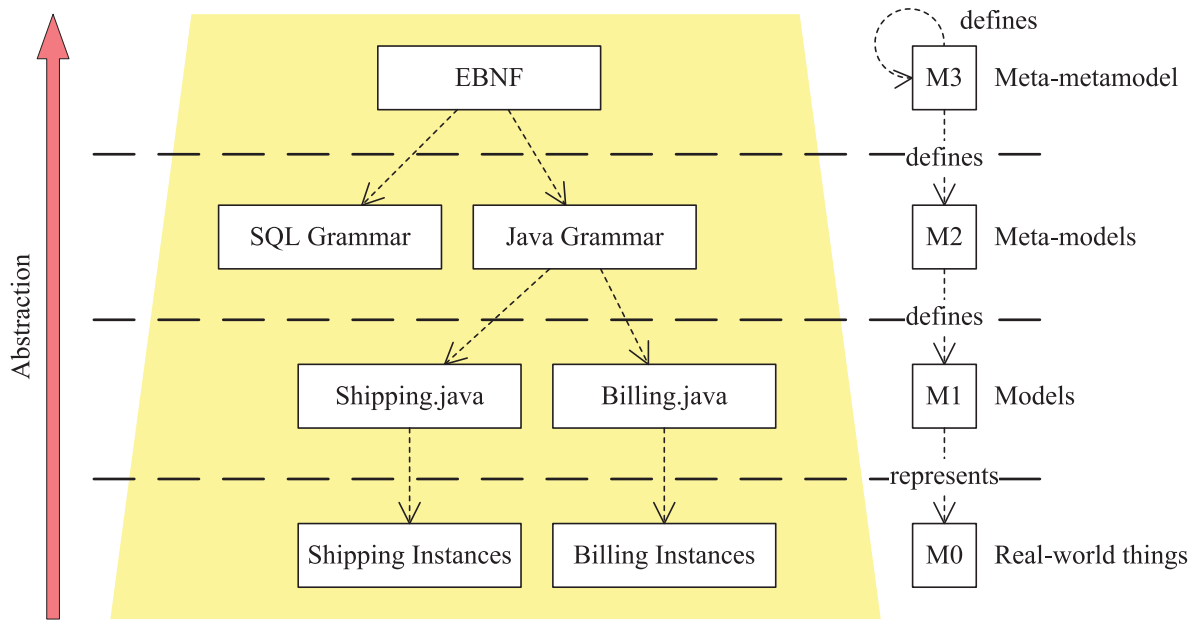


Figure 3.1: The Metamodelling Architecture of MDA, as discussed by Djurić et al. [80]

to general-purpose modelling languages, but these DSLs can be implemented in any model-driven environment. Consequently, while all MDE approaches are also MDD approaches, not all MDD approaches are MDE; it is perfectly acceptable to use general-purpose languages in a MDD approach, or to ignore model transformations completely.

### 3.1.5 Model Driven Architecture

*Model Driven Architecture* (MDA) is an ongoing software engineering effort to standardise model-driven approaches, and is advocated and trademarked by the OMG [243]. MDA focuses on developing *architectural* standards for the development and interchange of modelling approaches, and is a MDD approach. At the time of writing, only a draft outline<sup>3</sup> of the proposed architectures has been published [243]. In particular, MDA is currently based on two conceptual architectures: a four-layer *metamodelling architecture*, and a four-layer *viewpoint architecture*. These architectures are not mutually distinct, but considering both of these architectures simultaneously is useful.

#### Metamodelling Architecture

The first architecture behind MDA is based on a four-layer metamodelling architecture [243], as illustrated in Figure 3.1; each layer has a practical purpose, as described by Djurić et al. [80]. Each layer provides a metamodel for the layer below with an increased layer of abstraction, and the OMG has supplied a relevant modelling standard for each layer. This metamodelling architecture is used for all OMG modelling standards [185].

1. The *Reality layer* (M0) represents the real-world objects that are being modelled, which can include everything, including non-tangible things such as modelling languages and other abstract concepts. For example, real-world entities such as rocks or trees can be part of M0, as could a running Java application, or its source code.

<sup>3</sup>The technique of publishing a number of drafts before standardisation of a language was also used during the development of UML [188] and the HTML standards [333, 343].

Metamodel	Meta-metamodel
Java	EBNF [130]
SGML	ISO 8879 [164]
HTML 4.01	DTD [333]
HTML 5	English Language [343]
XML	EBNF [342]
DTD	EBNF [342]
Ecore	Ecore [314]
EBNF	EBNF [165]
MOF	MOF [251]
UML	MOF [253, 185]
English Language	English Language

Table 3.1: A list of popular metamodels and their meta-metamodels

2. The *Model layer* (M1) represents the model of the real-world object; these models are called *analysis models* in UML [188]. Individual models are known as *model instances*. For example, a Java application in M0 can be modelled with its Java source code in M1.
3. The *Metamodel layer* (M2) represents the model of the models in M1, also known as metamodels or *modelling languages*. For example, all Java source code in M1 must adhere to the Java language specification [130], which would be defined in M2. One OMG standard for metamodels is the *Unified Modeling Language* (UML) [254].
4. Finally, the *Meta-metamodel layer* (M3) represents the model of the metamodels in M2. For example, the definition of the Java language is provided in terms of *Extended Backus-Naur Form* EBNF [130]. EBNF would therefore be defined in M3. Most importantly, all M3 meta-metamodels are instances of themselves; that is, EBNF is defined formally as an instance of its own syntax [165], and there is no such thing as a *meta-meta-metamodel*. As a standard for meta-metamodelling, the OMG proposes the *Meta Object Facility* (MOF) [258].

Importantly, this architecture satisfies the problem on where increasing levels of metamodelling abstraction will end. Since models can themselves be defined in terms of metamodels, it is a realistic question to ask whether this redefinition could continue forever. The maximum number of layers within MDA are restricted to four, by restricting the metalanguage for M3 metamodels to themselves: “The OMG defined that all elements of layer M3 must be defined as instances of concepts of the M3 layer itself” [185, pg. 88].

For comparison, a selection of common metamodels and their associated meta-metamodels are listed in Table 3.1. A full description of each of these models will not be provided here. If the meta-metamodel for a given metamodel is identical to the metamodel, then this meta-metamodel is *M3-compliant* according to the MDA, and can be used in the development of an MDA-compliant modelling approach. There are two important relationships illustrated in this table: firstly, HTML 4.01 is defined in terms of DTD, which itself is not M3-compliant; and secondly, HTML 5 had only been defined using the English language at the time of writing [343].

A model instance is not limited to conforming to a single metamodel, and multiple metamodels are usually combined in a logically conjunctive way. For example, HTML 5 is currently defined only in



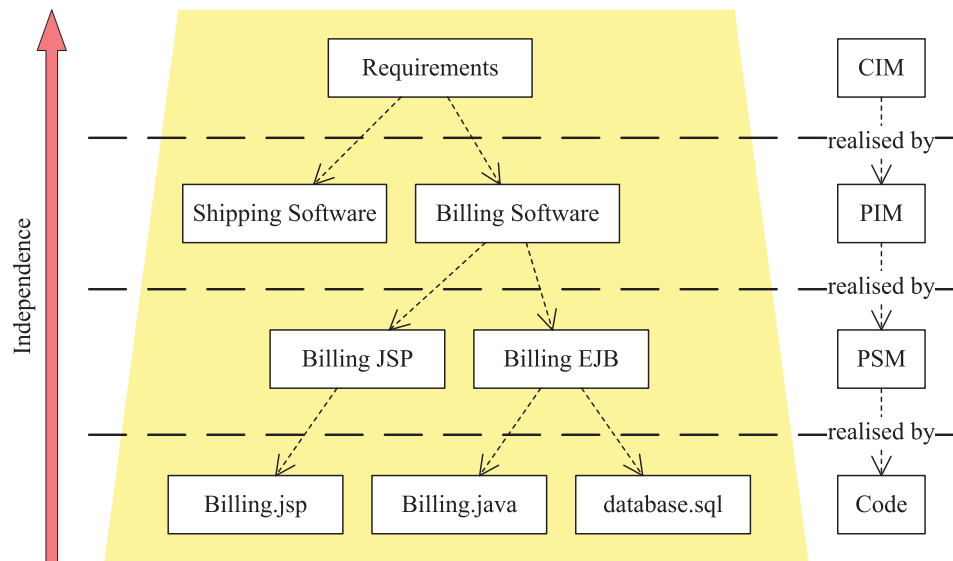


Figure 3.2: The Viewpoint Architecture of MDA, adapted from Mukerji and Miller [243]

terms of the English language, but in the future may also be defined using XML Schema or DTD [343], with each metamodel further restricting the valid range of HTML 5 instances to improve precision. Similarly, the Web Ontology Language (OWL) [351] is defined in three metamodels: EBNF abstract syntax, direct model-theoretic semantics and model mappings [337].

### Viewpoint Architecture

A second architecture proposed by the MDA is the four-layer viewpoint architecture [243, 185], illustrated in Figure 3.2. In this architecture, each layer tries to be an independent abstraction of the layer below it, with the intention of simplifying the development of complex model-driven approaches across an entire business, rather than just a software product. Kleppe et al. [185] discuss the implementation of this architecture using a real-world example.

The OMG does not define any standards for any layer, as each layer is instead advocating a particular architecture, although many of these layers could be implemented using existing OMG standards. For example, both a PIM and a PSM can be represented using UML [185].

1. The *Computation Independent Model* (CIM) is a software-independent model or business model used to describe a business system.
2. The *Platform-Independent Model* (PIM) is the model of a software system that is independent of any implementation technology, and these models have a very high level of abstraction. It is not possible to automatically translate a CIM into a PIM, because the decision of which parts of the business will be supported by software must be made by a human [185].
3. The *Platform Specific Model* (PSM) is the specification of a software system using a particular implementation technology. In some cases PSMs can be produced from the automatic translation of PIMs, but in many cases the model developer must complete missing parts of the PSM [185]. Importantly, PSMs are only understandable by people who are experienced in the specific implementation technology.



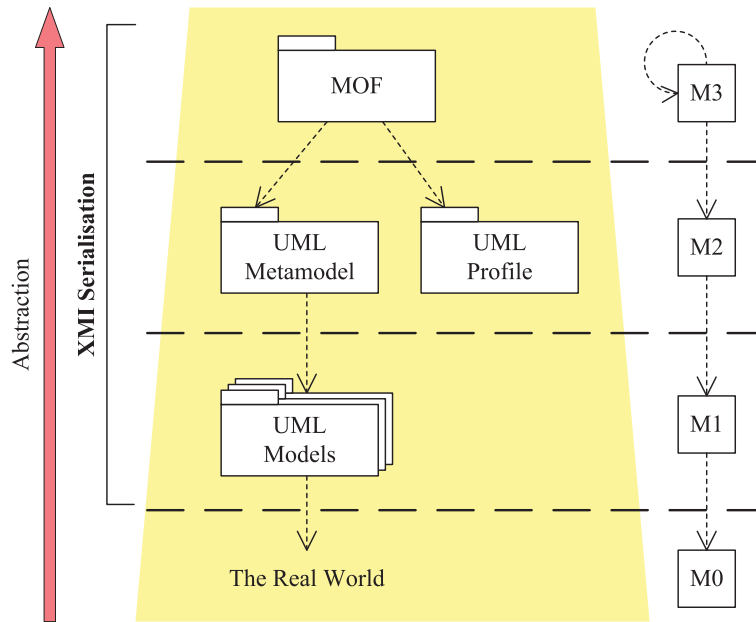


Figure 3.3: The layers of UML under the MDA, adapted from Djurić et al. [80]

4. Finally, the *Code* layer is the actual implementation of the PSMs. The transformation from PSM to code is relatively straightforward.

Kleppe et al. [185] argue that there are four main benefits from using this MDA architecture. Firstly, by shifting developer focus from platform-specific models to platform-independent models, the abstraction improves the efficiency of development. Secondly, everything specified at the PIM level is completely portable, depending on the automated transformation tools that are available; this also increases the interoperability of models. Finally, Kleppe et al. argue that since the PIM is at a higher level of abstraction than the PSM, the PIM also fulfills the function of high-level documentation.

### Meta Object Facility

The *Meta Object Facility* (MOF) is an OMG standard that was designed and published as the standard M3 language for all OMG models, and is used to define modelling languages [185]. As a standard defined at the level of M3, the MOF metamodel is defined in terms of the MOF metamodel, and the overall design was heavily derived from UML. The integration of the MOF with other OMG standards like UML is illustrated in Figure 3.3. The ambition of MOF – codified as the design goals of the standard – was to simplify the interoperability and communication between different modelling languages, and to clear up some of the definitions used in UML; the full list of design goals is published as part of the MOF specification [258].

The MOF model is also known as the *Complete MOF* (CMOF) package, which is built upon a subset of the UML 2.0 Infrastructure [253]. MOF also defines a smaller *Essential MOF* (EMOF) package, which simplifies the complexity needed for compliance<sup>4</sup> by providing a model focused solely on *kernel metamodeling* [258]. The Ecore metamodel of the Eclipse Modeling Framework, which also resides at the M3 layer, was built upon EMOF [12] and model instances can be serialised directly to EMOF [314, pg. 40]. A full discussion on EMF is provided later in Section 6.2.1.

<sup>4</sup>An implementation of MOF must adhere to at least one of the two *compliance points* of CMOF or EMOF.

## XMI

In order to share models, the XML Metadata Interchange (XMI) standard [250] was proposed by the OMG, which is an XML-based standard for sharing metadata. This standard integrated XML, UML and MOF together into one standard, which is now an international ISO/IEC standard [168]. Many model-driven CASE tools, especially those used for modelling UML such as ArgoUML [286] and Eclipse UML [44], support exporting and importing model instances formatted according to XMI, and a sample XMI model is illustrated in Section F. In the same way that XML has improved integration between different technologies, XMI can be used to improve integration between different model-driven technologies, and is an important technology behind MOF.

## The Big Picture

As the viewpoint and metamodelling architectures in the MDA are not distinct, it can be useful to consider a model-driven approach that uses *both* of these architectures at the same time, when trying to define the scope of different interacting models. Figure 3.4 illustrates a range of model-driven technologies that may be used to implement a RIA, and highlights their interaction with the two architectures of the MDA. For example, a RIA may be described in a platform-independent manner, which may then be translated into platform-specific EJB [172] and RIA user interface model instances. These may, in turn, be translated into instances of HTML [343], SQL [186] and other related technologies.

By considering both of these architectures simultaneously, it is easier to understand the interaction between each of these MDA architectures. In the domain of RIA development, it is simple to see how a single model described by a single metamodel (in the PIM) can be implemented using many different RIA technologies, and how every metamodel has an independent meta-metamodel. This figure also illustrates how the PIM model and metamodel should not have any reference to the specific technologies in the PSM or code, highlighting the platform-independence of this layer.

In this example matrix, it is unclear what the PIM M0 for modelling RIAs would represent (marked as '??' in Figure 3.4), and this remains an unanswered question. Would this represent the real-world mental model of our intentions? Would it represent the analysis or design of the modelling language in this thesis? Both of these appear to be valid answers, as the M0 layer can represent any concept.

### 3.1.6 Model Spaces

As discussed earlier, if you take something from the real world (M0) and then model it, you obtain a model instance on M1. However, this model instance can itself be considered a real world object on M0, and the model instance can be modelled (or abstracted away) again; that is, models can exist on different layers at the same time. In order to enable a deeper understanding of model-driven development approaches, a method is necessary to understand how models may concurrently exist within multiple domains.

Djurić et al. [80] propose the concept of a *modelling space*, which is a particular modelling architecture within a particular domain. A modelling space consists of all four layers from the MDA, with the models defined within this space representing the real world from one point of view. Conversely, each layer of the MDA and its contents are also part of M0, so it is always possible to abstract away anything into a separate model space.

For example, consider a developer that is trying to use UML to model a Java program, as in Figure 3.5. UML can model both the Java program and the Java grammar itself [80], and without

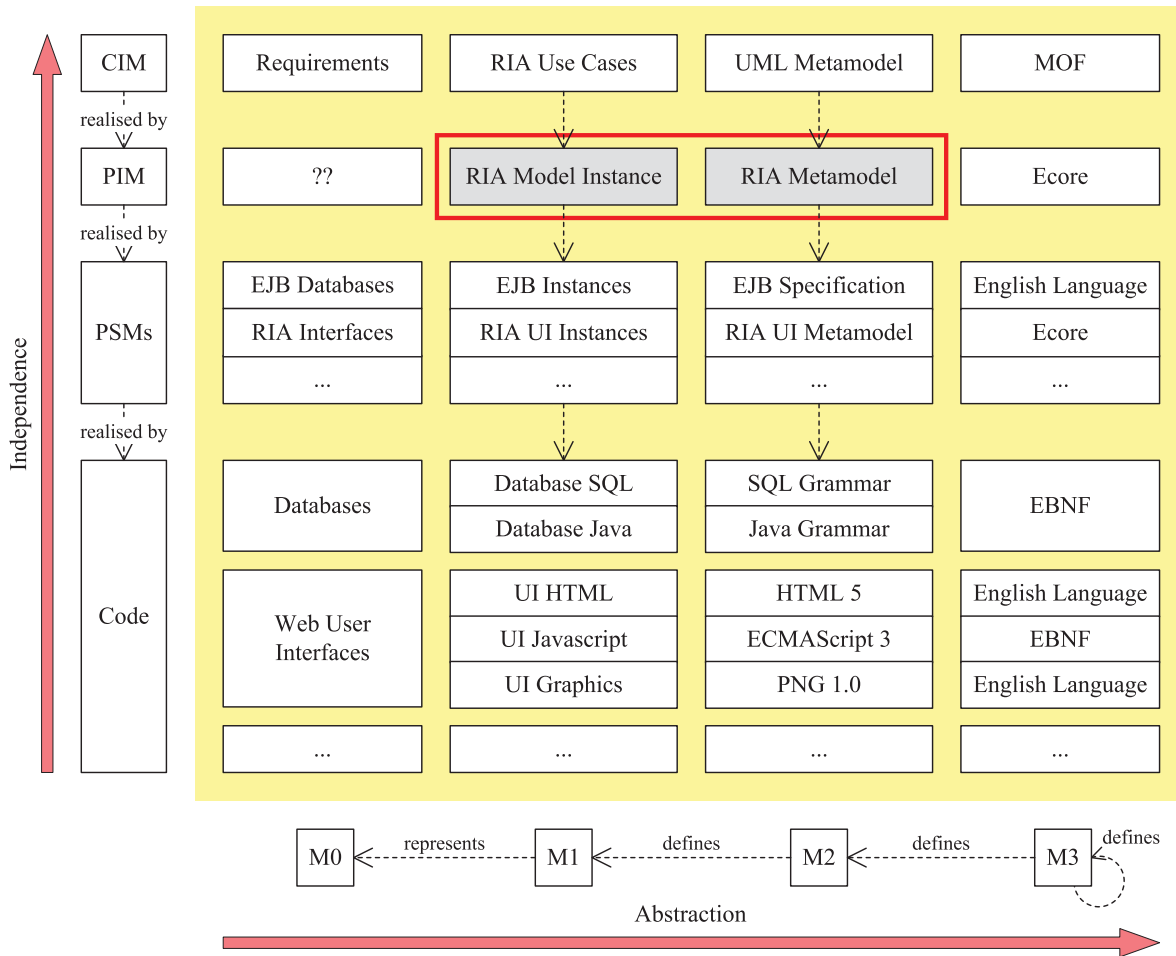


Figure 3.4: Using both architectures of the MDA to describe the model-driven development of RIAs

modelling spaces it can be unclear where this distinction lies. If we place UML models into a *MOF modelling space*, we can introduce a separate *EBNF modelling space* which holds models for the Java aspect of the system. This full process is described in detail by Djurić et al. [80].

These separate modelling spaces for Java software and the MOF metamodeling approaches can make it much clearer to understand where modelling concepts should reside. For example, the syntax of Java reflection would be modelled in the UML Java *grammar model*; but the source code of Java reflection actually being used would be modelled in the UML Java *program model*.

### 3.1.7 Model Transformations

As discussed by Czarnecki and Helsen [66], one of the important applications of model-driven development is in the generation of lower-level models, and eventually code, from one or more higher-level models. This process can be generalised as a *model transformation*, which encompasses all aspects of transforming one model instance into another; a formal description of a model transformation is discussed later in Section 3.1.10. This is particularly important within MDA, where many models within many layers of abstraction need to interact with each other [153]. For external domain-specific languages, this transformation step is essential if the language is to be used [102, pg. 46].

Model transformations can be implemented with many technologies and approaches, and a de-

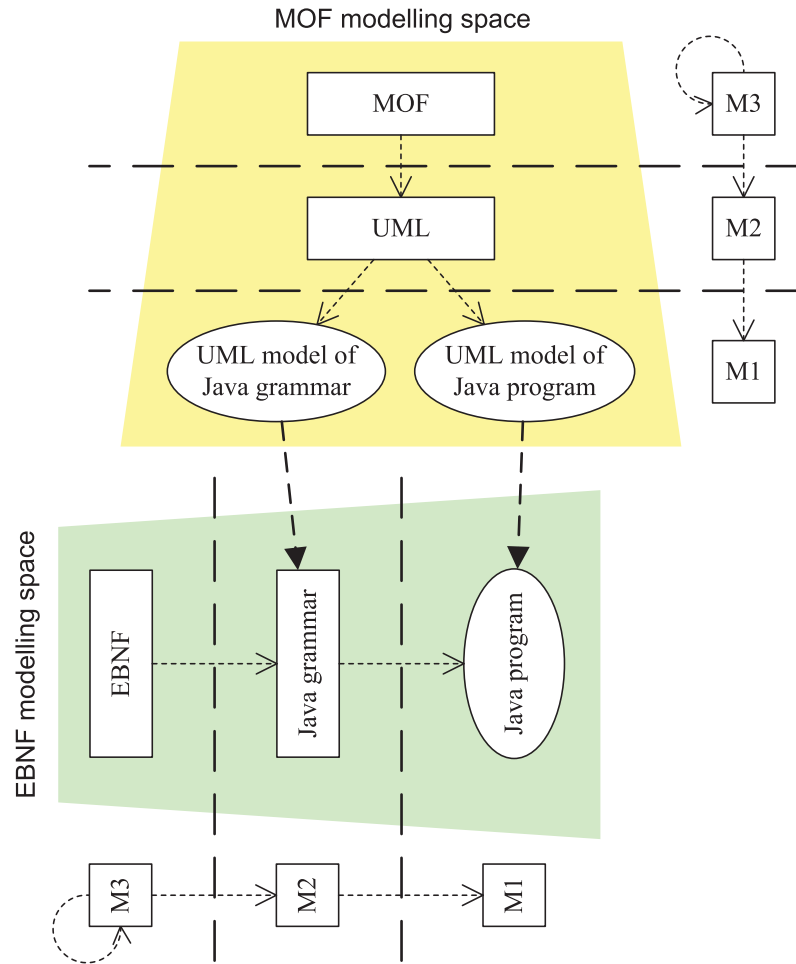


Figure 3.5: Dealing with different Modelling Spaces, adapted from Djurić et al. [80]

tailed evaluation of four existing technologies is provided later in Section 6.4. As Czarnecki and Helsen [66] discuss, each model transformation technology can support different features in the specifications of transformations. For example, technologies may support incremental transformations, or round-trip engineering through directionality and tracing.

It is possible to describe the architecture of model transformations using the metamodeling architecture of the MDA, where different model instances and metamodels span the M1, M2 and M3 metamodeling layers. Bézivin [33] proposes one common transformation pattern, which is illustrated here in Figure 3.6. The implementation of UWE uses this pattern heavily, as UWE model instances need to be transformed repeatedly in order to produce deployable artefacts [189].

### Code Generation

*Code generation* and *compilation* are two types of model transformation, and are especially common if programming languages are also considered as modelling languages. Both of these transformations take platform-independent models at a higher level of abstraction, and refine it into platform-specific models at a lower level of abstraction as *source code*. In particular, model transformation may be considered code generation if the target metamodel is on the *Code* layer of the MDA viewpoint architecture, or the model transformation can be considered a *model-to-text* transformation as discussed in the next section.

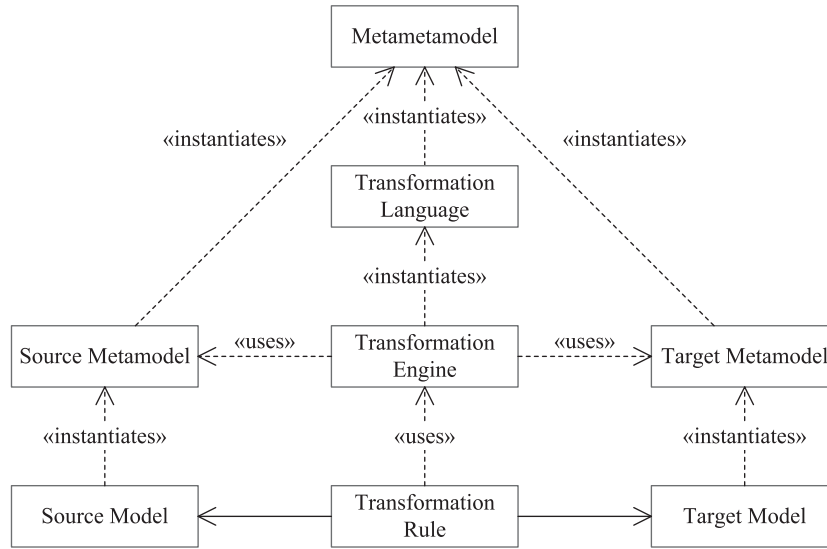


Figure 3.6: The Model Transformation Pattern in MDA, adapted from Bézivin [33]

It is unclear whether there is a difference between these two terms, and there is no consensus between the two terms in the research community [66]. In this research, the following distinction is made: while both techniques generate executable instructions, a *code generator* generates models (or source code) that are expected to be edited by a developer, whereas a *compiler* generates models (or source code) that are not expected to be edited manually.

### 3.1.8 Query/Views/Transformations Language

As part of the development of the MOF architecture, the OMG appreciated the need for a standard model-to-model transformation language. Consequently the OMG developed the Query/Views/Transformations (QVT) specification to satisfy this need [260]. QVT consists of three separate languages:

1. A high-level declarative language, *QVT Relations*. Instances of this language are generally represented using a graphical notation [190, pg. 107].
2. A low-level declarative language, *QVT Core*. *QVT Relations* transformations may themselves be translated into equivalent programs within *QVT Core* [174], and are generally represented textually.
3. An imperative language, *QVT Operational Mappings* or *QVTo*. Instances of this language are also generally represented textually.

Each of these languages have a different scope and area of intended use. For example, QVTr is designed for providing high-level model transformations, whereas QVTo is designed for implementing the code generation of higher-level models into lower-level and platform-independent models. Since this research was started, the OMG published the *MOF Model to Text Transformation Language* (MOFM2T) to address model-to-text transformations [255].

The QVT standard relies heavily upon MOF, and all three QVT languages are defined according to a MOF-compliant metamodel. QVT only supports model-to-model transformations between models that adhere to the MOF specification [260]. Unless a textual target language is first represented using

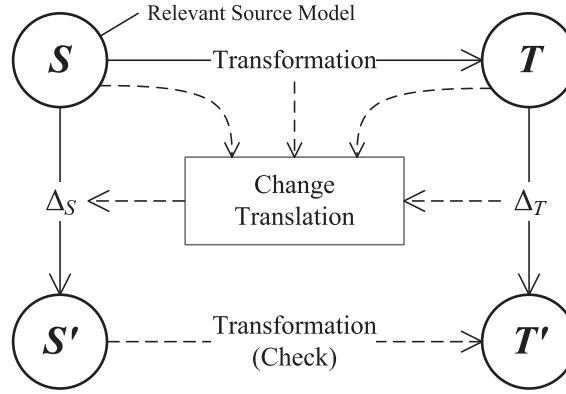


Figure 3.7: Using change models for round-trip engineering, adapted from Hettel et al. [153]

an intermediary modelling language, QVT-based approaches are not suitable for direct code generation. For example, the JavaML project [13] provides a metamodel for Java programs, similar to an abstract syntax tree, which can be utilised within model-to-model transformations [34].

The Atlas Transformation Language (ATL) [34] was one of the first implementations of the OMG’s proposal for QVT, and also provides an Eclipse-based implementation [173]. Jouault and Kurtev [174] investigate and discuss the architectural alignment of ATL and QVT, and find that “it is possible to have a reasonable interoperability between [the two languages]” [174, pg. 1194]. In particular, *QVT Core* and *QVT to* are very closely aligned to the design of ATL.

### Round-Trip Engineering

When using a model transformation that follows code generation concepts, it is often desirable to modify the translated model manually. These target models may be altered or extended, due to maintenance or changing requirements. However, the modified target model may no longer be an accurate result of the original source model transformation. *Round-trip engineering* (RTE) is the process used to ensure that the source model remains consistent when the target model is modified [153].

Hettel et al. [153] provide an excellent overview into the challenges in round-trip engineering, and also discuss some of the approaches used to solve this problem. One of these approaches, common in the model-driven development community, is on the use of *change models*, as illustrated in Figure 3.7. In change models, a change  $\Delta_T$  to the target model  $T$  is translated into a change  $\Delta_S$  from the source model  $S$ , as illustrated in this figure.

Round-trip engineering is very similar to *reverse engineering*, however, RTE is more concerned with keeping source and target models consistent by supporting both forward and reverse engineering. Chikofsky and Cross II define reverse engineering as a process of extracting *abstractions* from target systems, and these abstractions do not necessarily need to be identical to the abstractions of the original source models [55].

In general, the reverse engineering of model transformations within the *same* metamodels is very difficult. As Hettel et al. [153] discuss, this can only be achieved if the model transformation can be defined entirely using *injective* functions, each with an *injective inverse* function defined. However, even primitive arithmetic functions such as addition are not injective due to information loss<sup>5</sup>, making reverse engineering of arbitrary model transformations very difficult [153, pg. 33].

<sup>5</sup>For example, the expression “ $2+2$ ” may be transformed into “4”, but it is impossible to reverse engineer “4” back into “ $2+2$ ” without providing additional knowledge or making assumptions.

## Traceability

When translating requirements into the analysis, design or implementation from a set of requirements, it is often desirable for the relationships between these source and target artefacts to persist and be documented explicitly. This is known as *traceability*, and is desirable for model-driven approaches to improve understanding and maintainability [144, 185].

Traceability is particularly important when a PIM to PSM translation is not complete, and the user must fill in gaps in the PSM. If there is the possibility that the user may overwrite parts of the automatically translated code, and the environment cannot update the source model, then the user should at least be warned about any potential problems [185]. This is also closely related to round-trip engineering, where changes in the PSM need to be synchronised with the original PIM. Traceability is not a concern with frameworks, as frameworks are extensions of a source model within the same modelling layer.

Some Agile development proponents advocate skipping some or all traceability in favour of “travelling light” to reduce development overhead, as discussed by Hailpern and Tarr [144]. This approach is likely to be more successful if most of transformation processes are automated, and if model developers are supported by rich model-driven tools to deal with changing requirements. However, such an approach makes reverse and round-trip engineering much more difficult.

### 3.1.9 The Syntax and Semantics of Metamodels

While a metamodel is still being developed, it is acceptable for the meaning of the elements and structure within the metamodel to be contained within informal design specifications, use cases and test cases. As the language matures and is published however, this knowledge should be expressed formally, and possibly into a format that aids the computation and evaluation of the metamodel.

This specification process is also known as providing *semantics* to the modelling language, and there are many different ways in which this may be achieved. Harel and Rumpe [148] provide an excellent overview into the different types of semantics that a modelling language may be defined with. In particular, the *syntax* of a language represents its notation, and the *semantics* represents its meaning. In this section, four related concepts will be briefly discussed.

#### Syntax

In terms of modelling languages, the *syntax* or *structure* of the language defines how a valid model instance of that language may be constructed, and is often provided in terms of a formal language. For example, XML Schema [338] may be used to restrict instances of XML to a particular structure; similarly, EBNF is used to define the valid grammar of Java source code [130]. Syntax may be defined informally (such as in the English language), but this approach is often undesirable as informal definitions can introduce ambiguity or hinder machine evaluation. Syntax can also only provide a limited amount of meaning to a model instance; for example, a containment reference [314] can imply some sort of ownership or parent-child relationship.

#### Invariants

*Invariants* are a way to provide additional restrictions in a metamodel in situations where the restriction cannot be defined in the syntax of the metamodel. For example, the “acyclical class inheritance”



```

context Classifier::allParents() : Set(Classifier) body:
  allParents = self.parents()->union(
    self.parents()->collect(p | p.allParents())

context Classifier inv:
  not self.allParents()->includes(self)

```

Listing 2: Implementation of the *acyclical class inheritance* constraint of UML class diagrams in OCL [254, pg. 53–54]

restriction on UML class diagrams cannot be expressed in terms of MOF, but can be expressed as an invariant using the syntax of the OCL language [252], as illustrated in Listing 2.

### Formal Semantics

*Formal semantics* refer to the meaning that is provided through instances of a formally-defined language; that is, one that “has been endowed with precise and unambiguous definitions” [148, pg. 9], and are unambiguous enough that their assertions may be conclusively proven. A wide range of approaches provide frameworks for such formal semantic definitions, such as the operational frameworks of *structural operational semantics* [272], *Hoare logic* [155] and *abstract state machines* [141]. A full discussion on the functionality of each of these frameworks is well outside the scope of this thesis, however the interested reader is referred to a brief overview by Zhang and Xu [375].

### Informal Semantics

*Informal semantics* attempt to try and provide the same meaning to a modelling language as *formal semantics*, but within a language that permits ambiguity. Evolving modelling languages will often initially have their semantics defined informally – such as the definition of HTML 5 using the English language [343] – but informally-defined semantics cannot be proven without translation into some formal representation.

#### 3.1.10 A Formal Definition of Metamodels

In order to precisely understand the relationships between models, metamodels and model transformations, it is necessary to define the semantics of these terms and their relationships formally. In this thesis, the definition published earlier by Wright and Dietrich [369] will be used as a basis for discussing the formal semantics of metamodels, and this section will briefly summarise these definitions. In terms of the metamodeling architecture of the MDA, the meta-metamodel (M3) of these definitions is first-order logic [98].

One can consider a model to consist of a set of artefacts  $M$  which occur within the universe of all possible artefacts, i.e.  $model \in 2^M$ . The types of artefacts in this model are not restricted, but in order for a model to have any meaning, a particular model needs to be restricted against certain semantics in the modelled domain. To achieve this restriction, the metamodel  $\mathcal{S}$  is defined as the valid range of all possible models,  $\mathcal{S} \subseteq 2^M$ , and this metamodel represents a modelling language.

For example, consider a model universe  $2^M$  which contains all possible UML classes and UML class inheritance relationships; any instance of a UML class diagram *model* would then be a member of this model universe  $model \in 2^M$ . Since the UML specification defines a number of restrictions –



for example, class inheritance must be acyclical [254, pg. 69] – this universe needs to be restricted according to all of these restrictions in the UML metamodel  $\mathcal{S}_{UML}$ . Any set of artefacts within the model universe  $2^M$  that is also within the metamodel  $\mathcal{S}_{UML}$  would therefore be a valid UML model instance and UML class diagram instance.

This restriction can be defined using any number of mechanisms or technologies, including English language or mathematical definitions<sup>6</sup>. Most metamodelling technologies, such as UML and EMF (as discussed later in Section 6.2), support the restriction of model instances using structural techniques such as cardinality constraints and containment references. Restrictions that cannot be defined within the metamodelling environment may be introduced using specialised constraint languages such as OCL [254], as discussed later in Section 3.3.

### A Formal Definition of Model Transformations

As discussed earlier in Section 3.1.7 by Czarnecki and Helsen [66], a model transformation is the process of converting one or more source model instances into one or more target model instances; however, in this thesis, the formal definition of a model transformation will be restricted to only those transformations that process *one* source model instance into *one* target model instance.

With respect to two metamodels  $\mathcal{S}_1, \mathcal{S}_2$ , a model transformation can be portrayed as a function  $T : \mathcal{S}_1 \rightarrow \mathcal{S}_2$ . If the target metamodel is also the source metamodel, then the transformation can be simplified as a single-metamodel transformation  $T_1 : \mathcal{S} \rightarrow \mathcal{S}$ .

#### 3.1.11 The Bigger Picture of Modelling Approaches

The modelling community has not yet arrived at a consensus on how to consolidate and compare these different modelling approaches. While all of these approaches advocate the use of models in one way or another, and many can be used simultaneously without conflict, many do not describe their relationships with other modelling approaches.

In this thesis, these missing relationships are explored through the definitions proposed in Table 3.2, and this investigation is necessary in order to understand the bigger picture of modelling approaches. Many of these relationships are straightforward, and have been discussed in greater detail throughout this chapter.

The distinction between a modelling language (that is, a metamodel) and a programming language is hard to define, and needs to be discussed here. They are both abstractions of a system with well-defined syntax and semantics, and usually both have an implementation in a target language. The specific way in which an instance is represented does not affect the distinction, as modelling languages may be textual, and programming languages may be visual.

The main distinction between a modelling language and a programming language appears to be the difference in their scope. A programming language is often a general-purpose metamodel on the *PSM* layer of the MDA viewpoint architecture, used to define model instances of the *Code* layer; whereas a modelling language is often a domain-specific metamodel on the *PIM* layer.

However, domain-specific programming languages such as IA-64 assembly language [157], and general-purpose modelling languages such as UML [254], blur these distinctions. Clements [57] argues that there is a definite overlap between programming languages and modelling languages. In

---

<sup>6</sup>Wright and Dietrich [369] define a metamodel  $\mathcal{S}$  through the definition of a language  $L(R, F, C, V, =)$ , consisting of finite, pairwise disjoint sets of relations  $R$ , functions  $F$ , constants  $C$  and variables  $V$ , and a distinguished binary relation symbol  $=$  for equality.

Term	Definition
Model	A set of artefacts representing a simplified abstraction of reality, that are valid to the constraints of a modelling language.
Metamodel	Defines a valid type of model for a particular domain according to a set of defined syntax and semantics.
Programming Language	A general-purpose modelling language with a specific syntax, semantics, and compilation into a lower-level language.
Framework	The extension and abstraction of a programming language, contained within the same language, to simplify common tasks.
Domain-Specific Language (DSL)	A domain-specific programming language with a specific syntax, semantics, and transformations into a lower-level language.
Model Transformations	The translation of one model instance into another model instance, within the same, or across different, metamodels.
Code Generation	A model transformation with a target metamodel on the <i>Code</i> level of the MDA viewpoint architecture.
Model-Driven Development (MDD)	A development approach that considers models as first-class development citizens.
Model-Driven Engineering (MDE)	A development approach that encourages domain-specific languages and automated model transformations.
Model Driven Architecture (MDA)	Two four-layer architectures illustrating the relationships between different models in an integrated model-driven environment.

Table 3.2: The relationships between different modelling approaches

this thesis, a language is defined as a *programming language* if its intent is to be general-purpose, and as a *modelling language* or *metamodel* otherwise.

### 3.1.12 Metamodelling Environments

As discussed earlier in Section 2.3.1, a modelling environment should be supported with software tools in order to maximise the benefits of using models [180]. Such tools can include tools to define model instances, code generators, and analysis tools [300].

While the abstraction of complex systems has always been a part of software development, it is only recently with the development of new modelling software platforms that general-purpose model-driven software has become common. New model-driven technologies such as the Eclipse Modeling Framework [314] and the Graphical Modeling Framework [136] can provide a platform which encourages the cheap and quick development of model-driven approaches. A comprehensive evaluation of some of these existing frameworks is provided later in Section 6.2.

## 3.2 Model Completion

When designing a modelling language, a key challenge is balancing the level of detail in its design [369]. A language that is too simple will result in a rigid approach that cannot adapt to many situations; conversely, too much flexibility will force the developer to create and maintain large monolithic models. Software frameworks have to deal with this problem as well; the frameworks must be flexible, but also simple to develop with. Web application frameworks such as Ruby on Rails [321] and Symphony for PHP [274] have proposed to resolve this balance through the use of *documented conventions*.

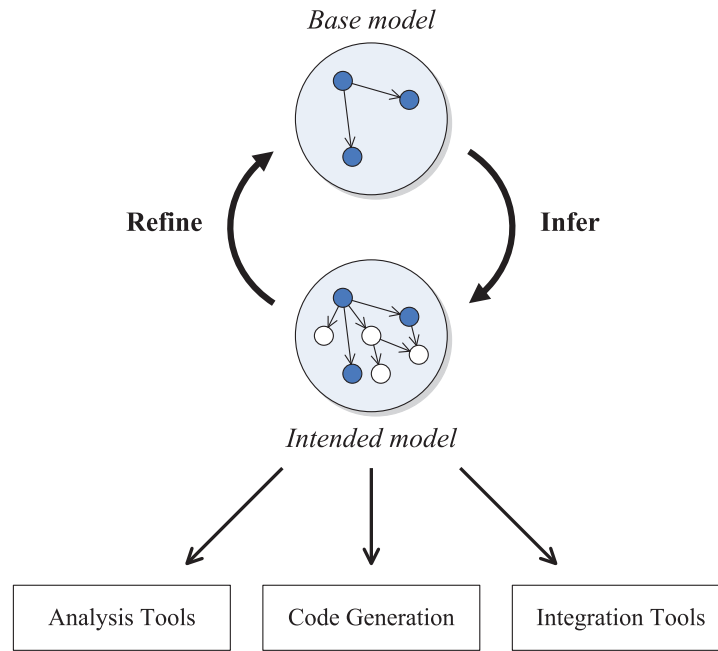


Figure 3.8: The model completion process within model-driven development, adapted from Wright and Dietrich [369]

*Model completion*, as proposed by Wright and Dietrich [369], adapt the concepts behind software frameworks to the model-driven domain in order to provide this flexibility and simplicity to the model-driven development of systems. The remainder of this section will briefly discuss the semantics behind this process, which are necessary to ensure the consistency of the approach; for more detail, the interested reader is referred to the paper [369].

### 3.2.1 Documented Conventions

Within these frameworks, *documented conventions* are an approach to automatically complete missing parts of the application [369]. The intent behind this process is to improve the productivity and efficiency of using the framework in common situations. For example, as web applications often require database integration, the Symfony web application framework supports the definition of an abstract database schema, which is then translated automatically into code for a particular database platform.

Three important aspects must be considered when using this completion process. Firstly, the conventions must be well-documented, so that the developer can anticipate changes without having to inspect source code. Secondly, the intent of the developer must always override the intent of the framework, and these intentions must never be overwritten; the developer must always be able to override these conventions, otherwise these conventions would become restrictions. Finally, if the framework has too many “magic” conventions, then it is likely that the complexity of the framework will become a mental burden for the developer.

### 3.2.2 Model Completion Concepts

Model completion is a single model transformation operating on a *base model* defined by a single metamodel, illustrated in Figure 3.8. Following documented conventions, a CASE tool takes the initial

model and transforms it into an *intended model* complete with all necessary detail. As discussed earlier in Section 2.3.1, implementing this process within a CASE tool is essential in order to maximize the benefits of MDE [180].

This approach also maps well onto evolutionary development processes, as once the base model is transformed into an intended model, it may be used as input to other tools such as code generators or analysis tools [300], and evaluated in order to obtain feedback. This feedback can then influence refinements to the base model as part of a new development iteration [369].

Importantly, model completion needs to complete the intended system based on incomplete knowledge, a process known as *non-monotonic reasoning* [115]. Applied to our domain, this states that when we add additional information to the system, any information inferred may be retracted in the presence of new knowledge. This reasoning philosophy provides a great deal of flexibility.

To illustrate this point, consider a user interface element representing a boolean property within the system. This interface element is part of a platform-independent model, allowing us to ignore the technical details of its implementation. It is reasonable to assume that by default, this property should be rendered by a checkbox. However, the developer may instead wish to represent this property with a drop-down list containing the values *yes* and *no*. In this case, the default checkbox should be removed; it should no longer be generated through model completion, nor override this new knowledge.

If negative existentials are used in defining part of a convention, non-monotonicity may be a consequence. For example, this *default checkbox rule* can be expressed non-monotonically as:

$$\begin{array}{ll} \text{IF} & (\text{there exists a boolean property}) \\ \text{AND} & (\text{there does not exist an editor for it}) \\ \hline \text{THEN} & (\text{create a checkbox editor for it}) \end{array}$$

Model completion restricts retraction within non-monotonic reasoning to only facts inferred by the reasoner itself; that is, the reasoning process can never retract any information in the base model. This is important to ensure developer effort is never inadvertently discarded.

### 3.2.3 Model Completion Semantics

In order to prove the consistency and correctness of this inference process within a model-driven implementation, we need to investigate the formal semantics of models and the model completion operation. This definition is essential to ensure that model completion preserves *consistency* with the original model [203]. Using the metamodel definitions provided earlier in Section 3.1.10, the model completion process itself will now be defined formally; for more detail, the interested reader is referred to Wright and Dietrich [369].

Given a metamodel  $\mathcal{S}$  such that  $model \in 2^M$  and  $\mathcal{S} \subseteq 2^M$ , model completion is defined as a function  $C : \mathcal{S} \rightarrow \mathcal{S}$  operating within the same metamodel. That is, all completed models will also be valid models in our domain. For a model completion  $C(X)$  operating on a model  $X \subseteq \mathcal{S}$ , two conditions need to be imposed:

1. *Extensive*: Model completion must not retract any existing information in the base model, i.e.  $X \subseteq C(X)$ . This is in contrast to many rule-based reasoners which permit the retraction of facts within the base model [311].
2. *Idempotent*: Once the intended model has been completed from a base model, applying model completion on this intended model will not change the model, i.e.  $C(X) = C(C(X))$ .

$$\begin{aligned}
\Phi_1 &= \text{property}(\mathbf{a}) \\
\Phi_2 &= \text{property}(x) \wedge \neg \exists y : \text{editor}(y) \wedge \text{editorFor}(x, y) \\
&\quad \rightarrow \text{checkbox}(\text{newCheckbox}(x)) \wedge \text{editorFor}(x, \text{newCheckbox}(x)) \\
\Phi_3 &= \text{checkbox}(x) \rightarrow \text{editor}(x) \\
\Phi_4 &= \text{dropdown}(x) \rightarrow \text{editor}(x)
\end{aligned}$$

Figure 3.9: Definition of the rule program  $C$  for the *default checkbox rule* convention, adapted from Wright and Dietrich [369]

$$\begin{aligned}
A &= \{\text{property}(\mathbf{a})\} \\
C(A) &= \{\text{property}(\mathbf{a}), \text{checkbox}(\text{newCheckbox}(\mathbf{a})), \\
&\quad \text{editorFor}(\mathbf{a}, \text{newCheckbox}(\mathbf{a}))\}
\end{aligned}$$

Figure 3.10: Model completion  $C(A)$ , adapted from Wright and Dietrich [369]

These two conditions are part of Tarski’s classical axioms for inference operators [317]. Monotonicity is not a necessary condition; in the face of new information within a base model, previously inferred knowledge may need to be retracted, i.e.  $X \subseteq Y \not\Rightarrow C(X) \subseteq C(Y)$  [369].

In order to implement these conditions on the model completion function, new elements must be created according to a *factory function*, and introduced into the intended model using the concept of *stratification* [327]. This allows for the conventions to be represented as a set of rule formulae in a *rule program*, which may be evaluated against a model instance to complete the intended model. By using an *insertion cache* in the implementation of the factory function, the model completion process has been demonstrated to satisfy the extensive and idempotent conditions on  $C(X)$  [369].

Within this definition of model completion, the *default checkbox rule* may be expressed using the rule program  $C$  illustrated in Figure 3.9. This can then execute on an initial model  $A$  to create the completed model  $C(A)$ , as in Figure 3.10. The full expansion of the steps necessary to complete the intended model in this example through model completion is not provided here; the interested reader is instead referred to Wright and Dietrich [369].

### 3.3 Model Instance Verification

As discussed earlier in Section 2.3.1, an important feature requirement of a modelling language for RIAs is to support integration with formal verification tools. Software verification of an implemented application is the standard method for verifying correctness properties on that application [230], such as affirming design constraints or identify potential security flaws. Verifiable web application properties include non-functional requirements such as broken links, syntax validation, load testing, and page response time [26].

However, in many cases it is preferable to verify a model instance instead of an implementation, as the relative cost of fixing an error dramatically increases over time [278, pg. 197]. This also allows properties of web applications such as concurrency and user interaction to be evaluated directly against the system model [305], and often with less performance impact than testing an implementation [41]. There is a significant body of existing work on verifying model instances *reverse-engineered* from an implementation [25, 75, 303], keeping in mind that it is usually unfeasible to extract structural design intent from code [220].

### 3.3.1 Validation vs. Verification

Validation and verification are two closely related concepts, and are often combined together into the general term *verification & validation* (V&V). The differences between these two terms are fairly subtle, as illustrated by the wide range of definitions published by existing work.

For example, Adrion et al. [4, pg. 188] defines *validation* as “determination of the correctness of the final program or software produced from a development project with respect to the users needs and requirements” by verifying each stage of the development life cycle, and *verification* as “in general, the demonstration of consistency, completeness, and correctness of the software at each stage and between each stage of the development life cycle.” Adrion et al. also distinguishes *valid input* separately from validation, as data that lies within a particular domain.

Conversely, Sargent [294, pg. 37] defines *verification* as “ensuring that the computer program of the computerized model and its implementation are correct”, but it is unclear what “correct” is referring to; and *validation* as “substantiation that a computerized model within its domain of applicability possesses a satisfactory range of accuracy consistent with the intended application of the model”. These definitions appear to be at odds with those defined by Adrion et al.

In this thesis, the definitions by Pressman [278, pg. 479] and Boehm [37, pg. 37] are adapted to the model-driven domain in order to distinguish the differences between these two terms. In particular, the following definitions of V&V are used in this thesis:

**Model verification** refers to the set of activities that ensure that the model correctly implements a specific function; and

**Model validation** refers to a different set of activities that ensure that the built model is traceable to customer requirements.

In practical terms, this means that a requirements-independent check of a model instance – such as checking that all element IDs are unique, or that all private data is secure – is an instance of a *model verification* activity. Conversely, *model validation* is the process in checking that the model instance implements the requirements of the model – such as making sure that the “View Product” page is present, or that a particular key is secure.

These definitions are not consistent with some published research which erroneously use the terms interchangeably. For example, Ricca and Tonella [285] propose that there are two types of web application testing techniques, *static verification* and *dynamic validation*; but in this thesis, these tests would both be treated as instances of verification, as they do not test the application requirements themselves.

### 3.3.2 A Formal Definitions of Constraints

One may consider model instance verification as a process operating on syntactically correct models that identifies invalid models through constraint violation. Earlier in Section 3.1.10, a formal definition of metamodels and their model instances was provided based on Wright and Dietrich [369]; this definition may be extended to describe the process of model instance verification.

In particular, a model  $X$  consists of a set of artefacts from  $M$  constrained by a particular metamodel  $\mathcal{S}$ , i.e.  $X \in \mathcal{S} \subseteq 2^M$ , with  $\mathcal{S}$  defined by a language  $L(R, F, C, V, =)$ . A set of constraints may be used to identify valid models by defining a verification function  $V_f : 2^M \rightarrow \{0, 1\}$ ; a particular model  $X$  is valid according to this function if  $V_f(X) = 1$ .



A constraint is defined in terms of a verification language  $L_{ver}$ , consisting of type predicates<sup>7</sup>, n-ary relations, variables and standard predicate logic connectives and quantifiers. A constraint has the following form:

$$\forall x_1, \dots, x_n : T_1(x_1) \wedge \dots \wedge T_n(x_n) \rightarrow P(x_1, \dots, x_n)$$

where  $n$  represents the *arity* of the constraint;  $x_1, \dots, x_n$  are variables;  $T_i$  represent unary type predicates within the relations set  $L_R$ ; and  $P$  a constraint formula containing only the free variables  $x_1, \dots, x_n$ . A model is therefore valid with respect to this constraint if this formula is satisfied. The unary type predicates  $T_i$  ensure that the constraint is always satisfied if evaluated against instances of non-matching types.

The expressiveness of the verification language  $L_{ver}$  may impact on the performance and decidability of its constraints; for example, OWL 2 Full [351] is the most expressive language within the OWL family, but its expressiveness has been proven to be undecidable [240]. Similarly, relations expressed in CrocoPat may be queried by identifying graph patterns, but this subgraph isomorphism problem is NP-complete [32, pg. 143]. Different verification engines may therefore be classified on expressiveness restrictions, and on their maximum constraint tuple size.

### Expressiveness Criteria

In this thesis, the expressiveness of a verification language is defined in terms of the permitted syntax of constraint formulae. In particular, three non-exclusive categories of language expressiveness are considered:

1. **Functions:** In languages that can be used for verification such as OCL, constraint predicates are defined in terms of functions defined within the model instances, and additional functions may be defined within the language. It is not possible to define additional relations within the language.

For example, a constraint “all *Named Elements* must define a name” may be specified using the predicate  $P_{named}(x) = \exists n : name(x) = n$ . The resulting constraint may then be defined as the  $\forall_x T_{Named\ Element}(x) : P_{named}(x)$ .

2. **Relations:** In languages that can be used for verification such as OWL or Jena Rules, constraint predicates are defined in terms of relations, and additional relations may be defined within the language using first-order logic. Relations may use self-reference to support recursion.

A constraint “a *Frame* cannot redirect to itself” may first be specified through the relation  $R_{redirects}(x, y) = \exists e, w : T_{Event}(e) \wedge onAccess(x, e) \wedge T_{ECA\ Rule}(w) \wedge from(w, e) \wedge to(w, y)$ .

An additional relation may then define the transitive closure of this relation through  $R_{redirects}(x, y) = \exists z : R_{redirects}(x, z) \wedge R_{redirects}(z, y)$ .

The resulting constraint may then be defined as  $\forall_x T_{Frame}(x) : \neg R_{redirects}(x, x)$ .

3. **Higher-order Logic:** In languages that can be used for verification such as OCL and OWL, the verification language supports the definition of additional relations or predicates using higher-order logic.

<sup>7</sup>In this thesis, types are represented as unary predicates; for example, the type *Visible Thing* is represented as the type predicate  $T_{Visible\ Thing}(x)$  on a model artefact  $x$  of that type, as illustrated by Wright and Dietrich [369].

For example, a relations-based verification language may provide the higher-order operator **TC** to represent transitive closure for a particular relation. This would permit the transitive closure definition of the previous constraint to be defined instead as  $\mathbf{TC}(R_{\text{redirects}}(x, y)) \Rightarrow R_{\text{redirects}}(x, y)$ .

Similarly within a functions-based verification language, the higher-order aggregation function **fold** may be used to evaluate a function over a set of artefacts and reduce it into a single value.

If the level of expressiveness of each language impacts on the performance of a model instance verification process, it may be desirable to support many verification languages simultaneously. This would allow simple constraints – implemented using a functions-based verification language – to be evaluated quickly and regularly, while more complex constraints – implemented using a relations-based language – may be evaluated less frequently, due to their resource requirements.

### Model Checking

*Model checking* is defined by Baier and Katoen [14, pg. 11] as “an automated technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds for (a given state in) that model.” In particular, these constraints are checked against the entire state space of the system [149].

For a system to be evaluated using model checking, it must first be translated into a list of potential states and the permitted transitions between these states; this *system model* may then be evaluated against a property using a language such as LTL [14]. These constraints cannot be described using the general constraint form defined earlier in  $L_{\text{ver}}$ , and a formal definition of model checking is well outside the scope of this thesis; for a comprehensive and complete description of the principles of model checking, the interested reader is referred to Baier and Katoen [14].

However, since a model checker must navigate through *all* of the possible states of a modelled system, there is a significant performance penalty in evaluating this process. Consequently model checking may only be evaluated infrequently – for example, when a model instance is used to generate a production system, or as part of a long-running automated process to catch bugs – and it is often preferable to express a constraint using one of the verification languages discussed earlier.

#### 3.3.3 Discussion

As each of these verification approaches have different performance characteristics, model instance verification will be implemented with regards to three categories of language expressiveness. This will allow a model developer to evaluate a model instance against increasingly precise definitions of correctness, dependent on the resources available to the developer. In this thesis, these three categories are defined as follows:

1. *Function Language*: Functions-based constraint languages will be used to quickly verify constraints, and will therefore be evaluated frequently against the most common design problems.
2. *Relation Language with Higher-order Logic*: Relations-based constraint languages with support for higher-order logic constructs will be used to verify more complex constraints, and will therefore be evaluated less frequently against more complex design problems.
3. *Model Checking Language*: Finally, a model checking approach will be used to evaluate behavioural correctness of the modelled systems. As this approach requires significant amounts of resources, it is intended that this approach will be evaluated fairly infrequently.



### 3.4 Visual Modelling

As discussed by Moody [236], visual representations of models can be more effective than textual representations, because they can tap into the capabilities of the powerful human visual system. In particular, he argues that diagrams are believed to convey information more effectively, concisely and precisely than textual language, and the information is more likely to be remembered. Visual notations are also particularly important when communicating with end-users and customers.

Depending on the requirements of the artefacts, visual modelling can range from strictly formal approaches – visual models can be exactly as formal and structured as textual models [236] – to informal approaches as used by individual designers. UML models are intended to be drawn by hand, as communication was thought to be more important than automation (and therefore precision). However, recent revisions of UML have focused on improving its architecture and formalism in order to support software integration and reasoning [188].

Visual representations can be particularly powerful when applied to the domain of model-driven development, as visual representations are themselves models. The combination of a domain-specific language and an accompanying visual representation can be termed as a *domain-specific visual language*, as defined by Grundy et al. [137]. This visual language can be defined from a metamodel specification with a certain level of automation [308]. However, when developing a visual language for the representation of a model, it is important to document the rationale and design decisions used [236, 137].

There are situations where visual modelling may not be appropriate, and a textual representation would be better. Vlissides and Linton [329] argue that graphical models are best for domain-specific languages, rather than general-purpose languages. Graphical languages generally lack *efficiency of expression*; that is, it is generally very difficult to visually design a complex algorithm [308]. Finally, without the support of a graphical modelling environment or framework, the implementation of a visual modelling language is often more difficult than the implementation of a textual modelling language.

#### 3.4.1 Visual Metaphors

A *metaphor* is a linguistic device where one or more words for a concept are used outside their conventional meaning, to express a *similar* concept [201]. Metaphors are very popular in software development<sup>8</sup>, in particular *interface metaphors* for user interface design, and *system metaphors* for the development of the software system. These metaphors reduce the mental load for developers, and improve the accessibility of the system [278]. Metaphors can also apply to visual modelling as a *visual metaphor*; that is, a metaphor used to relate a visual representation of a model instance with a foreign concept.

Barr et al. [20] expand on the work of Lakoff and Johnson [201] to propose a taxonomy of interface metaphors. This taxonomy includes *orientational*, *ontological*, and *structural* metaphors. Structural metaphors – those that deal more directly with physical objects in the real world – are particularly prominent in user interface design [20]. For example, the “desktop metaphor”, which represents files within a file system as objects on an actual desk, is an instance of a structural metaphor.

---

<sup>8</sup>The Rational Unified Process does not use term *metaphor* as a linguistic device, but as a short story that replaces the notion of system architecture [273].

### 3.4.2 Visual Metaphors in Existing Models

Existing visual modelling languages already use visual metaphors in their design and implementation, and this thesis will discuss some of the metaphors identified in WebML and UWE. Each of the WebML models represented visually has a particular visual metaphor, although the WebML specification does not define these metaphors directly. The WebML hypertext model, for example, appears to follow the visual metaphor of *a web application is the flow of data*, because the relationships between content units are expressed as a flow.

Some papers on WebML claim that each *content unit* in the model uses a visual metaphor; for example, Di Martino et al. describe that “the *MultiMap* Unit is a visual metaphor for a map viewer able to render both vector and raster data arranged in layers” [76, pg. 2]. According to the definition of *visual metaphor* in this thesis, this is not a visual metaphor, because their description only defines the visual representation of the *rendered* unit, and *not* the visual representation of the unit itself within a model instance.

Each model of UWE also has a particular visual metaphor, although these metaphors are also not explicitly discussed. Many UWE models are extensions of existing UML models – such as class diagrams and activity diagrams – so these models reuse their existing metaphors. On the other hand, some UWE models use entirely new metaphors; for example, the *navigation structure* model represents the web application using the metaphor of *web application navigation using states and menus* [192]. Since UWE reuses common visual metaphors (from UML), it is easier for new users to pick up and understand these models, and it also promotes sharing these models with users who already understand the metaphors of UML. It would therefore be beneficial for a modelling language to reuse existing visual metaphors where appropriate.

### 3.4.3 Visual Modelling Software

While visual models can be drawn by hand and exchanged manually, software can be used as part of the process. Visual modelling software is a strictly optional component of model-driven approaches, but it can improve the efficiency of model design and development, and the quality of communication with other end-users. These editors, sometimes called *graphical object editors*, allow a user to manipulate the graphical representations of model instances directly [329]. Visual modelling software has already been discussed in depth by existing literature [329, 236, 286]; a range of existing visual modelling frameworks is discussed later in Section 6.3.

Since these visual editors provide interactive representations of an underlying model instance, these editors map naturally to the model-driven development approach advocated in this thesis. Since different graphical editor instances often share similar functionality – such as printing, layout and shape types – metamodels may be used to simplify the implementation of a model-driven graphical editor. These *model-driven graphical modelling environments* are represented as a model instance, and translated using model transformations into the source code of the final graphical editor.

### 3.4.4 Visual Notation

For instances of a modelling language to be represented visually, a corresponding visual notation needs to be designed, either formally or informally. The development of visual representations for modelling languages is a very large research area, and is not the main focus of this research; a full discussion on existing research into visual notations is therefore well outside the scope of this thesis. However,

Variable	Power	Capacity
Horizontal position (x)	Interval	10–15
Vertical position (y)	Interval	10–15
Size	Interval	20
Brightness	Ordinal	6–7
Colour	Nominal	7–10
Texture	Nominal	2–5
Shape	Nominal	Unlimited
Orientation	Nominal	4

Table 3.3: Information encoding capacity of different visual variables, adapted from Moody [236, pg. 770]

this section will briefly discuss some related literature on the design of effective visual notations for modelling languages.

### Notation Information Capacity

To evaluate the effectiveness of a visual notation, Moody [236] proposes measuring *cognitive effectiveness* through the systematic evaluation of a number of notation variables. By measuring these variables in a quantitative way, the effectiveness of two similar visual notations can be effectively compared, and this section will discuss two of these approaches.

The first approach – *ontological analysis* – measures the effectiveness of mapping the visual notation to the underlying model instance. In particular, four detrimental anomalies are identified, with respect to a desirable one-to-one mapping between constructs and notation [236, pg. 759]:

1. **Construct deficit**, a model concept does not have a corresponding visual notation;
2. **Construct redundancy**, multiple visual notations can represent a single model concept;
3. **Construct overload**, a visual notation can represent multiple model concepts; and
4. **Construct excess**, a visual notation does not represent any model concepts.

The second approach focuses on measuring the *information capacity* of a visual element with respect to a number of different visual variables, each of which have a particular information capacity. For example, Moody argues that we can distinguish between an unlimited number of element shapes, but we can only distinguish between a few element textures. The capacity of various visual variables as discussed by Moody are illustrated in Table 3.3; in particular, *power* represents the highest level of measurement that can be encoded, and *capacity* represents the number of perceptible steps.

### Cognitive Dimensions Framework

The *cognitive dimensions* framework proposed by Green and Petre [133] defines thirteen dimensions for the subjective evaluation of user interfaces. This framework may be used to evaluate the usability of visual modelling languages, as illustrated by the evaluation of the View Mapping Language (VML) by Grundy et al. [138]. A summary of these thirteen dimensions is provided in Appendix H.

As discussed by Green and Petre [133, pg. 139], no given notation can satisfy all of these dimensions simultaneously; designers must understand the trade-offs between these dimensions in order to design the most effective visual notation. These trade-offs are most apparent when proposing a change to an existing visual notation, rather than trade-offs involved in developing a new visual notation from scratch. For example, the introduction of an additional *abstraction* into a visual notation may reduce the *viscosity* of the notation and introduce *hidden dependencies*, but the change to the *closeness-of-mapping* entirely depends on the existing visual notation and the form of the new abstraction [133, pg. 164–165].

### Graphical Language Guidelines

Rumbaugh [291] proposes a list of twelve guidelines for the development of graphical modelling languages. One cannot evaluate the satisfaction of a visual notation against all of these guidelines through quantitative means; however, some of these guidelines – such as “*must fax and copy well using monochrome images*” and “*easy to draw by hand*” – are still useful to consider when designing a visual notation.

Some of the guidelines proposed by Rumbaugh can be indirectly evaluated in terms of the visual notation guidelines proposed by Moody [236]. For example, Rumbaugh’s “*no overloading of symbols*” guideline can be considered equivalent with Moody’s guideline of *construct overload*; and “*uniform mapping*” equivalent to Moody’s *ontological analysis* goal of having a one-to-one mapping between concepts and constructs [236].

## 3.5 Conclusion

In this chapter, different approaches and technologies to model-driven development have been investigated to assist in the design and implementation of a new modelling languages for RIAs. In this proposed modelling language, both the viewpoint and metamodeling architectures of the MDA will be considered; model transformations will be used to translate models into executable code; existing visual metaphors will be used for the visual representation of the modelling language where appropriate; and the technique of *model completion* will be introduced to improve the flexibility of model instance development.



## Chapter 4

# Rich Internet Application Modelling Concepts

As discussed earlier in the Background chapter, a modelling language for RIAs may need to support the expression of many web concepts, such as events, users or access control. This chapter will investigate and discuss a range of existing approaches and implementations of each concept, but will not explicitly discuss the suitability of each approach with respect to RIA modelling languages. These discussions will instead be used to inspire the modelling language definition in the following chapter.

### 4.1 Managing Complexity through Structural Decomposition

When developing a modelling language, it is important to consider the complexity of the resulting model instances, which can be strongly influenced by the structural design of the language. It is often more desirable to work with many simple model instances than to work with a single monolithic model instance [278, pg. 344]; this is particularly true when working with a visual modelling language [236].

This section will investigate the techniques discussed by Pressman [278, pg. 348–349] for using structural decomposition to deal with the complexity of model instances<sup>1</sup>. These techniques are not mutually exclusive, but rather represent a variable measure of the extent that each technique is used; for example, different systems can have differing degrees of modularity [16].

#### 4.1.1 Modularity

The most common way of reducing complexity of large systems is to structurally divide them into smaller parts or *modules*, a technique called *modularity* [229, pg. 39-66]. These independent modules work together, with different levels of interconnectivity representing differing *degrees* of modularity [16]. As Pressman [278, pg. 348–349] discusses, modularity may be achieved through horizontal or vertical partitioning, and Figure 4.1 illustrates the use of *horizontal partitioning*.

Modularity can encourage system reuse as long as the system has been implemented correctly [229, pg. 39], as modules with few dependencies on other modules may be more easily reused in other contexts. Modularity also encourages the use of small units, rather than larger ones which may be more difficult to understand. However, a poorly-implemented modular system can be confusing and

---

<sup>1</sup>The complexity of a modelling language, rather than its instances, has been discussed earlier with metamodeling metrics in Section 2.6.2.

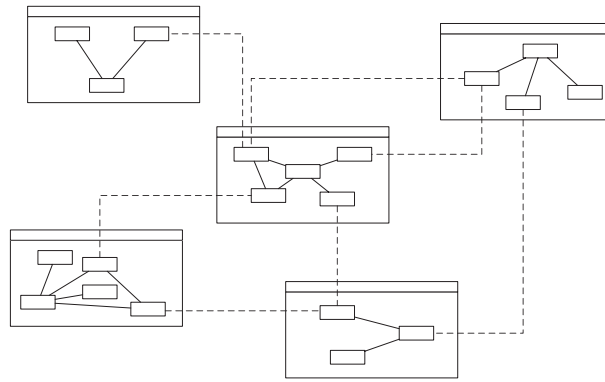


Figure 4.1: Managing complexity through horizontal partitioning

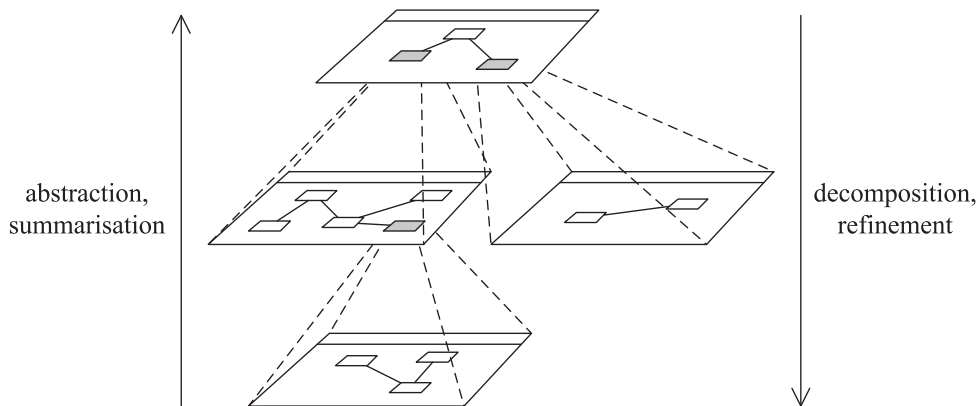


Figure 4.2: Managing complexity through hierarchical modelling, adapted from Moody [236]

difficult to maintain, if modules are made too small or too large [278, pg. 344], or there are too many dependencies between modules. Without adequate documentation it can also be difficult to understand the bigger picture of a modular system. The effectiveness of a module is often expressed in terms of *cohesion* and *coupling*, as discussed by Pressman [278, pg. 353-355].

It appears that WebML [51], as discussed earlier in Section 2.4.1, has been designed to support modularity in its resulting model instances through horizontal partitioning. In particular, an entire web application is expressed in a single monolithic model, but certain areas – such as pages, and operations – may be grouped together, in a manner similar to a module.

### 4.1.2 Hierarchical Modelling

As discussed by Moody [236, pg. 767], *hierarchy* is “one of the most effective ways of organizing complexity for human comprehension as it allows systems to be represented at different levels of detail, with complexity manageable at each level.” It supports the decomposition of a complex system into separate models with each model becoming a part of a higher-level model, as illustrated in Figure 4.2. This reduces the susceptibility to side-effects from changes applied to lower-level modules, increasing maintainability of the system [278, pg. 349].

Hierarchy is also known as structural modularity through *vertical partitioning* [278, pg. 349] and is distinguishable from horizontal partitioning. In horizontal partitioning, the relationships connecting modules are those between *individual* model elements encapsulated as modules, whereas the relationships in hierarchy are those between the higher-level layers. This means that a structurally modular

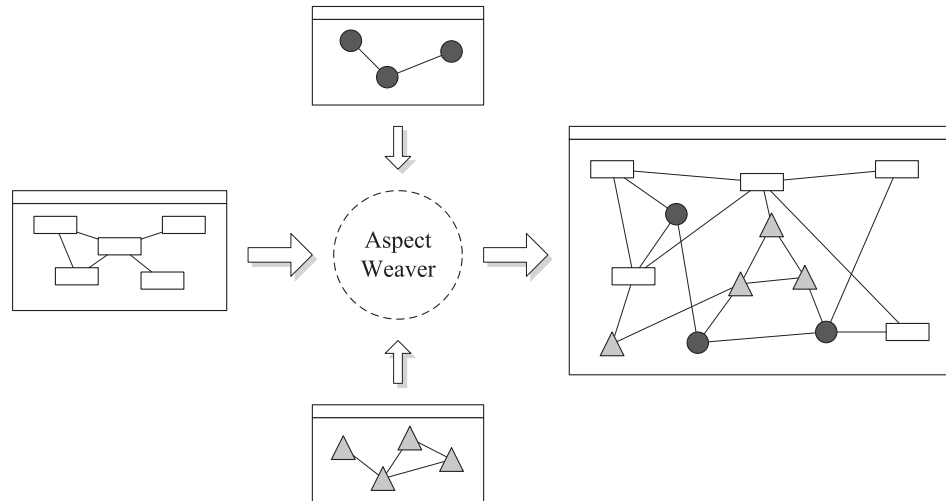


Figure 4.3: Managing complexity through aspect-oriented modelling, adapted from Kiczales et al. [182]

system may be both horizontally and vertically decomposed, as discussed by Meyer [229, pg. 40].

Hierarchical modelling encourages both top-down design and bottom-up design, in the sense that a model developer may abstract away complexity into many layers. It is important to provide appropriate context in a hierarchical environment, to orient the current view with the rest of the modelled system [236]<sup>2</sup>. One example of a modelling language designed with hierarchy in mind is UML [254], where a system may be progressively decomposed through deployment diagrams, component diagrams, class diagrams, and activity diagrams.

### 4.1.3 Aspects

Aspect-oriented programming is not a technique for structural decomposition, but is a relatively recent innovation that separates different functionality concerns into new abstractions known as *aspects* [97, 182]. Aspect-oriented modelling adopts aspect-oriented programming to a model-driven environment, supporting the modelling of separate concerns of a model instance in separate models, which are then integrated together into a target model [103]. Concerns that span over the entire model are known as *cross-cutting concerns*, which are integrated as aspects together using an *aspect weaver*, as in Figure 4.3.

Using aspect-oriented models has already been done in the domain of modelling web applications. For example, UWE uses aspect-oriented modelling to support the modelling of web application adaptivity [22]. Other web application concerns such as access control, logging and user authentication can also be modelled using aspects with various degrees of independence.

As a relatively recent innovation, aspect-oriented approaches still have a number of important issues that need to be resolved. Aspect-oriented programming always depends on the underlying code that they weave, hindering their testability and reuse [7]. Understanding the interaction between many diverse aspects in a system, and how these interactions evolve over time during maintenance – known as *cognitive distance* – remains a critical issue [5]. The success of an aspect-oriented modelling approach is therefore very dependent on its implementation.

<sup>2</sup>In the proof-of-concept implementation of IAML, this is achieved by using breadcrumbing and shortcuts, as discussed later in Section 7.4.4.



#### 4.1.4 Discussion

These techniques are not mutually exclusive, and may be combined in different ways [229, pg. 40]. As discussed earlier in Section 2.7.2, software development should adapt existing software process models to the individual needs of the project; similarly, complexity decomposition approaches should be adapted to meet the needs of a modelling language.

For example, it would be possible to design a system using aspect-oriented modelling, with the modelled aspects following a hierarchical design; conversely, it would be possible to design a system using hierarchical modelling, and then apply aspect-oriented modelling to the different levels of hierarchy. Each of these systems would have different benefits and drawbacks; in the case of the former, it would be difficult for lower levels of the aspect hierarchy to communicate, whereas with the latter the aspects could not consider the higher levels of the hierarchy.

## 4.2 Events

As discussed earlier in Section 2.3.1, RIAs may be considered event-driven applications, and an RIA modelling language should have strong support for modelling web application events. In this thesis, the definition of an *event* is adapted from the UML specification; that is, an event is defined as “the specification of some occurrence that may potentially trigger effects” [254, pg. 440].

An event may trigger a particular action, and this invocation relationship may be restricted with a condition; this definition forms an *Event-Condition-Action* (ECA) rule, forming the basis of event definitions in this thesis. As discussed by Papamarkos et al. [268], ECA rules have been used in many settings, including active databases, publication/subscription technology, and in the implementation and specification of business processes. An ECA rule has the general syntax [79, 268]:

*on event if condition do action*

In order to select a conceptual approach for the modelling of events, the following sections will briefly illustrate and informally evaluate a range of existing approaches in terms of expressibility and simplicity. A similar approach was performed during the development of the R2ML rule modelling language [125]; Wagner et al. [354] evaluated seven existing approaches for visual rule modelling as the basis of event modelling in R2ML. For each modelling approach, the representation of a generic ECA rule will be illustrated in order to highlight the differences in syntax.

### 4.2.1 UML

UML permits the modelling of event handlers using *AcceptEventActions* [254, pg. 235], which can be used in UML activity diagrams to trigger behaviours. In UML, both signals and operation calls can be considered as events, and consequently *SendSignalActions* can be used to model a triggered event, with incoming parameters modelled using *InputPins*. This specification allows for an ECA rule to be modelled as illustrated in Figure 4.4.

Depending on the complexity of the ECA rule, a number of visual optimisations may be performed; these optimisations are not illustrated in Figure 4.4. If the condition is simple or the action is not restricted by a condition, the use of the *DecisionNode* with attached *decision behaviour* is not necessary. If the condition can be reduced into an instance of *ValueSpecification*, this specification can be used as a guard condition on a single *ActivityEdge*.

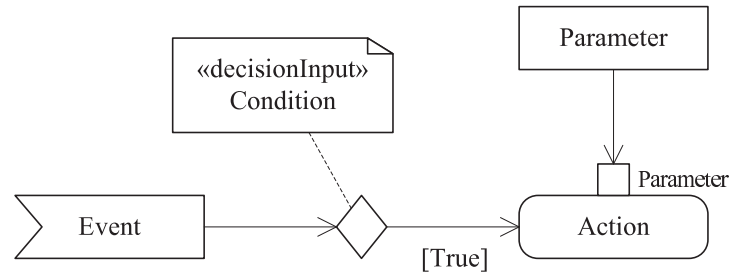


Figure 4.4: Event-Condition-Actions in UML activity diagrams [254]

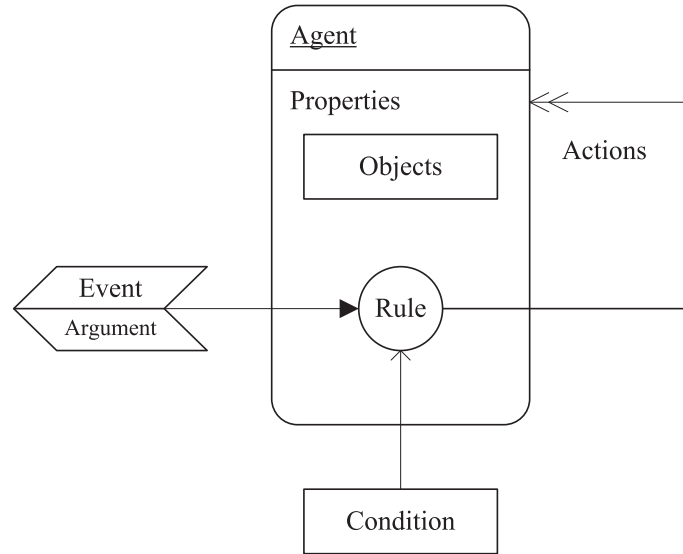


Figure 4.5: An AORML Interaction Pattern diagram, adapted from Wagner et al. [354]

UML also defines a number of fundamental events, such as *ChangeEvent* and *MessageEvent*, which are all instances of the *Event* abstract class. In order to define new events on the UML level, it appears that a UML Profile must be defined; conversely, to define new events on the model instance level, it appears that a *SignalEvent* may be used to handle instances of *Signals* as events.

### 4.2.2 AORML

The *Agent-Object-Relationship Modelling Language* (AORML) [352] is an agent-based modelling language that can describe the interactions between multiple agents in a system. As the name implies, AORML is intended to model agents rather than software components, with ECA rules owned by a parent agent; this may hinder the use of AORML in systems which are not agent-based. This language supports a wide range of visual notations for event modelling; for example, AORML can be used to describe ECA rules in terms of an agent as in Figure 4.5 [354].

### 4.2.3 URML

The REVERSE Working Group I1 developed the *UML-based Rule Modelling Language* (URML) [355] to allow visual rule modelling, and this metamodel largely overlaps with the metamodel of R2ML. In particular, URML can be “considered as a language that is derived from R2ML in order to provide UML-based rule modelling” [355].

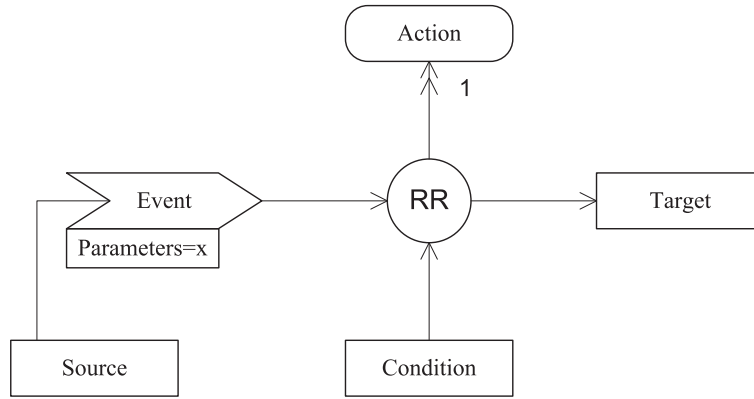


Figure 4.6: Modelling Reaction Rules in URML, adapted from Wagner et al. [355]

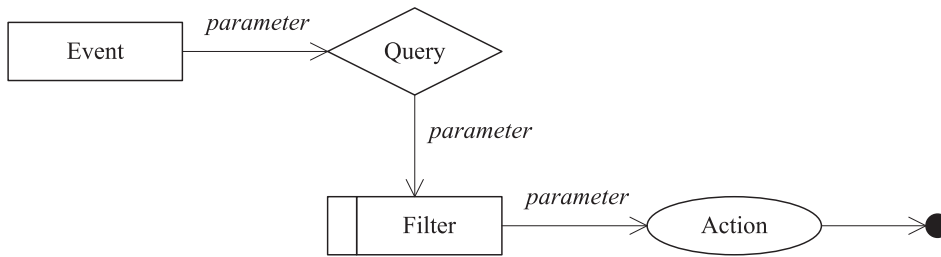


Figure 4.7: Visual Event Handler definitions in Kaitiaki, adapted from Grundy et al. [139]

URML supports three types of rule modelling – derivation rules, production rules, and reaction rules – but only reaction rules will be considered in this section, as their design is the most similar concept to ECA rules. The visual modelling of reaction rules in URML is illustrated in Figure 4.6, as defined by Wagner et al. [355].

#### 4.2.4 Kaitiaki

As described by Liu et al. [215], the *Kaitiaki* project uses the *Event-Query-Filter-Action* metaphor in order to describe and define event handlers, which is very similar to ECA rules. This events metaphor has been integrated with the web service composition specification language *ViTABaL-WS*, as discussed by Li et al. [209]. The *Kaitiaki* project is supported with a proof-of-concept implementation within the Eclipse-based *Marama* framework [139].

#### 4.2.5 Event Algebra

Events can also be modelled in a more formal way using *event algebra*, especially when modelling the timeline of the interaction between different event instances. As discussed earlier in Section 3.1.9, the definition of the formal semantics of a modelling language may be used to identify undesirable properties of a modelled system, and these benefits similarly apply in the case of event algebra. Similarly, event algebra may be used to improve the expressiveness of model verification constraints.

A full discussion on the different types of event algebras is well outside the scope of this thesis, but one of these event algebras will be briefly discussed to illustrate the concept. Zimmer and Unland [377] propose a metamodel for modelling complex events in active database systems, and some of the operators defined by their metamodel are listed in Table 4.1. These operators indicate some of the potential interactions that may occur between two distinct event instances.

Operator	Summary
$ei_1 ; ei_2$	<i>Sequence</i> : instances have to occur in the given order.
$ei_1 == ei_2$	<i>Simultaneous</i> : instances have to occur simultaneously.
$ei_1 \wedge ei_2$	<i>Conjunction</i> : instances have to occur, in any order.
$ei_1 \vee ei_2$	<i>Disjunction</i> : at least one of the instances have to occur, in any order.
$ei_1 \neg ei_2$	<i>Negation</i> : the specified events cannot occur within the specified instance period.

Table 4.1: Selected event algebra operators for composing complex events, adapted from an event metamodel proposed by Zimmer and Unland [377]

```
// Introduced in DOM Level 2:
interface MouseEvent : UIEvent {
    readonly attribute long        screenX;
    readonly attribute long        screenY;
    readonly attribute long        clientX;
    readonly attribute long        clientY;
    readonly attribute boolean     ctrlKey;
    readonly attribute boolean     shiftKey;
    readonly attribute boolean     altKey;
    readonly attribute boolean     metaKey;
    readonly attribute unsigned short button;
    readonly attribute EventTarget relatedTarget;
    void                        initMouseEvent(...);
};
```

Listing 3: The IDL definition of the MouseEvent interface, adapted from the DOM Events specification [334]

### 4.2.6 DOM Events

As part of the definition of HTML, the W3C defined an events model known as DOM Events<sup>3</sup> which specifies a range of event interfaces on many different web browser elements. An event may be *bubbleable*, in that the event will “bubble” up the element hierarchy and can be intercepted by any of its parents; and an event may be *cancelable*, in that the default behaviour of the event can be overridden by an event handler [334].

For example, the DOM event *click* is defined for “most elements” in the HTML 4.01 definition, and occurs “when the pointing device button is clicked over an element” [333]. This event is bubbleable and cancelable, and occurs after the *mousedown* and *mouseup* events have been fired. This event can therefore be used to intercept a user clicking on a hyperlink, and cancelled to prevent the web browser normally navigating to the hyperlink destination.

When a DOM event is triggered, an event object instance is also created and provided to an event handler with specific information about the event. For example, the *click* event creates an instance of the MouseEvent interface, which provides attributes such as the position of the mouse on the screen when the click occurred; the state of keyboard modifier keys, such as `shift` and `ctrl`; and the button that triggered the click. The IDL definition [248] of this event interface is provided here in Listing 3.

<sup>3</sup>DOM events are also named *DOM Level 2 Events*, to distinguish the event model from a deprecated event model not formally defined by the W3C known as *DOM Level 0* [334].

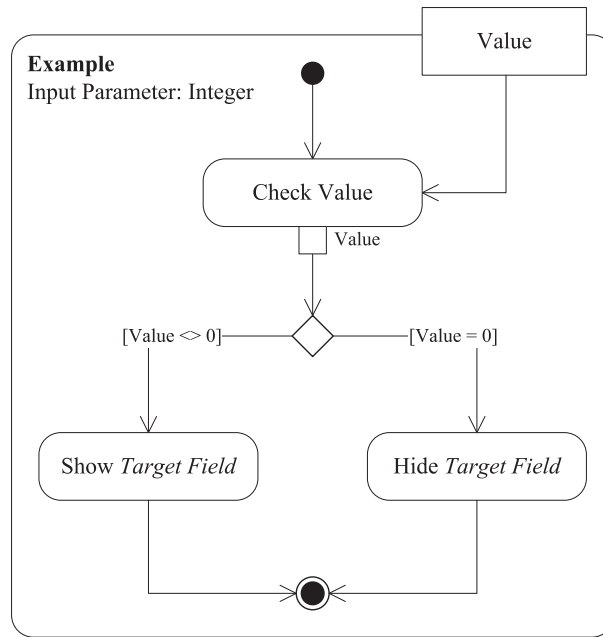


Figure 4.8: Operation modelling using UML activity diagrams [254]

### 4.3 Operation Modelling

Whilst events and actions refer to different types of activities, *operation modelling* refers to the modelling of the lower-level behaviours that make up these activities. This type of modelling emphasises the execution flow, data flow and conditions in these behaviours. A well-defined model-driven development approach can allow a modelling language to support multiple forms of operation modelling; for example, an activity could be modelled in a visual language, or textually in a general-purpose programming language. In this section, two visual operation modelling languages will be briefly discussed.

#### 4.3.1 UML Activity Diagrams

A UML activity diagram [254, pg. 295–417] models behaviours of different aspects of a system at a lower level than other UML diagrams [28], and “emphasizes the sequence and conditions for coordinating lower-level behaviors, rather than which classifiers own those behaviours. These are commonly called control and object flow models” [254, pg. 295]. A sample UML activity diagram is included in Figure 4.8; this activity implements the operation of changing the visibility of a particular user interface element, based on the value of an input parameter.

The UML activity diagram notation does not support primitive data operations, such as arithmetic<sup>4</sup>; this type of behaviour must either be defined informally, or by instantiating an extension within a UML Profile. UML activity diagrams can model a significant range of behaviours, but it is not possible to consistently compile these diagrams into code, because UML does not have a defined execution semantics (as discussed earlier in Section 2.4.2).

The specification of UML proposes that formal execution semantics may be provided to the UML activity diagram by translating diagrams into UML statecharts [254]; however, such a translation is

<sup>4</sup>The UML specification does not discuss why arithmetic is not supported. Perhaps this is due to the language design goal of staying platform-independent, or perhaps it is intended that expression languages such as OCL should be used.

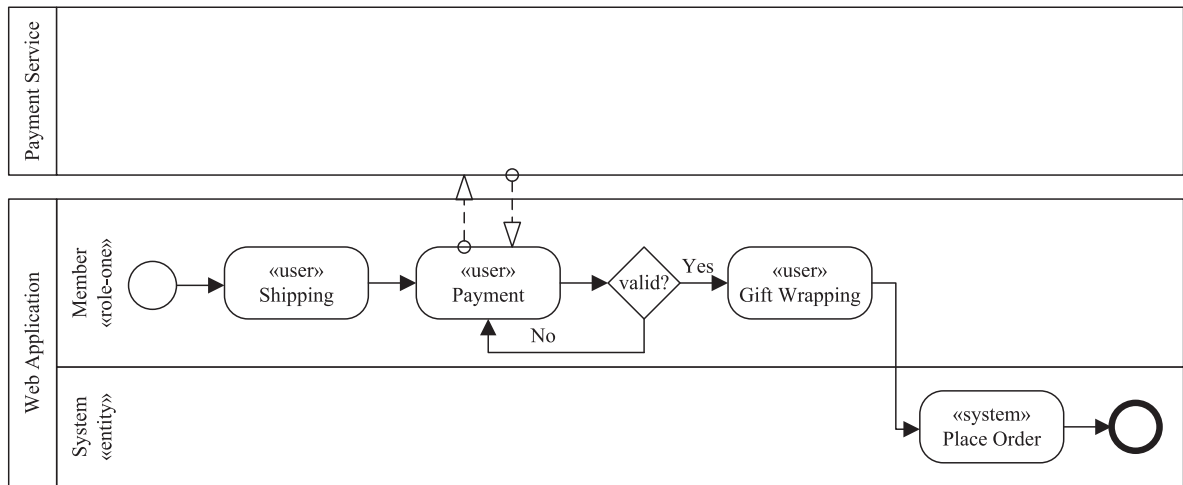


Figure 4.9: Operation modelling of an online shopping business process using BPMN, adapted from Torres et al. [324]

not provided in the latest edition of the UML specification, and Eshuis and Wieringa [89] argue that this process is inherently flawed as activity diagrams are too expressive. Eshuis and Wieringa instead translate activity diagrams into *workflows* in order to provide execution semantics to UML activity diagrams [89].

### 4.3.2 Business Process Modelling Notation

The *Business Process Modelling Notation* (BPMN), defined and specified by the OMG [257], is a visual modelling language for graphically representing the higher-level behaviours involved in business processes, with a notation that is intended to be immediately understandable by all business users. In terms of common business workflow events, the expressibility of BPMN has been found to be similar to UML activity diagrams by White [361], however it is not clear whether the two notations are fully equivalent in terms of expressibility.

Similarly to UML activity diagrams, the formal semantics of the behaviour of BPMN model instances have not yet been defined. Wong and Gibbons [365] argue that the existing approach to translate BPMN instances into Petri nets in order to obtain execution semantics is incomplete. However, BPMN has already been used in the web application modelling domain to implement workflows, as discussed by Brambilla et al. [42].

A sample BPMN diagram of the business process behind a simple online shopping web application is illustrated in Figure 4.9, adapted from Torres et al. [324]. This diagram illustrates the responsibilities of each entity within the process and the communications between them, and also highlights how similar the BPMN visual notation is to the UML activity diagram notation.

### 4.3.3 Predicate Modelling

In both UML and BPMN, predicates are often written textually, as illustrated earlier in Figures 4.8 and 4.9. While this simplifies the model instance, it also makes it difficult to translate the predicate into executable code. However, predicate modelling can be considered a form of operation modelling, on the restriction that the operations must return either **true** or **false**; a formal distinction between operations and predicates is provided later in Section 5.4.1.

This implies that operation modelling techniques – such as UML activity diagrams, or BPMN – may be used to model predicates themselves. However in some situations, the additional complexity of designing a simple expression in a visual modelling language may not be desirable. Embeddable textual expression languages, such as OCL [252] and XPath [347], may be used to simplify the instantiation of simple conditional expressions. For example, the UML specification permits the definition of conditional constraints using both English language and OCL [254, pg. 58].

## 4.4 Type Systems

Being able to model the definition, access and creation of data is essential for a web application modelling language, and is the basis of all web modelling languages (termed *data-intensive modelling languages* by Ceri et al. [51]) such as WebML [50]. It is still debated whether programming languages should be typed; while they potentially increase the workload of a developer, a well-designed type system can capture many routine programming errors before the system is deployed [47].

A particular type system can possess a variety of features and designs; for example, the system can be typed or untyped; provide static, or dynamic type checking; and support safe or unsafe typing. These type system definitions are discussed in detail by Cardelli [47], and will be briefly summarised:

- **Dynamically checked language:** “A language where good behaviour is enforced during execution.” This often means that an object can have many different types over its lifetime.
- **Statically checked language:** “A language where good behaviour is determined before execution.” This often means that an object can only ever have *one* type.
- **Typed language:** “A language with an associated (static) type system, whether or not types are part of the syntax.”
- **Strongly checked language:** “A language where no forbidden errors can occur at run time (depending on the definition of forbidden error).” Strongly checked languages prevent the implicit conversion of one type instance to another.
- **Weakly checked language:** “A language that is statically checked but provides no clear guarantee of absence of execution errors.” Weakly checked languages permit the implicit conversion of one type instance to another.

### 4.4.1 Primitive Types

In a type system, a *primitive type* refers to the smallest datatype that can be expressed, and may also be termed an *atomic type* [339]. In other words, the type can not be expressed as the composition or restriction of any other types in the type system<sup>5</sup>. These primitive types are often then used in the composition of more complex types, and instances are often immutable. Because they refer to the smallest expressible type of data, all languages which possess a type system will have at least one primitive type.

<sup>5</sup>This constraint is discussed in the XML Schema specification: “primitive datatypes are those that are not defined in terms of other datatypes; they exist *ab initio*” [339].



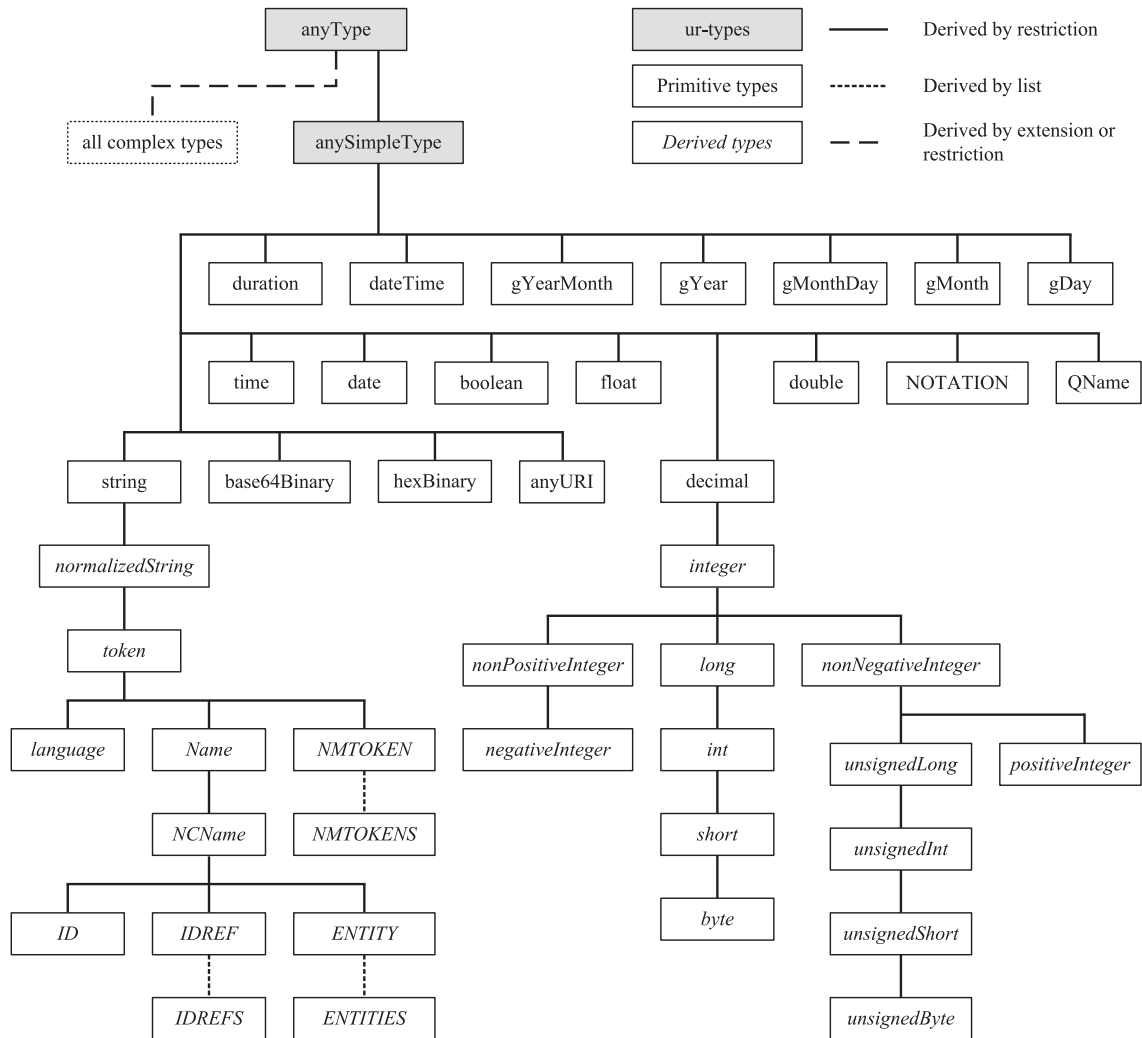


Figure 4.10: The built-in datatype hierarchy of XML Schema Datatypes, adapted from W3C Group [339]

#### 4.4.2 Derived Types

Primitive types can be used to define *derived types* through a number of mechanisms, including composition, restriction, or extension. These mechanisms may be defined formally within the type system, allowing for *type reasoning* to ensure the validity of the typed system [47]. General-purpose programming languages often support restriction-based derived types, such as *subtyping* in object-oriented languages; composition-based derived types, such as *type unions*, are less commonly supported. One notable exception is the Web Ontology Language (OWL), which also supports inferred types through class equivalence [351]. Derived types may be defined separately in type documents (for example, XML Schema) or directly in the system definition itself (for example, Java classes and interfaces).

One technology for representing type systems in XML documents is *XML Schema* [338]. XML Schema allows the formal definition of the syntax and semantics of a type system, which will be used to construct and verify instances of XML documents. This includes the definition of derived types based on existing types in XML Schema. For example, an *email* datatype can be naïvely defined as the composition of two strings, *identifier* and *host*; i.e., *identifier@host*<sup>6</sup>.

<sup>6</sup>The definition of a valid e-mail address is much more complex according to its specification in RFC 5322 [283], as



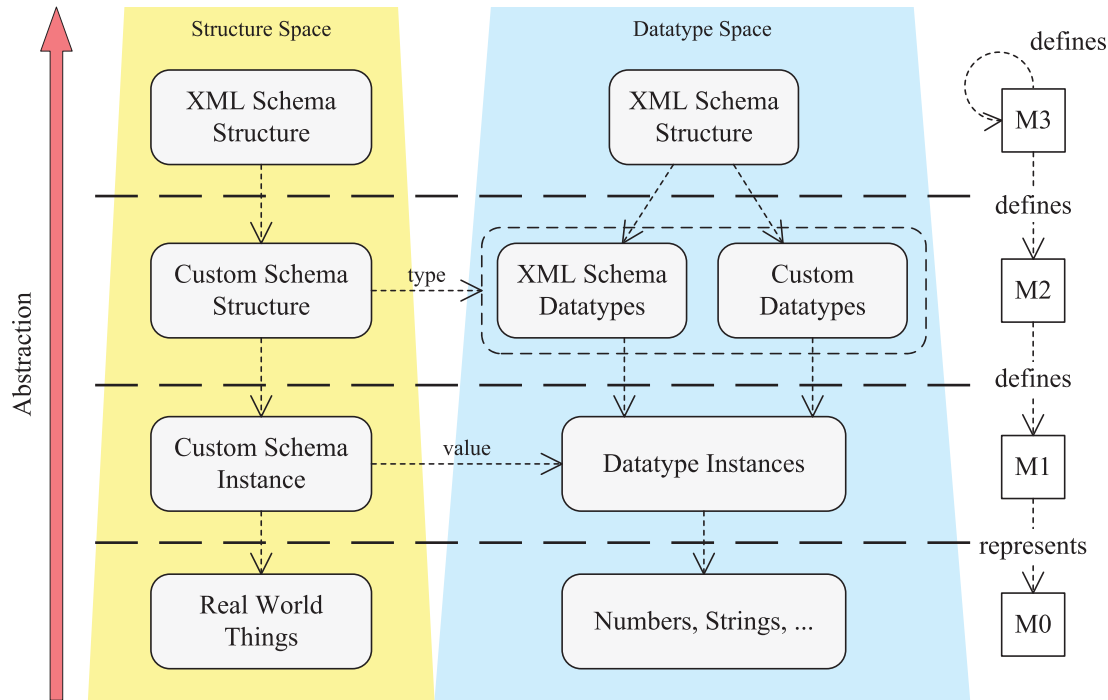


Figure 4.11: Representing XML Schema Datatypes within the metamodeling architecture of MDA using modelling spaces

XML Schema also defines the *XML Schema Datatypes* recommendation, which provides a hierarchy of built-in datatypes and derived datatypes through restrictions such as range bounding and regular expressions. This includes the definition of a *ur-type*, which is a built-in datatype with “an unconstrained lexical space, and a value space consisting of the union of the value spaces of all the built-in primitive datatypes” (and their subsequent lists) [338]. The hierarchy of these datatypes is illustrated here in Figure 4.10.

#### 4.4.3 Type Systems in the Model Driven Architecture

Type systems can also be integrated into the metamodeling architecture of the MDA, as introduced earlier in Section 3.1.5. The integration of type systems into the MDA specifically is not discussed in any existing standard, but it is fairly straightforward to adapt the metamodeling architecture of XML Schema Structure [338] and XML Schema Datatypes [339] in this way. The resulting architecture is summarised in Figure 4.11.

In this architecture, a custom type schema is placed in the metamodel layer (M2), and instances of this schema are placed in the model layer (M1). The real world objects that the schema instance represents remain in M0. Importantly, the XML Schema datatypes are also part of the metamodel layer (M2), because the custom schema can refer to these datatypes. Custom datatypes as defined by the schema are also placed in M2. This is because these datatypes can be considered the metamodel for the actual datatype instances in M1.

The concept of modelling spaces [80] as discussed earlier in Section 3.1.6 can be applied to simplify the complexity of this architecture, and to aid in its comprehension. This allows the metamodeling architecture to be split into the two modelling spaces of a *Structure Space* and a *Datatype Space*,

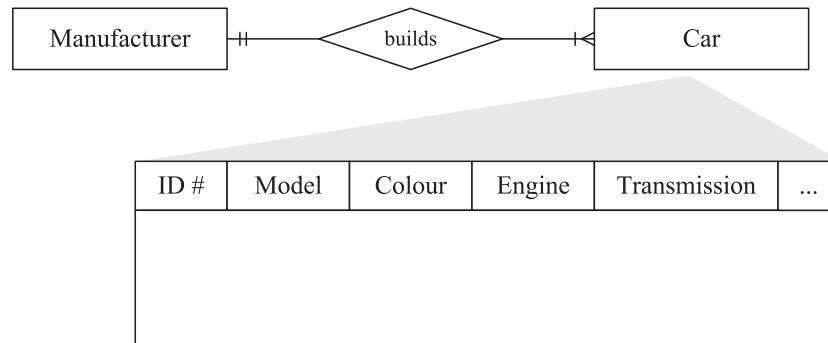


Figure 4.12: A sample ER Diagram and Data Object Table, adapted from Pressman [278]

as illustrated in Figure 4.11. It is important to note that the definition of XML Schema datatypes may simultaneously reside in *both* M2 and M3 at the same time; Djurić et al. [80] argue that this is acceptable for metamodeling architectures.

## 4.5 Domain Modelling

The range of data and datatypes used in a web application is rarely limited to primitive types such as strings and integers. Depending on the web application, the data in the web application needs to be modelled in terms of the *domain* of the application. For example, a shopping web application will likely need to model concepts specific to the domain such as products, taxes, and so on.

The modelling of these domain-specific data structures is known as *domain modelling*, and often involves the extension and composition of primitive types. The definition of a particular structure is known as a *schema*, and is often conceptually separate from the method necessary to access or modify a particular instance of the schema. Instances of schemas can be stored using a variety of technologies, such as relational databases, object-oriented databases, or other structured storage methods<sup>7</sup>.

These techniques discussed in this section are selected from technologies discussed earlier in this thesis; from domain modelling technologies discussed by Pressman [278]; and from document-specific technologies discussed by Nussbaumer and Gaedke [246], and had an associated implementation readily available. For example, ER diagrams are used within the domain modelling approach of WebML [51, pg. 61–76], and XML Schemas may be used to define datatypes within web applications [246, pg. 121–122], and is also used as part of the definition of HTML 5 [343].

### 4.5.1 ER Diagrams

*Entity-Relationship* (ER) diagrams can be used to model relationships between entities; these diagrams were originally proposed by Chen [54] for the design of relational database systems, and have been extended by others [278, pg. 307]. For example, Figure 4.12 shows a system where a *Car* has a relationship to a *Manufacturer* through the relationship *builds*. The relationship edge arrowheads illustrate the modality and cardinality between the relationships.

ER diagrams cannot model the structure of the entities themselves; that is, ER diagrams cannot show that a *Car* has a *colour*. In this case, a *data object table* or schema is often also supplied to complete the model [278]. This data object table is also illustrated in Figure 4.12.

<sup>7</sup>Non-relational databases, known as NoSQL, falls under the “structured storage” category [205].

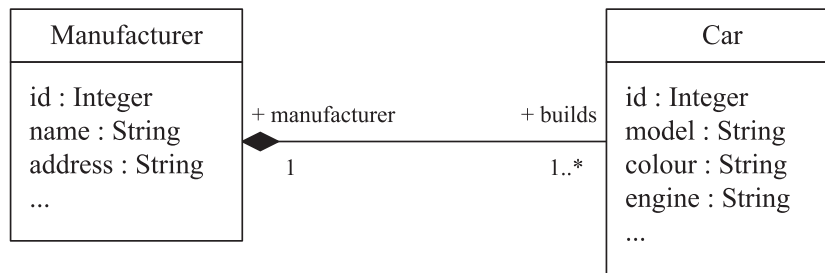


Figure 4.13: A sample UML class diagram [254]

Because ER diagrams only illustrate the relationships between entities, they are good at giving an overall understanding of a system. However, the relationship edge arrowheads can be confusing to people without experience in the ER modelling field, making ER diagrams unsuitable for communicating with non-technical stakeholders. Finally, unlike the other three techniques discussed in this section, ER diagrams do not support the object-oriented concepts of subtyping or type inheritance.

### 4.5.2 UML Class Diagrams

UML class diagrams [254] include both the entity relationship modelling of ER diagrams and the schema definition of data object tables. For example, Figure 4.13 models the same Car/Manufacturer system as the combined ER/data object table example in Figure 4.12. Modelling the structure is optional, and this can simplify the diagram into a format similar to ER diagrams. Because UML class diagrams use text (e.g. “1..\*”) instead of line connectors to mark modality and cardinality, these diagrams can be easier for non-technical stakeholders to understand, yet reduce visual expressiveness [236, pg. 770].

UML class diagrams are also much better suited for modelling object-oriented software systems than ER diagrams, as they support type system concepts such as abstract classes, interfaces and type inheritance. UML class diagrams can also support more advanced types of relationships, such as aggregation, association and composition.

### 4.5.3 XML Schema

As discussed earlier, XML Schema [338] can be used to define derived datatypes through the composition, restriction or extension of other datatypes, and instances are serialised using XML. XML Schema can also be used for domain modelling by defining *complex types*, by defining each entity as an `element` associated with a `complexType` representing the entity’s schema.

However, XML Schema cannot directly model the constraints necessary to match the UML class diagram in Figure 4.13; in particular, it seems difficult (if not impossible) to enforce that both ends of the relationship are correct<sup>8</sup>. It is also difficult to enforce cardinality constraints based on primary or foreign keys; in some situations, this can be achieved by adding regular expressions to the attributes.

An XML Schema is therefore not suitable for domain modelling except in the simplest of circumstances due to its lack of expressiveness. However, it still remains suitable as a first step for validating model instances serialised using XML, if the model instance is loaded using a *schema-validating XML parser* [338].

<sup>8</sup>That is, a car’s manufacturer must also have the same car listed in the list of cars it has manufactured; this constraint could not be expressed in Figure 4.13.

#### 4.5.4 EMF Ecore

As discussed in greater detail later in Section 6.2.1, the Eclipse Modeling Framework (EMF) may be used to define metamodels for model-driven approaches [314]. The underlying meta-metamodel for EMF metamodels is the Ecore metamodel, which is defined in terms of itself. This self-definition allows for Ecore to reside at the M3 layer of the metamodeling architecture of MDA. As discussed earlier in Section 3.1.5, the Eclipse Modeling Framework was built on EMOF [12] and model instances can be serialised directly to EMOF [314, pg. 40].

One issue with the Ecore metamodel is that it only permits a single attribute on a given `EClass` to be the *ID* for the `EClass`; that is, it is not possible to define the concept of a multi-valued primary key, which is a common scenario for relational databases used in the web application domain. This problem is discussed in further detail later in Section 5.6.2.

### 4.6 Design Patterns for Data Access

The previous section described four modelling technologies that defines *what* a domain schema is modelling; however, none of these technologies define *how* the data within a schema can be accessed. This problem is usually left to the model developer to solve, but this thesis will look at proposing a common way of accessing data, in order to improve security and simplify the development of model instances. *Design patterns* formally document a solution to a common design problem in a technology-independent way [118], and two of these design patterns will be briefly discussed here as inspiration for the following chapter.

#### 4.6.1 Database Broker

Bennett et al. [28, pg. 469–474] propose the *Database Broker* design pattern as one form of data access, using a `TBroker` intermediary class which retrieves and stores instances of a particular class `T`. This data access pattern is illustrated by the sequence diagram in Figure 4.14. Bennett et al. use class inheritance on the abstract supertype `TBroker` to define how the data instances are stored; for example, the broker may be defined by the subtype `RelationalBroker` of the in order to access a relational database.

Because the Database Broker can abstract away from the underlying representation of the data storage (e.g. relational database, in memory, etc.), a broker can simplify the maintainability of the system, because different storage representations can be used transparently. However, if a Database Broker can only return one instance from a select, it is necessary to execute many queries to retrieve multiple instances. If a broker can return a list of elements instead, it is easier to retrieve multiple instances, but simply returning a list of elements can be detrimental if there may be many results.

#### 4.6.2 Iterator

The *Iterator* design pattern “provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation” [118, pg. 257–271], and is illustrated in Figure 4.15. That is, an Iterator can be used to access a source of data without needing to understand its storage representation, whether it is a relational database or simply in memory.

The standard Iterator design pattern does not specify that a given Iterator can be applied to many different types, such as through parameterised types (known as *generic types* in the Java language

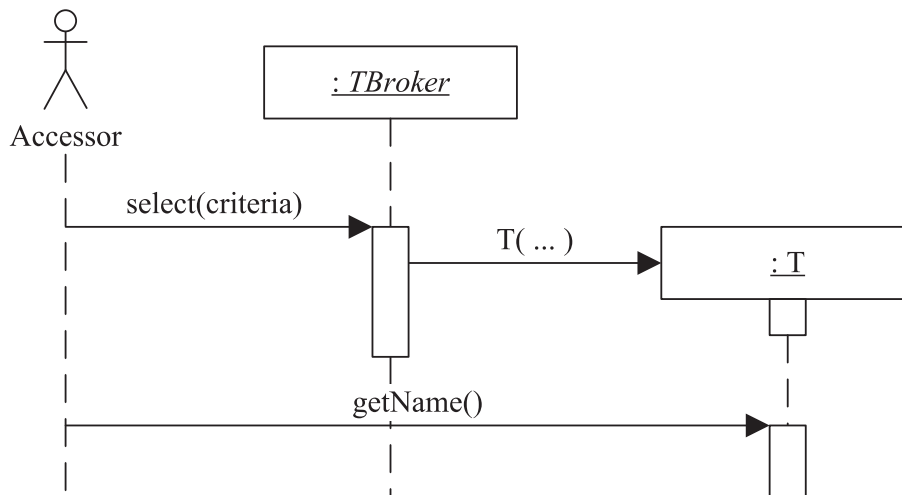


Figure 4.14: The Database Broker design pattern represented as a UML sequence diagram, adapted from Bennett et al. [28, pg. 469]

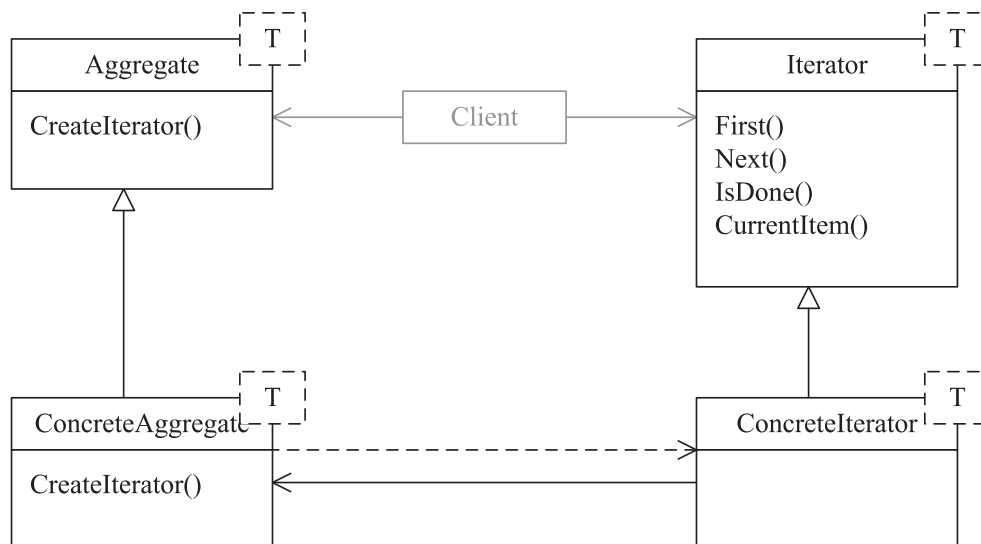


Figure 4.15: The Iterator design pattern using parameterised types, adapted from Gamma et al. [118] and using a UML class diagram syntax [254]

[45]). In Figure 4.15, these parameterised types have been added to the standard Iterator design pattern, represented here using the UML syntax of *template parameters* [254, pg. 633].

Both the Iterator and Database Broker design patterns include instances of the *Abstract Factory* design pattern [118, pg. 258], which can improve the maintainability of the designed system as the underlying source of the data can be abstracted away. The current state of the Iterator may be stored as a *cursor* to permit the same access algorithm to be used across different sources of data [118, pg. 261]. Importantly, an Iterator design pattern may also improve the scalability of the system; because an Iterator can fetch results “on-the-fly”, computationally expensive methods can be deferred until the result is actually necessary [77].

One drawback of the Iterator design pattern is that each Iterator may keep a connection open to the original source of the data until the Iterator is finished or destroyed; in web applications, this is particularly important if the original source is a database. In this case, it is important to design the Iterator as *closeable*, so the Iterator may release resources back to the system [118, pg. 266].

Method	Annotation	Summary
n/a	@BeforeClass	Initialises the test state before a class containing unit tests are executed.
setUp()	@Before	Initialises the test state before a particular unit test.
testXXX()	@Test	Executes a particular unit test.
tearDown()	@After	Cleans up after a unit test has completed.
n/a	@AfterClass	Cleans up after a class containing unit tests has completed.

Table 4.2: JUnit lifecycle methods and annotations [24]

## 4.7 Lifecycle Modelling

As discussed earlier in Section 2.3.1 by Wright and Dietrich [368], RIA development may be simplified by supporting the concept of lifecycle management. In this thesis, lifecycle modelling does not refer to a software engineering lifecycle, which is the more common usage<sup>9</sup>; but rather refers to the lifecycle of a particular component, scope or application in a software system.

In particular, lifecycle modelling can be considered as the modelling of the different *states* of a component; the *transitions* between these states; and integrating these transitions to fire *transition events* when a state transition has occurred. To illustrate how lifecycle modelling may improve the effectiveness of modelling certain software systems, three existing lifecycle implementations will be briefly discussed in this section as examples.

### 4.7.1 Implementation-Level Examples

#### JUnit

JUnit is a popular framework for developing unit test cases for Java software [15], and is also discussed in further detail in Section 7.9. To assist writing tests, JUnit supports a number of lifecycle methods and annotations, as illustrated in Table 4.2. These lifecycle methods and annotations permit the unit test case developer to extract common functionality, reducing code duplication which would negatively impact the maintainability and understandability of test suites.

#### OSGi

OSGi is a dynamic component platform for Java, where applications and components (known as *bundles*) may be added, updated, installed or remove at run-time [318]. The OSGi specification provides a component lifecycle for each bundle, represented here as a state diagram in Figure 4.16. The lifecycle events as part of a particular bundle can be listened to by other bundles<sup>10</sup> in order to perform complex bundle lifecycle management.

<sup>9</sup>For example, Pressman [278] describes the linear sequential and evolutionary models as *lifecycles*, which are instead defined in this thesis as *software process models* in Section 2.7.2.

<sup>10</sup>In OSGi, this is achieved through the BundleListener interface.

## Potential Lifecycles of Rich Internet Applications

As part of the survey of existing modelling languages for RIAs, Wright and Dietrich [368] discuss a range of potential lifecycles as part of a Rich Internet Application. They do not specify a modelling approach or the actual events that may occur, but provide an overview of the types of lifecycles where modelling may be necessary. A summary of these potential lifecycles are reproduced here in Table 4.3.

## UML State Diagrams

UML supports modelling of lifecycle events explicitly through the UML state diagram [254, pg. 523–584]. Every *Transition* from one state to another can specify a *Behavior* which will be performed when the transition fires, and domain-specific events may be described using *Signals*. *Transitions* can also be specified to execute or capture particular events, including the sending or receiving of *signal actions*.

## 4.8 Users and Access Control

As discussed earlier in Section 2.3.1, a *user* represents both people and non-human entities that need to access a particular web application, and it would be beneficial to support user modelling in a RIA modelling language. User modelling is very closely related to access control modelling, as user authentication often requires both authorisation and authentication.

There are many different techniques and methods for describing the roles of users in secure systems, and for describing how resources may be accessed or permissions granted. Depending on the scope and requirements of the modelled system, the access control model can either be very simple, or very detailed. In this section, a number of existing access control methods will be discussed; however, a full discussion on the security properties of each of the following systems is well outside the scope of this thesis.

Because access control is generally an overarching property of the system, the complexity management techniques discussed earlier in Section 4.1 may be applied to access control modelling. For example, aspect-oriented modelling can be used to define system access separately from other system functionality; UWE uses this approach to model access control in web applications by using UML state machines [374]. However, the ISO/IEC 9126 software quality model defines access control as a functional requirement of a software system [166].

As Ceri et al. [52] discuss, although some user information is generic – for example, most web application users have an e-mail address and a password – other information is domain-specific. A modelling language for web applications therefore needs to support profile modelling for specific domains, whilst still supporting common user modelling scenarios. In many ways, user modelling can be considered a form of domain modelling, and this concept is discussed further in Section 5.10.1.

### 4.8.1 Access Control Lists

An *Access Control List* (ACL) is perhaps one of the simplest forms of user access control in computing, and forms the core of many of the other approaches discussed here. ACLs allow an object to be associated with a list of permissions against particular users and groups. When the object needs to



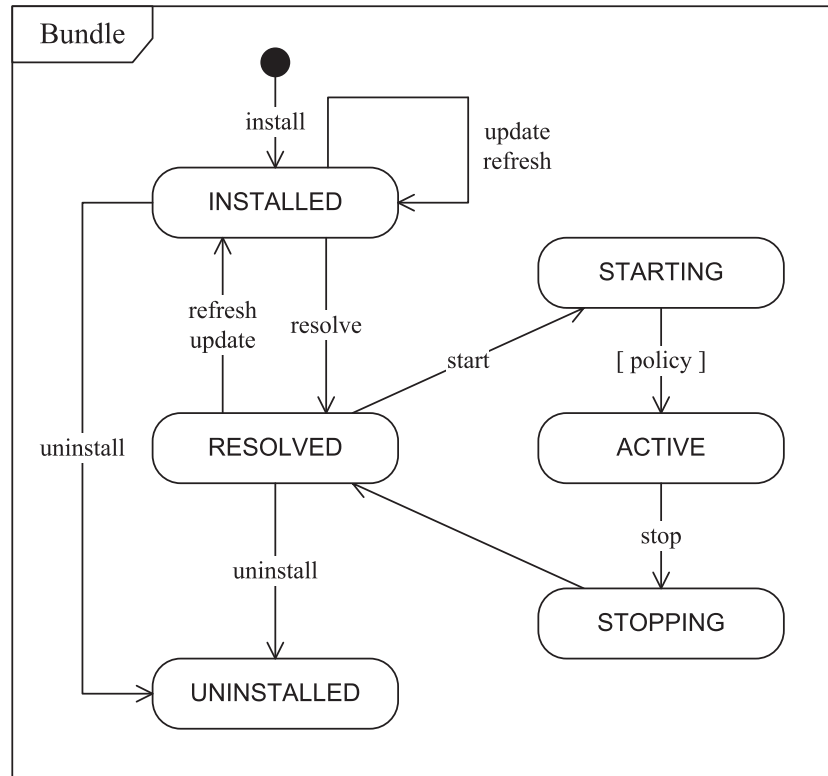


Figure 4.16: Component lifecycle events in OSGi [318], represented using a UML state diagram

Scope	Represents	Example Use
Component	elements in the DOM	Once an interactive map has loaded, move focus to a text box.
Request	a single HTTP request	Transform generated XML to HTML at the end of a request.
Page	many requests over AJAX	Close a previously-opened window when the parent page is closed.
Session	a single web application session [246]	Delete a temporary shopping cart on session timeout.
Login	user authentication over many sessions	Save a temporary shopping cart to the database on logout.
User	the user itself	When a user closes their account, delete all their previously-saved blog entries.
Application	the RIA itself	When an application is initialised, create temporary administrator accounts.

Table 4.3: Possible lifecycle layers of Rich Internet Applications and their potential use, adapted from Wright and Dietrich [368]



	File 1	File 2	File 3	Program 1
Ann	own read write	read write		execute
Bob	read		read write	
Carl		read		execute read

Table 4.4: Example of an Access Matrix, adapted from Samarati and di Vimercati [292]

be accessed by a client, the client is evaluated against the list of permissions provided by the ACL in order to identify its current permissions.

*Traditional Unix permissions* are a simple form of ACLs applied to file systems, where every filesystem object is provided with three permissions (read, write, and execute) for three types of users (owner, owner's group, and other users). Unix permissions are widely used in computing and in the deployment of web applications. However, Unix permissions can become too simple and limiting when different objects regularly need to be shared with specific people. Since permissions cannot be assigned per-user but only per-group, providing the same permission to two users involves the creation of an entirely new group.

#### 4.8.2 Discretionary Access Control

*Discretionary Access Control* (DAC) policies are based on individual users, and on access rules stating what permissions each user possesses [292]. Every user in the system has a set of permissions against every object in the system, and this set may be empty. These rules are often represented using access matrices as in Table 4.4, but as the matrix is usually sparse, policies are often implemented using ACLs instead [135].

As secure systems often deal with large numbers of both users and objects, DAC policies can quickly become overwhelming. Without a concept of roles or groups, it can be difficult to accurately apply new permissions or objects to a common group of users. However since permissions are defined explicitly and per-user, it is not possible to override any of these permissions from elsewhere in the system, improving the inherent security of the approach.

#### 4.8.3 Mandatory Access Control

*Mandatory Access Control* (MAC) policies are based on mandated regulations, determined by a central authority [292]. Groenewegen and Visser [135] explain how this policy can be implemented on assigning labels (e.g. *TopSecret*, *Secret*, *Unclassified*) to objects, and users are given a clearance label that indicates their access level. The relative importance between labels is defined by a partial order, and users can only create objects at a level they can access. The central authority regulations are then used to determine access permissions for a particular user against a particular object.

As Groenewegen and Visser argue [135], MAC policies are mainly aimed at preserving confidentiality of information, by preventing the unsafe transfer of information contained within objects to other security labels. The use of a MAC-based system is made easier if labels can override lower-level

labels, but the resulting partial order between labels can cause problems if label order is not evaluated as a whole.

#### 4.8.4 Role-based Access Control

The *Role-Based Access Control* model (RBAC), as described by Sandhu et al. [293] and proposed as a NIST standard [95], allows a great deal of flexibility in defining access and permissions. RBAC is essentially an extension of ACLs with the concepts of roles, in order to support the large-scale management of permissions and roles. A set of roles are defined, and each role provides particular permissions; these roles can then be globally applied to individual users, or temporarily applied within a session.

Compared with RBAC, MAC does not support individual permissions, however both models support a partial ordering on a hierarchy of roles. Samarati and di Vimercati [292] argue that as RBAC is more expressive, it is attracting increasing attention as an alternative access control model to the traditional DAC and MAC models. Groenewegen and Visser argue that the main benefits of the RBAC model include that role assignments are separated from users and permissions; many applications naturally consist of a hierarchy of roles; and a user can activate the minimum role able to perform a task, increasing security [135, pg. 9].

### 4.9 Describing Reusable Patterns within a Metamodel

When designing many different systems within a particular domain, there are often a number of common patterns of functionality that emerge. As an example in the RIA domain, it is a common requirement to keep the values of two objects synchronised; for example, keeping a form on the client synchronised with its stored representation on a server<sup>11</sup>. A RIA modelling language could therefore provide some way of reusing these common patterns, named *reusable patterns* in this thesis<sup>12</sup>.

#### 4.9.1 Independence of the Reusable Pattern Metamodel

One option for modelling reusable patterns is to integrate the patterns directly into the same metamodel; for example, a `Text Field` could have a reference “`synchronised with`”, which references another `Text Field` in the model. This approach is useful if there is a predetermined set of potential patterns; however, as supported patterns are directly encoded into the metamodel, it would be difficult for a third party to create new patterns, as the metamodel must be modified directly.

Alternatively, these patterns could be supported using aspect-oriented modelling, discussed earlier in Section 4.1.3. Here, pattern instances are defined as an instance of an entirely separate metamodel (a “patterns” metamodel), separating the additional functionality of the patterns from the base model itself. With this approach, it is easy for third parties to extend or create new patterns using this approach, and the set of available patterns does not need to be restricted.

<sup>11</sup>This design pattern is described as a *Sync Wire* later in Section 5.9.1.

<sup>12</sup>This nomenclature is provided to distinguish these types of patterns from the concept of architectural software *design patterns* as discussed by Gamma et al. [118].

### 4.9.2 Implementation of the Reusable Pattern Metamodel

Once a metamodel can support the instantiation of common reusable patterns as part of a model instance, it is also necessary to describe how the pattern will actually be implemented. With respect to the “synchronised with” reference discussed above, this would involve the creation of methods, events and any other related functionality necessary to implement the intent of the reusable pattern.

One option would be to implement the reusable pattern logic as part of the model transformations – or code generation templates – used in the model-driven development process. This would mean that the reusable pattern must be integrated into the model transformation, making it difficult for third parties to extend these patterns, or to create new patterns. This approach would also make it difficult for a model developer to override the functionality of a reusable pattern, as the pattern has essentially become an intent of the metamodel itself.

Alternatively, if the source metamodel already supports the manual definition of the reusable pattern, then *model completion* could be used to implement the reusable pattern within the same model instance. This approach leaves the model transformation stage intact, as it moves the pattern logic into the model completion stage, as discussed in Section 3.2. This can also be considered a form of aspect-oriented modelling as the reusable pattern logic is removed from both the metamodel and the implementation into a separate completion process.

Both of these approaches could be simplified by using additional abstraction layers in the language implementation. That is, an intermediary metamodel can be used to make the source and target metamodels smaller, at the expense of additional steps in the modelling environment. As discussed earlier in Section 2.4.3 this is the approach taken by UWE, where each metamodel instance represents a particular aspect of the intended design [190].

If both the design and implementation of reusable pattern modelling support is achieved in an aspect-oriented modelling way, then the reusable pattern could be removed entirely from the base metamodel and considered an independent extension. The benefit of this approach is that the metamodel and model transformations are not polluted by a range of reusable patterns, and would also improve third-party interaction. However, the drawbacks of aspect-oriented modelling as discussed earlier – such as understanding the interaction between two aspects – remains a concern.

## 4.10 Conclusion

The development of a modelling language for RIAs requires support for modelling many domain concepts – such as events, types and lifecycles – and each of these modelling concepts may be implemented using a number of existing approaches. In this chapter, a variety of approaches for each of these modelling concepts have been evaluated and discussed, along with comparisons to existing work where relevant. In the next chapter, these design concepts will be integrated into a new modelling language for RIAs.

## Chapter 5

# The Internet Application Modelling Language

This chapter will provide the definition of the *Internet Application Modelling Language* (IAML), a modelling language for Rich Internet Applications. It will informally discuss the design and behavioural aspects of the language, the rationale behind each design decision, and illustrate example model instances. The complete definition of IAML – including the structural syntax, concrete visual notation, inference rules, informally-defined behaviours and implementation guidelines for each language element – is provided in Appendix I. Outstanding future work on the proof-of-concept implementation are listed as future issues<sup>1</sup>, as discussed at the beginning of Chapter 7.

Throughout this chapter excerpts of an example web application, *Ticketiaml*, will be provided to illustrate the use of certain IAML metamodel elements. As described later, this application is the IAML-based implementation of the *Ticket 2.0* benchmarking application, and its full definition is provided later in Section 8.3.2, and represented using XMI in Appendix F.

### 5.1 Applying Software Engineering Principles

Earlier in Chapter 2, this thesis discussed the requirements of RIA modelling languages and how no existing language could completely model all of the requirements of RIAs. This section will now fully define the scope for this research, and define the design approach for this language, which will be named the *Internet Application Modelling Language* to reflect the intended final scope of the language.

#### 5.1.1 Requirements Planning

Once the requirements for a RIA modelling language had been identified earlier in Section 2.3.2, it became clear that the resources necessary to implement such a language together with a validating proof-of-concept implementation far exceeded the resources available in a single Ph.D. The target set of requirements will therefore be simplified to a smaller set by omitting rarely-used features of RIAs.

After investigating our list of 59 modelling requirements, we found that some of these requirements could be characterised as describing the system complexity, distributed functionality, reliability and performance requirements of enterprise-level applications. These requirements were focused on describing large RIAs, rather than the features that made RIAs considerably different from other web

---

<sup>1</sup>For example, the outstanding Issue 281: *Refactor model elements into documented packages*.

Category	Requirements
Offline data	D7, D8
Server transactions	E9, E10
Multiple interfaces	U1, U9
Plugins and dynamic scripting	A6, A8, A9
Internationalisation	T4
Multiple domain support	T5
User collaboration	E7
Import external libraries	U11

Table 5.1: Additional Requirements for *Full RIAs*

applications. By removing these requirements, RIAs may be classified into two types based on their characteristics: *Full RIAs* and *Basic RIAs*.

For example, the offline application requirements of *Offline Data* and *Offline Resources* are considered in this thesis as instances of enterprise application requirements, since they improve web application availability, and most RIAs do not yet support these features natively<sup>2</sup>. Likewise, internationalisation and providing multiple user interfaces may be considered enterprise-level requirements for complex websites [101, pg. 344].

The requirements set of a *Basic RIA* can be used to describe and construct the vast majority of Rich Internet Applications on the Internet, such as simple mashups, personal web sites, and small online shopping sites. Larger enterprise-level applications such as web-based e-mail and social networking platforms can instead be described as a *Full RIA*. A *Full RIA* has the modelling requirements of a *Basic RIA* with the additional thirteen requirements in Table 5.1; this reduces the feature scope for modelling Basic RIAs from 59 requirements to a more reasonable 46 requirements (a 22% decrease).

This requirements restriction is not detrimental to the overall contributions of this thesis; as discussed by Wright and Dietrich [368], no existing approach can model either *Basic* or *Full RIAs*, so the proposal and implementation of a modelling language for *Basic RIAs* will still be a significant research contribution. Nevertheless the language will be designed to support *Full RIAs* in the future where possible, to allow the implementation of these missing requirements as future work.

### 5.1.2 Modelling Language Development Approach

As discussed earlier in Section 2.7, there are three methods of developing a language: the extension, or restriction of an existing language; or an abstraction into a new language. The approach taken depends on how well existing languages match our intended domain. However, since no existing language maps well onto RIA concepts, and many RIA concepts do not easily map into existing languages such as UML, the extension or restriction of an existing language is not a desirable approach.

This research proposes abstracting existing RIA platforms into a *new* domain-specific language, but also reusing concepts and semantics from established languages. This approach is advocated by Kelly and Pohjonen [179], who argue that it is a good idea to reuse basic ideas and concepts such as data flow, control flow and type inheritance in the development of new modelling languages.

For example, UML defines UML class diagrams [254] which are generally accepted by industry

<sup>2</sup>For example, at the time of writing only a few Google applications support offline mode, and these require the installation of browser plugins or setting web browser organisational policies [70].

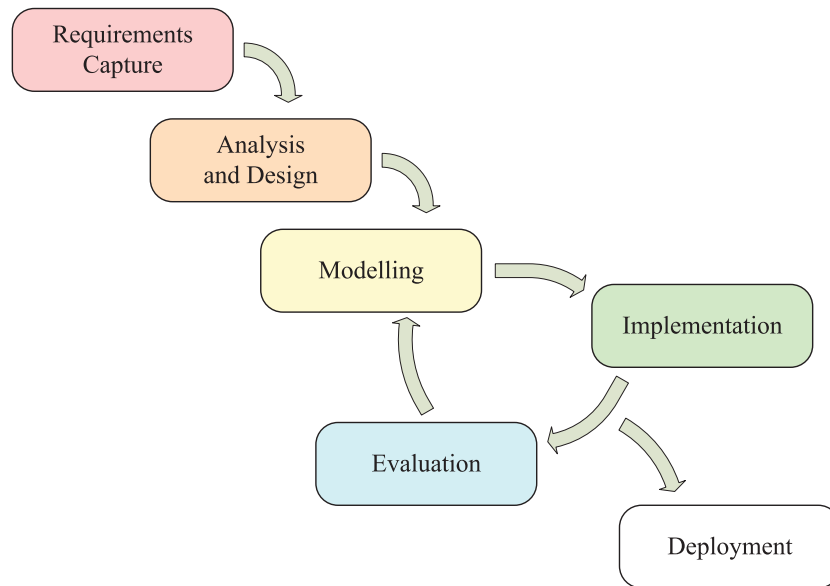


Figure 5.1: The hybrid modelling language process model used in the development of IAML

and extensively used to model the schema and structure of object-oriented systems [28]. The concepts and semantics of UML class diagrams could therefore be reused and adapted to modelling schemas or objects within RIAs, without the language being burdened by having to implement the entire UML specification<sup>3</sup>.

### 5.1.3 Software Process Model

This research will attempt to capture the benefits of both the linear sequential and evolutionary software process models, discussed earlier in Section 2.7.2. Since neither approach is purely suitable for research as discussed earlier, a hybrid approach may be beneficial, as illustrated in Figure 5.1. This approach is very similar to the *MDA software development process* advocated by Kleppe et al. [185]. The only difference is that as a research project, the analysis and design is done in a sequential manner.

In this hybrid approach, the *requirements capture* and *analysis and design* steps will be performed sequentially, to ensure that all of the desired requirements will be captured. This hybrid approach also includes the development of a proof-of-concept implementation of the language; as discussed earlier in Section 2.3.1, this implementation can be used as a reference implementation, and increase acceptance of the language within development communities [102].

The development, implementation and validation steps will be performed iteratively and in an evolutionary manner. This will ensure that this research will at the very least result in a fully validated partial language, rather than a complete language that cannot be fully validated. To improve the quality of the software implementation, some of the concepts of test-driven development discussed earlier in Section 2.7.3 will also be incorporated into the implementation process.

<sup>3</sup>The OMG has recognised that the size and complexity of the UML specification is significant; the focus of the upcoming UML 2.5 specification is to “greatly simplify the specification itself” to reduce architectural debt and enable additional improvements [60].

### 5.1.4 Evaluation

As discussed earlier in Section 1.3, the research method underlying this thesis proposes a number of methods in which to evaluate the proposed modelling language. In Chapter 8, five different evaluations will be performed:

1. *Feature Comparison Evaluation*: Earlier in Section 2.4.7, a feature comparison was performed to compare existing modelling languages for web applications against RIA features described by Wright and Dietrich [368]. These requirements will therefore be re-evaluated against the completed IAML metamodel, to compare the functionality of the proposed IAML metamodel with these existing languages.
2. *Modelling Requirements Evaluation*: The list of detailed modelling requirements for RIAs [367], discussed earlier in Section 2.3.2, will be evaluated against the proposed language in a requirement-matching evaluation. This comparison will identify any language requirements of *Basic RIAs* that are not yet supported.
3. *Benchmarking Application Evaluation*: The implementation of the proposed RIA benchmarking application, *Ticket 2.0* [367], will be used to illustrate that the language can be used to implement real-world applications, and that the modelling language is not excessively complex. In order to highlight any differences in development effort, the system metrics defined earlier in Section 2.6.3 will be used to compare the IAML-based implementation with the manually implemented benchmark.
4. *Metamodelling Metrics Evaluation*: The evaluation of the metamodelling metrics defined earlier in Section 2.6.2 will be used to compare the IAML metamodel with other similar metamodels, illustrating if the complexity of the metamodel could be considered unexpected.
5. *Visual Notation Evaluation*: The syntax and implementation of the visual notation of IAML model instances will be evaluated using two approaches discussed in Section 3.4.4. That is, the syntax will be compared by evaluating the information capacity of its notation [236]; and the implementation will be evaluated using the *cognitive dimensions* framework [133].

### 5.1.5 Design Goals

In order to clarify the intent of the proposed modelling language and to improve understanding of the necessary decisions that will be made during its design, it will be helpful to summarise the *design goals* of this new language as follows:

1. The IAML metamodel will fully support the 46 requirements of *Basic RIAs*, with consideration also given to the additional thirteen requirements of *Full RIAs*, as discussed earlier in Section 5.1.1. This design goal is evaluated later in Section 8.2.
2. IAML will reuse concepts from existing modelling languages where appropriate, as discussed earlier in Section 5.1.2. This design goal is evaluated later in Section 8.1.
3. IAML will be supported with a visual notation to improve the end-user effectiveness of designing model instances, as discussed earlier in Section 3.4. The implementation of this design goal is covered later in Sections 5.15 and 7.4.3.



4. The proof-of-concept implementation of IAML will be implemented iteratively in an evolutionary manner, and the metamodel will be frequently refactored as appropriate, as discussed earlier in Section 5.1.3. This implementation is introduced in greater depth in Chapter 7.
5. IAML and its proof-of-concept implementation will be published under an open source license to encourage use within industry, seek community feedback and to reduce defects, as discussed earlier in Section 2.7.4. This design goal is discussed in greater depth later in Section 7.1.1.
6. During its development, IAML will be frequently evaluated according to the evaluation criteria discussed earlier in Section 5.1.4, in order to guide its development in an evolutionary manner. The results of these evaluations are provided later in Chapter 8.
7. The implementation of IAML will be supported by a suite of model instance verification tools that can formally verify different design and correctness properties of modelled web applications. This goal was discussed earlier in Section 1.5, defined in further detail in Section 3.3, and evaluated later in Section 7.7.

## 5.2 Metamodel Design Principles

Before the IAML metamodel is discussed, it is important to identify the principles used to answer design decisions that will arise during the development of the language, along with techniques for evaluating these principles.

### 5.2.1 Addressing Security Risks through Modelling Language Design

As discussed earlier in Section 1.3, this thesis will focus on the development of a modelling language that increases the security of subsequently modelled applications. The design of IAML will therefore focus on providing an interface that encourages the development of secure web applications. In many cases this may be done through restriction of the underlying interface; by reducing the number of possible attack vectors in a system, security may be increased.

For example, the preferred option to prevent SQL injection in a web application is through the use of a safe parameterised API [319, pg. 7]. Some web application frameworks now only permit database access through such a safe API, such as the Propel database abstraction layer within the Symfony framework [274]. This does not make it impossible to execute unsafe SQL, but does make it much more difficult for a developer to unintentionally introduce this vulnerability into a web application.

### 5.2.2 Language Design Principles

The principles of design for a modelling language can be adapted from the principles of designing programming languages. For example, Hoare [156] introduces concepts such as simplicity and efficient object code as general design criteria. On a more practical level, Wirth [364] discusses some of the design decisions that need to be considered, such that mathematical formalisms of basic abstractions may identify inconsistencies<sup>4</sup>, and a language is useless without an implementation and documentation.

---

<sup>4</sup>Wirth also highlights how a balance must be achieved between formal and informal documentation; in particular, arguing that a “formal definition cannot be a substitute for an informal presentations and for tutorial material” [364, pg. 29].



Programming language design principles can be adapted to instruct the design of modelling languages; for example, Karsai et al. [177] proposes 26 guidelines on the design of domain-specific languages. While it is desirable for a modelling language to satisfy all of these principles and guidelines, many of these principles conflict with each other [177]. For example, adding language concepts to increase the functionality of a modelling language directly conflicts with a principle of keeping the modelling language simple.

This thesis will select four design principles as particularly important to guide the design of the IAML metamodel, with respect to the research questions proposed at the start of this thesis. In particular, IAML will be developed primarily according to the design principles of *scalability*, *simplicity*, *consistency* and *standards-compliance*.

### Scalability

As discussed by Paige et al. [267, pg. 15], “the principle of *scalability* states that a modelling language should ideally be useful for both small and large systems.” This can be measured by evaluating the modelling environment; for example, different views can be provided under the discretion of the model developer that hide unnecessary detail. The design of IAML to support hierarchical model instances is one way of satisfying this design principle, as discussed in Section 5.2.3.

### Simplicity

Paige et al. describe a *simple* language as one that is “small and memorable” [267]. Karsai et al. suggest many guidelines to achieve simplicity, such as limiting the number of language elements; avoiding unnecessary generality; and avoiding conceptual redundancy [177]. One useful set of metrics to evaluate simplicity should therefore be the metamodeling metrics discussed earlier in Section 2.6.2. In general, a metamodel with a fewer number of model elements should be a simpler metamodel.

Kelly and Pohjonen [179] also argue that developing a domain-specific approach is not about achieving perfection, but about developing an approach that works in practice: “It will always be possible to imagine a case that the language can’t handle.” They advocate concentrating on the core concepts in a domain, and initially building a prototype language for these features. Even languages such as UML are still undergoing evolution; at the time of writing, UML has been through 17 years of iterative development<sup>5</sup>.

### Consistency

Paige et al. describe the principle of *consistency* within modelling languages as that “there is a purpose to the design of the language” and that all features of the language must support this purpose [267, pg. 13], and this definition is used in this thesis. As the purpose of IAML is to support the design and implementation of Rich Internet Applications, this means that every element within the language must have an analogous component in the web application domain. For example, IAML is designed to model RIA functionality, and not the layout of user interfaces; IAML therefore delegates this responsibility to the CSS language [344].

The *consistency* language design principle also encompasses the goal of designing a secure modelling language to prevent the inadvertent introduction of vulnerabilities into a RIA, discussed in the

---

<sup>5</sup>As discussed by Object Management Group [247], the development of UML began in late 1994.

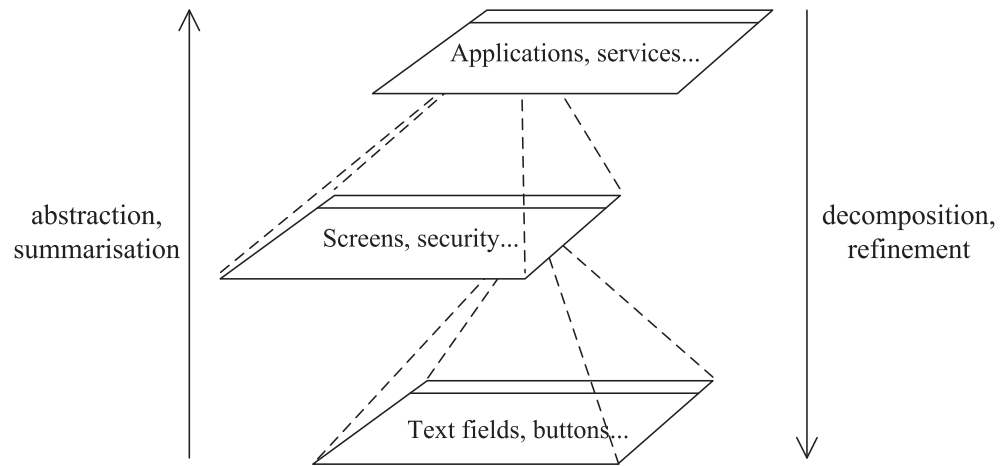


Figure 5.2: Hierarchical Modelling in IAML

previous section. For example, web applications should not support the execution of arbitrary commands on the server, as this introduces security and scalability issues; IAML should therefore never include support for such primitives. Similarly, database access should only be permitted through a safe parameterised API.

The design consistency of a modelling language must not be confused with the consistency of the model instances produced using the language [267], which may be evaluated using model instance verification as discussed earlier in Section 3.3. Similarly, modelling language consistency must not be confused with the consistency of the textual or visual *syntax* of the language, which is covered earlier as part of the cognitive dimensions framework in Section 3.4.4.

Finally, consistency can also be considered in terms of the high-level design of model instances, which is also a different concept to modelling language consistency. Depending on the development environment, design consistency can be addressed through naming patterns, coding conventions [229, pg. 875–902] and reusing architectural design patterns [118].

### Standards-compliance

*Standards-compliance* refers to the degree in which the language is “convenient and compatible with” widely-used specifications and standards [364, pg. 28], and should occur wherever possible either through composition or definition reuse [177]. This design principle is already reflected as a design goal of the language in Section 5.1.2. Importantly, standards-compliance may be at odds with simplicity and consistency principles, and may impact on proof-of-concept implementations; a metamodel which reused the *entire* UML metamodel would be much difficult to implement than one which only reused the UML state diagram metamodel.

### 5.2.3 Hierarchical Modelling Approach

In terms of the three complexity management techniques discussed earlier in Section 4.1, the IAML metamodel is designed to use a hierarchical modelling approach. Such an approach will allow model developers to see a high-level overview of a modelled system, but also zoom into the individual details of a particular model element. Each level of the hierarchy is designed to support a particular domain scope, as illustrated in Figure 5.2.

It is important for each viewed layer to provide appropriate context to the model developer, to orient the current view with the rest of the modelled system. In the IAML metamodel, it is envisaged that the proof-of-concept implementation can be used to provide this context. Hierarchical modelling is also very easy to integrate into the *model completion* framework, discussed earlier in Section 3.2.

A benefit of using a hierarchical modelling approach for modelling RIAs is that it becomes possible to use many visual metaphors at once. For example, the overview of a web application may use a deployment diagram metaphor; individual parts of the application may use a navigation metaphor; and the contents of methods may use an activity diagram metaphor. This technique is used by UML; for example, an overview of a system can be visualised using a *deployment diagram*; this diagram can be decomposed into *class diagrams*; and individual methods of a given class can be decomposed into *activity diagrams* [254]. Similarly, UWE conceptualises web application navigation into *navigation space models* and *navigation structure models* [191].

#### 5.2.4 Guidelines for Metamodel Refactoring

During the development of a modelling language, a number of architectural decisions on the metamodel itself will arise. For example, a set of elements could be refactored into a single element with an enumerated property; conversely, a single element with an enumerated property could be split into a set of related elements.

During the development process of the metamodel for IAML, there were no known guidelines for when to apply such metamodel refactorings. This may be due to the fact that modelling language development is a relatively recent area, and such guidelines often come from experience in the development of modelling languages. However, two guidelines will be proposed that were obtained by adapting existing refactoring guidelines to the metamodeling domain.

##### Extracting Supertypes

If a set of related elements all have common functionality, then the common functionality can be extracted into a supertype for these elements. For example, [Buttons](#) and [Input Text Fields](#) both represent user interface elements that need to be rendered in some fashion; it may be preferable to let these metamodel elements inherit this “renderable” functionality, to improve component reuse and simplify the metamodel development. However, it is important to note that such a metamodel refactoring will increase the number of classes in the metamodel, impacting the *simplicity* language design principle discussed earlier.

In the design of the IAML metamodel, this refactoring was encouraged according to the “three strikes and you refactor” guideline proposed by Fowler [100, pg. 57]; that is, once three metamodel elements share the same common functionality, this functionality can be extracted into a supertype. In terms of the previous example, this involved the creation of a [Visible Thing](#) supertype as discussed later in Section 5.13.2.

##### Merging Elements

This metamodel refactoring process is almost the converse of the *extracting supertypes* process. In this case, if a set of related elements all have the same supertype and there is very little different functionality between each type, then these subtypes can be merged together into a single type, and the differences in functionality represented using an attribute or a property. The number of elements

in the metamodel can therefore be reduced, satisfying the language design principle of *simplicity*, and reducing the mental load on the developer.

However, a merged element is much more difficult for a third party to extend in commonly-available typed systems, as most object-oriented language supports object inheritance through third parties, yet few languages support direct manipulation of an object property. It is also important that the property-based distinction be easily visible, otherwise design-time errors may occur. Finally, any constraints on the original elements must be merged in a consistent way, impacting the simplicity of constraints.

This implies that related elements may be implemented in two conflicting ways. The following guidelines were consequently used in the refactoring process of the IAML metamodel:

1. Is it likely that the original supertype will be extended by a third party? If so, the supertypes should be kept.
2. Are there significant differences in the individual implementations of each supertype? If so, the supertypes should be kept, as model transformation languages such as openArchitectureWare provide better support for exploring inheritance than exploring enumerations of properties.
3. Otherwise, the supertypes should be merged together into a parent type, and a new enumerated property should be provided to select between predefined behaviours, as previously permitted by the subtypes.

## 5.3 Language Overview

The remainder of this chapter will discuss the design of IAML in detail, but in this section, a brief overview of the general modelling approach of IAML will be discussed. In particular, this overview will be supported through the example implementation of a common concept in the development of web applications that deal with user profile data – the ability for a logged-in user to update their profile details, while ensuring that the supplied profile details are syntactically valid.

In Figure 5.3, a partial view of a valid IAML model instance is illustrated. The intent of this model is to provide two *text fields* that will allow a user to modify the e-mail address and country details stored in their existing user profile. The rest of this model will not be discussed in this thesis; for a more complete example model instance, the reader is referred to the implementation of the *Ticket 2.0* benchmarking application in Section 8.3.2. This example model illustrates a number of important IAML concepts:

1. All user interface elements and domain attributes are *typed* (Section 5.5) to define a valid range of values for that element. In this example, both the e-mail profile attribute and the text field used to modify this attribute are typed to `iamlEmail`, meaning that these elements can only ever have a valid e-mail address value, according to the semantics defined in Section 5.5.2.
2. While user interface elements and database values may be updated manually using concepts such as events, conditions, actions and rules (Section 5.4.3), in this model this common scenario of keeping two values synchronised is represented as a [Sync Wire](#) (Section 5.9.1). When this model is generated into code, model completion will add the necessary logic to implement this synchronisation.

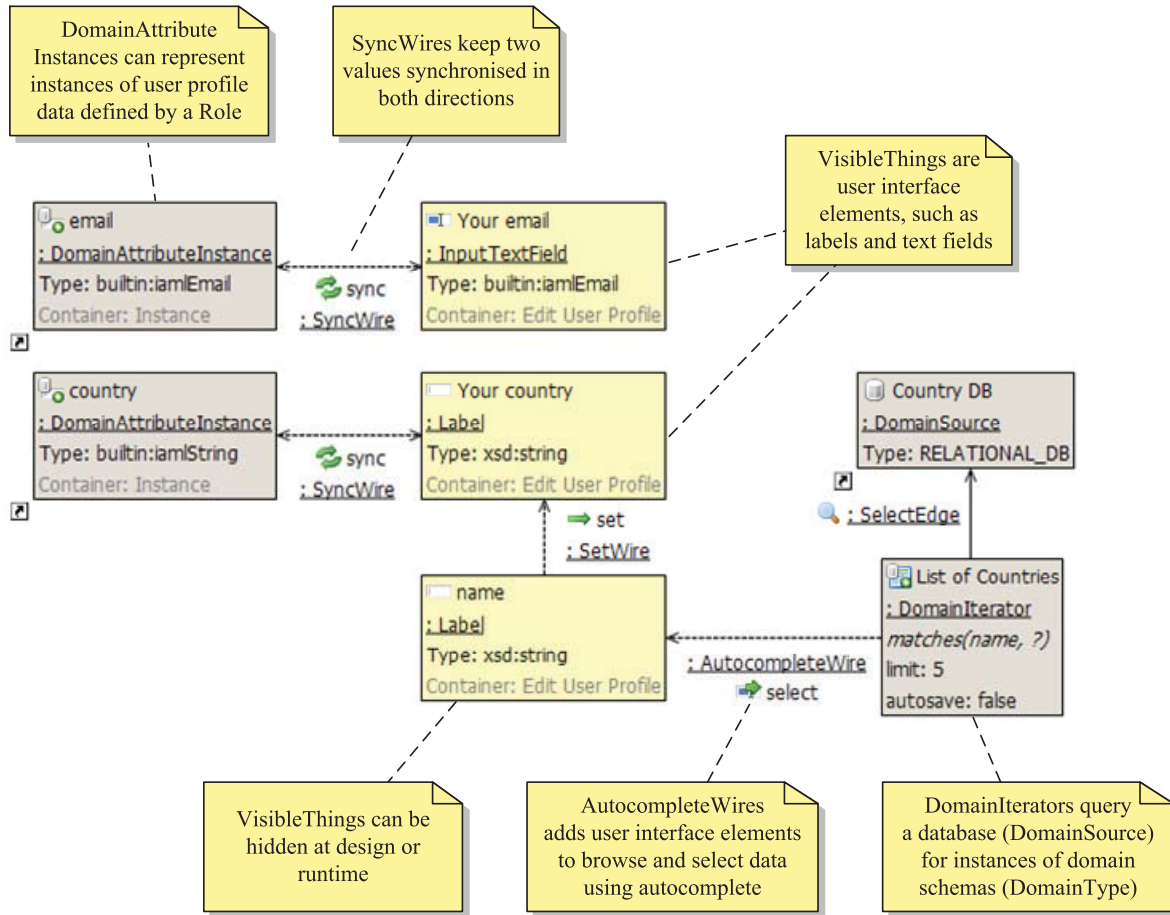


Figure 5.3: An overview of some of the important concepts of modelling RIAs using IAML

## Edit User Profile

Your email

Your country **New Zealand**

### List of Countries

Search by name

New Zealand  
Netherlands  
Nepal

Figure 5.4: The generated implementation of the IAML overview model instance defined in Figure 5.3

3. User interface elements can be hidden either at design-time or run-time. In this example model, the [Label](#) named *name* is defined to be *hidden* at runtime, to prevent the user interface element from being displayed. This property is not reflected in IAML visual notation, and is available through the extended properties of the model element<sup>6</sup>.
4. Once a domain schema has been defined through a [Domain Type](#) (Section 5.6.1), the resulting database may be queried through a [Domain Iterator](#) (Section 5.6.3), which allows the web application to browse over and select valid instances of data. In this example model, a [Domain Iterator](#) is used to select a maximum of five valid country names that match an incoming parameter.
5. Domain schemas may also be used to define user profile attributes through the definition of a [Role](#) (Section 5.10.1), and runtime instances of user profile data is available through [Domain Attribute Instances](#) (Section 5.6.3). In this example model, the current user profile data for *country* and *email* is used to populate the initial value of the *country* and *email* text fields.
6. IAML also supports the definition of complex behaviour often expected within an RIA. In this example model, a user may select their user profile country by selecting it from an existing database of country names using *autocomplete*, and this common functionality may be included by using an [Autocomplete Wire](#) (Section 5.9.4). As per the implementation of [Sync Wires](#), model completion will automatically define the user interface elements and additional logic necessary to implement this autocomplete.

Every model element within a valid IAML model instance – with the exception of the root [Internet Application](#) (Section 5.14), which defines the web application itself – is contained by a parent element through a hierarchical model design. The contents of this example model in particular are contained within a [Frame](#) (Section 5.13.4), representing content that may be accessed and rendered independently by a web browser.

The source code for this web application may then be generated from this IAML model instance, including the client-side functionality for the user interface, and the underlying server-side logic. For this example model, the generated user interface would appear similar to the screenshot illustrated in Figure 5.4. This screenshot illustrates that the user may, at runtime, modify the *email* attribute of their user profile, and also modify the *country* attribute of their user profile using autocomplete against a list of existing countries.

### 5.3.1 Package Overview

To encourage modularity in the architecture of IAML, the metamodel has been architected using *packages* in a similar fashion to the packages of the UML metamodel [253, pg. 11]. An overview of the IAML metamodel packages and their dependencies is illustrated as a UML package diagram in Figure 5.5; in this figure, each package is annotated with the number of classes defined within that package as described in this chapter<sup>7</sup>.

<sup>6</sup>This is an intended instance of a *construct deficit* in the visual notation, as discussed later in Section 7.4.3.

<sup>7</sup>While the proof-of-concept implementation of IAML uses EMF packages, these do not yet correspond to the packages described in this chapter. This package refactoring process represents future work, as discussed in Issue 281: *Refactor model elements into documented packages*.



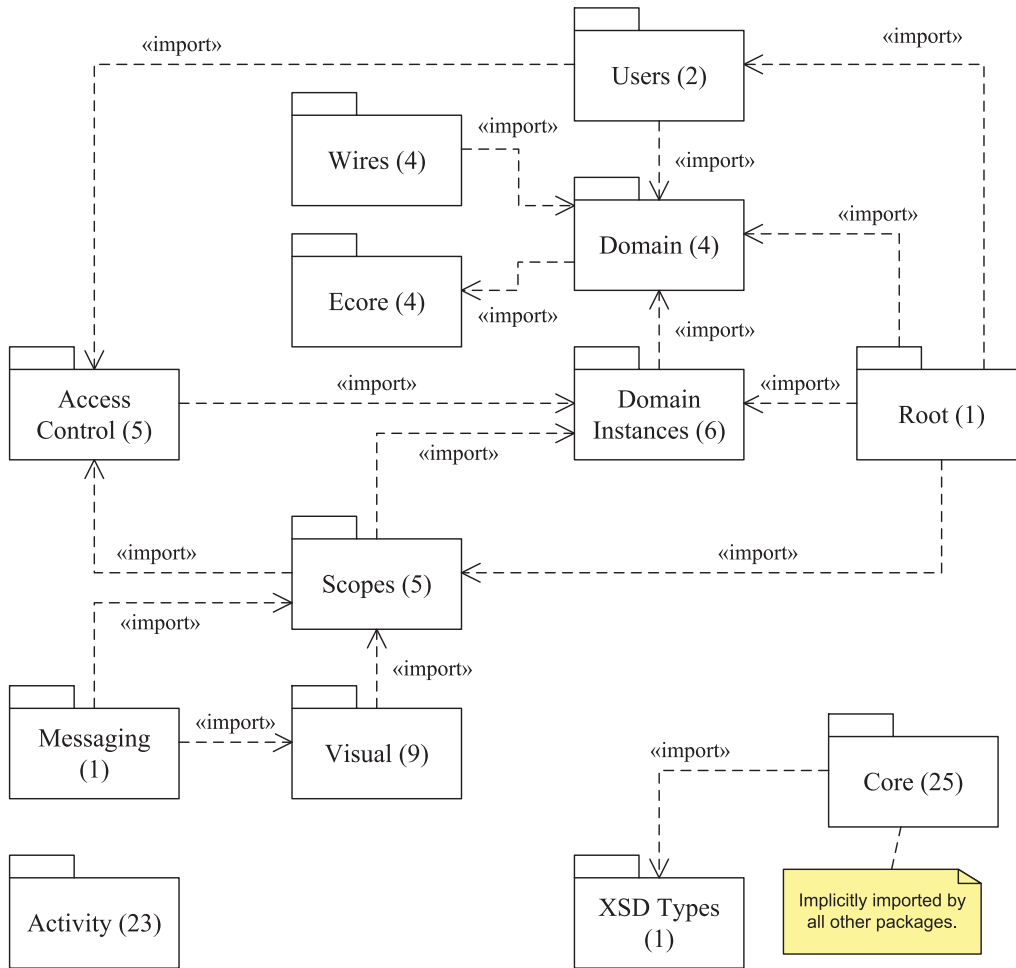


Figure 5.5: Dependencies between packages of related model elements within IAML

In this package diagram, the *Core* package is implicitly referenced by most of the other packages in the IAML metamodel, and forms the architectural *kernel* [253, pg. 12]. These packages are architected in an independently layered fashion to improve the design of the language. Inspired by the layered conceptual diagram of the Java SE platform [264], this package diagram can be reinterpreted as the *layered architecture diagram* illustrated in Figure 5.6. The remaining sections of this chapter will discuss the contents of each package in the IAML metamodel.

## 5.4 Metamodel Core

With the overall design principles of the modelling language discussed, the remainder of this chapter will focus on the metamodel structure of IAML, and on discussing the rationale behind the design decisions involved. In particular, each of the design concepts discussed earlier in Chapter 4 will be explored and implemented as part of the IAML metamodel.

The *Core* package forms the architectural kernel of the IAML metamodel in terms of its basic constructs – such as [Predicates](#), [Conditions](#), [Events](#) and [Actions](#) – and is therefore the most critical. The description of this package will be split into five aspects: the underlying logic model; the function model; the event-condition-action model; the wires model; and a constructs model which unifies these separate models.

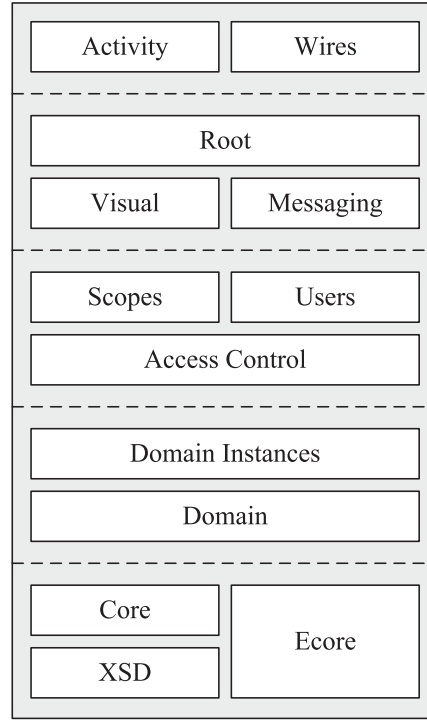


Figure 5.6: Layered architecture diagram illustrating the layered design of packages within the IAML metamodel

### 5.4.1 Logic Model

As discussed by Wirth [364], it is desirable to describe core concepts – such as conditions, operations, functions and values – on a rigorous mathematical definition, to provide a strong foundation for the rest of the language. The core of the IAML metamodel is therefore closely related to *first-order logic* [98], with a partial syntax provided here in Figure 5.7. The elements in this syntax are then mapped to IAML model elements according to the mapping in Table 5.2; the resulting structure of the core logic metamodel is introduced in Figure 5.8<sup>8</sup>.

The fundamental building blocks of IAML logic elements are **Functions** and **Parameter Values**, which are associated through an instance of a **Complex Term**. A **Function** defines a function signature; a sequence of *slots*, each with a name and type, and the **Function** itself has a name and type. A **Complex Term** associates a **Function** with a set of **Parameter Values**, using the **Parameter** named association class<sup>9</sup>.

When a **Complex Term** is referenced, each incoming **Parameter Value** to the **Complex Term** instance is bound according to the **name** of the associating **Parameter**, to the named slots of the **Function**. This creates variable bindings; that is, the value of a **Parameter** named *x* is bound to the slot *x* of the **Function** of the **Complex Term**. An OCL constraint is provided to ensure that all slots of a **Function** are provided with exactly one matching named **Parameter Value**. The *evaluation* of a **Function** will *return* a new typed value according to the semantics of the given **Function** and the respective incoming bound variables.

The fundamentals of logic can then be built from these basic elements. A **Predicate** is a **Function**

<sup>8</sup>OCL constraints defined throughout this chapter are implemented in the *Model Verification with OCL* component, discussed later in Section 7.7.2.

<sup>9</sup>A named association class is preferable over an ordered association relationship, as discussed later in Section 5.4.1.



<i>Sentence</i>	$\rightarrow$	<i>AtomicSentence</i>   <i>ConnectiveSentence</i>
<i>ConnectiveSentence</i>	$\rightarrow$	<i>Sentence</i> <i>Connective</i> <i>Sentence</i>
<i>AtomicSentence</i>	$\rightarrow$	<i>Predicate</i> ( <i>Term</i> *)
<i>Term</i>	$\rightarrow$	<i>Constant</i>   <i>Variable</i>   <i>ComplexTerm</i>
<i>ComplexTerm</i>	$\rightarrow$	<i>Function</i> ( <i>Term</i> *)
<i>Connective</i>	$\rightarrow$	$\vee$   $\wedge$   $\oplus$
<i>Predicate</i>	$\rightarrow$	$\neg$   $=$   ...
<i>Function</i>	$\rightarrow$	$+$   $-$   <i>concat</i>   ...

Figure 5.7: A partial syntax of first-order logic, adapted from Fitting [98]

FOL Concept	IAML Model Element
<i>Sentence</i>	Condition
<i>ConnectiveSentence</i>	Complex Condition
<i>AtomicSentence</i>	Simple Condition
<i>Term</i>	Parameter Value
<i>ComplexTerm</i>	Complex Term
<i>Connective</i>	AND   OR   XOR
<i>Predicate</i>	Predicate
<i>Function</i>	Function
<i>Constant</i>	Value
<i>Variable</i>	(not supported)

Table 5.2: Associations between first-order logic elements and IAML elements

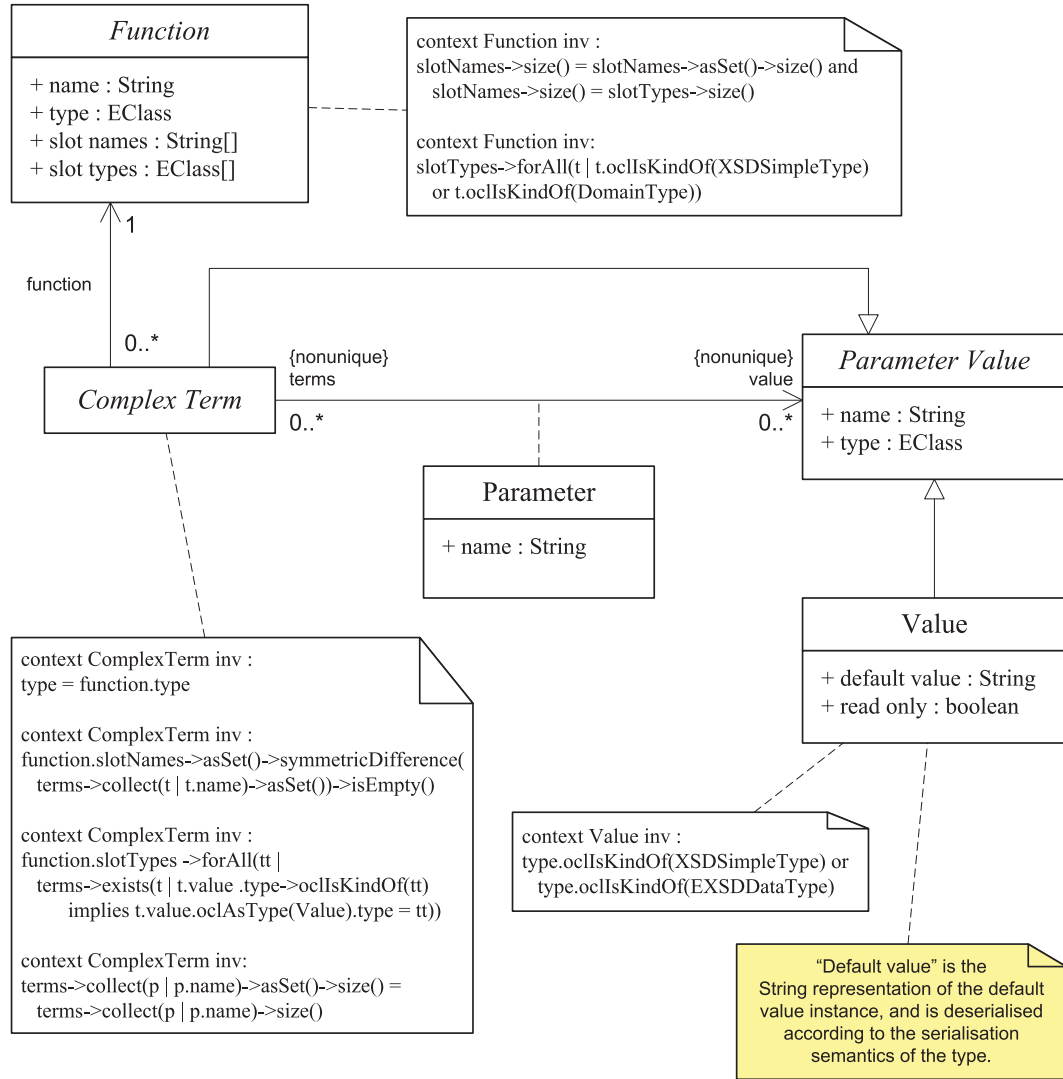
that always returns a `boolean`, as discussed earlier in Section 4.3.3. Similarly, a `Condition` is the instantiation of a `Predicate` with a set of `Parameter Values`, mirroring the `Complex Term` concept; consequently, a `Condition` can be defined as a `Complex Term` using a `Predicate`.

To support the definition of *ConnectiveSentences* – that is, a *Sentence* connected using *Connectives*, such as  $\vee$ ,  $\wedge$  and  $\oplus$  – the concept of a `Condition` is split into `Simple Conditions` and `Complex Conditions`. Boolean functions such as negation and equality are implemented as `Functions`; for example, negation is provided by the XQuery function `fn:not` which accepts a single `boolean`-typed argument [350]<sup>10</sup>, and equality provided by a number of XQuery functions.

*Variables* are not currently supported in IAML – that is, these logic constructs only provide terms. Support for variables would require some method to define how to select elements within a model, and also some method to define how the selection process may access the model instance. This could be implemented using the XPath language [347] to select elements from the Document Object Model (DOM) of the model instance [336]. Throughout the development of the IAML logic metamodel, support for *variables* has not been necessary<sup>11</sup>; quantifiers such as  $\forall$  and  $\exists$  are therefore not supported.

<sup>10</sup>An example model illustrating the use of negation is provided later in Figure 5.25.

<sup>11</sup>One consequence of this design decision is that events must be modelled with respect to a triggering element, as discussed later in Section 5.7; it is not possible to model “catch-all” events.

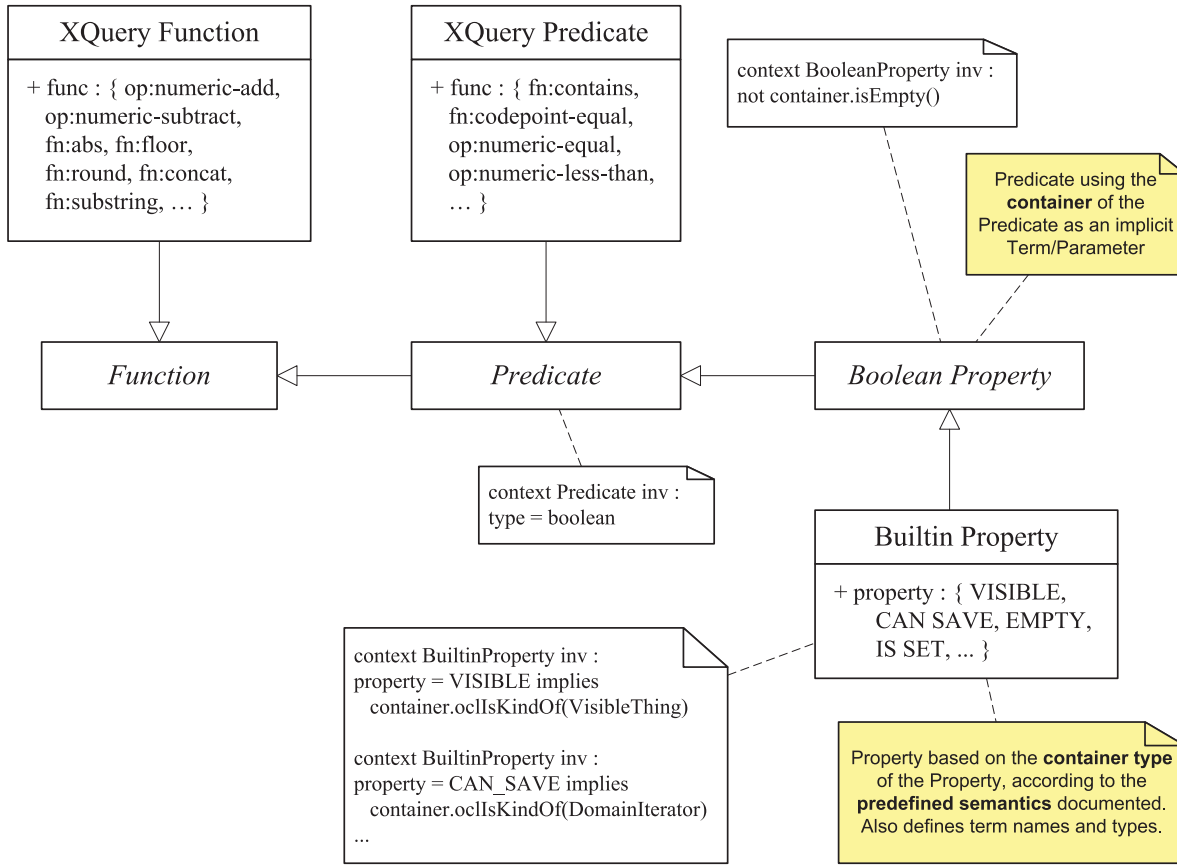
Figure 5.8: UML class diagram for the *Core* Package: Logic Model

### Named Parameters

One important question is on the design of parameters used in **Complex Terms** within the metamodel. One option is to create a simple association between the **Complex Term** and **Parameter Value** elements and label it as an “{ordered}” association [254, pg. 42]. That is, the relations are defined using *positional arguments* rather than *slotted arguments* [38]. However, such an approach directly impacts on the usability of the resulting model instance development environment.

In particular, this means that developers must always be mindful of the order of the arguments, and arranging the order of elements by accident may introduce significant problems that cannot be checked at design-time. *Named parameters* rather ensure that serialisation order does not matter, and missing parameters can easily be detected.

Parameters are therefore designed as an explicit association class [254, pg. 46–48], with each parameter named; conversely, each **Function** defines a set of **slot names**. The association ends for the named parameters must also explicitly be non-unique, to allow for situations where a **Complex Term** uses the same **Parameter Value** for more than one slot; for example, the expression  $5 + 5$  uses the same **Value** instance twice within a single **Complex Term** instance.

Figure 5.9: UML class diagram for the *Core Package: Function Model*

### 5.4.2 Function Model

The **Function** metamodel element is extended to introduce two new model elements – **Boolean Property** and **Builtin Property** – to simplify the development of the remainder of the IAML metamodel, as illustrated in Figure 5.9. The intent of these extensions is to permit model objects to contain **Predicates** specific to the instance of the model element. For example, a **Domain Iterator** contains the **Predicate** “empty”, and the value of this predicate is unique to each iterator instance.

This may be achieved by providing the instance as an positional argument to the **Predicate**  $f$ , such as  $f(C, a_0, \dots, a_n)$  where  $C$  represents the container. Within object-oriented languages, this syntax is deprecated in favour of  $C.f(a_0, \dots, a_n)$  to highlight that the container *owns* the particular function or method through its type<sup>12</sup>. This latter approach is used for IAML to define a **Boolean Property**, which is a **Predicate** using its *container* as an implicit *slot* with a predefined value.

Model element instances may be contained by other model element instances, and this relationship is defined as a *containment reference*. This reference may be defined explicitly, however to simplify the design of the metamodel, it is instead desirable to specify that *all* model elements have an implicit *container* relationship as an opposite to every containment reference. This is identical to the semantics of the **eContainer** method specified by the Eclipse Modeling Framework [314, pg. 31–32]. This *container* relationship is not explicitly modelled in the class diagrams throughout this chapter.

<sup>12</sup>As discussed by Meyer [229, pg. 447], this syntax represents a *qualified procedure call* and rules that the function  $f$  must be available to the class  $C$ .

### On the slot types of Functions

In the version of first-order logic used within this thesis, a *Function* has no restriction on the acceptable types of terms. A *Function* can therefore operate on anything within the defined universe<sup>13</sup>.

However, in an EMF-based implementation of a modelling language, the definition of an *EClass* must explicitly state a type for each of its references, even if that type is to the generic *EObject* type. That is, the *slot types* must each be typed in the metamodel design. The question thus becomes one on the selection of the most appropriate type for *Function* terms.

Within the design concepts of IAML, a *Function* can only be applied to two types of source data within the model – primitive types through *XSD Simple Types* discussed in Section 5.5.1, and complex types through *Domain Types* discussed in Section 5.6. Four options for restricting the domain of *slot types* were considered:

1. The type of *slot types* could be defined as the union of *XSD Simple Types* and *Domain Types*. However, *Ecore* does not support the concept of *type unions* in references.
2. A reimplementaion of *XSD* within *EMF* would allow the definition of a new common ancestor between *XSD Simple Types* and *Domain Types* i.e. the type of *slot types* would be a new interface *IAML Type*, with both *XSD Simple Type* and *Domain Type* implementing this interface. However, this represents additional implementation complexity and effort, and any updates to *XSD* would need to be propagated to the *IAML* implementation.
3. Remove the concept of *Domain Types*, and only allow *XSD Types*, which includes both *XSD Simple Types* and *XSD Complex Types*; i.e. the type of *slot types* would be the abstract *XSD Type*. The drawbacks of this approach are discussed later in Section 5.6.2; in particular, *XML Schemas* do not support the definition of operations, conditions or events on complex types.
4. Retain the concept of *Domain Types*, but specify that *Domain Types* are an extension of *XSD Complex Types*; i.e. the type of *slot types* would be the abstract *XSD Type*. This approach is also discussed later in Section 5.6.2; in particular, the complexity of *XSD Complex Types* would not easily map to relational databases – a fundamental target platform for this research<sup>14</sup>.

While *EMF* does not support the concept of type unions, this design can be emulated using *OCL* constraints on the metamodel. The reference type of *slot types* is therefore defined to be *EClass*, which is the only common ancestor of *IAML* model elements and *XSD* types, as discussed later in Section 7.2. This specification is subsequently constrained to only be an instance of *XSD Simple Type* or *Domain Type* using *OCL* constraints, as illustrated in Figure 5.8. It is not desirable to leave the type unconstrained to *EClass*, as this would support higher-order logic<sup>15</sup>.

#### 5.4.3 Event-Condition-Action Model

Events are placed within the *Core* package of the metamodel as they are considered a fundamental aspect of RIAs within this thesis. While considering the evaluation of existing approaches to the visual modelling of events earlier in Section 4.2, the corresponding metamodel for ECA rules is illustrated

<sup>13</sup>*Functions* can also accept other *Functions* as terms, however this represents a higher level of logic [223] than first-order logic, which is outside the scope of this thesis.

<sup>14</sup>Use Case 1: *View Data*.

<sup>15</sup>For example, a *Function* with an instance of the *slot types* reference typed to *Function*.

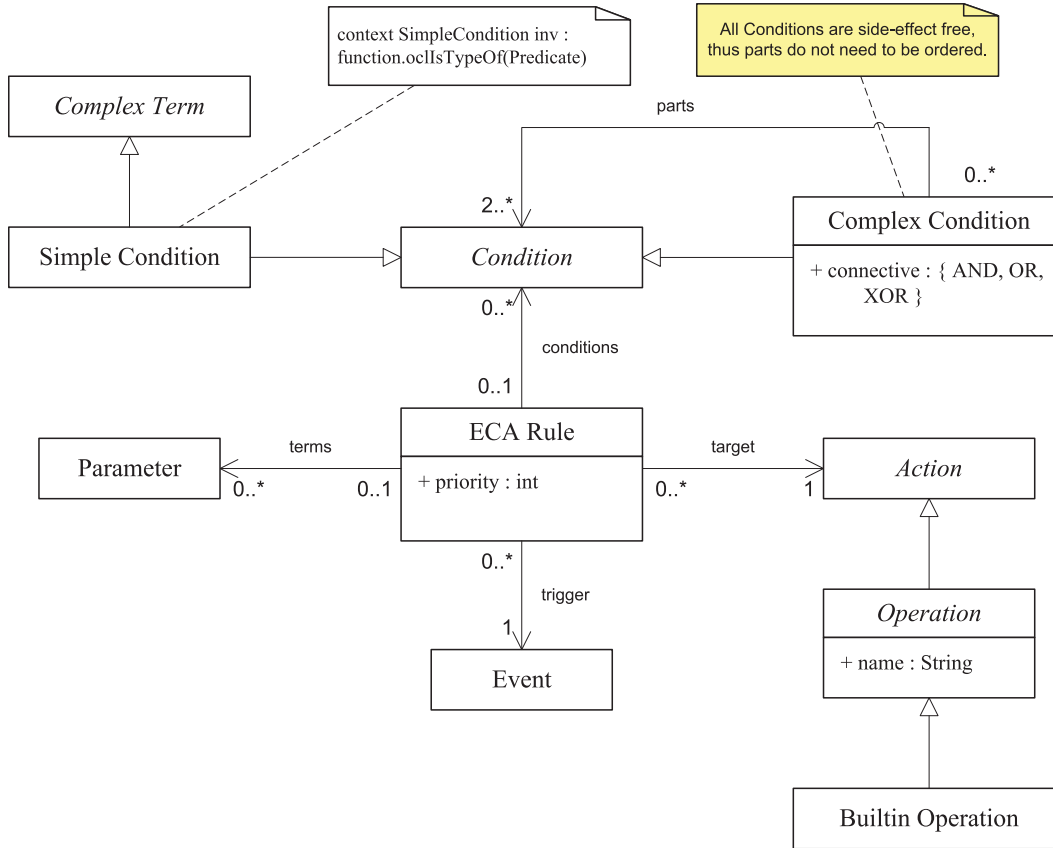


Figure 5.10: UML class diagram for the *Core* Package: Event-Condition-Action Model

in Figure 5.10. In particular, an **ECA Rule** states that an **Event** will execute a particular **Action** when the event is *triggered*, optionally constrained by the **Conditions** on the rule itself.

Corresponding instances of ECA rules may be represented visually as in Figure 5.11, illustrating an instance of an **ECA Rule** within the Ticketiaml application; here the visual notation for a UML *SendSignalAction* [254] is used to represent the **Event**. This rule is triggered by the *onAccess* **Event** of the containing **Label** (as defined later in Section 5.7.3), and a **Parameter** named “message” is provided to the target **Operation**.

It is not possible for a model developer to directly define their own domain-specific events in IAML, because the definition of an event also requires the definition of the semantics of when the event would be triggered. This would therefore require the definition of an *event modelling language*, which is outside the scope of this research.

## Actions

In IAML, the UML concept of an *Activity* is reused as the **Action** abstract metamodel element; that is, a “specification of parameterized behavior” which may contain actions of various kinds [254, pg. 315]. An **ECA Rule** may be provided a set of named **Parameters**, which will be passed along to the **Action**; for example, as parameters to an **Operation**, as discussed later in Section 5.8.2.

IAML does not use UML’s *Action* element, as an **Action** can only be used and executed within a single *Activity*. It would be more desirable to be able to execute an action from many different contexts, which an *Activity* permits. However, IAML retains the name “Action”, to match the *Event-Condition-Action* paradigm. All actions are contained directly by a target element, which can be referenced by

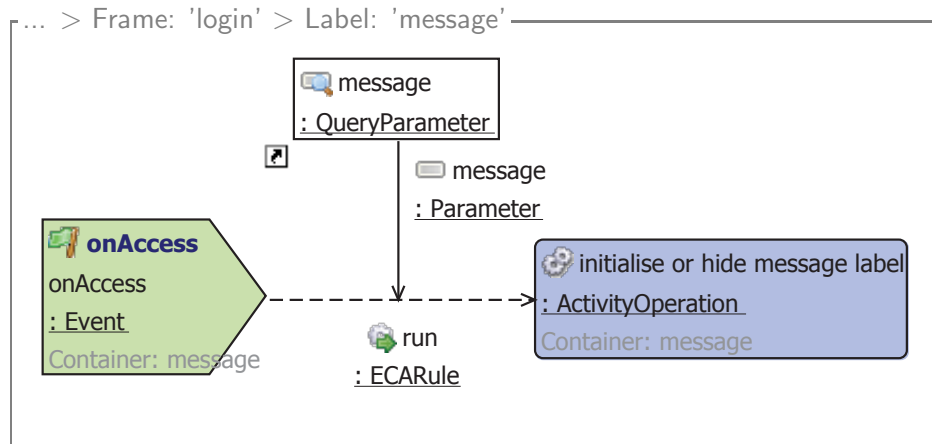


Figure 5.11: Ticketiaml: An ECA Rule connecting an Event to an Operation

the action itself through the implicit *container* relationship. Events are contained by a source element, and this is the element that triggers the event itself. More information on this technique is discussed in Section 7.4.4.

#### 5.4.4 An example decomposition of a Simple Condition

To illustrate the mechanism by which a **Simple Condition** – a **Complex Term** – evaluates the instances of its provided **Parameters**, the following simple conditional expression will be decomposed as an example:

**gig.minAge.value < textfield.value**

This expression is given in a pseudo-textual format, and ‘<’ represents the XQuery comparison function `op:numeric-less-than` [350]. The subsequent decomposition of this expression is illustrated in Figures 5.12 and 5.13. (The design rationale behind domain modelling types such as **Domain Instances** is discussed later in Section 5.6.)

The syntax used in Figure 5.12 is based on the UML syntax for defining instances of classes, as illustrated in the UML infrastructure specification [253, pg. 201]. Since the IAML metamodel is defined in terms of standard classes and not stereotypes [254, pg. 670], UML stereotype notation should not be used to illustrate the metaclass name of each element. Generally, metaclasses in UML model instances are distinguished based on visual notation, and not by explicitly labelling the metaclass name; however in this example, these metaclass name labels will improve the understandability of this decomposition.

The full decomposition as in Figure 5.12 is not normally provided to the developer, as this syntax is very complex. The complexity of this representation is expected, and would parallel the alternative decomposition of the textual expression into an abstract syntax tree. The representation provided to the model developer is illustrated instead in Figure 5.13.

In this model developer view, some model elements (such as **Simple Conditions** and **Parameters**) are reduced to edges; most of the model element attributes are moved into a separate *properties view*; and some referenced elements are removed entirely (such as the **Domain Type** and **Domain Attribute** classifiers of the first **Parameter** instance). A full discussion of the visual representation of IAML model instances is provided later in Section 7.4.





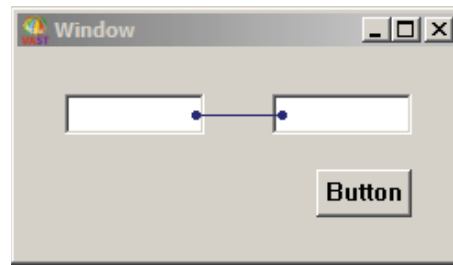
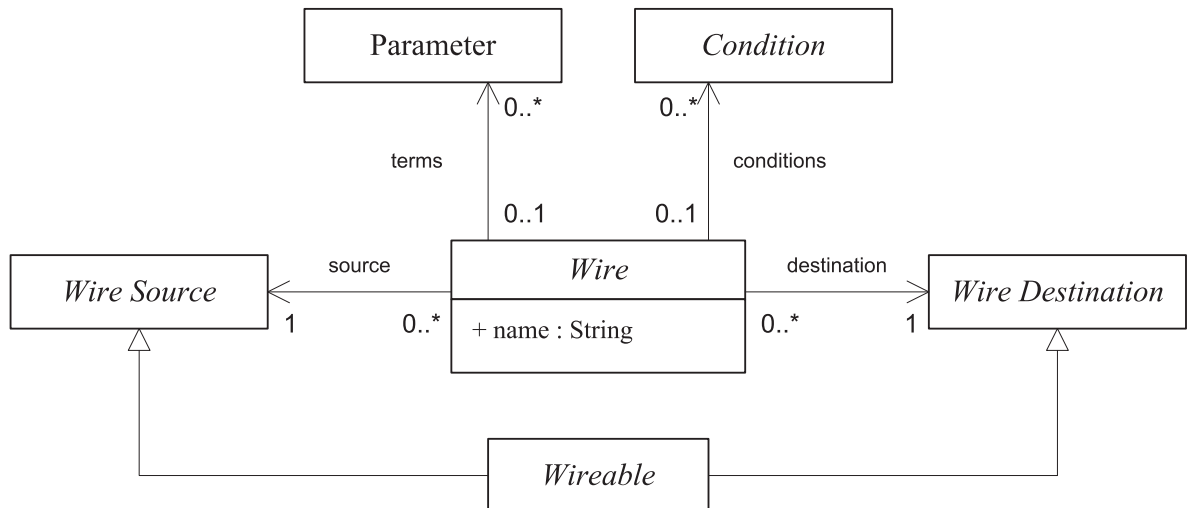


Figure 5.14: VisualAge for Smalltalk: Using Connections

Figure 5.15: UML class diagram for the *Core* Package: Wires Model

### 5.4.5 Wires Model

In order to support the inclusion of common reusable patterns in a modelling language for RIAs, IAML proposes the concept of a *Wire* as an abstract connection that represents additional functionality. Wires are inspired by the *connection* concept in VisualAge for Smalltalk [210]. VisualAge for Smalltalk supports a visual rapid application development approach, as in Figure 5.14; here, a simple application consisting of two text fields and a button are defined. The two text fields are associated with a *connection*, connecting the *value* property of each text field. When executed, the running application automatically keeps the two text fields synchronised.

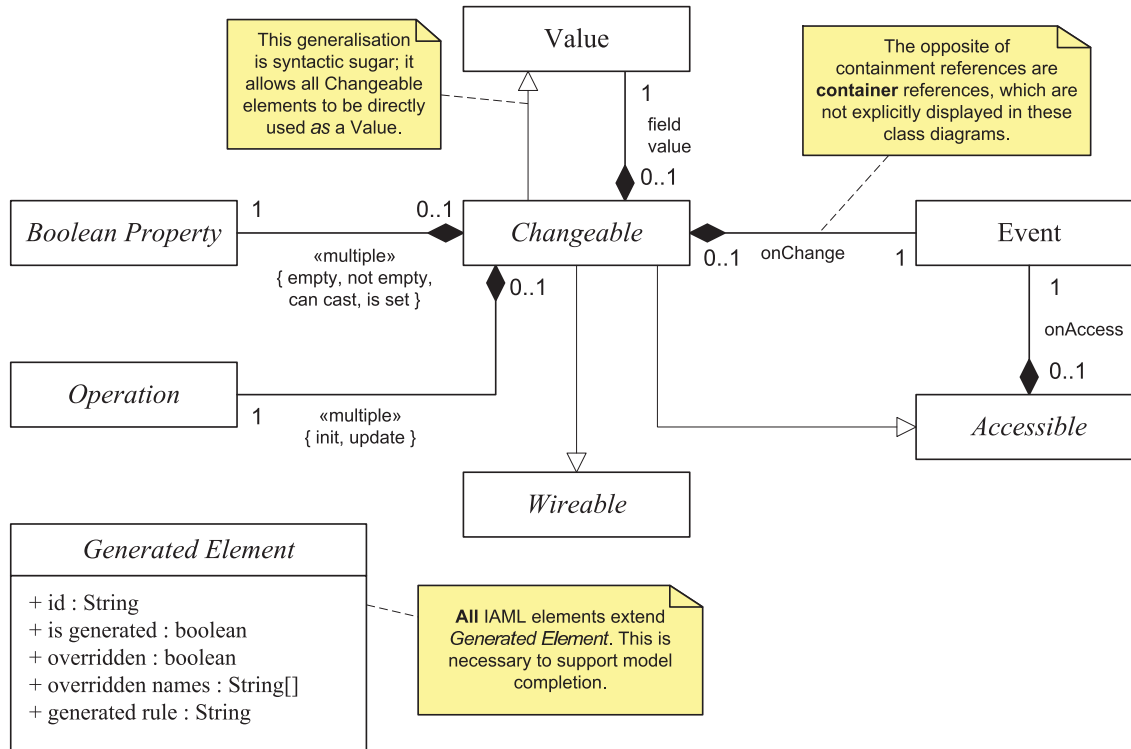
Wires are implemented using *model completion* as discussed earlier in Section 4.9.1, which allows for the reusable pattern logic to be executed independent of the underlying metamodel implementation. Each reusable pattern instance is represented as a subtype of *Wire*, and the range of *Wires* subtypes within IAML are discussed later in Section 5.9. This infrastructure forms part of the *Core* package of the IAML metamodel as illustrated in Figure 5.15.

### 5.4.6 Constructs Model

The final aspect of the *Core* package of the IAML metamodel is the definition of the *Changeable* and *Accessible* interfaces, as illustrated in Figure 5.16. These interfaces are used to simplify the design of other IAML metamodel elements by providing a common infrastructure.

The *Changeable* interface is used to define a model element which has a *single Value* named *field value*. To simplify the development of model instances, *Changeable* is specified as a subtype of *Value*



Figure 5.16: UML class diagram for the *Core Package: Constructs Model*

as a form of syntactic sugar. This means that **Changeable** elements may be used directly as **Parameters** in **Complex Terms**, and the element itself becomes an alias for the contained **field value**.

In order to support one of the use cases of model completion – that it should be possible for a model developer to complete an intended model, and then later remove these generated elements<sup>16</sup> – the abstract class **Generated Element** is also defined in Figure 5.16. All elements within the IAML metamodel should be subtypes of this abstract class to enable these use cases; the meaning of these attributes are discussed later in Section I.56.

Figure 5.16 also introduces the use of the «multiple» stereotype to simplify the UML class diagrams throughout this thesis. A «multiple» stereotype on an association is an alias for multiple identical associations on the two participating model elements, each with the same multiplicities for each association end, but with different association names as listed on the alias association. For model elements with multiple associations between the same element types – for example, the many **Boolean Properties** defined for a **Domain Iterator** later in Figure 5.22 – using this stereotype significantly simplifies the resulting UML class diagram.

## 5.5 Type System

With respect to the type systems described earlier in Section 4.4, the IAML type system is designed to be *statically checked* [47]; **Values** can only ever have a single type<sup>17</sup>. Statically checked languages are often easier to verify using verification technologies, and this is one of the design goals of IAML.

<sup>16</sup>This action is implemented by the *remove contained generated elements* action implemented in the *Diagram Actions* component, as discussed later in Section 7.4.5.

<sup>17</sup>It is important to note that a statically checked type system does not prevent a single type from having many supertypes through a *subtype relation* [47].

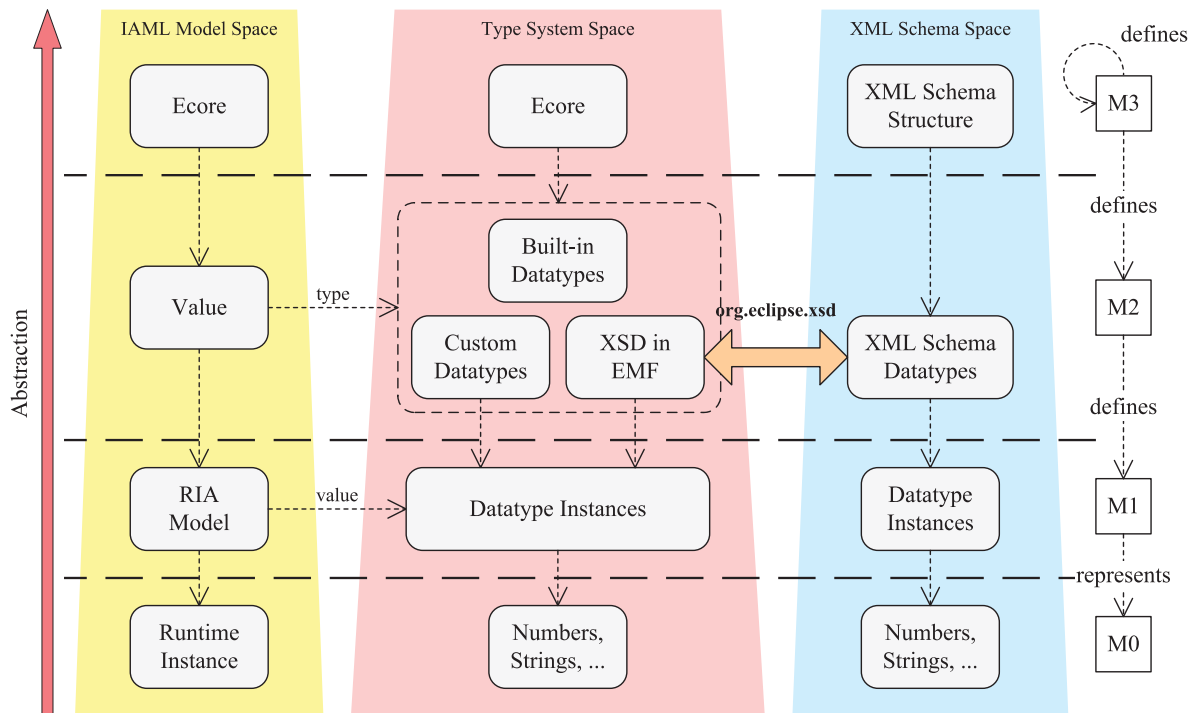


Figure 5.17: Adapting the metamodeling architecture of MDA to the use of primitive type systems

However, the type system is also *weakly checked* [47]; a type instance can be implicitly translated into another type instance if necessary through a *cast*, as discussed later in this section. A weakly checked language may simplify model instance development, as type instances do not need to be explicitly translated. However weak typing can hinder the verification of system correctness, unless these implicit casts are well-documented.

### 5.5.1 Primitive Types

As discussed earlier in Section 4.4.2, XML Schema [338] provides a framework of builtin datatypes and a framework for the definition of derived datatypes. Primitive datatypes in IAML are therefore based on the existing XML Schema datatypes [339], satisfying the design goal of reusing existing standards where possible. IAML can also support the definition of new simple types through the restriction or extension of existing datatypes, according to the XML Schema specifications. IAML does not explicitly support XML Schema complex types, as discussed later in Section 5.6.2.

The Eclipse Modeling Framework also provides its own datatype definitions – such as the datatypes `EString` and `EInt` – and reusing these definitions was considered. However, EMF datatypes must be implemented with Java methods such as `convertXXXToString()` and `createXXXFromString()` [314, pg. 390–391], whereas XSD datatypes are defined using a platform-independent formal semantics [339]. This would mean that any EMF-based custom types would need to be implemented in Java, reducing the platform-independence of IAML.

### Primitive Type Systems in the Model Driven Architecture

The primitive type systems architecture discussed earlier in Section 4.4.3 may be used to understand the role of these primitive types within a model-driven architecture. The primitive type system design

of IAML is illustrated here in Figure 5.17; this architecture shows how predefined, built-in and custom datatypes (and their instances) interact with type definition references in the same model instances.

Within IAML, the suite of predefined datatypes are obtained directly from the XML Schema Datatypes definition [339], as supplied through the `org.eclipse.xsd` plugin developed by the Eclipse Foundation<sup>18</sup>. These predefined datatypes are extended to supply built-in datatypes such as `iamlEmail`, as discussed in the next section. Finally, developers may introduce their own custom datatypes into the environment, which represents the definition of a new metamodel element within the M2 layer.

## Comparison to MOF

The architecture of the IAML type system was partially inspired by the *semantic domain model for constructs* defined in the MOF [251, pg. 53–64]. This domain model was not used directly in the IAML metamodel as this domain model was deemed too complex for defining primitive types, with respect to the desire to reuse first-order logic within the metamodel core.

In particular, the IAML type system does not define metamodel elements for structural associations, instance slots<sup>19</sup>, value specifications or structural features. However, the reuse of the EMOF-compatible Ecore metamodel [12], as discussed later in Section 5.6, does define associations and structural features within the scope of domain modelling.

## Casting

*Casting* refers to the conversion of one instance of a datatype to another value of a different datatype. Ideally, a cast will not result in loss of precision (for example, an *integer* into a *real number*), but there are situations where a loss of precision cannot always be prevented (for example, a *real number* into an *integer*). In the latter case, the magnitude of precision loss is dependent on the original value; a cast without any loss of precision is deemed a *successful cast* within IAML.

The implementation of every primitive type in IAML must also implement a method to evaluate whether a particular instance of the type may be successfully cast into an instance of `string`, and whether a particular instance of type `string` may be successfully cast into an instance of that given type. A selection of these conditional casts with respect to certain XSD datatypes are illustrated in Table 5.3<sup>20</sup>; these casts are separated into the three categories of *permitted* (✓), where the cast is guaranteed to be successful; *conditional* (?), where cast success depends on the runtime value of the instance; and *prohibited* (✗), where the cast is guaranteed to be unsuccessful for *all* values.

XML Schema defines how one instance of a given datatype may be converted into a different datatype, by defining the *lexical space* of a datatype [339]. Alternatively, when a datatype is being used with an XPath query [347], the conversion is defined as part of the *constructor function* of the particular target type<sup>21</sup> [350]. Within IAML, the casting semantics of the latter process are used to cast XSD datatype instances.

<sup>18</sup>The `EXSD Data Type` metamodel element is necessary to bridge this gap between XSD and IAML.

<sup>19</sup>The MOF definition of a *Slot* is different from the definition of a *slot* in the IAML metamodel, as highlighted by its metamodel specification [251, pg. 54]. That is, a MOF *Slot* is owned by an *InstanceSpecification*, whereas an IAML *slot* is owned by a *Function*, as discussed earlier in Section 5.4.1.

<sup>20</sup>In this table, `email` refers to the IAML-defined `iamlEmail` type introduced in the next section, as XML Schema does not define an “email” datatype.

<sup>21</sup>For example, the process of converting the `string` instance “3” into an `integer` instance would be expressed in XPath as the expression `xs:unsignedInt("3")`.

	Source Type				
Target Type	string	int	dateTime	email	default type
string	✓	✓	✓	✓	✓
int	?	✓	?	✗	?
dateTime	?	✓	✓	✗	?
email	?	✗	✗	✓	?
default type	✓	✓	✓	✓	✓

Table 5.3: A selection of permitted, conditional and prohibited datatype casts in IAML

### 5.5.2 Additional Built-in Primitive Types

While XML Schema provides a wide range of built-in datatypes, these do not include datatypes common to the web application domain. For example, e-mail addresses and URLs are both common elements of data within Rich Internet Applications, but these datatypes are not supported natively in the XML Schema definitions. These common datatypes are therefore included as built-in primitive datatypes in the implementation of IAML, and some of these types will be briefly discussed here.

#### Default Type

IAML supports the concept of a “default type”, which represents an untyped value; in some languages, this is also known as an *Any* type. Following the design of the type system as a weakly checked language, such a value instance must first be implicitly cast to a specific type before it can be used in a typed scenario; this is in contrast to languages such as PHP, where the type of a value is determined at runtime depending on the context of its use [2].

All datatypes must also support the serialisation of instances into the default type (and this serialisation often follows the same semantics as casting to `string`). In particular, IAML requires that all type instances must be serialisable into a string-based format without a loss in precision.

#### iamlEmail

This datatype represents an e-mail address, such as the value “`jevon@openiaml.org`”. However, the complete definition of an e-mail address in RFC 5322 [283] is very difficult to implement in terms of a regular expression; RFC 5322 allows arbitrary labels and whitespace throughout a valid e-mail address, vastly increasing the complexity of any regular expression.

#### iamlURL

This datatype<sup>22</sup> represents instances of Uniform Resource Locators (URLs) [31], such as the value “`http://openiaml.org/model#InternetApplication`”. XML Schema already defines a datatype for URIs (`anyURI`), but does not define URLs, which are a subset of URIs. A separate URL datatype is necessary in order to represent locations on the Internet; URIs, on the other hand, can represent content such as text strings or ISBN codes.

<sup>22</sup>Issue 269: *Implement iamlURL builtin datatype.*

## iamlOpenIDURL

The `iamlOpenIDURL` datatype represents a validated OpenID URL; that is, in order to be a valid instance, it must be both a valid URL, and has been validated using the OpenID protocol [281]. Consequently, the validity of its string representation can change over time; if a value is cast to a `string` instance and then back to an `iamlOpenIDURL` instance, the user may need to re-authenticate with their identity provider. The full rationale behind this datatype and its semantics, such as how a valid instance may lose its validity if cast, is discussed later in Section 5.10.4.

## Proposed Datatypes

A number of other builtin datatypes have been considered, but have not yet been defined in IAML. This is due to resource constraints; as a fundamental part of IAML, the definition of a new primitive type requires subsequent implementation and evaluation in [Domain Types](#) (Section 5.6.1), [Gates](#) (Section 5.11.4), different [Visible Thing](#) types (Section 5.13.2), and input validation scenarios (Section 5.5.4). The implementation and evaluation of the following datatypes remains future work:

- `iamlPassword`, representing a password. By explicitly typing value instances as passwords, we can execute additional verification checks (e.g. never display a `iamlPassword` in clear text) and improve security (e.g. all `iamlPassword` instances must be stored through a hash function) as default properties of RIAs.
- `iamlFile`, representing a file. By providing a built-in datatype for files a developer would not need to conceptualise files differently from any other representation of data. This type could be further restricted, e.g. to only text files, binary files, and so on. A built-in datatype would also improve the platform-independence of such a RIA model, as files could be stored using a number of technologies depending on the scenario<sup>23</sup>.
- `iamlImage`, representing an image, and could be a restriction on `iamlFile`. Such a datatype would therefore introduce similar advantages to those gained in defining the `iamlFile` datatype.
- `iamlIdentity`, representing a user identity along with the mechanism that may be used to verify a users' identity. For example, this could be an instance of an `iamlOpenIDURL` as verified through OpenID [281], or an e-mail address used by Microsoft Passport. This proposed datatype is discussed in further detail later in Section 5.10.4.

### 5.5.3 Value Instances

Now that the underlying type system of IAML has been defined, it is necessary to describe how instances of these types can be defined, created, modified and used. A [Value](#) represents an instance of a particular type within a particular scope, and the value of this instance may change over time.

If an instance of a [Value](#) model element is contained by a [Scope](#) (as discussed later in Section 5.11), then at runtime there will be many different instances of this single [Value](#), and each instance will exclusively belong to a single instance of each unique [Scope](#) instance at runtime. The accessibility of

<sup>23</sup>For example, files can be stored directly on the file system, or within relational databases, or distributed globally as part of a content distribution network; each of these approaches have various benefits and disadvantages, as discussed by Eichinger [88].

a particular **Value** instance is dictated by the *containing scope* of the **Value**, a property calculated in the following manner:

- If the **Value** is contained within an **Operation**, then the containing scope of that value is the containing scope of the **Operation**'s container.
- If the **Value** is contained within a **Scope**, then the value instance is unique to that **Scope**. For example, a **Value** contained within a **Session** can only have one instance of that **Value** per **Session** instance, and instances of that **Value** cannot be accessed by other instances of the same **Session** scope.
- Finally, if the **Value** is not stored within a **Scope**, then the value instance is available globally according to the root **Internet Application**. Only one instance of the **Value** can ever exist.

**Values** are used in the composition of complex objects. For example, an **Input Text Field** also possesses the values **field value**, **default value** and **current input**; and the syntactic sugar of the **Changeable** interface defines the *value* of the **Input Text Field** as its **field value**. These values can be referenced and modified directly, however it is often more desirable to use an associated **Operation** such as **update** (as discussed in Section I.59). Model instance developers can also define their own **Values** within almost any other model element instance.

While the **default value** of a **Value** only supports the string representation of the instance value, it may be desirable in the future to support more complex method of defining default values. Expression languages such as XQuery [349] or OCL [252] could be used to define values based on the DOM of the underlying IAML model instance.

### On merging model elements representing value instances

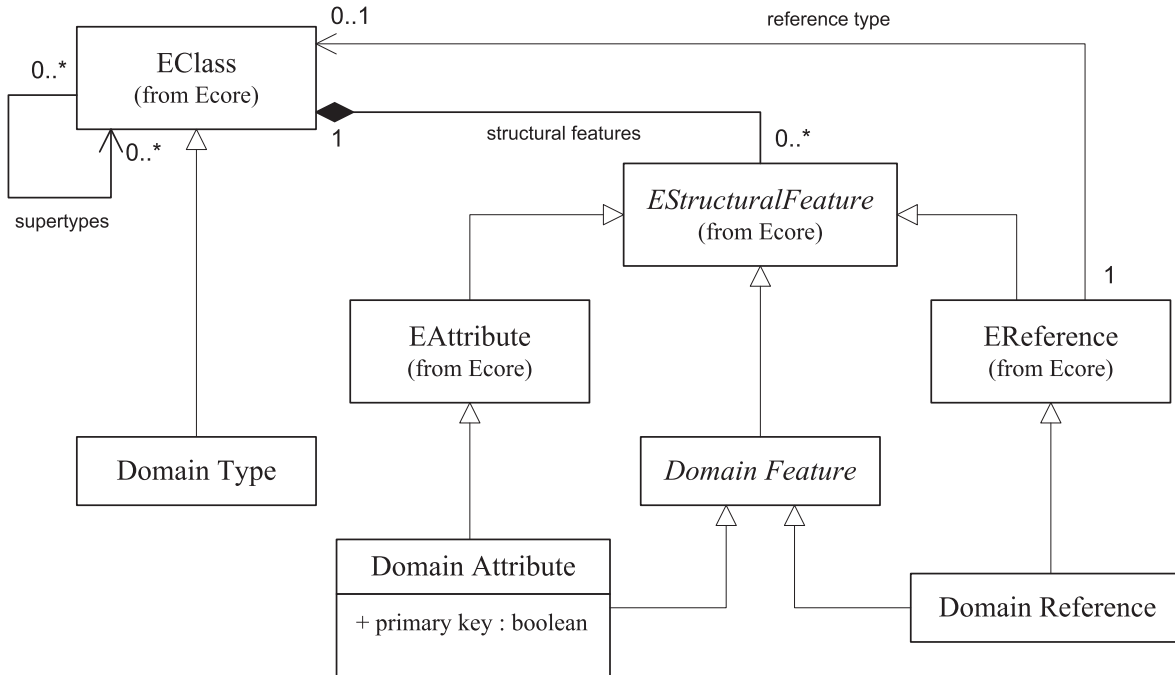
The four model elements of **Value**, **Query Parameter** (discussed later in Section 5.11.5), **Activity Parameter** and **Temporary Variable** (both discussed later in Section 5.8) are all concerned with the storage of value instances. As per the metamodeling design discussion of Section 5.2.4, it may be beneficial to merge these elements together into a single element to reduce the size and complexity of the IAML metamodel.

It would be undesirable to merge the **Value** and **Query Parameter** model elements together, as it is unlikely that a model developer would wish to change the storage semantics of a **Value** element to those of a **Query Parameter**. Allowing instances of **Value** to be set externally by the client via the request URL means that all **Values** may now contain untrusted data [319]. Similarly, a **Parameter** can only be set via an **ECA Rule**, whereas a **Value** can be set using any mechanism, which may similarly introduce a security problem.

The key difference between **Value** and **Temporary Variable** instances are the semantics of where the element may be accessed, and how long data persists within that instance. It may be desirable in the future to merge these two elements together, by introducing a new attribute for **Value** that defines the persistence of data; however, this attribute could conflict with the semantics of the *containing scope* property.

#### 5.5.4 Client-side Input Validation

Because all client-side elements such as **Input Text Fields** have an associated type, IAML can automatically implement some client-side input validation. For example, consider an application model

Figure 5.18: UML class diagram for the *Domain* package of IAML

that specifies a [Frame](#) containing an `iamlEmail`-typed [Input Text Field](#); if at runtime a user enters in an invalid e-mail address into this field, the application can automatically highlight the invalid field, and inform the user that the value is invalid. This free client-side input validation logic is provided through the model completion framework. For certain visual elements, the *onInput Event* may also permit validation to occur simultaneously while the user is entering in data, corresponding to the classical *asynchronous input validation* use case<sup>24</sup> of RIAs [368].

## 5.6 Domain Modelling

As the primitive type system has now been defined, these definitions can be used in the composition of complex types to support *domain modelling*, discussed earlier in Section 4.5. In this thesis, domain modelling is split into two related concepts: the *definition* of domain modelling schemas, expressed as [Domain Types](#); and the *access* of instances of these domain schemas, expressed as [Domain Iterators](#).

### 5.6.1 Domain Types

The IAML domain metamodel, used to define the structure and attributes of these domain-specific complex types, is adapted from the Ecore metamodel [314]. As illustrated in Figure 5.18, this metamodel replicates the structure of the Ecore metamodel, but this duplication is necessary in order for these metamodel elements to also subtype the [Generated Element](#) abstract class, necessary to support model completion.

<sup>24</sup>Use Case 19: *Asynchronous Form Validation*.



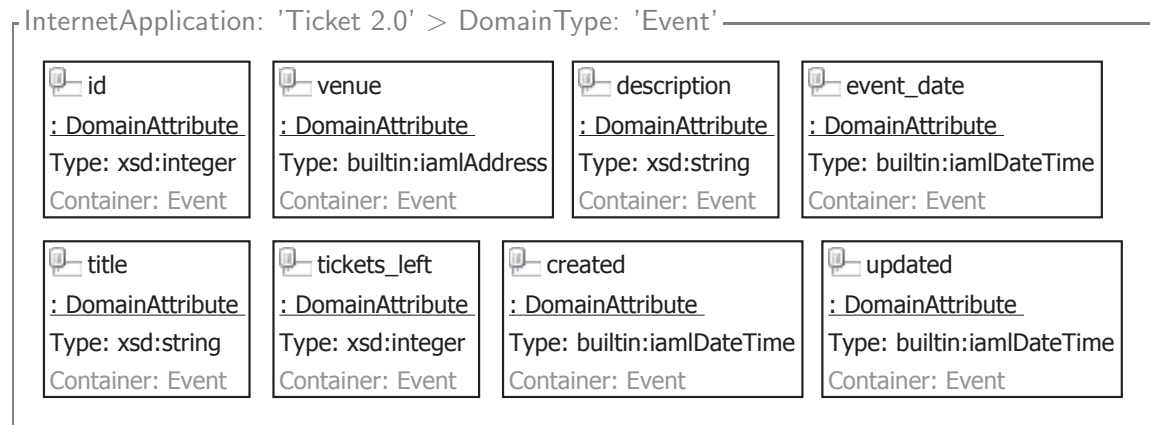


Figure 5.19: Ticketiaml: The defined **Domain Attributes** of the **Domain Type** *Event*

## Domain Type

A **Domain Type** may compose many primitively-typed instances of data into an overall complex type. Each primitively-typed instance of data is represented as a **Domain Attribute** contained by a particular **Domain Type**. Instances of a particular type may also reference other types; this reference is represented as a **Domain Reference**.

**Domain Types** may subtype others to support a simple implementation of multiple inheritance; multiple inheritance is necessary to support **Roles** as discussed later in Section 5.10. The IAML meta-model reuses the **supertypes** property of **EClass** in order to define these subtype relations, expressed as instances of **Extends Edge**.

## Domain Feature

As discussed earlier, one of the fundamental target platforms<sup>25</sup> of IAML is relational databases, where complex types are represented as tables, and relationships between table instances represented as foreign keys. Relational tables also often have uniquely-identifying attributes collated into a *primary key* [28, pg. 456–457]; these primary keys are essential for association tables.

This is at odds with the Ecore definition of an **EClass**, where only a single **EAttribute** may be uniquely identifying, as specified by the **ID** of the **EAttribute** [314]. Consequently it is necessary to also define a boolean **primary key** attribute on **Domain Feature**, representing that this attribute or reference may be combined with other attributes into a primary key within a relational database. If a **Domain Type** does not define *any* primary key, then by default model completion will insert in a new integer-typed **Domain Attribute** as a primary key.

## Domain Type Example

Since a **Domain Type** simply consists of a set of **Domain Attributes**, the corresponding visual representation is fairly simple. In Figure 5.19, the eight **Domain Attributes** used to define an *Event Domain Type* within Ticketiaml is represented; this includes the integer-typed ID attribute, string-typed title and description attributes, and *iamlDateTime*-typed date attributes.

<sup>25</sup>Use Case 1: *View Data*.



## 5.6.2 Design Decisions on Modelling Domain Types

### Reusing Existing Type Systems

Similarly to the reasoning behind the design of the primitive type section, it is necessary to decide whether a new complex type system should be designed or if an existing complex type system can be reused. As discussed in Section 4.5, four existing metamodels were considered: ER models, UML classes, XML Schemas, or EMF models.

ER diagrams are not suitable for modelling complex types in IAML as the language does not support the concept of type inheritance, which is necessary to model users as discussed later in Section 5.10. The initial definition of ER diagrams [54] did not explicitly label relationship cardinality, object composition, or the direction of relationships; later extensions to ER diagrams included these concepts [278]. Finally, ER diagrams also require the use of the Data Object Tables model to define the inner attributes of classes, adding further complexity to a model instance.

Alternatively, a **Domain Type** could be defined as an extension of a **XSD Complex Type**, providing a complete implementation of XML Schemas within IAML. However, instances of **XSD Complex Type** can represent arbitrarily deep trees and sequences of complex data. Without arbitrarily restricting the depth of children permitted in complex types, implementing this functionality using relational database concepts – a fundamental target platform of this research<sup>26</sup> – would be difficult<sup>27</sup>.

The UML class diagram supports complex type definitions through UML elements such as *Classifier* and *Property*. However, the infrastructure behind UML classes are very complex; the latest UML specification [254] comprised of 55 separate model element types for class diagrams, including elements such as *Slot*, *OpaqueExpression* and *BehavioredClassifier*, negatively impacting the simplicity language design principle of IAML. Furthermore, there is no standard implementation or defined semantics of UML class diagrams, and a model developer could not rely on any form of interoperability between existing models.

The Ecore metamodel of EMF represents the best choice for modelling complex types in IAML when considering these alternative approaches. At 17 elements, the Ecore metamodel uses much fewer model element types than UML classes; there is a well-defined sample implementation of Ecore metamodel instances [314]; and this implementation has been ported to other platforms such as C++ [127]. Nevertheless, some Ecore element types such as **EDataType** and **EGenericType** are not necessary for the IAML metamodel, so these element types will not be supported within IAML; as discussed earlier, the underlying primitive type system of IAML uses XML Schema datatypes [339].

### Defining Domain Types using Metamodelling

Since the model of a domain type can be considered the definition of a metamodel for instances of that type, one can reasonably consider that the definition of domain types should reside within a separate metamodel, which is then used as the metamodel of a model instance. This scenario is illustrated in Figure 5.20, where the complex domain type *Gig* and an associated instance *my gig* is defined with a hypothetical domain modelling metamodel.

However, this approach does not satisfy the metamodelling architecture of the MDA discussed earlier in Section 3.1.5. If the metamodel layers are added to Figure 5.20, the IAML metamodel def-

<sup>26</sup>Use Case 1: *View Data*.

<sup>27</sup>For example, consider a single complex type which defines a *Company* to contains many *Persons*, each with their own *Address* complex type. At what point should these inner types be serialised to separate tables?

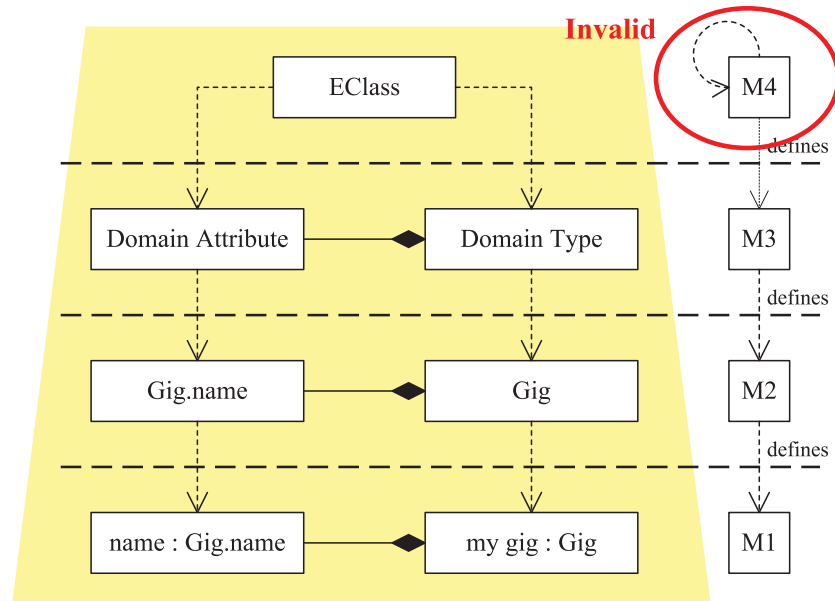


Figure 5.20: If domain types are defined using metamodeling, the resulting models are *incompatible* in terms of the MDA metamodeling architecture

inition must reside on M3. Since IAML is defined in terms of the M3-compatible Ecore metamodel, and not in terms of itself, this situation would violate the conditions of an M3 layer – the metamodeling architecture of MDA does not support any M4 layer.

In UML, the approach taken is to define an *InstanceSpecification* type, which has a *classifier* reference to a *Classifier* instance, as illustrated in the UML infrastructure [253, pg. 54]. Both *InstanceSpecification* and *Classifier* reside within the same metamodel, meaning both domain types and instances of the domain types must reside within the same model instance. The IAML metamodel follows this approach as illustrated in Figure 5.21, and this design is compliant according to the metamodeling architecture of the MDA.

It is still possible to separate the concepts of domain types and domain type instances by separating the models into layers. This approach would be beneficial if the metamodel for domain modelling already exists as an independent metamodel, such as XML Schema or UML class diagrams; or if the domain type instances subsequently defined needed to be reused in model instances outside of RIAs.

### Implementing Multiple Inheritance

As discussed by Meyer [229, pg. 519–568], multiple inheritance is theoretically favourable for object-oriented systems, as real-world objects can simultaneously be many types of common things. The type system of IAML is therefore designed to support multiple inheritance. The main challenge is in its implementation – for example, reliably permitting developer access to identically-named attributes derived through multiple inheritance. It is therefore important to consider how a multiple inheritance system within IAML may be implemented.

It is possible to decompose a multiple inheritance hierarchy into a system that only supports single inheritance, increasing system complexity. As discussed by Crespo et al. [63], these techniques include *emancipation* (all inheritance is flattened into a single monolithic class); *composition* (inheritance relationships are transformed into composition relationships); *expansion* (multiple inheritance relationships are expanded into many single inheritance relationships); and using a *variant type* (all

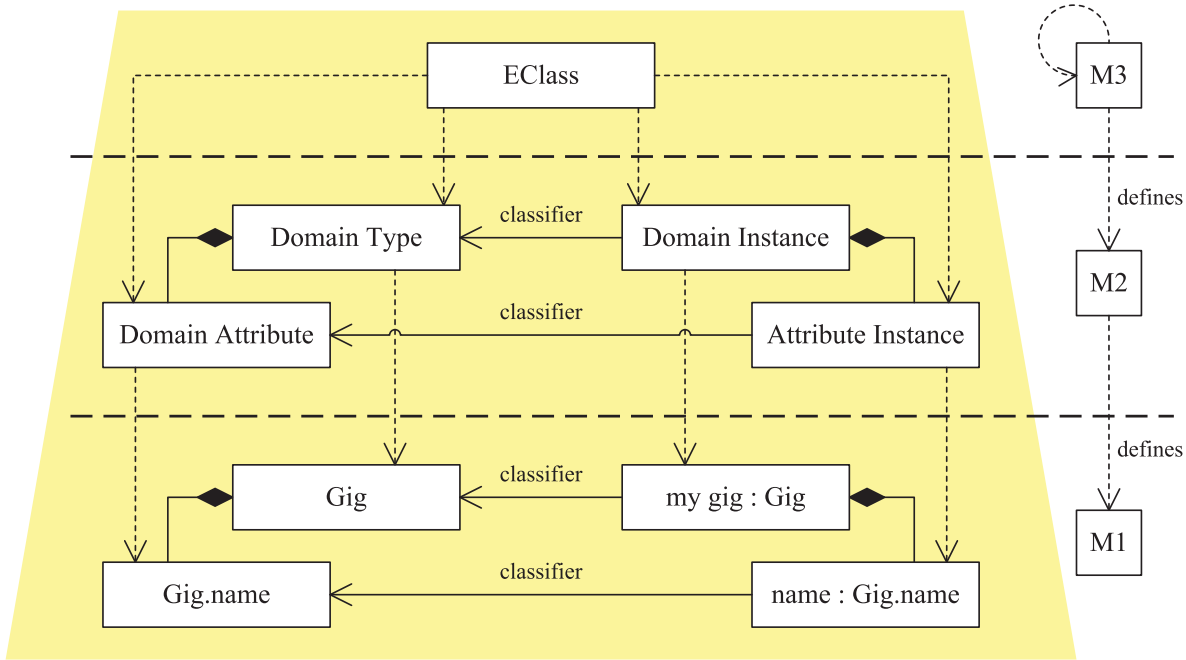


Figure 5.21: Defining instances using references instead of instantiation is MDA-compatible

inherited attributes are collated into a class which uses dispatch based on runtime type).

Both WebML, UML and UWE support single inheritance as part of their domain modelling approach; UML also supports multiple inheritance in UML class diagrams [28]. As UWE reuses much of the UML approach, it seems that UWE can also support multiple inheritance, although multiple inheritance is not considered in any existing work. WebML, on the other hand, only supports single inheritance in defining database models<sup>28</sup>.

### 5.6.3 Domain Iterators

In order to support access to instances of [Domain Types](#), IAML uses the Iterator design pattern [118] discussed earlier in Section 4.6.2 as the structural basis for data access in the language. In particular, all data access is provided through a [Domain Iterator](#), which has explicit references to a source of data ([Domain Source](#)) and a data schema ([Domain Type](#)). This structure is implemented according to the UML class diagram illustrated in Figure 5.22<sup>29</sup>.

In order to access data, the [Domain Iterator](#) contains a single [Domain Instance](#), representing the current instance of the [Domain Type](#) of that iterator. This instance is subsequently composed of a number of [Domain Feature Instances](#), each representing the current value instance of each [Domain Feature](#) within the specified [Domain Type](#). As the iterator is evaluated, accessed and navigated, these instances are populated with the values from the current *instance pointer* of the current result set. The instance data within a particular [Domain Iterator](#) can be reloaded from the specified [Domain Source](#) by calling the *reload* operation within that iterator. A full semantic description of the behaviour of instances, iterators, their properties and operations is provided later in Section I.38.

Instances of [Domain Iterators](#) are stored and accessible according to the *containing scope* of the iterator itself in an identical fashion to the *containing scope* of [Value](#) instances, discussed earlier in

<sup>28</sup>“Each entity is defined as the specialisation of at most one super-entity.” [51, pg. 66]

<sup>29</sup>Within the implementation of IAML, the *select* reference of a [Domain Iterator](#) is implemented as the [Select Edge](#) model element. Similarly, the *schema* reference of a [Domain Source](#) is implemented as the [Schema Edge](#) model element.

Section 5.5.3. For example, multiple [Sessions](#) can access the same [Domain Source](#) using the same [Domain Iterator](#), but as each session is unique, each session has a unique view over the domain source.

### Domain Iterator Query

To access data from a particular data source, a parameterised query<sup>30</sup> must be provided to select which data should be retrieved. If no query is provided, then the iterator will simply return all available instances of that type. An [Domain Iterator](#) query is provided as a single text string, but this does not prevent language extensions from using alternative query definition methods such as graphical queries or filters. For example, the query “name = :name” will select instances of a [Domain Type](#) where the [Domain Attribute](#) named *name* matches the value of the incoming [Parameter](#) named *name*.

One important aspect of SQL is that it only provides a limited range of SQL functions, and these functions may not be provided by particular relational database implementations [186]. Each database vendor also independently defines their own functions, and many common functions – such as MATCH, which may be used to search against text patterns – are not defined in SQL99. Consequently, platform-independent database access must be done through database abstraction layers, such as Propel or PDO for PHP [207, 274].

IAML therefore defines its own query functions which are then translated into the function specific for a particular database. These query functions are discussed later in the documentation of [Domain Iterator](#) in Section I.38.

### Domain Source

A [Domain Source](#) defines the source of a particular [Domain Type](#). In particular, it defines where the data is stored; for example, if it is stored within a database, a file, or stored remotely (for example, an RSS feed). Clients using a [Domain Iterator](#) do not need to know where the source data is stored or accessed, or how it is saved.

Figure 5.23 illustrates a partial model used to define the *Edit Event* page of Ticketiaml. This [Domain Iterator](#) will select a single instance of an *Event* [Domain Type](#) as specified through the [Domain Source](#) connected to the iterator. This iterator uses a specific [query](#), “id = :id” in order to select a specific *Event*, and this value is provided as a [Query](#) to the iterator from an existing [Value](#).

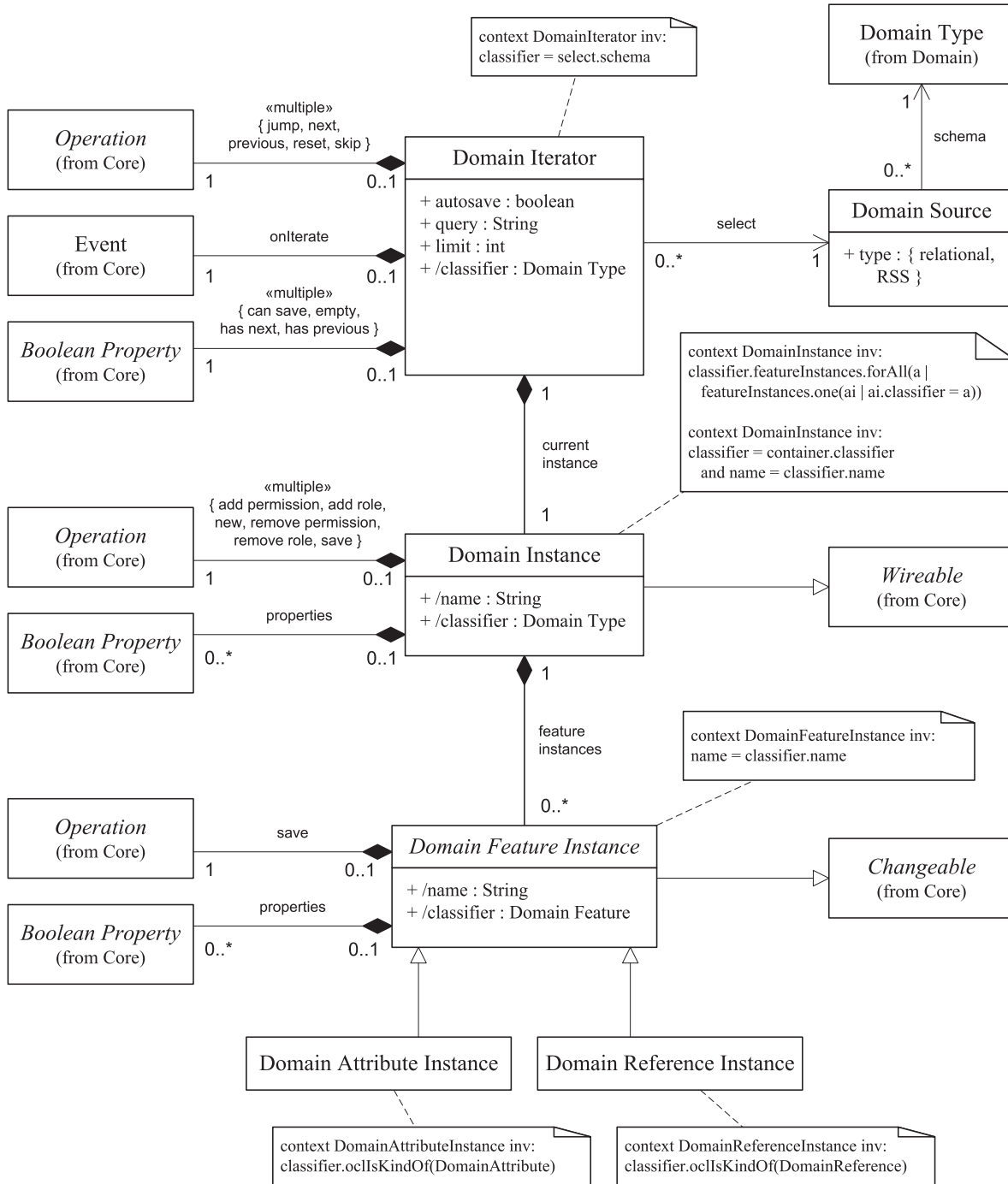
### RSS Feeds

By default, a [Domain Source](#) is assumed to be a local database on the same server as the web application. IAML also supports the use of an external RSS feed as the source of the [Domain Type](#), as specified through the [type](#) attribute of the source. Such a feed will be cached locally and periodically updated<sup>31</sup>. The feed can be manually updated using the [reload](#) operation, or it will be updated automatically after a specified number of seconds (the [cache](#) attribute).

Through model completion, a schema provided by an RSS domain source will automatically include the schema of RSS [290] as its [Domain Type](#). For example, the [Domain Type](#) will include the attributes *title*, *link*, *description*, and so on; however, these attributes will only be populated at runtime if the particular RSS feed provides them. It is up to the developer to keep these attributes

<sup>30</sup> A [Domain Iterator](#) only accepts parameterised statements in order to improve the security of the modelled web applications, as described earlier in Section 5.2.2.

<sup>31</sup> The use of an RSS feed as a [Domain Source](#) of data must not be confused with *providing* an RSS feed through an RSS-typed [Frame](#), as discussed later in Section 5.13.4.

Figure 5.22: UML class diagram for the *Domain Instances* package of IAML

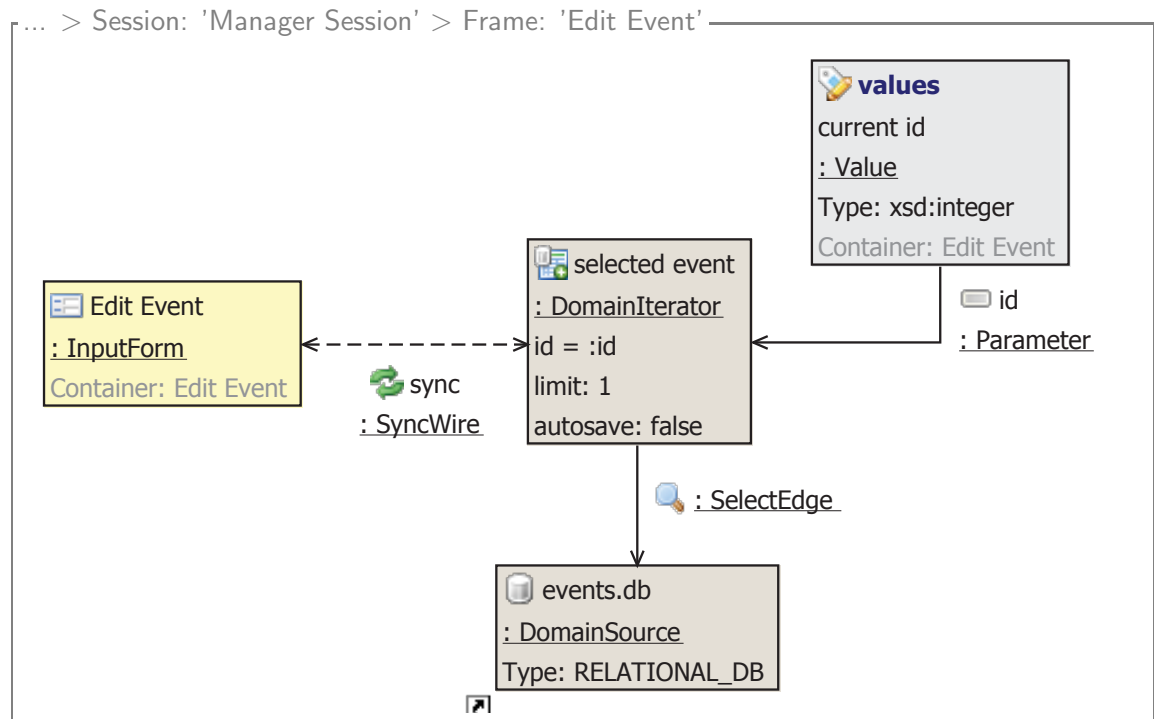


Figure 5.23: Ticketiaml: Selecting an instance of the *Event Domain Type* through a *Domain Iterator* connected to a *Domain Source*

synchronised with other attributes in the schema, for example by using *Sync Wires*. This schema is included automatically using the model completion framework<sup>32</sup>.

### Failure Handler Rules

By associating an *ECA Rule* as a *failure handler* to a *Domain Iterator*, the failure handler will be used to capture exceptions or problems during the evaluation of the iterator, preventing the exception from otherwise crashing the application. This rule is specified as a failure handler by defining the *name* of the rule to “fail”.

If an error occurs during the operation of a *Domain Iterator* – for example, the current *instance pointer* is requested to go out of bounds, or the *Domain Source* is no longer available – then the *failure handler* will be executed, according to the semantics of *ECA Rule* (Section 5.4.3). If the iterator does not have a failure handler defined, then the failure handler semantics of the containing *Scope* are used instead as discussed later in Section 5.11.3.

### 5.6.4 Domain Instances

A *Domain Iterator* contains a single *Domain Instance* representing the currently selected instance of the *Domain Type*. Through model completion, this instance is consequently populated with *Domain Feature Instances* corresponding to each *Domain Feature* defined within the selected *Domain Type*. Each *Domain Iterator* has an associated current *instance pointer*, indicating the position or *cursor* within the evaluated result set. This value can be accessed at runtime through the *currentPointer Value* of the iterator.

<sup>32</sup>Issue 218: Include RSS Type elements automatically in RSS Domain Sources.

A **Domain Iterator** also provides a number of operations and predicates to control navigation over the result set, such as `next`, `previous`, `hasNext`, `hasPrevious` and so on. These methods may modify the *instance pointer*, and are discussed in more detail later in Section I.38.

This **Domain Instance** may then be connected to other model elements using concepts such as **Sync Wires** and **Set Wires**, introduced later in Section 5.9; since a **Domain Iterator** may only ever contain a single **Domain Instance**, these wires may also connect directly to the iterator itself, in order to simplify development. This is illustrated earlier in Figure 5.23, where the selected *Event* is kept synchronised with an **Input Form**, allowing the manager to edit the *Event* instance directly.

By default, a **Domain Iterator** only selects at most one result from the specified **Domain Source**. The `limit` attribute can be used to select many results, rather than just one, making it an iterator over multiple results. This resulting set of results can then be subsequently sorted using the `orderBy` reference. Modifying the `limit` of the iterator does not affect the **Domain Instance** within the iterator, but restricts the maximum number of results that may be iterated over.

### Modifying Domain Instances

A **Domain Iterator** can be used not only for reading from a domain source, but also to modify and subsequently save changes to the domain data. Every **Domain Attribute Instance** contained within its **Domain Instance** can be modified, and once the iterator is saved these changes will be written to the original domain source. An iterator may also automatically save all changes by setting the `autosave` property of the **Domain Iterator** to `true`.

However, some domain sources such as RSS feeds are read-only, and trying to modify such a **Domain Instance** will result in an error. An application therefore should first evaluate the `canSave` predicate of the iterator to check that the iterator can be successfully saved. Alternatively, the failure handler discussed earlier can be used to handle these exceptions.

### New Result Instances

A **Domain Iterator** may also be used to create *new* instances of a **Domain Type** within a particular **Domain Source**, by setting the `query` of the iterator to “new”. If the `autosave` attribute of the iterator is set to `false`, then this new instance will not be committed to the database until the `save` operation is executed – for example, through a **ECA Rule**.

This approach is illustrated in Figure 5.24, which is adapted from the Ticketiaml implementation of the *Signup* page. By connecting this **Domain Iterator** to a **Signup Form** via a **Sync Wire**, a new user can create a new profile directly. As this **Domain Iterator** has its `autosave` attribute set to `false`, the `save` operation must be executed manually.

## 5.6.5 Design Decisions on Modelling Domain Instances

Similarly to the design of **Domain Types**, a number of design decisions were considered during the development of the **Domain Iterator** model elements, in particular with respect to the underlying first-order logic model discussed earlier in Section 5.4.1.

### Domain Attribute Instances should not be Complex Terms

A **Domain Attribute Instance** could be considered as a **Complex Term**; that is, a **Function** operating on a **Domain Instance** that returns the current value of the given attribute. However, this is only true if



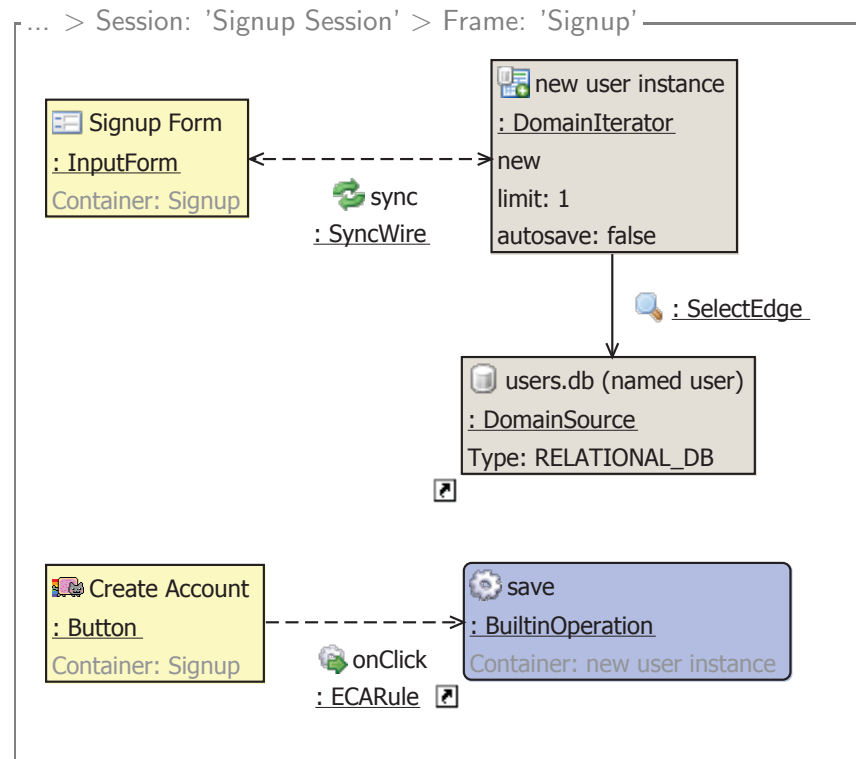


Figure 5.24: Ticketiaml: Creating a new instance of an *Event* using a *Domain Iterator*

the value is read-only. *Domain Attribute Instances* can be modified, and can throw events when they are modified. This conflicts with the meaning of a *Function*, as *Functions* must not change the state of the system.

A *Domain Instance* may still provide both *Domain Attribute Instances* and corresponding *Functions* for each attribute instance – in essence, providing two methods of accessing the same data in two different contexts. However, this violates the principle of *uniform access* discussed by Meyer [229, pg. 57], where there should be no distinction between a module's services and its underlying implementation.

### Attribute instances in *Domain Instances* are not directly *Value* instances

*Domain Attribute Instances* could also be considered as *Values* which is accessible and modifiable. However, it is not desirable for these attribute instances to directly be instances of a *Value*, according to the design philosophy of *Value* as described in Section 5.5.3; in particular, a *Value* cannot contain any operations, predicates or events, as a *Value* represents a *Variable* in first-order logic concepts. As discussed earlier, each *Domain Attribute Instance* contains a *save* operation, a *can save?* predicate and a *onChange* event, making it incompatible with the design philosophy of *Value*.

However, the IAML metamodel specifies that an *Domain Attribute Instance* is also an instance of a *Changeable* in order to provide the *onChange* event to attribute instances. Due to the syntactic sugar introduced in Section 5.4.6 which specifies that all *Changeables* can also be used as *Values*, this means that a *Domain Attribute Instance* eventually inherits the *Value* abstract type.



## Navigation through foreign keys of **Domain Type** instances

Currently in the IAML metamodel there is no way to navigate between **Domain Type** instances through foreign keys, such as through instances of **Domain References**. This remains future work<sup>33</sup>, and a number of potential modelling approaches have already been identified. For example, a **Domain Attribute Instance** may define a **Domain Instance** for each domain object instance referenced via a foreign key. Alternatively, a **Domain Iterator** could contain sub-iterators for each foreign key, which could then be navigated normally.

Nevertheless, it is generally not desirable to support extensive foreign key navigation, if domain modelling is considered in an object-oriented software development viewpoint. The *Law of Demeter* [212] provides a guideline or heuristic for developing high-quality object-oriented programs; in particular, Lieberherr and Holland [212] suggest that objects should only have limited knowledge about other objects, and knowledge should be restricted to “friends” of an object. Navigating over foreign keys should therefore be the exception in a well-designed IAML model instance.

## 5.7 Events

The rationale behind the use of events as part of Event-Condition-Action modelling has been discussed earlier in Section 5.4.3, as it forms an important part of the *Core* metamodel package. Events can also be used to implement lifecycle modelling, as discussed later in Section 5.11. In this section, each possible instance of an **Event** will be briefly introduced; for the full definition of their behaviours, the interested reader is referred to the definition of **Event** in Section I.44.

An **Event** must be owned by a model element which may trigger that event, with this relationship represented as a containment relationship in the IAML metamodel, as this vastly simplifies the proof-of-concept implementation of events. A triggered event does not have access to the previous state of triggering or parent objects; this may be a consequence of not supporting free variables within the core metamodel of IAML. For example, when an *onChange* event is triggered, it is not possible to access the previous value of the changed **field value**. Similarly, when an *onFailure* event is triggered for an **Email**, no failure message is available to the developer. This information has not proven necessary within the current proof-of-concept implementation, but has been noted as an avenue of further research<sup>34</sup>.

### 5.7.1 onChange

The *onChange* event is intended to be fired when the **field value** of a modelled element changes. This is particularly important for visual elements such as **Input Text Fields**, discussed later in Section 5.13; in this case, the *onChange* event is fired when the control both loses the input focus *and* its contained **field value** has been modified since it gained focus. The event is derived from the DOM Level 2 *change* event [334], and is also similar to JavaFX’s *onReplace* event [358].

### 5.7.2 onInput

The *onInput* event is similar to the *onChange* event, but is defined in order to support the richer interfaces provided by RIAs. This event is fired whenever a *user* modifies the displayed value of the displayed element on the client-side, but before it has been committed to the **field value** of the element.

<sup>33</sup>Issue 228: *Permit navigation through foreign keys of instances of Domain Types.*

<sup>34</sup>Issue 198: *Populate Events with event information when events are fired.*

However, if this event is triggered too frequently, the performance of the web application may suffer. Consequently it is not guaranteed that the event will ever fire, or that it will fire regularly (for example, upon every character keystroke) – but this is to guarantee the performance and usability of the web application. This event may be buffered during periods of intense input or performance reasons and to prevent excessive network requests.

### 5.7.3 onAccess

The *onAccess* event is defined for all [Scopes](#) and [Accessible](#) elements, and is triggered when the [Scope](#) is accessed or the element is rendered<sup>35</sup>, or. For example, a control contained within a [Frame](#) will cause *onAccess* to be fired whenever the control is rendered on the frame.

When a [Scope](#) is accessed, the *onAccess* [Event](#) is triggered for its parent [Scope](#) before the event is triggered for the current [Scope](#), unless the current [Scope](#) has is the root [Internet Application](#) element. This cannot be circumvented; it is therefore not possible to prevent the *onAccess* [Event](#) of the root [Internet Application](#) from triggering. This allows parent [Scopes](#) to selectively prevent access to its contained elements, as discussed later in Section 5.11, and is fundamental to the implementation of [Gates](#).

### 5.7.4 onInit

The *onInit* event is defined for all [Scopes](#), and allows the scope to be initialised for the first time. It is triggered when an instance of that [Scope](#) is initialised, and is guaranteed to trigger *before* the *onAccess* event of the [Scope](#) is triggered.

Since *onInit* is *only* called when the scope is initialised, it is guaranteed that the root [Internet Application](#) will only trigger its *onInit* event once throughout the lifecycle of the application. The triggering of *onInit* events also follows a containment hierarchy identical to the triggering semantics of *onAccess* events.

### 5.7.5 onClick

The *onClick* event is defined for all visual elements (see Section 5.13) to describe scenarios where the user clicks a rendered element. This event is once again derived from the definition of the DOM Level 3 *click* event [336].

For composite visual elements, the *click* event will *bubble up*<sup>36</sup> the containment hierarchy of [Visible Things](#), as per the notion of DOM event bubbling discussed earlier in Section 4.2.6 [334]. For visual elements such as [Buttons](#), *onClick* also covers the scenario where the button is clicked using the Enter key, as this simulates a pointer click.

### 5.7.6 onSent

The *onSent* event is only defined for [Email](#) elements (discussed further in Section 5.12.1), and represents the successful *delivery* of a given e-mail. It is important to note that the *onSent* event does not cover a successful *receipt* of an e-mail, but only a successfully delivery.

<sup>35</sup>The *onAccess* event could therefore have been named *onRender*.

<sup>36</sup>Unlike the DOM definition, there is no way to cancel a bubbled event in IAML. Describing this scenario is future work, but a workaround can be provided by using state variables and conditions.

For example, if the e-mail is mistakenly deleted by an intermediary mail daemon, or later deleted by a spam filter, this event may fire but the e-mail is still not received. If the e-mail is lost during delivery, the *onFailure* event may be triggered in the future to signal a failed delivery; that is, an *onFailure* event may occur after an *onSent* event.

It is not possible to guarantee that any given e-mail message has been successfully delivered *and* received by a particular recipient. While *Delivery Status Notifications* (delivery receipts) [237] and *Message Disposition Notifications* (read receipts) [146] can be used to verify a received email, not all e-mail clients support this extension, and the user may disable these notifications for privacy or performance reasons. Another attempt is through the inclusion of *web beacons* – invisible images that are uniquely identified to track user activity, as discussed by Lawton [204] – which may also be disabled for privacy or performance reasons.

### 5.7.7 onFailure

The *onFailure* event is in some ways the opposite of the *onSent* event, as it represents the unsuccessful *receipt* of a given e-mail instance. It is not guaranteed that this event will fire in the case of a delivery failure, unless this failure information is provided to the IAML implementation. For example, if the e-mail is mistakenly deleted by a mail daemon or deleted by a spam filter without any return notification, this event may not fire.

An implementation of the IAML metamodel may allow the definition of an “email timeout”, describing the length of time before a queued e-mail (which has not yet triggered *onSent*) is tagged as a failed delivery and *onFailure* is instead triggered. This would allow the implementation to guarantee that an e-mail will eventually either pass or fail within a specified period of time.

### 5.7.8 onIterate

The *onIterate* event is defined for all *Domain Iterator* model elements, as discussed earlier in Section 5.6.3. It allows the application to update parts of the application when the current result changes; for example, by using “previous” and “next” buttons on a results page. As described later in Section I.38, this event is fired after the instance pointer of the iterator changes, or the contained *Domain Instance* is reloaded.

### 5.7.9 Client-side Events

In IAML, there is no distinction between client-side and server-side events, and events are intended to execute identically on both platforms according to the triggering definitions of the events. For example, if the *field value* of a visual element is modified, then the *onChange* event *must* be triggered for this visual element, even if the element is not currently visible or rendered.

This transparency between client-side and server-side events allows a model developer to focus on the functionality of an element, without needing to keep track of the rendering state of the entire application. This reduces the complexity of the language, and is one of the design goals of IAML. However, this approach places a technical burden on the web application implementation, as now the application needs to emulate the behaviour of the entire application on the server-side if the client-side interface is not being displayed, or cannot be displayed due to browser limitations.

## 5.8 Operations

As discussed earlier in Section 4.3, operation modelling refers to the modelling of the lower-level operations that make up higher-level behaviours, and this modelling may occur through a number of mechanisms. IAML supports modelling behaviours through two model elements, each a subtype of the abstract class `Operation`.

IAML defines a range of built-in `Builtin Operations` which can be used in both composite operations, and directly in the web application itself. This design follows the similar architecture of `Functions`, by defining a range of built-in `Builtin Properties` for certain metamodel elements. For example, the IAML metamodel element `Input Text Field` (introduced later in Section 5.13.2) defines the operations `hide` and `show`, which change the visibility of the text field. A full description of the behaviour of built-in operations within IAML is outside the scope of this chapter; the interested reader is instead referred to Appendix I.

However, it is not possible to define a non-trivial web application without having to express some aspects of complex behaviour and logic; in many cases, this represents the domain logic of the web application. This logic may be expressed independently of the IAML model instance – for example, through the manual extension of the generated web application – but it may be preferable to model this logic directly within the same instance, to obtain model-driven benefits such as code generation and model instance verification.

### 5.8.1 Modelling Complex Behaviour

As an example, consider a web application that needs to `hide` an `Input Text Field` if the value of a separate text field is zero, or `show` the field otherwise. This behaviour may be implemented using only `ECA Rules`, `Parameters` and `Complex Terms` as illustrated in Figure 5.25<sup>37</sup>. Here, an instance of a `Complex Term` used as a `Condition` for one `ECA Rule` is reused as a `Parameter`.

However, there are a number of negative aspects that may arise when using this simplistic approach to model complex behaviour, that need to be considered.

1. Conceptually, complex behaviour would no longer be represented by an `Action`, but would instead be represented by the constraints operating on a particular `ECA Rule` as it applies to a single instance of an `Event`. This has important ramifications: most importantly, it means that an `ECA Rule` instance must support the definition of *many* `Event triggers`, otherwise complex behaviour must be duplicated for each triggering `Event`.

This also means that an `Action` would no longer represent “a specification of parameterized behaviour” as defined earlier in Section 5.4.3, but would instead represent a single form of predefined behaviour that cannot be specified by the model developer. This also means that behaviours could not be externally referenced – if the `onChange` and `onInput` events should perform the same action, for example, the behaviour would need to be duplicated.

2. As the complexity of a particular behaviour increases linearly, the complexity of each `ECA Rule` increases exponentially. For example, consider a behaviour which may execute one of eight different operations based on the evaluation of three boolean input variables. In this example, eight `ECA Rules` would need to be provided, each with a different expression of the same input

<sup>37</sup>This model example also illustrates the use of the XQuery Function `fn:not` in order to implement negation.

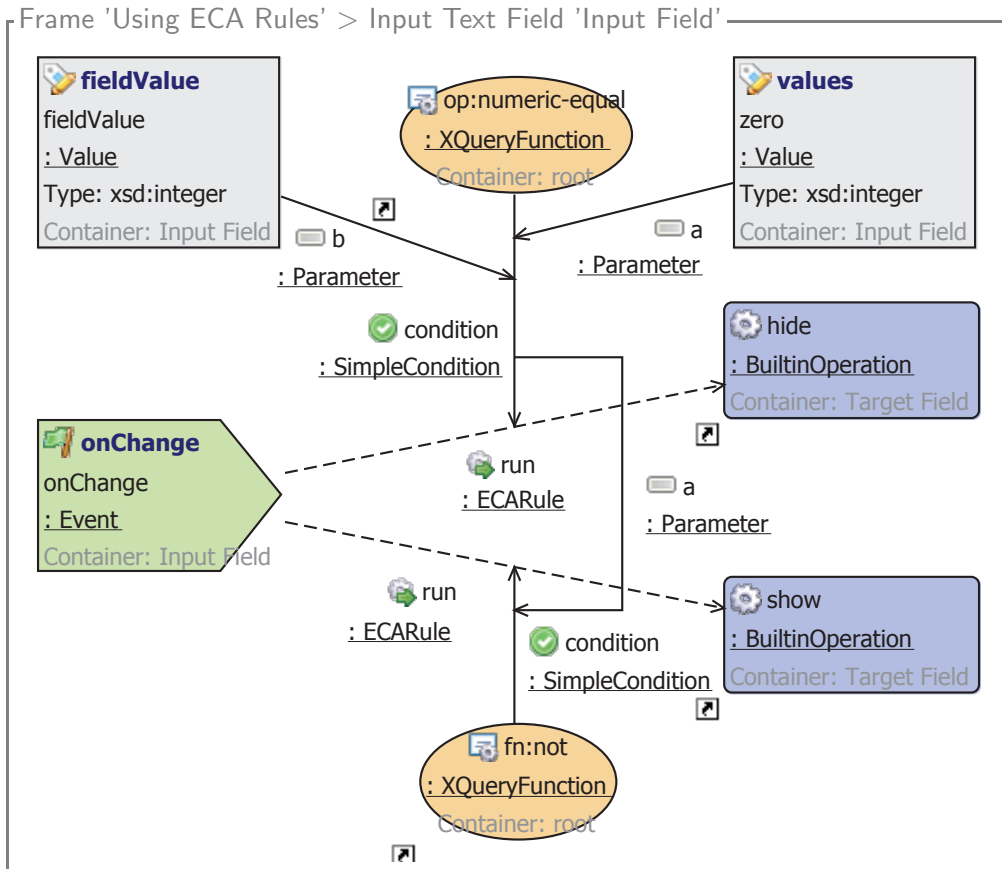


Figure 5.25: Representing complex behaviour within IAML using two ECA Rules

variables. This represents a significant amount of developer effort, and may result in logic errors.

3. The visual complexity of Figure 5.25 is a consequence of the visual notation used, where every model element has a corresponding node or edge. This issue is discussed in further detail later in Section 9.3.3. A new visual notation would need to be proposed, and this notation may need to incorporate grouping related elements, a hybrid textual/visual syntax, or directly supporting event algebra [377] and behaviour composition operators.

For example, an ECA Rule could specify a textual condition “[*op:numeric-equal(text.fieldValue, 0)*]” on the edge, which would implement the intended logic. This approach is extremely similar to the use of OCL within UML [254], and would require the definition of a mapping between model instance elements to references within the textual syntax. However, this approach would vastly simplify the definition of complex conditions on a rule.

4. With this approach it is not possible to specify procedural-based behaviour such as loops<sup>38</sup>, or concurrency-based behaviour such as multithreading. Some procedural behaviour may be emulated using special element wrapper structures – for example, a loop may be implemented by allowing an ECA Rule to re-trigger its triggering Event.

<sup>38</sup>As discussed earlier in Section 3.1.2, Fowler [102] argues that domain-specific languages should not exhibit Turing completeness through general-purpose language concepts such as loops. IAML follows this guideline with the exception of describing Activity Operations, as general-purpose language concepts vastly simplify the implementation of domain logic.

5. Finally, this concept of *event-oriented behaviour* implies that the lowest level of the hierarchical design of the IAML metamodel, as illustrated earlier in Figure 5.2, would be focused on the definition of components. In order to improve the manageability of the complexity of the model instance, it may instead be desirable to support a further hierarchical decomposition into an *operation modelling* layer.

It is therefore more desirable to support modelling complex logic and behaviours within an [Action](#), than on the [Conditions](#) of an [ECA Rule](#). Two approaches to modelling complex behaviour will be considered, with respect to the modelling language development principles discussed earlier in Section 2.7: the reuse of an existing textual expression language; or the reuse of a subset of UML. It is not appropriate to introduce a new form of logic into the IAML metamodel; any approach for modelling complex logic should reuse existing IAML concepts where appropriate.

### Reusing an Existing Textual Expression Language

There are many existing languages for defining expressions and behaviour, and these could be used to define [Operations](#) within IAML. For example, a new model element named [OCL Operation](#) could be added to the model, and contain an OCL operation [252]; similarly, a model element named [Java Operation](#) could support the execution of a method within a Java class.

The main issue with using external languages for defining expressions and operations is of the resulting implementation. For each target platform where the expression must be evaluated, either an instance of that language must be available to evaluate the expression directly, or – if there is no instance of the language in that target platform – the expression must be translated into another form where it *can* be evaluated. For example, an [OCL Operation](#) would require an OCL interpreter for both Javascript and PHP, as discussed later in Section 7.6.3.

### Reusing UML Activity Diagrams

As discussed earlier in Section 4.3.1, UML activity diagrams may be used to model the lower-level object and control flow of an activity. While the same behaviour may be implemented though UML interaction sequence diagrams, Bennett et al. [28, pg. 106] argue that activity diagrams are more useful to model business behaviours. Some lower-level constructs such as arithmetic are however not supported, and it is not possible to consistently translate these models into code. The metamodel specific to UML activity diagrams is also very large; at the time of writing, this included 52 separate modelling concepts, and 31 common behaviours [254].

As the intent of an IAML [Action](#) is based upon the UML *Activity* model element, UML activity diagrams should be well-suited towards the intent of operation modelling within IAML. UML activity diagrams allow for the complex behaviour of the example illustrated earlier in Figure 5.25 to be implemented as an independent *Activity*. In Figure 4.8, the same complex behaviour of Figure 5.25 has been expressed with a UML activity diagram.

It is not desirable to directly use UML activity diagrams within the IAML metamodel due to the size of the UML metamodel. However, it would be desirable to reuse aspects of the UML activity diagram through metamodel restriction, where a smaller subset of the UML activity diagram is reused along with their concepts, and concrete execution semantics are defined for this smaller metamodel. Therefore IAML does not directly include UML activity diagrams, but adapts many of the elements and their relevant visual notations and behaviours.



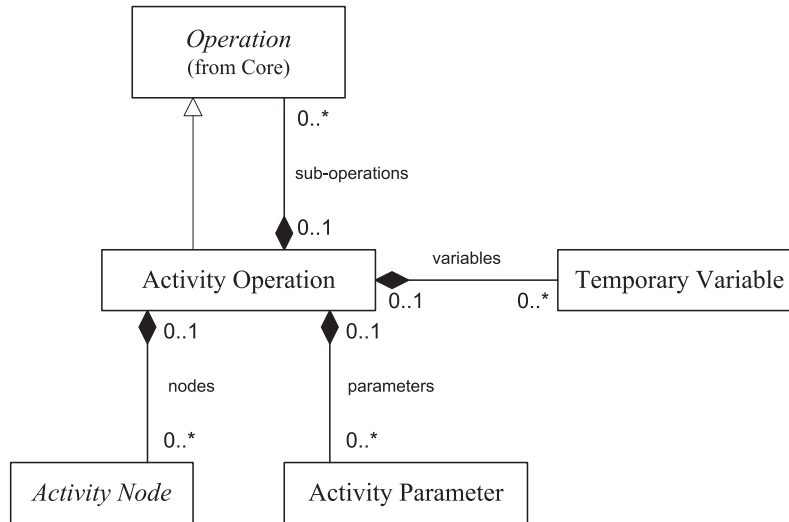


Figure 5.26: UML class diagram for the *Activity* Package: Operations Model

## 5.8.2 Activity Operations

IAML model developers may therefore describe the complex behaviour of web applications using **Activity Operations**, composed of operational elements derived from the UML activity diagram meta-model. At 23 model elements<sup>39</sup>, the metamodel necessary to define **Activity Operations** is much smaller than the UML activity diagram metamodel discussed in the previous section. The UML class diagram of this metamodel is illustrated in Figures 5.28 and 5.29.

As illustrated in Figure 5.26, an **Activity Operation** is defined as the composition of a number of **Activity Nodes**, which are connected via **Execution Edges** and **Data Flow Edges**. **Execution Edges** represent the execution flow of the particular operation, and **Data Flow Edges** represent the flow of data within the particular operation.

Importantly, IAML defines **Activity Operations** as an extension of the IAML metamodel, meaning that this subset of the metamodel may safely be removed if model completion is not used within the development environment. This design requirement represents the motivation for model elements such as **External Value** and **Operation Call Node**, which act as a bridge to core model elements.

An **Activity Operation** allows the complex behaviour defined earlier in Figure 5.25 to be implemented as the **Activity Operation** represented in Figure 5.27; it is important to note the structural similarities between this example model and the UML activity diagram representation of the same behaviour in Figure 4.8. It is also important to note that this figure does not include the single **ECA Rule** instance used to execute the **Action**.

### Execution Flow Nodes

In an **Activity Operation**, the *execution flow* is analogous to the *control flow* of UML activity diagrams [254, pg. 295], and is defined by connecting *execution nodes* with **Execution Edges**. A full discussion of each subtype of **Execution Node** are discussed later in Appendix I, but a UML class diagram for these nodes are illustrated in Figure 5.28.

<sup>39</sup>The abstract classes **Execution Node** and **Data Flow Node** described in Figures 5.28 and 5.29 are not included in Appendix I, but are introduced here as empty interfaces in order to simplify the class diagram structures.



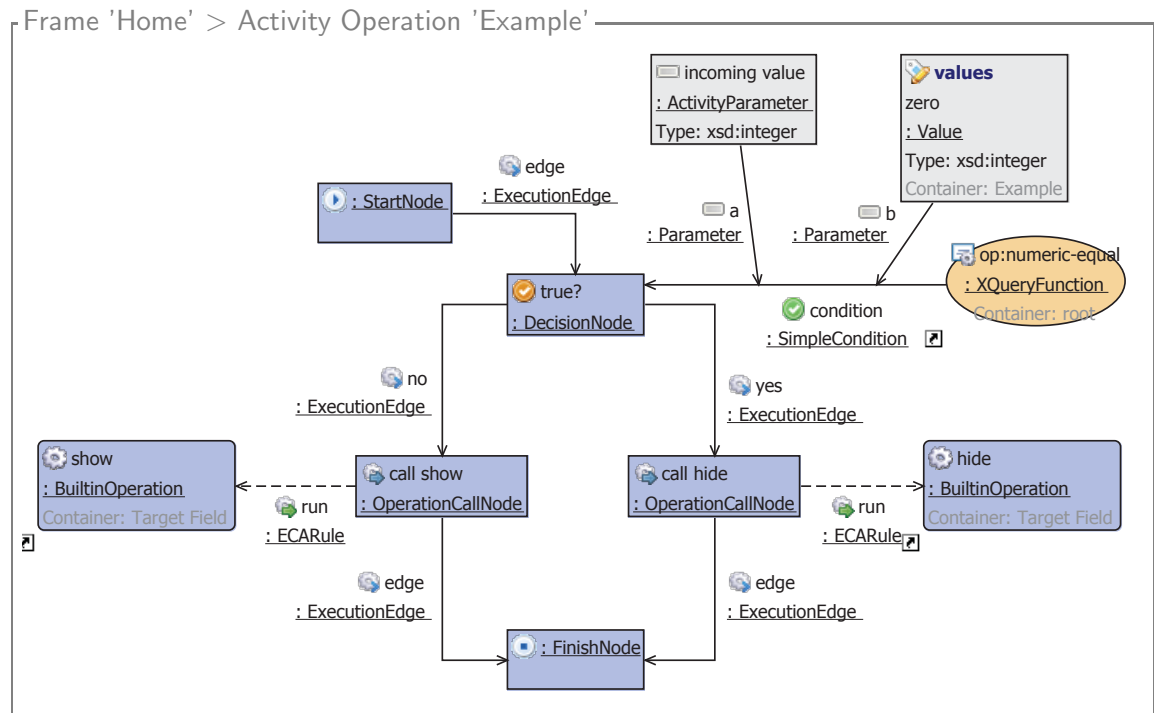


Figure 5.27: The same complex behaviour of Figure 5.25 expressed with an Activity Operation

Decision Nodes and Operation Call Nodes both support the concept of *failure edges* and *success edges*, which apply to outgoing Execution Edges from that node. For each node, all outgoing Execution Edges with a name beginning with the characters *n* or *f*<sup>40</sup> represent the *failure edges*. All other Execution Edges from each node represent the *success edges*.

Activity Operations also support the concept of multithreading through the Split Node and Join Node model elements, which are conceptually similar to the *ForkNode* and *JoinNode* model elements of UML activity diagrams [254, pg. 295–417]. In a similar fashion to UML, modelled concurrency within IAML does not necessarily imply a distributed implementation; for example, an implementation of IAML may group these threads together into a single thread, and execute them sequentially in a manner which preserves its behaviour.

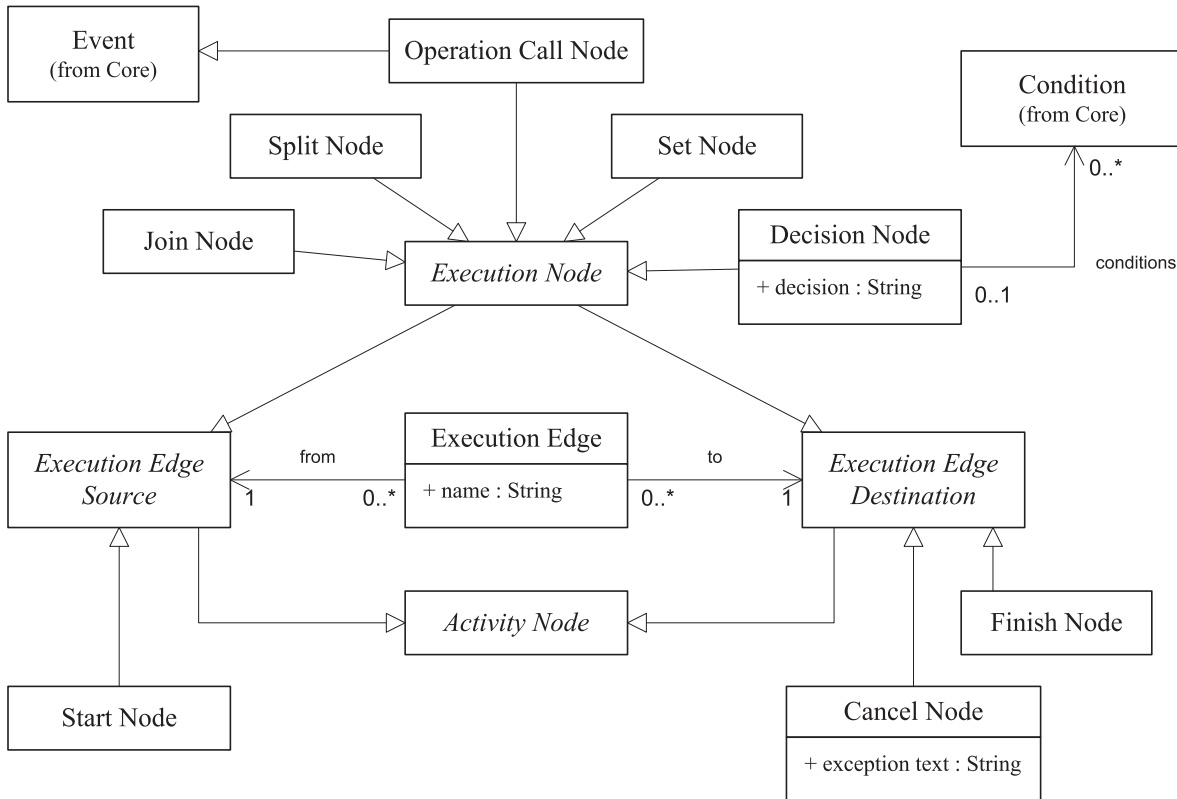
Multithreading is a fairly novel concept within RIAs, as Javascript does not internally support multithreading<sup>41</sup>, with the exception of recent extensions such as *WorkerPool* through Google Gears [129]. A modelling language with concurrency support should also define constructs to control this concurrency, such as access locks or *volatile* properties, or defining the semantics of multithreaded access to shared properties. However, these constructs fall well outside the scope of the *Basic RIA* requirements, as discussed earlier in Section 5.1.1, and adding these constructs to IAML remains future work.

### Data Flow Nodes

In an Activity Operation, the *data flow* represents the *object flow* of UML activity diagrams [254, pg. 295], and is defined by connecting *data flow nodes* with Data Flow Edges. This data flow graph is

<sup>40</sup>This definition covers the most common ways of specifying a negative result in English, including *n*, *no*, *fail* or *false*.

<sup>41</sup>Within a web browser, each window has a single thread for executing Javascript; even AJAX callbacks must wait to be processed by this thread.

Figure 5.28: UML class diagram for the *Activity Package: Execution Model*

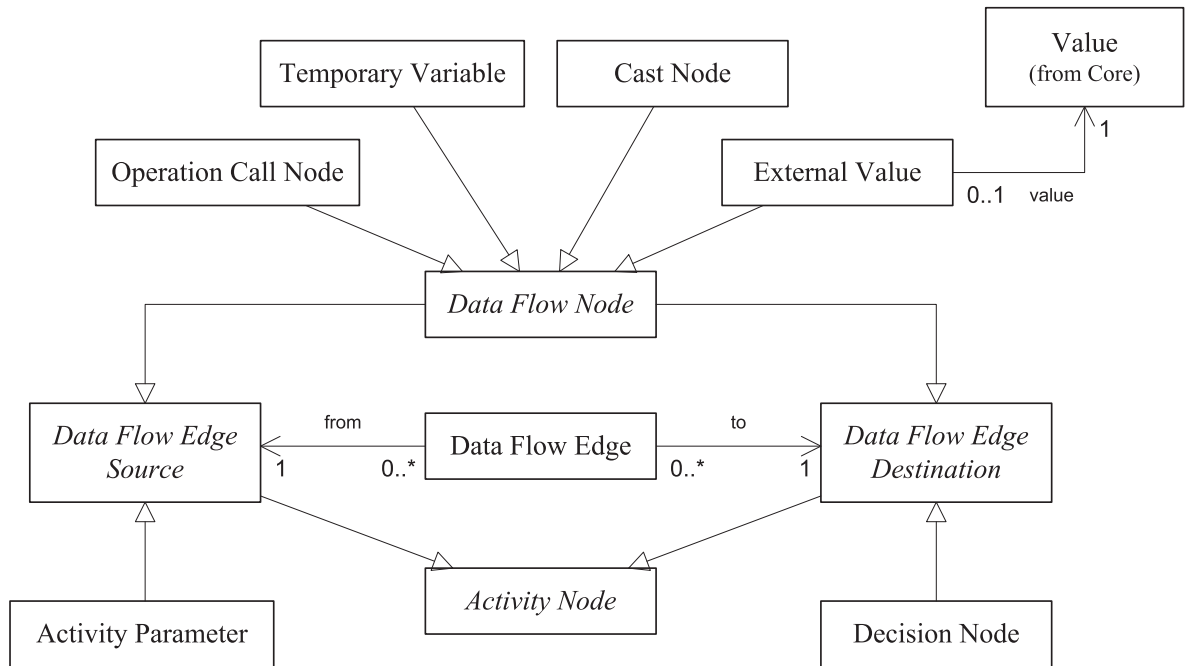
similar to a *data flow diagram* [278, pg. 311], except that the data flow relationships are not transitions themselves. A full discussion of each subtype of *Data Flow Node* are discussed later in Appendix I, but a UML class diagram for these nodes are illustrated in Figure 5.29.

*Activity Parameters* allow an *Activity Operation* to specify named parameters, allowing the operation to be reused in other contexts. If an *Activity Operation* is executed as an *Action*, any incoming *Parameters* to the connecting *ECA Rule* are passed as *Activity Parameters* to the *Activity Operation*.

### 5.8.3 Activity Predicates

In an analogous fashion to allowing a model developer to describe complex behaviour using *Builtin Operations* and *Activity Operations*, IAML allows the model developer to describe complex conditions using *Builtin Properties* and *Activity Predicates*. A full description of each *Builtin Property* is provided in Appendix I.

The *Activity Predicate* model element is defined in a similar fashion to the *Activity Operation* model element, as illustrated in Figure 5.30. However, the semantics of the *Finish Node* and *Cancel Node* elements are modified within the context of a *Activity Predicate*. If the execution flow of an *Activity Predicate* arrives at a *Finish Node*, the *Predicate* is taken to be *successful*. Similarly, if the execution flow of an *Activity Predicate* arrives at a *Cancel Node*, the *Predicate* is taken to be *unsuccessful*.

Figure 5.29: UML class diagram for the *Activity Package*: Data Flow Model

### 5.8.4 Replacing the Default Behaviour of Model Elements

Throughout the definition of the IAML metamodel, all model element operations are defined as instances of the *Operation* abstract class; at runtime, these empty references are replaced with instances of *Builtin Operations* through model completion rules. This design is intentional, as *Activity Operations* – also subtypes of *Operation* – may replace these builtin operations to replace default functionality. For example, it would be possible for a developer to replace the logic of the *update* operation of *Input Text Field* to also trigger additional functionality, or to prevent the text field from ever being updated through that operation. Similarly, *Builtin Properties* may be replaced with *Activity Predicates* to redefine the builtin behaviour of element properties, as both are subtypes of *Predicate*.

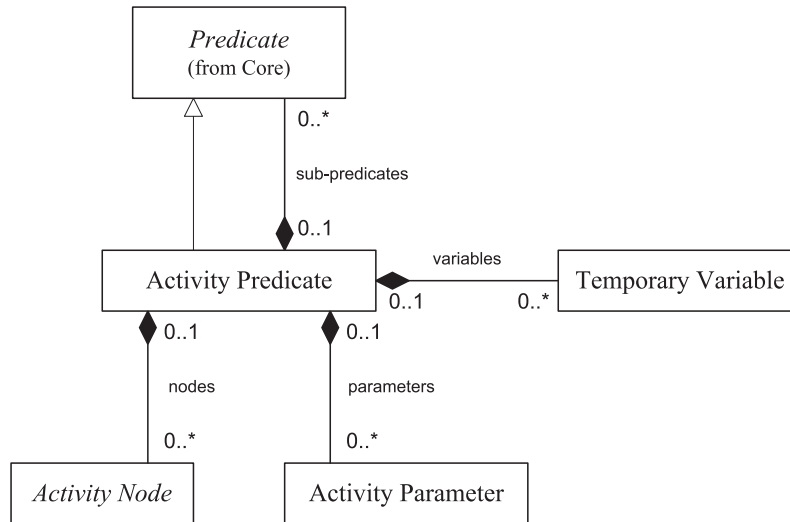
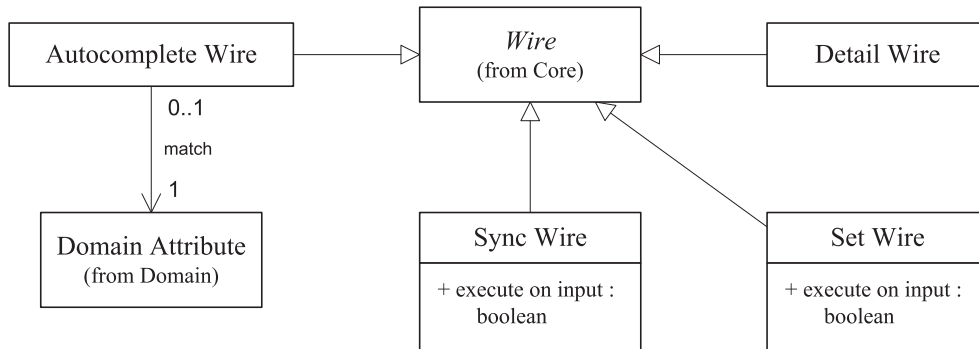
## 5.9 Wires

As described earlier in Section 5.4.5, common web application design patterns are implemented as instances of *Wires*, which are then used to complete the model instance according to the design of the *Wire* through model completion. Each different design pattern is implemented as a subtype of the *Wire* abstract class, illustrated by the UML class diagram in Figure 5.31.

Depending on the design pattern behind them, *Wires* may either be unidirectional or bidirectional. A unidirectional *Wire* has both a *source* and a *target*; whereas a bidirectional wire simply has *targets*. In some cases, it may be possible to replace a bidirectional wire with two unidirectional wires<sup>42</sup>.

It is important to note this requirement as the distinction between *Actions* and *Wires*; an *Action* has a defined functionality and behaviour defined as part of its instance, whereas a *Wire* is an abstract description of higher-level behaviour, which is later implemented by *ECA Rules* through model completion. A *Wire* instance will *not affect* the generated application without first performing model completion on the model instance.

<sup>42</sup>For example, replacing a bidirectional *Sync Wire* with two unidirectional *Set Wires*.

Figure 5.30: UML class diagram for the *Activity* Package: Predicates ModelFigure 5.31: UML class diagram for the *Wires* package of IAML

It is also important to note that *Wires* are a distinct concept from *ECA Rules*. *ECA Rules* specify the actions that a specific event may conditionally execute when the event is triggered; whereas *Wires* specify higher-level behaviours which are implemented through many individual model elements. For example, the implementation of a single *Sync Wire* will usually include at least four *ECA Rules* between the two connecting elements, as discussed later in Section I.94.

In this section, four different *Wires* will be discussed. The inference rules necessary to complete the functionality of each *Wire* will not be discussed here due to lack of space, but is rather discussed in Appendix I. A summary of the model completion rules used to implement each *Wire* is provided later in Table 7.4.

### 5.9.1 Sync Wire

As discussed earlier in Section 4.9, one of the most common design patterns in web applications is the need to keep the values of two elements synchronised. A *Sync Wire* can be used in situations such as keeping user interface forms synchronised across different interfaces; or for synchronising both a client-side interface and its server-side representation within a database.

An element may be connected to any number of *Sync Wires*; it is therefore possible that a *Sync Wire loop* may exist within a web application. This is allowable within IAML, and it is a requirement of the model completion and code generation frameworks to implement *Sync Wires* in such a way that

the application will not result in any infinite loops.

An important feature of a **Sync Wire** is to also keep the *children* of two elements synchronised as well. That is, if an **Input Form** with three contained text fields is connected to an empty **Input Form**, the **Sync Wire** will automatically populate the second **Input Form** with text fields in order to keep these forms synchronised<sup>43</sup>. These generated text fields will then subsequently be connected to the text fields within the original form using additional **Sync Wires**.

An example of a **Sync Wire** is illustrated earlier in Figure 5.24; here, an instance of a *named user* may be modified through an **Input Form**. The behaviour and functionality of a **Sync Wire** is implemented through model completion rules, and the interested reader is referred to the detailed decomposition of this process in Appendix B.

### 5.9.2 Set Wire

This unidirectional wire focuses on another common design patterns in IAML models; the need to keep the value of one element updated with the value of another element, but not vice versa. A **Set Wire** can be used in situations such as keeping a client-side interface updated with changes in its server-side representation within a database, or for keeping a local database synchronised with a remote one. An example of a **Set Wire** is illustrated later in Figure 5.32. In this model, a **Domain Iterator** is used to populate the contents of an **Iterator List** in a read-only fashion, preventing modifications to the underlying data from changes in the **Iterator List**.

Similar to a **Sync Wire**, a **Set Wire** may also be used to construct scenarios of looping **Set Wires**, and these two wire types may be linked together. A **Set Wire** will also automatically keep the *children* of two elements synchronised in a unidirectional manner; a **Set Wire** will generate **Labels** instead of the **Input Text Fields** generated by a **Sync Wire**.

### 5.9.3 Detail Wire

As discussed earlier in Section 5.6.3, a **Domain Iterator** may be used to iterate over the contents of a data source. This **Domain Iterator** can be connected to a user interface using a **Set Wire**; for example, if a **Domain Iterator** is connected via a **Set Wire** to an **Input Form**, upon model completion the form will contain a summary of labels of the current instance within the **Domain Iterator**.

However, it is often the case that a simple interface is necessary to navigate quickly over a result set, but this interface is later expanded into a full interface that provides more detail for a selected instance. The unidirectional **Detail Wire** represents a way of “zooming in” to a particular instance at runtime.

While the generated *target view* is normally connected to the source **Domain Iterator** via a **Set Wire** – that is, the detailed view is only provided in a read-only fashion – it may be desirable to instead replace this wire with a **Sync Wire**. In this way, the *target view* becomes a way to *update* the details of a selected instance as obtained via an iterator.

#### Example

The use of a **Detail Wire** is best illustrated through an example from the Ticketiaml example application. In Figure 5.32, a **Detail Wire** is connected from a **Ticket Domain Iterator** to a **View Ticket Frame**.

<sup>43</sup>It is not desirable for this to always occur; for example, password fields in a database should *not* be displayed to anonymous users. IAML supports this scenario by designing *overridable elements*, as discussed later in Section 7.5.2.

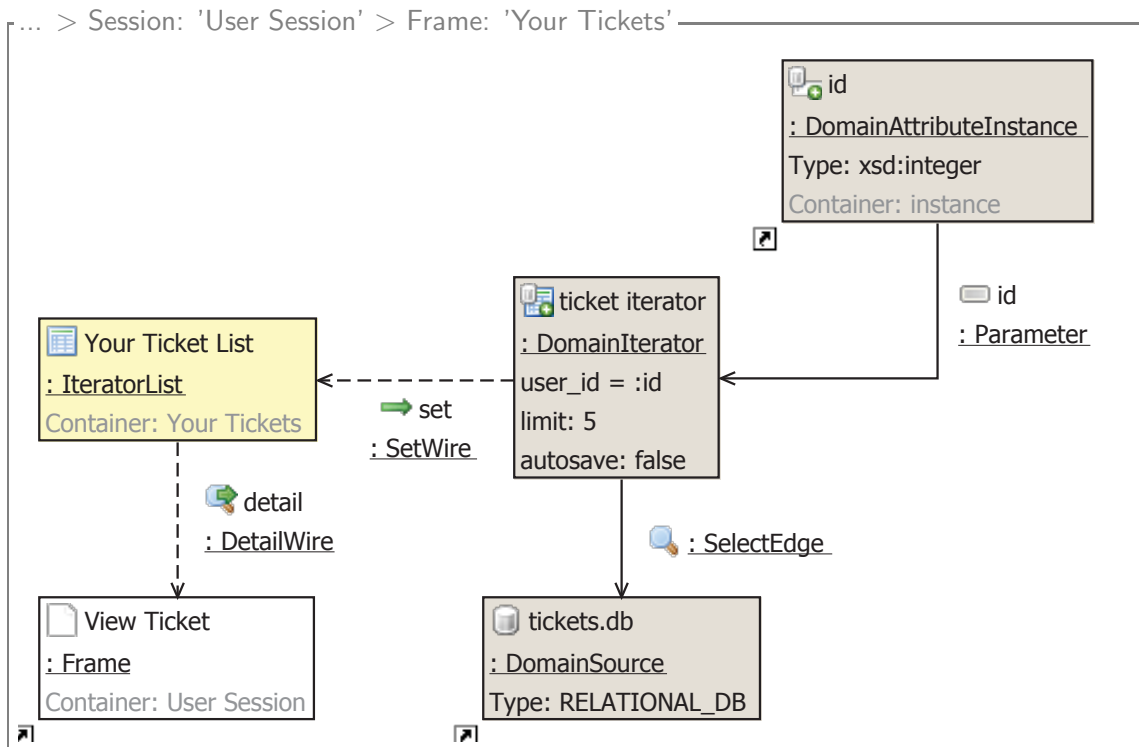


Figure 5.32: Ticketiaml: Connecting a [Detail Wire](#) to a [Domain Iterator](#)

This will allow the user to view more details about each selected *Ticket* instance illustrated through the [Iterator List](#). In particular, model completion will have the following effects:

- The *Your Ticket List* [Iterator List](#) will now include a [Button](#). When a user clicks on this [Button](#), the user will be redirected to the *View Ticket* [Frame](#) (the *detail frame*). As expected, this behaviour is subsequently implemented through an [ECA Rule](#).
- The *View Ticket* [Frame](#) will include an [Input Form](#) to view the details of the selected *Ticket*. This form will be populated with the same instance that the user selected in the first [Frame](#), by defining a new [Domain Iterator](#) selected via a [query](#) based on an incoming [Query Parameter](#), and connecting this iterator to the form via a [Set Wire](#).

#### 5.9.4 Autocomplete Wire

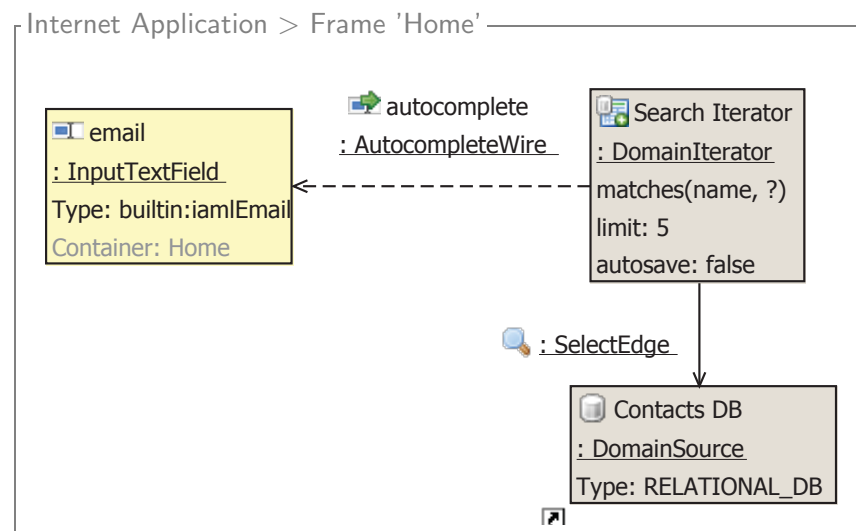
*Autocomplete* allows a user to complete an input field by performing a quick inline search based on a search query, which occurs asynchronously without the user having to leave the current page. One of the motivating use cases for IAML was to support *autocomplete*<sup>44</sup>, and this use case has already been discussed by Wright and Dietrich [368]. IAML consequently defines the [Autocomplete Wire](#) model element to support this common design pattern.

Autocomplete was one of the first well-known features of AJAX websites, as illustrated in Figure 5.33 by the Gmail web application. A search box allows the user to navigate through a large address book by typing in the first few characters of a name, e-mail address, or both. The search results pop up underneath the input text, and can be navigated by using the keyboard. Once selected by

<sup>44</sup>Use Case 69: *Autocomplete*.



Figure 5.33: An instance of autocomplete in Gmail

Figure 5.34: Autocomplete implemented in IAML using an [Autocomplete Wire](#)

the keyboard or mouse, the resulting address is inserted into the input text field, and another address can then be searched.

### Example

Autocomplete behaviour may be implemented within IAML using an [Autocomplete Wire](#) as illustrated in Figure 5.34. Here, a [Domain Iterator](#) is specified as the source, and an [Input Text Field](#) as the target. This target field will be updated with the value of a selected attribute once it has been searched for and selected by the user. The [match](#) reference of the [Autocomplete Wire](#) is set to the “name” [Domain Attribute](#) of the underlying [Domain Type](#).

The resulting application generated for this example model is dependent on the implementation of the modelling language, code generation framework, and its runtime libraries. With respect to the proof-of-concept implementation of IAML discussed later in Chapter 7, this example model will generate an interface similar to Figure 5.35. Here, the user has entered in the search query “J”, and a list of three matching results have been displayed; if the user clicks on one of these, the resulting e-mail address will be inserted into the *email Input Text Field*.



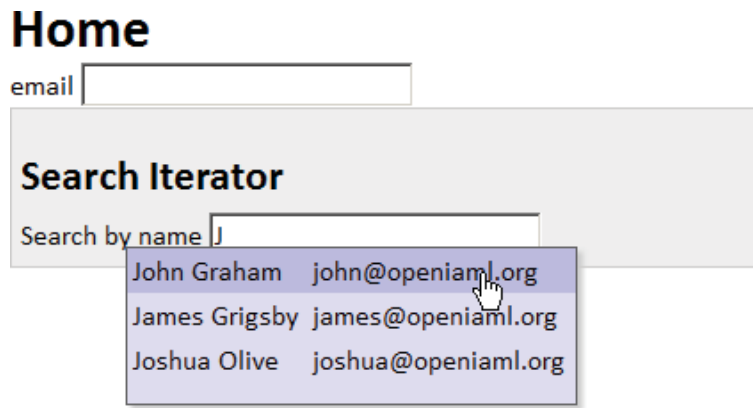


Figure 5.35: The user interface for an [Autocomplete Wire](#), as generated by the proof-of-concept implementation of IAML

## 5.10 Users and Access Control

As discussed earlier in Section 4.8, both users and access control are important domain concepts for web applications. A modelling language for RIAs should not only support the design of secure applications, but also simplify common scenarios involving users and their roles and permissions.

### 5.10.1 Roles and Permissions

Based on the four access control models proposed earlier in Section 4.8, IAML uses the Role-Based Access Control (RBAC) model as described by Sandhu et al. [293]. However, the restriction that users may only possess permissions via a role is relaxed; a user may obtain *any* permission regardless of roles. This approach is taken to increase flexibility; for example, one identified use case for RIAs allows a web application to assign specific permissions to individual users<sup>45</sup>, bypassing roles altogether. IAML defines the metamodel elements [Role](#) and [Permission](#) to represent the RBAC concepts of roles and permissions respectively.

#### Definitions

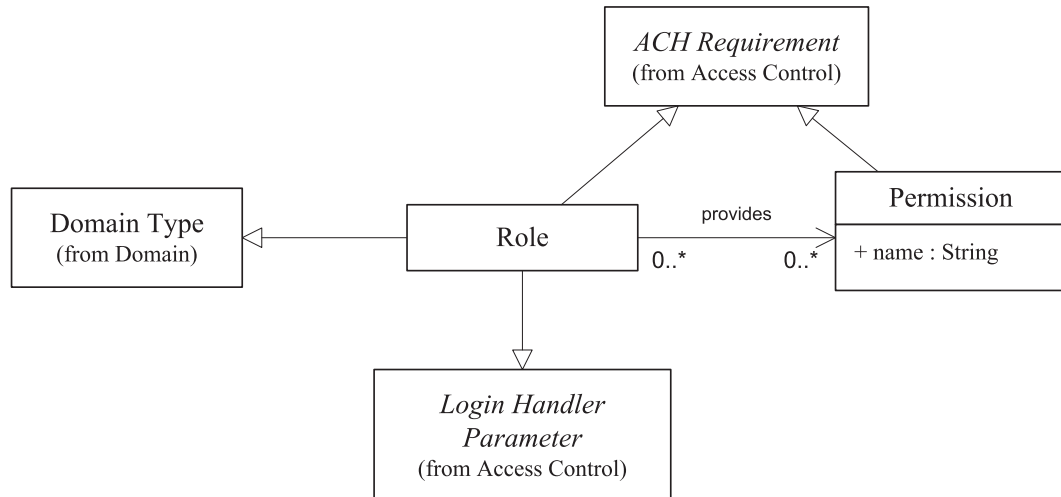
Ceri et al. [52] argue that while some user information is generic – for example, e-mail addresses and passwords – other user information is domain-specific. User profile modelling is therefore very similar to the existing domain modelling approach supported by [Domain Types](#), as discussed earlier in Section 5.6.1.

IAML therefore reuses these domain modelling concepts to support user modelling as illustrated in Figure 5.36. In particular, a [Role](#) is a subtype of a [Domain Type](#); the [Domain Attributes](#) of the [Role](#) represent the *profile attributes* of that role; and [Roles](#) may be defined in a multiple inheritance hierarchy in order to define a *role hierarchy*.

In order to support the common user modelling functionality as described above, all [Roles](#) in a model instance inherit a *default role* containing default user attributes, such as an `iamlEmail`-typed [Domain Attribute](#) named “email”, and a `string`-typed [Domain Attribute](#) named “password”<sup>46</sup>. By

<sup>45</sup>Use Case 36: *User Content Access Control*.

<sup>46</sup>This datatype is typed to `string` until the datatype `iamlPassword` is defined, as discussed earlier in Section 5.5.2.

Figure 5.36: UML class diagram for the *Users* package of IAML

defining a default role, model developers do not need to manually define user profile attributes that are common to most web applications.

### User Instances

As user modelling is based upon domain modelling, IAML reuses the concepts of data access in Section 5.6.3 for defining the semantics of user instances. That is, user information may be stored in any valid [Domain Source](#) (such as a relational database), and a [Domain Iterator](#) may be used to select valid *users* in a web application according to a specific [Role](#)<sup>47</sup>. A [Domain Iterator](#) may also be used to create new user instances.

### 5.10.2 Login Handlers

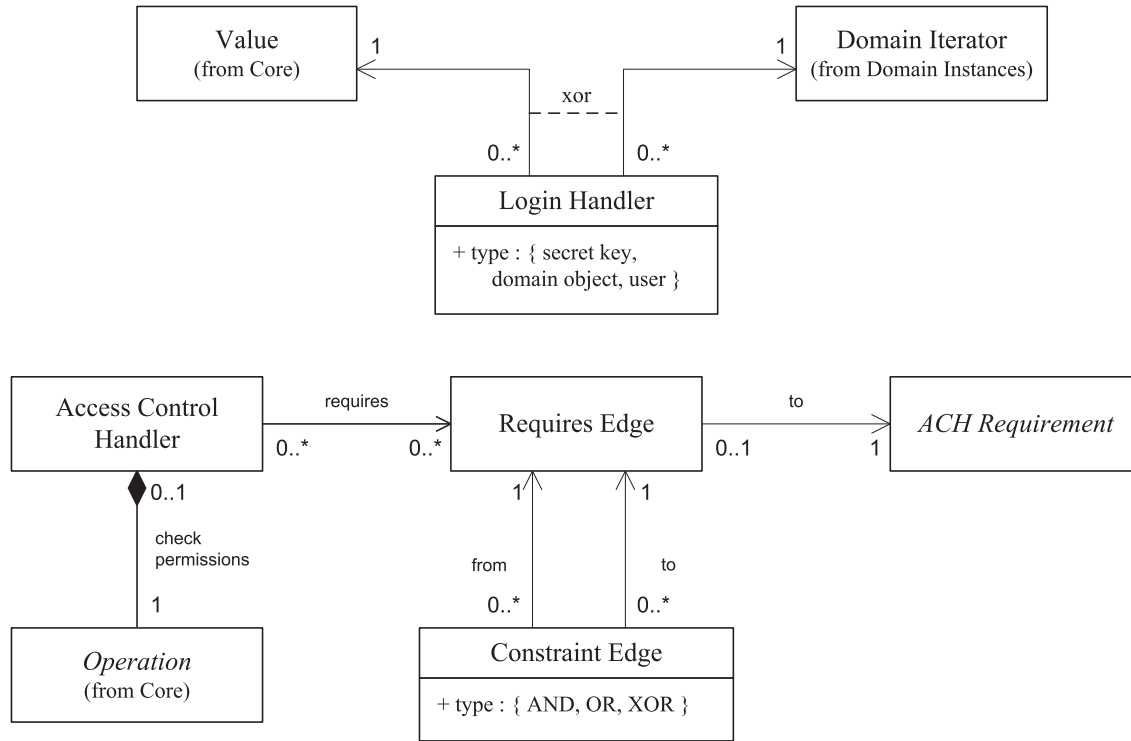
While it is possible to use [Domain Iterators](#) and [Gates](#) directly to prevent access to [Scopes](#), access control is a very important aspect of web applications, and there should be modelling support for these common scenarios. A [Login Handler](#) can be used to prevent access to a [Scope](#) based on an incoming requirement.

While a [Gate](#) also prevents access to a [Scope](#), the key difference is that a [Login Handler](#) provides most of the repetitive scaffolding for common authentication scenarios – such as user logins, or passwords – whereas with a [Gate](#) this functionality must be implemented manually by the model developer. A [Login Handler](#) controls access through two mechanisms, as illustrated in the class diagram in Figure 5.37: secret keys obtained from [Values](#), and domain objects and users (instances of [Domain Types](#) and [Roles](#), respectively) obtained from [Domain Iterators](#).

Similar to a [Wire](#), the functionality of a [Login Handler](#) is implemented through *model completion* so a developer may modify the generated behaviour. In particular, a [Login Handler](#) will generate login and logout pages, and add [Gates](#) or [ECA Rules](#) to redirect the user to these [Frames](#) if the current user does not satisfy the requirements of the handler. These model completion rules are discussed in further detail in Section I.64.

A *secret key* [Login Handler](#) requires that the user provides a single *password*. This is the simplest

<sup>47</sup>In this case, the [Schema Edge](#) from the [Domain Source](#) will point to a [Role](#) rather than a [Domain Type](#).

Figure 5.37: UML class diagram for the *Access Control* Package

form of a [Login Handler](#) which does not require a valid user instance. Alternatively, a *domain object* [Login Handler](#) requires that a valid instance of a [Domain Type](#) exists, as specified by the incoming parameters of the [Login Handler](#). This parameter must be an [Domain Instance](#) or [Domain Iterator](#).

### 5.10.3 Access Control Handlers

An [Access Control Handler](#) may be used to restrict user access to a [Scope](#) to a particular set of [Roles](#) and/or [Permissions](#), and is also implemented through the model completion rules discussed later in Section I.1. An [Access Control Handler](#) is similar to the [Login Handler](#), except that a [Login Handler](#) only considers a single access requirement, and may create login and logout pages through model completion; an [Access Control Handler](#) is only focused on the composition of complex role and permission requirements, and the two model elements may be used simultaneously. The class diagram for the model elements necessary to implement an [Access Control Handler](#) is also illustrated in Figure 5.37.

An [Access Control Handler](#) is also similar to a [Gate](#) (Section 5.11.4), except that [Gates](#) are only focused on satisfying incoming [Conditions](#). A [Gate](#) may be used in the implementation of an [Access Control Handler](#) (i.e., through model completion), except that the model developer would have to implement their own [check permissions Operation](#) manually.

### Example

The [Login Handler](#) and [Access Control Handler](#) model elements may be used together to implement complex access control requirements; in particular, the [Login Handler](#) provides functionality to allow users to login, and the [Access Control Handler](#) provides functionality to check the roles and permissions of the current user.

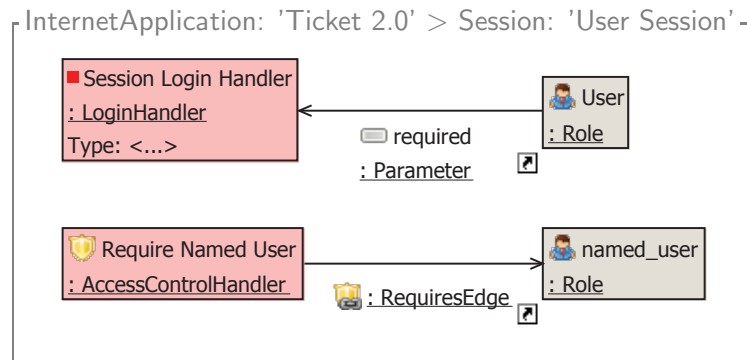


Figure 5.38: Ticketiaml: Protecting access to a [Scope](#) via an [Login Handler](#) and [Access Control Handler](#)

This pattern has been used in the Ticketiaml example to implement access control for application users, as illustrated in Figure 5.38. The [Login Handler](#) ensures that a *User* has logged on, and the [Access Control Handler](#) ensures that the *User* also possesses the *named\_user Role*.

#### 5.10.4 OpenID

While password authentication is a common web application scenario for user authentication, more recently the concept of *decentralised authentication* has become popular, and in particular the OpenID standard [281]. OpenID allows for users to control their own identity verification, and this improves both reliability, security and privacy. Modelling OpenID verification is an important use case<sup>48</sup> of Rich Internet Applications.

Web applications that use OpenID do not need to store sensitive personal information like e-mails or passwords in order to perform identity verification, but can instead store a single URL which will independently perform this service. OpenID only handles the verification of identities rather than profile data; this decision was made by design to reduce the scope of the project. The OAuth standard [145] may be utilised to support user profile management.

On a separate level, OpenID authentication can be considered as merely a *specific type* of authentication; other types of authentication include Google Accounts or Microsoft Passport, and it should be simple to include these as separate authentication protocols. Furthermore, it may be possible to unify all these different authentication methods into a single “authentication” datatype, such as the datatype `iamlIdentity` proposed earlier in Section 5.5.2.

It is important to let the developer decide which types of authentication methods they support, rather than simply assuming that the developer would want all authentication types. For example, OpenID does not provide identity management, and is easier to subvert or modify than a Google Account or a Microsoft Passport identity due to its decentralised nature; consequently there may be some scenarios of authentication where the developer would consider OpenID identity verification as too insecure.

#### Rationale behind the type `iamlOpenIDURL`

Many options were considered in methods of representing OpenID authentication. One option was to define a particular type of [Domain Attribute](#) with an [isSecure](#) attribute which would represent authen-

<sup>48</sup>Use Case 50: *Single Sign-In Solutions*.

tication attributes. However this would mean OpenID authentication could only apply to instances of **Domain Types**.

Another option was to define a subtype of **Gate** – such as an **OpenID Gate** model element – which would restrict access to a **Scope** if a valid OpenID was not provided to the gate. However this approach is too platform-dependent – this would mean that each separate authentication method would have to have a separate **Gate** subtype within the IAML metamodel.

The decision was made to extend the existing datatype framework and provide a new derived type named `iamlOpenIDURL` to represent valid OpenID instances. This meant that much of the existing gates and validation frameworks could be reused; for example, a **Gate** requiring a valid `iamlOpenIDURL`-typed **Value** instance is one implementation of the proposed **OpenID Gate**.

This approach may break an assumption that cast instances can always be recast back to an original type. One can consider this as a digital signature on the instance type; if the signature is lost by converting the instance into a different format, then the digital signature (and consequently validity) of the original instance is lost.

## 5.11 Scopes

As discussed earlier in Section 4.7, the ability to model lifecycle and handle lifecycle events may be a useful technique in the development of RIAs. IAML supports lifecycle modelling by defining *scopes* through the abstract model element **Scope**, and assigning various lifecycle events to these scopes.

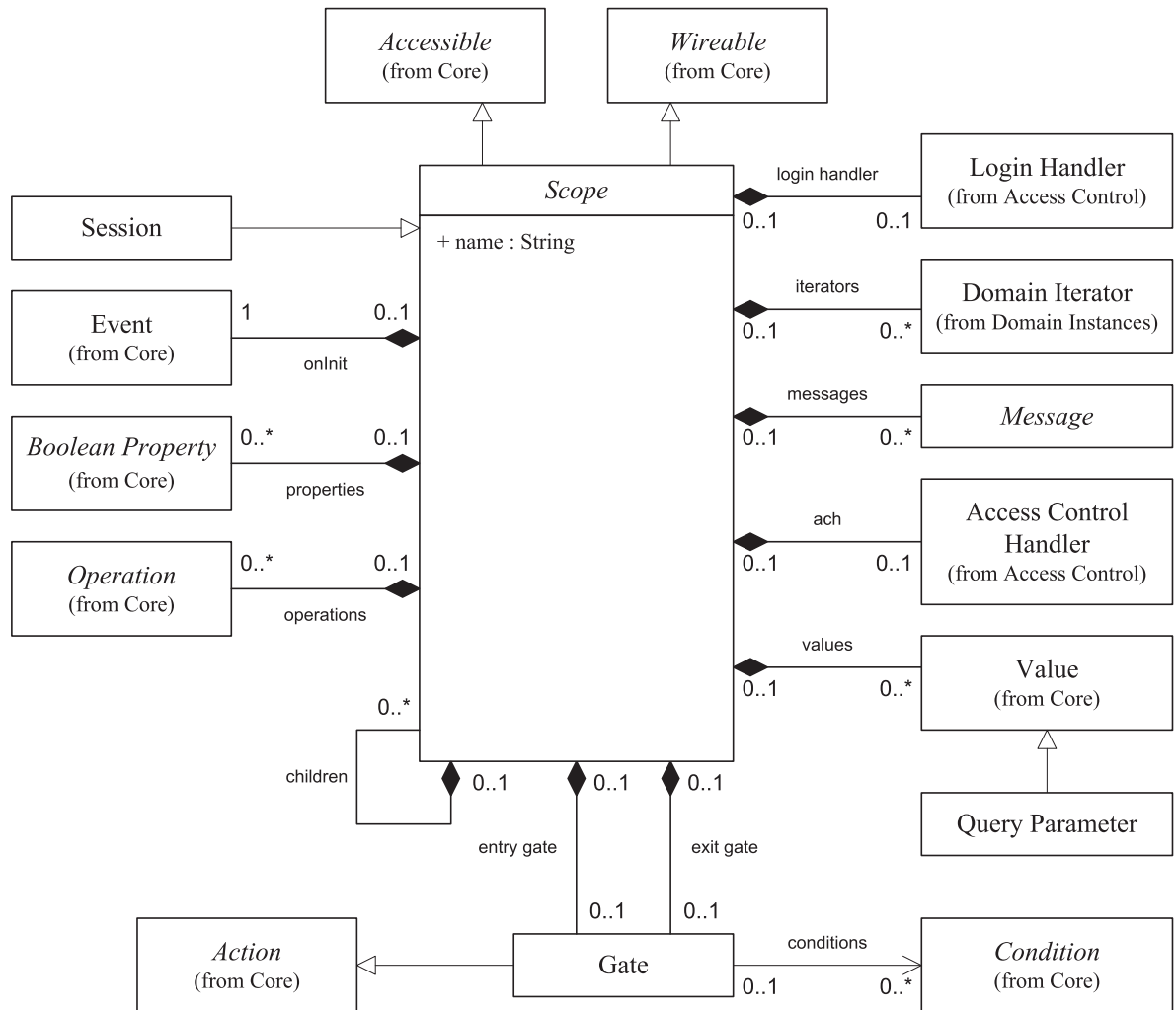
The scopes in IAML are based on the possible lifecycle layers of Rich Internet Applications [368] discussed earlier in Section 4.7.1. Each **Scope** has different initialisation and storage semantics; for example, an **Internet Application** is accessible to all users, whereas a **Session** is only accessible by a single browser instance. **Scopes** in IAML therefore follow a hierarchy, where one **Scope** can contain another **Scope**, as illustrated in the class diagram in Figure 5.39.

A **Scope** may contain any number of value instances, including primitive type instances (**Values**, Section 5.5.3) and complex type instances (**Domain Iterators**, Section 5.6.3). The semantics of how these instances are represented, stored and shared across different users form the *storage semantics* of that **Scope**.

When a **Scope** is rendered, the lifecycle events for that **Scope** must be executed before any elements within the current **Scope** are rendered. However, the lifecycle events for any parent **Scopes** – that is, up the scope hierarchy – must be executed *before* the lifecycle events for the current **Scope**. This allows for the designer to protect access to children scopes through parent scopes; for example, in order to access a **Frame**, the current user must first verify their identity through a parent **Session**.

### 5.11.1 Internet Application

An **Internet Application** instance represents an entire web application, and is consequently the root element of the model (and the indirect parent of all model elements). Because the *onInit Event* is only ever called once, this event can be used to perform application initialisation, such as registering web services, initialising databases, and so on. The structure of the **Internet Application** model element is discussed in further detail later in Section 5.14.

Figure 5.39: UML class diagram for the *Scopes* package of IAML

### 5.11.2 Session

An instance of a [Session](#) represents a user session in the web application; that is, a “sequence of Web transactions issued by the same user during an entire visit to a Web site”, as discussed by Cardellini et al. [48, pg. 268]. Because a web application can support multiple [Sessions](#), it is possible for a user to possess many session instances at any point in time; this may prevent data leakage, as instances of data remain independent between different [Sessions](#).

### 5.11.3 Failure Handlers

It is necessary to support the graceful failure of errors in a web application. In IAML, the concept of a *failure handler* is used as a form of exception handling, and each [Scope](#) in the web application has an associated failure handler. In the case of an exception, this failure handler is executed.

An outgoing [ECA Rule](#) from a particular [Scope](#) with the [name](#) “fail” is defined as a *failure handler* for the given scope. If an error occurs during the execution of the given [Scope](#), then the failure handler for that [Scope](#) is executed according to the semantics of the [ECA Rule](#) (Section 5.4.3). For example, if an [Operation](#) executed via an [ECA Rule](#) *fails*, all subsequent [Action Edges](#) in the *current execution set* are cancelled and the *failure handler* for the [Scope](#) containing the source of the

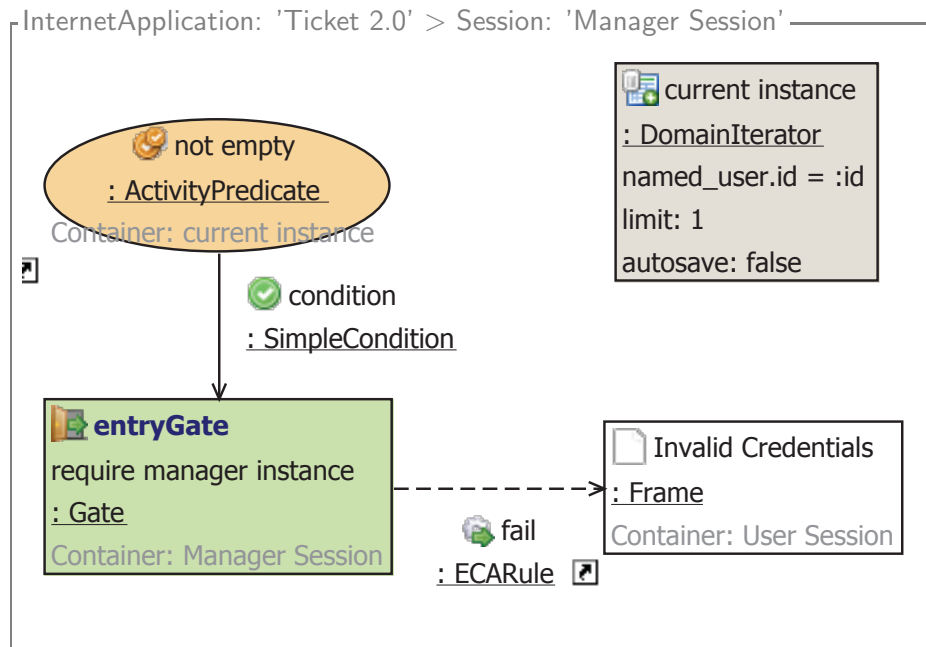


Figure 5.40: Ticketiaml: Protecting access to a [Scope](#) via an entry [Gate](#)

[ECA Rule](#) is executed.

If the current [Scope](#) does not define a *failure handler*, then the *failure handler* of the containing [Scope](#) is used as the failure handler. If an exception is not caught through the containment hierarchy of [Scopes](#), then the exception is caught by the runtime environment and the web application will display a runtime error.

Since exceptions are essentially a form of triggerable event, it may be desirable to instead implement exception handling through the event modelling approach of IAML. This remains future work; for example, an event trigger such as *onException* could be introduced into the IAML metamodel, using the same failure handler semantics as discussed before.

#### 5.11.4 Gates

One common use case in web application development is to selectively prevent access into portions of the web application, without first obtaining the appropriate permissions. Another is to force a user to view an advertisement before they may proceed with the current page.

In IAML, a [Gate](#) may be used to address these scenarios, to restrict access into, or out of, a [Scope](#) according to some condition. However, a web application developer should use [Gates](#) with caution, as web application users may become frustrated if it is too difficult to satisfy a condition of a particular [Gate](#), and simply abandon the web application.

Two types of [Gates](#) are defined. An entry [Gate](#) restricts access *into* a given [Scope](#), and will prevent entry until a particular [Condition](#) is satisfied. Likewise, an exit [Gate](#) restricts access *out of* a given [Scope](#), and will prevent exit until a particular [Condition](#) is satisfied. The functionality behind [Gates](#) are strongly dependent on the method in which the [Scope](#) may be accessed.

An exit [Gate](#) cannot actually prevent the user from closing down their web browser, or manually navigating to an external URL. This is a restriction of the current implementation of web browsers. An exit [Gate](#) may still be used to prevent any further interaction within the modelled web application.



In the Ticketiaml application, an entry *Gate* is used to implement a role check for the *manager Session*. As illustrated in the partial visual model in Figure 5.40, this *Gate* will fail if the incoming *Condition* – a *Predicate* defined as part of the *Domain Iterator* within the same session – is not true. Consequently, if a logged-in user tries to access any *Frame* within the protected *Session*, they will be redirected to the *Invalid Credentials Frame*.

### 5.11.5 Query Parameter

A *Query Parameter* is an extension of the *Value* model element discussed earlier in Section 5.5.3, and represents a parameter from the current browsers' URI request. For example, a request URI `target?name=value` would set the “name” *Query Parameter* to the value “value”. Conceptually a *Query Parameter* is different from a *Value*; the *Value* belongs to the containing *Scope* and persists across calls within that scope, whereas a *Query Parameter* is global and is entirely dependent on the request URI. Therefore, it is necessary to have a separate element to distinguish the two.

### 5.11.6 Discussion

During the design of the IAML metamodel, other model elements within the IAML language were also considered as potential *Scopes*. An *Operation* could be considered a *Scope*; here, the *onAccess* event would be fired whenever the operation is executed. A *Visible Thing* could also be considered a *Scope*, and would follow the *onAccess* event semantics of *Frame*.

Similarly, a request itself could be modelled as a *Scope* [368], but this would require a precise definition of a user request. For example, can only requests triggered explicitly by the user be considered a request, or would requests via AJAX also be considered a request? Such a *Scope* could define events for when the request begins, finishes, or is redirected to another URL.

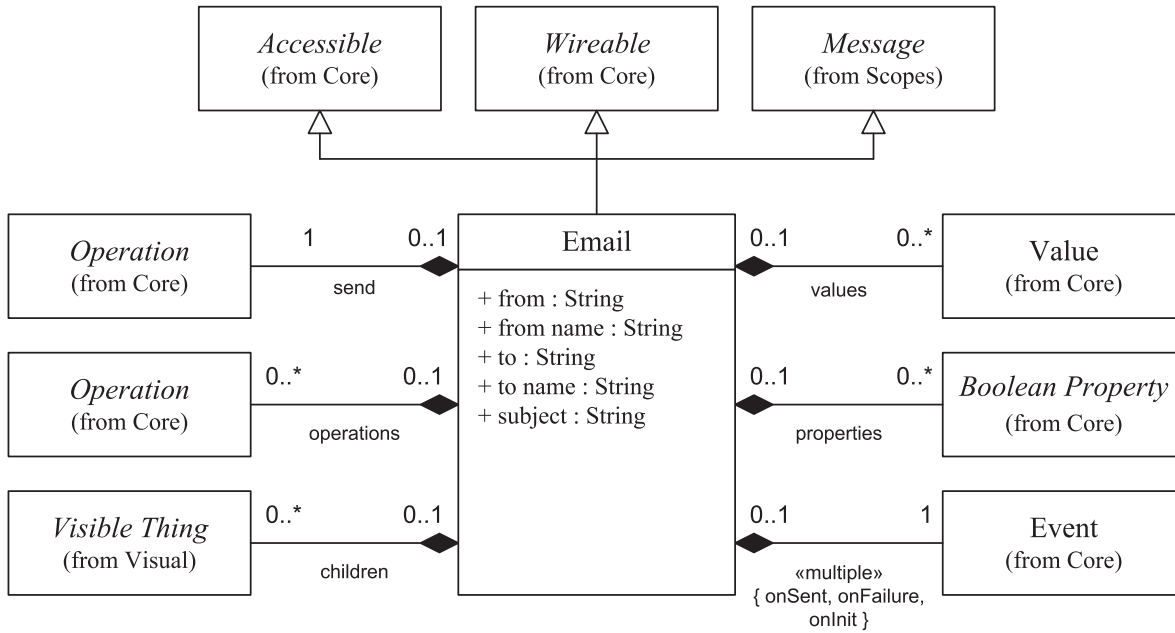
The idea of supporting user-definable scopes was also considered; for example, allowing the developer to specify a “within the administration area” scope, or to designate scopes using tagging. This was not one of the identified use cases of RIAs, and the implementation of this concept should not be difficult as future work. However it is not possible to describe arbitrary events without requiring an event modelling language, as discussed earlier in Section 5.4.3.

The IAML metamodel does not define pre- or post-lifecycle events, as these definitions have not proven necessary. For example, an *onFinish* event for *Sessions* could be triggered when a *Session* times out [246, pg. 114]; however, one cannot guarantee this event will ever be triggered.

## 5.12 Messaging

Messaging refers to the ability to encapsulate content into a form which may be sent from a source to a recipient. Web applications are mostly *pull-based*; that is, they provide content in response to a request, rather than actively pushing out content (*push-based*). However, web applications still have a number of ways of pushing out content, even though their underlying implementations may be pull-based. The most common technique is through sending e-mails<sup>49</sup>.

<sup>49</sup>Use Case 33: *E-mailing Users*.

Figure 5.41: UML class diagram for the *Messaging* package of IAML

### 5.12.1 Email

An e-mail or *mail message*, as defined by RFC 2821 [184], is a message sent from a sender to a receiver through a number of intermediary servers. The message itself has a number of headers and a *body* which may be encoded according to MIME [110]. IAML supports the composition and sending of e-mails through the **Email** model element, the structure of which is represented in Figure 5.41.

The **Email** model element is a wrapper for a number of **Values**, each representing the various required headers for the e-mail message; and also contains events and operations necessary for sending the actual e-mail, such as the **send Operation**. The **Email** model element also defines a number of element-specific attributes, which are used as default values if a similarly-named **Value** is not present or not set. As an **Email** is **Wireable**, its content may be synchronised through **Set Wires**.

When an **Email** is composed in order for delivery, the *body* of the **Email** will be composed of all contained **Properties** and their values at the time the **Email** was sent. The format of this body may use a *custom template* if this template is specified and supported by the language implementation.

### 5.12.2 Additional Messaging Types

There are a number of other ways of delivering messages in web applications, as identified in the list of use cases in Appendix A. As the proof-of-concept implementation of IAML has been strictly limited to Basic RIAs as discussed earlier in Section 5.1.1, these messaging types have not been implemented in IAML. However, they will be briefly discussed here as potential future work.

1. A short message may be sent to a user via a text message or SMS for a mobile device<sup>50</sup>. IAML could therefore define a **Text Message** modelling element to support this scenario, which would likely be similar in composition to the **Email** element above. That is, there could be *onSent* and *onFailure* events defined, however there would be no need for a **subject Value**, as SMSes do not support subject fields.

<sup>50</sup>Use Case 32: *Mobile Phone Communication*.

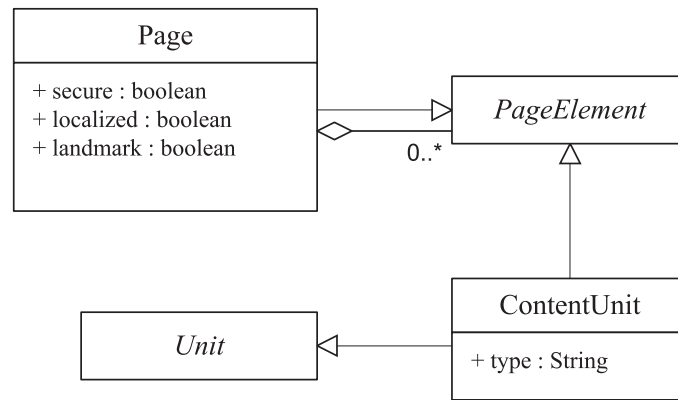


Figure 5.42: Instantiation of user interface elements in WebML: an excerpt from the WebML metamodel [238]

2. Web applications may communicate with web services<sup>51</sup>, as discussed earlier in Section 2.3.1. There are different technologies that may be used in communicating with web services; the service may be defined formally according to standards such as SOAP [340] and XML-RPC [363], or in a proprietary format using JSON [65].

## 5.13 User Interface Modelling

As discussed earlier in Section 2.3.1, the user interface is a very important aspect of RIAs as their interactivity and richness depends directly on its interface. *User interface modelling* is supported in IAML to describe the composition and structure of user interfaces, and is an important part of the modelling language.

### 5.13.1 Existing Approaches in Modelling User Interface Types

WebML defines different types of *Content Units* such as *Entry Units* (similar to [Input Forms](#)) and *Index Units* (similar to [Iterator Lists](#)) [51]. However within its UML metamodel, these different types are treated as “black-box plugins to the three existing models, rather than constituents of an independent modeling layer” [238]. To illustrate this design, an excerpt from the WebML metamodel as described by Moreno et al. [238] is illustrated in Figure 5.42.

The WebML approach is similar to defining a [Visible Type](#) element within the WebML metamodel, allowing different types of elements to be loaded through a library. The reliability of the WebML approach is fairly weak<sup>52</sup> compared to the UWE approach; by defining a *type* of type `String`, there is no guarantee that the given type of `ContentUnit` will actually exist in the system, and all aspects of the implementation must essentially implement their own type checking system. That is, a metamodeling environment that implements the WebML metamodel must be extended to identify different `ContentUnit` instances. However, this does solve the issue of defining behavioural semantics, because all of the semantics are moved into third-party plugins.

<sup>51</sup>Use Case 27: *Web Service*.

<sup>52</sup>This can be considered a specific instance of the *Primitive Obsession* design anti-pattern discussed by Fowler [100, pg. 81–82], where the differences between different types are expressed via a primitive, rather than through an object type hierarchy. This anti-pattern can result in issues of referential integrity, as there are now two different typing mechanisms in the system.

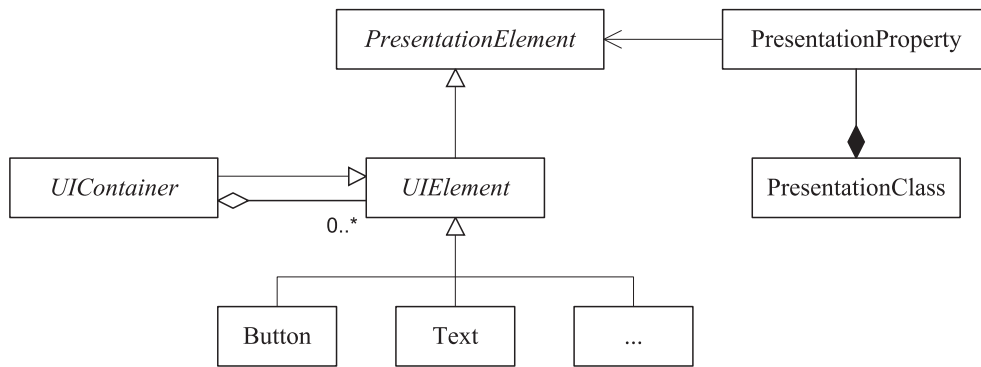


Figure 5.43: Instantiation of user interface elements in UWE: an excerpt from the UWE meta-model [196, pg. 13]

UWE, on the other hand, defines separate types of UIElements for each type of interface element within the metamodel, such as Button and Text. An excerpt from the UWE metamodel as described by Kroiß and Koch [196] is illustrated in Figure 5.43. Instances of the user interface elements are notated in UWE model instances using stereotypes [196, pg. 22]. UML itself does not provide any standard way of modelling user interfaces, and extensions such as UMLi have been proposed [68].

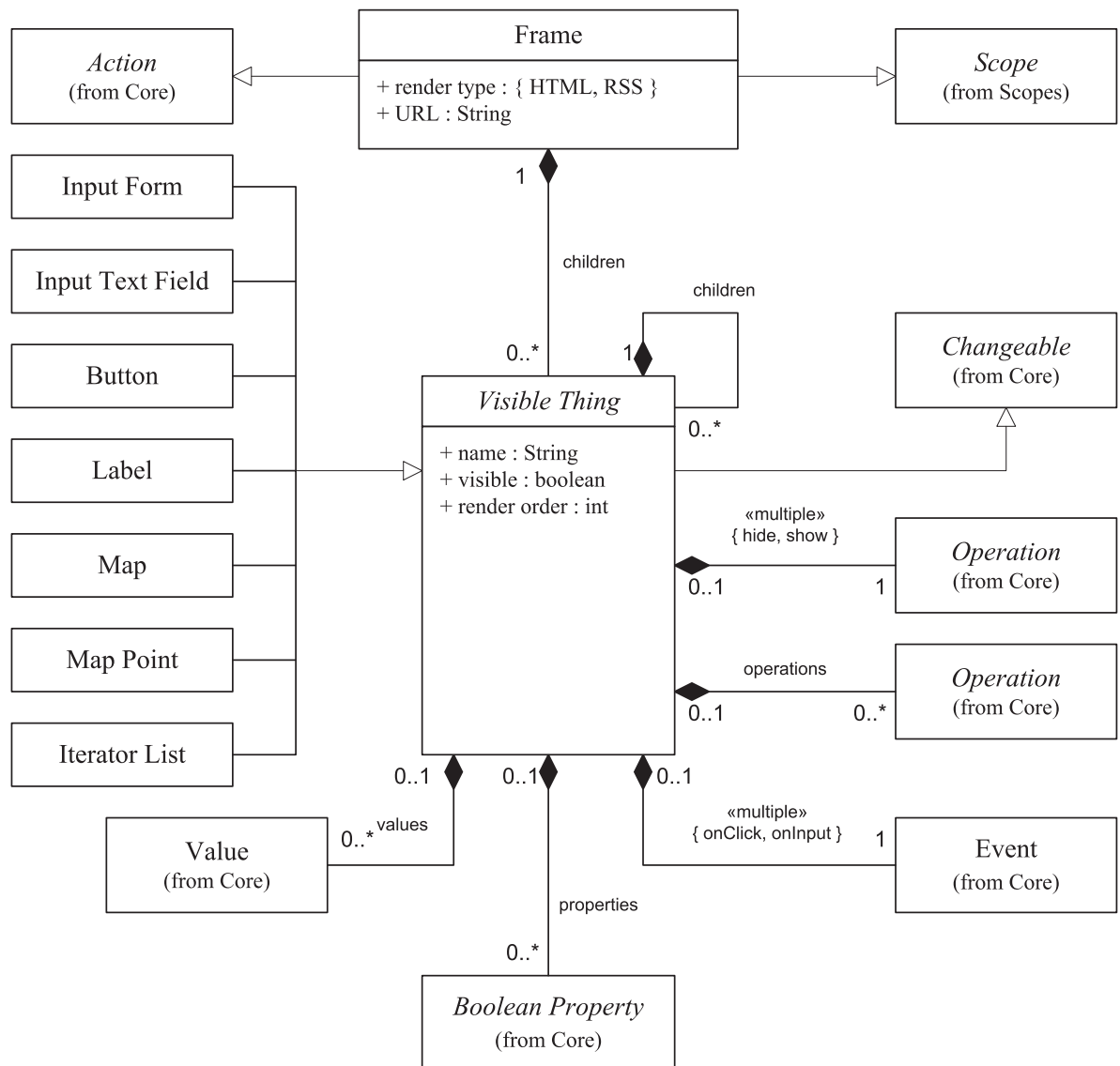
As these two investigations show, both approaches can be used to model the user interface of a web application. More research is necessary to identify when one approach is beneficial over the other. However, the design of the user interface modelling aspect of IAML is most similar to the UWE approach, where each user interface element is defined as a subtype.

### 5.13.2 Visible Thing

In IAML, the *Composite* design pattern [118, pg. 163–173] is used to design user interfaces, as illustrated in the UML class diagram in Figure 5.44. Visual elements may be used to compose interfaces, and these elements themselves may be composed of other visual elements. The composite design pattern also aligns nicely with the hierarchical modelling approach in IAML, as top-level visual elements can hide the complexity contained within.

The **Visible Thing** type is the abstract type for all visual elements in the language. A **Visible Thing** may be hidden on the user interface, by modifying the value of the **visible** property of the element at either design-time or runtime. Since all visual elements are designed to be renderable to the user, all **Visible Things** have additional events related to user interactivity – such as the *onClick* and *onInput* events discussed earlier in Section 5.7 – and are all **Changeable** (and therefore have a **field value**). This chapter will briefly introduce each subtype of **Visible Thing** within IAML; these are discussed in further detail in Appendix I.

It is also necessary to define the way in which a user interface is *rendered* to the user. The spatial layout of a user interface is important, so each **Visible Thing** also specifies its **render order**. While this spatial information is placed within the model instance itself, it may be desirable to move this information into a separate layout model in the future to encourage the separation of concerns between the interface layout and the underlying model.

Figure 5.44: UML class diagram for the *Visual* package of IAML

## Button

A [Button](#) represents a clickable button in the user interface, and is derived from the HTML `BUTTON` element specification [333]. If triggering the *onClick* event does not change the state of the web application – for example, it does not modify any databases or interact with any external component – then the [Button](#) may be rendered as a text hyperlink.

## Label

A [Label](#) represents a static block of text that is not user-editable, but may still be modified programmatically and trigger *onClick* events. This element is derived from the HTML `LABEL` element specification.

## Input Text Field

An [Input Text Field](#) represents a text field that accepts text input from the user, and is derived from the HTML `INPUT` element specification [333].

## Input Form

An [Input Form](#) represents a group of related input elements, and is derived from the HTML `FORM` element specification [333]. An [Input Form](#) is particularly useful when using a [Sync Wire](#) or [Set Wire](#), as discussed earlier in Section 5.9.1. In particular, if a [Domain Iterator](#) is connected via one of these wires to an empty [Input Form](#), the form will be populated with the necessary user interface elements to read or to edit the given iterator instance.

## Map

A [Map](#) represents a geographical area as an interactive map – that is, the map can be navigated from within the browser. Compared to the other defined [Visible Things](#), including the [Map](#) model element is particularly interesting, as it has no analogy in either HTML 4.01 [333] or the upcoming HTML 5 standards [343]; however, if a platform-independent map element is introduced into an upcoming HTML standard, the IAML metamodel will be automatically forward-compatible. A [Map](#) may contain many [Map Points](#) in order to show multiple locations simultaneously.

## Map Point

A [Map Point](#) represents a single geographical point, whereas a [Map](#) represents a single geographical area. A [Map](#) may contain any number of [Map Points](#), all which will be rendered within the current [Map](#) as separate points; however, a [Map Point](#) does not need to be contained by a [Map](#), in which case the point is rendered individually. In order to model an indeterminate number of [Map Points](#) within a [Map](#), an [Iterator List](#) may be used as discussed in the next section.

## Iterator List

All of the [Visible Things](#) discussed so far in this section only support single instances; that is, the number of these elements that may be rendered is defined at design-time, and cannot be controlled

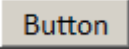




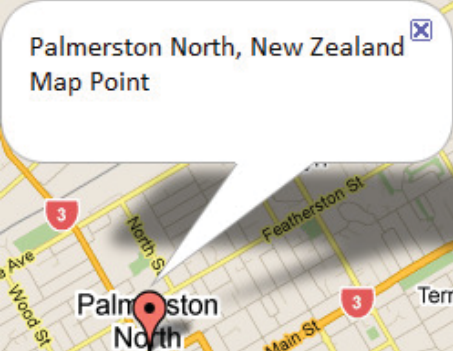
Model Element	Sample Representation								
Button									
Input Text Field									
Input Form									
Label									
Map									
Map Point									
Iterator List	<table border="1"> <thead> <tr> <th>value</th><th>key</th></tr> </thead> <tbody> <tr> <td>Scope</td><td>Frame</td></tr> <tr> <td>Visible Thing</td><td>Input Form</td></tr> <tr> <td>Visible Thing</td><td>Label</td></tr> </tbody> </table>	value	key	Scope	Frame	Visible Thing	Input Form	Visible Thing	Label
value	key								
Scope	Frame								
Visible Thing	Input Form								
Visible Thing	Label								

Table 5.4: Sample visual representations of interface modelling elements in IAML



at runtime. An **Iterator List** allows for a set of **Visible Things** to be rendered an arbitrary number of times, controlled at runtime via an instance of the **Domain Iterator** model element.

For example, if an **Iterator List** contains a single **Label** and the connected **Domain Iterator** has five **results**, then five **Labels** will be rendered at runtime. If the **Domain Iterator** is **empty**, then the **Iterator List** will also be empty.

### 5.13.3 Sample Representations

A sample rendering for each **Visible Thing** subtype is illustrated in Table 5.4. In this summary, the visual representations of **Map** and **Map Point** are those provided by the Google Maps library.

### 5.13.4 Frame

A **Frame** is used in IAML to represent a block of content that may be accessed and rendered independently by a web browser, and as an **Action** may be connected by an **ECA Rule** to support user navigation. By default, the interface to a **Frame** will be provided through the combination of web technologies such as HTML and Javascript. However, if the **render** attribute of the **Frame** is set to **RSS20**, then the content of the **Frame** will be available as an RSS 2.0 feed<sup>53</sup> [290] – particularly useful if the **Frame** contains an **Iterator List** to populate the contents of the RSS feed.

The concept of a **Frame** includes both the concepts of a top-level page, and a block of content within another **Frame**. Top-level pages can also specify a custom **URL** in order to specify the intended URL of the **Frame**. An important piece of future work is in supporting the definition of sub-frames, and the composition of these with parent **Frames** in order to support richer web applications. This technique could be used to implement functionality such as wizard-based workflows [328].

By specifying that an **Frame** is an **Action**, an **ECA Rule** may also be used to redirect the browser to a different **Frame**. Any incoming **Parameters** to the connecting **ECA Rule** are therefore passed as **Query Parameters** to the target **Frame**.

### Example

Within Ticketiaml, the implementation of the *Browse Events* page is implemented through the definition of a **Frame** consisting of a number of contained **Visible Things**, as illustrated in the partial model of Figure 5.45. Here, an **Domain Iterator** is used to populate an **Iterator List** of matching events via a **Set Wire**, which then populate a **Map** via another **Set Wire**. This iterator is also restricted via an incoming parameter from an **Input Text Field**.

## 5.14 Internet Application

In the previous sections of this chapter, each of the modelling concepts used in the IAML metamodel have been defined. The final remaining element that needs to be defined is the **Internet Application** element, which is the root element of any IAML model instance. An **Internet Application** defines the **Scopes**, **Domain Types** (including **Roles**), **Domain Sources**, **Permissions** and **Predicates** of the modelled RIA, as discussed later in Section I.60.

<sup>53</sup>Use Case 26: *Data Feeds*.

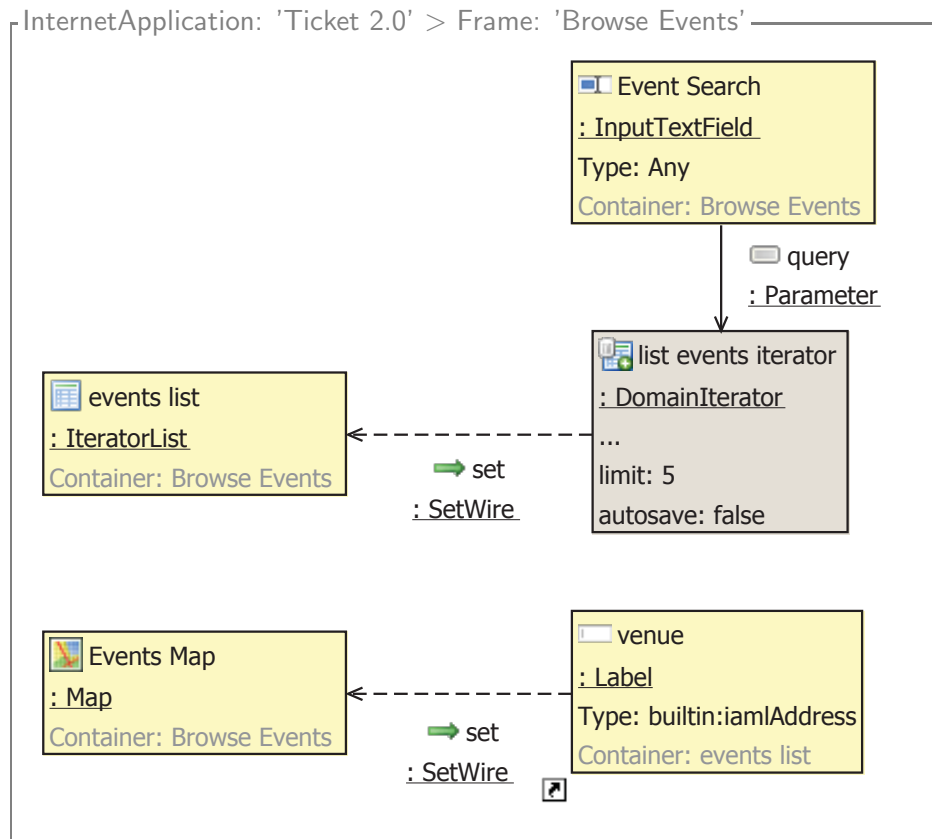


Figure 5.45: Ticketiaml: Designing the user interface for the *Browse Events* page using Visible Things

## 5.15 Visual Modelling Metaphors

As discussed earlier in Section 5.1.5, the development of a visual modelling language is an important design goal for IAML. One part of the design of the IAML visual modelling language included the selection or design of *visual metaphors* – the reuse of familiar concepts outside of their conventional meaning to express a similar concept – in order to reduce the mental load for developers, and improve system accessibility, as discussed earlier in Section 3.4.1.

Each of the hierarchical layers proposed earlier in Section 5.2.3 can afford their own visual metaphor. For example, the operation modelling layer can reuse the visual metaphors from UML operation modelling [254]. It is important for a visual modelling language to explicitly state the visual metaphors used in its design; a summary of these metaphors are provided in Table 5.5, similar to the visual metaphor summary of Grundy et al. [138].

### 5.15.1 Overview Layer

The top layer of the model instance, the *overview layer* represents the overall components within the final RIA. This includes elements such as database and domain sources, sessions, user databases and so on. The UML deployment diagram [254] seems to be an appropriate visual metaphor to adapt, as this layer describes the composition and distribution of application elements, and does not describe any navigation or behaviour.

Layer	Contents	Proposed Visual Metaphor
Overview	Domain sources, sessions, frames	UML deployment diagram
Navigation	Frames, sessions, access control, login handlers	Navigation model
Interface	Visual elements, interface events	User interface builder
Domain	Types, type relationships, attributes	UML class diagram
Operations	Activity nodes, conditional nodes, execution flow, data flow	UML activity diagram

Table 5.5: Visual Metaphors for a Rich Internet Application modelling language

### 5.15.2 Navigation Layer

This layer is concerned with the navigation between different pages, or targets, in the system. It is also concerned with sessions and access control. Within the development of web applications, *sitemaps* are often developed, which are hierarchical trees of content connected through navigation.

A *navigation model*, similar to the UWE navigation model [192], is therefore the ideal visual metaphor for this layer. A navigation model illustrates the intended top-level navigation structures between top-level elements in the application, similar to a sitemap.

### 5.15.3 User Interface Layer

The contents of a page, on the other hand, are inherently visual in Rich Internet Applications. This layer is oriented most towards visual designers. Consequently, a user interface metaphor is chosen; here, visual elements can be dragged and dropped onto the page. The spatial layout of the contents of the diagram could dictate the final layout of the page<sup>54</sup>.

As discussed earlier in Section 5.13, UML does not support any form of user interface modelling except through extensions to the language such as UML*i* [68]. Therefore the user interface visual metaphor used is the metaphor used in user interface builders such as VisualAge for Smalltalk [210].

### 5.15.4 Domain Modelling Layer

As discussed earlier in Section 5.6, domain modelling in IAML is strongly derived from the UML class diagram model [254] as **Domain Types** have an inheritance hierarchy and each contain attributes and references to other types. The UML class diagram visual metaphor therefore seems the most appropriate metaphor to reuse as the domain modelling layer visual metaphor in IAML.

### 5.15.5 Operation Modelling Layer

As discussed earlier in Section 5.8, operation modelling in IAML is strongly derived from the UML activity diagram model [254]; in particular, an **Activity Operation** contains a number of **Activity Nodes** connected with others through **Execution Edges** and **Data Flow Edges**. Reusing the UML activity diagram visual metaphor as a metaphor for operation modelling therefore seems appropriate.

<sup>54</sup>For example, the top-to-bottom layout of the visual model should be used to derive the top-to-bottom rendering order of the generated user interface.

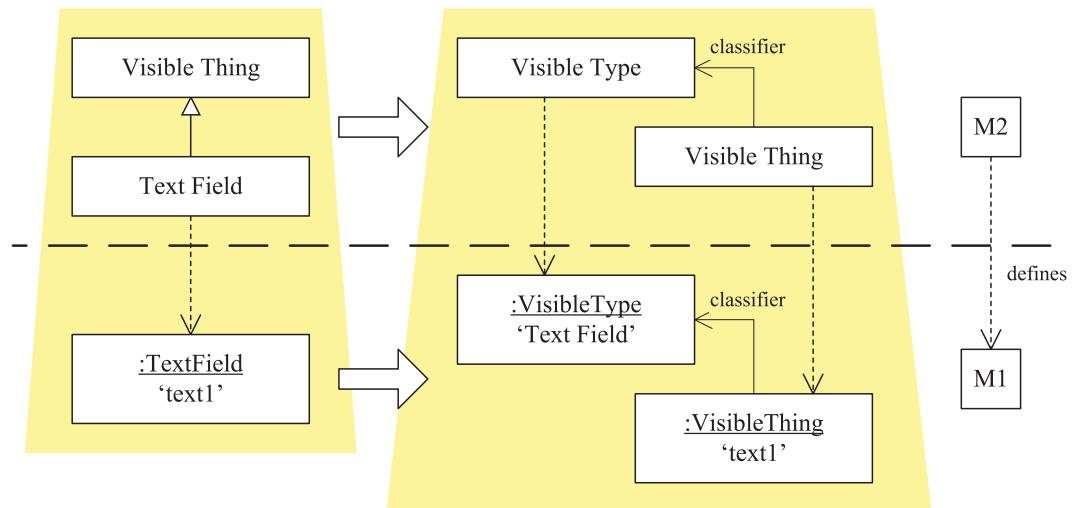


Figure 5.46: The consequences of unifying instance types within the metamodeling architecture of the MDA

## 5.16 Unification of Type Instantiation and Classification

During the development of the IAML metamodel, a discrepancy was identified between domain instance modelling and visual element instance modelling. For example, a [Domain Instance](#) must reference a [Domain Type](#) as a [classifier](#), whereas an [Input Text Field](#) has no such classifier reference. The fundamental difference between these two approaches is that a [Domain Type](#) *classifies* the [Domain Instance](#), whereas a given text field instance is *defined* by the [Input Text Field](#) type.

In terms of the metamodeling architecture of the MDA, this would change the IAML metamodel and the resulting instances of the metamodel as illustrated in Figure 5.46. This refactoring would introduce the following benefits:

1. A [Frame](#) no longer has to have separate references to [Visible Things](#) and [Domain Instances](#). This means that the instance specification model is more unified. This approach is also more aligned with the underlying infrastructure of UML models [253, pg. 54].
2. Parts of the IAML metamodel can be simplified, as visual element instances such as [Input Text Fields](#) and [Buttons](#) would be refactored out into the model instance layer. This would mean that a *library* of visual elements could be provided, perhaps automatically.
3. A model instance developer could in the future develop their own [Visible Types](#), allowing IAML to adapt to new standards and innovations in RIAs. For example, the recent acceptance of video as a first-level interface element with HTML 5 [343] could ostensibly be independently defined by a developer as a [Visible Type](#) named “Video”. As discussed in Section 6.2.1 and Section 6.4.3, EMF metamodels and Xpand templates can already be extended by third parties, so this scenario can already be supported<sup>55</sup>.

If these changes were made against the underlying IAML metamodel discussed throughout this chapter, the resulting metamodel would resemble the unified metamodel illustrated in Figure 5.47. In particular, both [Domain Instances](#) and [Visible Things](#) would follow the architecture of having a

<sup>55</sup>The extensibility of the proof-of-concept implementation of IAML is discussed later in Section 9.3.8.

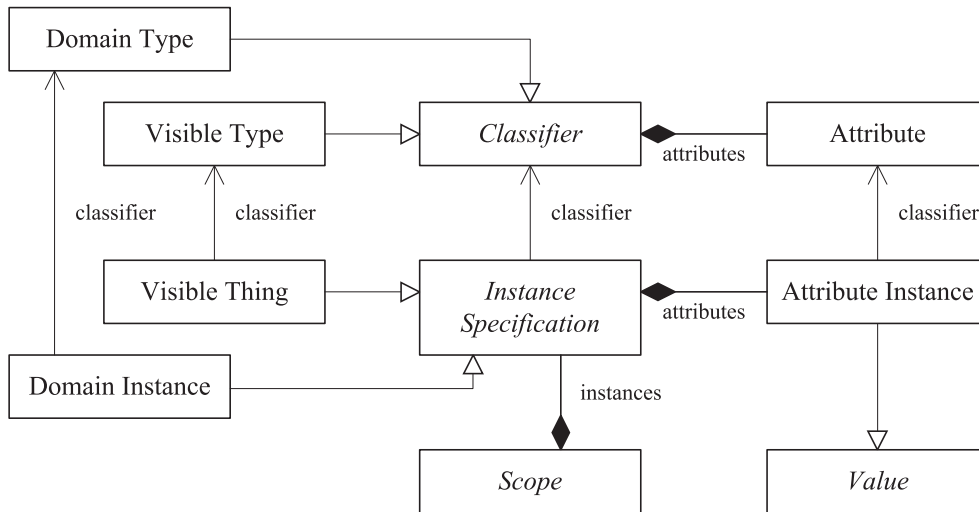


Figure 5.47: A proposed refactoring of the IAML metamodel to unify the instantiation of [Domain Instances](#) and [Visible Things](#)

*classifier* reference to an abstract [Type](#). However, the following negative effects would apply to this refactored metamodel:

1. Parts of the IAML metamodel must become more complex. Two additional types, [Visible Type](#) and [Instance Specification](#), must be introduced, along with additional references.
2. There is no benefit in moving instances of [Visible Types](#) to the model instance layer unless developers can create their *own* instances of [Visible Types](#). In the existing metamodel, different [Visible Types](#) have differing event behaviours and user interface appearance. It would therefore be necessary to also support the full definition of the [Visible Types](#)' corresponding semantics. This includes:
  - (a) Definition of arbitrary events. As discussed in Section 5.4.3, this would require the definition of an event modelling language, which is outside the scope of this thesis.
  - (b) Definition of code generation templates. A user-defined user interface element still needs some way to be rendered. This would therefore require the definition of a templating language, which is also outside the scope of this thesis.

Conversely, no support in defining these semantics for [Domain Types](#) was necessary, because all user-defined [Domain Types](#) *have the same behaviour and semantics*. That is, the behavioural difference between a “User” and a “Gig” is only the [Domain Attributes](#) inside; whereas the behavioural difference between a “Button” and a “Video” is fairly substantial.

Alternatively, the WebML “black box” approach as discussed earlier in Section 5.13.1 could be used here, where the model developer would define implementations completely independently from the modelling language according to an interface. However, this means that a “Video” element could not define its own events, such as the *onPlay* and *onPause* events of the HTML 5 video element [343].

3. It would no longer be possible to distinguish between different interface elements according to their notation. For example, to a model instance developer an instance of a [Button](#) would appear

identical to an [Input Text Field](#), except for a label describing the classifier. It is not appropriate for a metamodel element to define its own notation – for example, a [icon](#) field on [Visible Type](#) – as this breaks an underlying MVC model [195].

Finally, it was found that this unification process was not necessary in order to implement the benchmarking application *Ticket 2.0*. This discussion is provided later in this thesis in Chapter 8.

### 5.16.1 Guidelines for Applying Instance Unification in Metamodels

It is desirable to extrapolate the underlying reasoning from the previous discussion, to propose guidelines on applying this instance unification pattern within a metamodel. That is, it would be helpful to understand when it is appropriate to have type definitions within the model instance layer, or when to keep it (or move it into) the metamodel layer. In the development of IAML, the following guidelines are therefore proposed:

- The definition of a type, such as [Domain Type](#), should reside within the model instance layer when some or all of these criteria are met:
  1. There are none, or few, differences in behaviour between instances of the different types.
  2. It is expected that different model instances will have their own domain-specific instances of the types; that is, it is likely that a developer would have to implement their own instances of each type across each different model instance.
  3. There will be a significant number of unique instances of the type over the lifetime of the language within a suite of model instances of that language.
- Alternatively, the definitions of a range of type instances, such as [Visible Things](#), should reside within the metamodel instance layer when some or all of these criteria are met:
  1. There are substantial differences in behaviour between instances of the different types, that cannot be described within the current language, or the effort necessary to describe and implement these behaviours separately against a modelling language is too great.
  2. It is expected that different model instances will refer to predefined or built-in instances of the type, with third-party instances of the type considered an exception.
  3. There will be very few unique instances of the type over the lifetime of the language within a suite of model instances of that language.

Alone, none of these criteria may be strong enough to warrant the investment necessary to move the instance type definition to another layer. However, they may be useful in the development of new modelling languages that require the definition of types, and the subsequent instantiation of these types. During the development of the IAML metamodel, no such guide was available.

### 5.16.2 Applying Instance Unification to the IAML Metamodel

Other than [Domain Types](#) and [Visible Things](#), there are a number of other elements in the IAML metamodel that may be defined either as user-defined instances of types, or instances of user-defined types. The guidelines introduced in the previous section can be used to make this decision, and in this section this investigation will be performed.

### Type Instantiation through Classifiers

With respect to the instance unification guidelines discussed in the previous section, the following IAML concepts should support the definition of both types, and instances of these types, within a single model instance:

1. **Domain Types:** Different model instances will have their own domain-specific **Domain Type** instances, and there will be many unique types defined across all model instances. There is no difference in behaviour between instances of different **Domain Types**.
2. **Functions** and **Predicates:** The latest version of XQuery defines 180 different **Functions** [350], and most web applications will need to also define their own libraries of **Functions**. It is fairly simple to describe the differences in functionality of different **Functions**; new instances of **Functions** can be described using languages such as OCL [252], or within the IAML metamodel itself using **Activity Predicates**, as discussed in Section 5.8.

### Type Instantiation through Instances

Similarly, the following IAML concepts should only support the definition of instances in a single model instance, with the definition of the types of these instances residing in the metamodel definition for that model instance:

1. **Visible Things:** The behavioural differences between different types of user interface elements are significant, and the effort necessary to support these behaviours is significant. Most web applications will use a common range of user interface elements.
2. **Wires:** The behaviour of **Wires** are described with a complex set of model completion rules, as discussed in Section 5.9. As discussed by Wright and Dietrich [369], it is desirable for third parties to define their own model completion rules, however this process is not simple. Allowing **Wires** to be typed would mean that the IAML metamodel would need to support the elements of model completion directly – in other words, a model completion rule modelling language.
3. **Actions:** As discussed in Section 5.4.3, there are only two different types of **Actions** in the IAML metamodel – **Operations** and **Frames** for navigation – and it is not clear if there may be any other types of **Actions** for RIAs.
4. **Events:** As discussed in Section 5.7, a modelling language that supported the third-party definition of events would need some method for defining the *semantics* of the events themselves. This would therefore require the definition of an event modelling language, which is outside the scope of this thesis.

### Type Instantiation through either Classifiers or Instances

There is one IAML concept that may be defined using either of these two instance unification guidelines, as it is not yet clear which approach would be stronger. In the current IAML metamodel, the different types of **Scopes** are defined as part of the IAML metamodel.

1. **Scopes:** The only behavioural difference between **Scopes** is the method in which the **Properties** of that **Scope** is stored. However, it is not expected that a web application will have a wide range of **Scopes**, and most developers will simply use the builtin **Scopes** such as **Session** and **Frame**.



### Discussion

As discussed earlier in Section 5.1.4, the benchmarking application *Ticket 2.0* may be used to verify the design decisions taken in the development of the IAML metamodel. As discussed later in Chapter 8, this implementation suggests that these design decisions were correct, as no additional types needed to be defined within the language, and all of the types defined were unique to the application itself.

## 5.17 Conclusion

In this chapter, the complete formal definition of the Internet Application Modelling Language has been provided, including the metamodel structure, constraints operating upon model instances, and the functional semantics of each model element when translated into a generated application. The next chapter will focus on the implementation of this language into a CASE tool using existing model-driven development technologies.



## Chapter 6

# Implementation Technologies

As discussed earlier in Section 5.1.5, an important design goal of IAML is to provide a proof-of-concept implementation of the modelling language, in order to validate the metamodel. This implementation will be used in the evaluation of the modelling language; serve as a reference implementation for future work; and may increase acceptance of the language by the development community.

A range of technologies – from metamodeling environments to code generators – may be used in such an implementation, and each technology has many existing implementations. In this chapter, each of these technology implementations will be evaluated, and the best technology implementation will be discussed. Where applicable, this evaluation will include comparisons performed using the OpenBRR framework, detailed in Appendix C. This combination of selected technologies will form the basis of the proof-of-concept implementation of IAML as discussed in the next chapter.

### 6.1 Common Comparison Criteria

Each technology in this chapter will initially be evaluated against a suite of common criteria. For example, the execution or runtime environments of a technology can be described and documented; and project quality metrics can be applied if source code is available.

#### 6.1.1 Execution Environment Comparison Properties

In this chapter, the *execution environment* refers to the environment in which the technology is written and executes. This environment can be different from the implementation language; for example, most Eclipse plugins are written in the Java language, but need to be executed within the Eclipse framework [117]. For the proof-of-concept implementation of IAML it would be desirable to select technologies implemented with the same languages and executed within the same environments. The following common properties will be discussed for each technology in this chapter:

- *Language*: What general-purpose programming language is the technology implemented in? To simplify the implementation of a proof-of-concept environment, it would be beneficial to select components that are all implemented within a single language, as no cross-platform integration would be necessary.
- *Runtime Environment*: Applications using the particular technology may need to be integrated within another environment or container at runtime. This additional dependency increases the

overhead of using the technology, but may also provide a range of services necessary to simplify an implementation. For programming languages this does not refer to the development environment, but any runtime that compiled instances of the language must rely on.

- *Open Source*: Is the technology available under an open-source license approved by the Open Source Initiative<sup>1</sup>? As discussed earlier in Section 2.7.4, open source software can reduce defects [234], improve functionality and increase developer accessibility when compared to closed source software.
- *Version*: This property simply states the version of the technology evaluated.

## Eclipse Framework

Many of the technologies discussed in this chapter are provided as *plugins* to the Eclipse Framework – a software development framework designed to support a wide variety of development languages and approaches through a rich plugin interface, as discussed by Steinberg et al. [314, pg. 3] and Gamma and Beck [117]. These plugins are implemented as OSGi *bundles* [318] – discussed earlier in Section 4.7.1 – allowing components to be installed and removed at runtime in a flexible manner. All Eclipse plugins therefore have the Eclipse framework as a *runtime environment* dependency.

### 6.1.2 Open Source Comparison Criteria

Some technologies discussed in this chapter are provided under open source licenses, and it is therefore appropriate to also evaluate the open source quality of these technologies. Two existing frameworks for evaluating open source components are the *Qualification and Selection of Open Source Software* (QSOS) [265] and the *Business Readiness Rating* (OpenBRR) [263] frameworks.

Deprez and Alexandre [74] evaluated both of these frameworks and found that while QSOS provides more extensive rating criteria and supports versioning, OpenBRR provides scoring with less ambiguities and results can be tailored towards a specific context. At the time of writing, however, the progress of OpenBRR appears to have stalled, with no documentation released since 2005<sup>2</sup>. Nevertheless, in this thesis the OpenBRR will be used to form the basis of open source evaluation criteria.

The most recent release of OpenBRR [263] defines 28 metrics categorised into eleven categories, along with a *representative set* of metrics to illustrate the model. The authors argue that these representative metrics should be tailored towards the particular domain. In this thesis, no such tailoring was performed as it was not necessary; the full range of metric values were present through the evaluations, so no reweighting was necessary.

While every category in the OpenBRR ranking system represents an important part of the overall ranking, five of these categories have been selected as particularly important for the proof-of-concept implementation – *quality*, *support*, *documentation*, *adoption* and *architecture*. These categories are those most likely to suggest a simple implementation process, and are described by the OpenBRR standard [263] as follows:

- *Quality*: “Of what quality are the design, the code, and the tests? How complete and error-free are they?”

<sup>1</sup>The Open Source Initiative: <http://www.opensource.org>.

<sup>2</sup>At the time of writing, the OpenBRR website <http://www.openbrr.org> simply states that “we will soon be updating this site with new content.”

- *Support*: “How well is the software component supported?”
- *Documentation*: “Of what quality is any documentation for the software?”
- *Adoption*: “How well is the component adopted by community, market, and industry?”
- *Architecture*: “How well is the software architected? How modular, portable, flexible, extensible, open, and easy to integrate is it?” In particular, a system with a high *architecture* score will most likely support third-party extensions.

The categories of *usability*, *performance*, *security* and *stability* are not too relevant for proof-of-concept implementations, as their impact is reduced within an experimental environment. Additionally, the *community* category is very related to the *support* and *adoption* categories, so may be omitted. The values for every OpenBRR metric evaluated against each technology is provided in this thesis in Appendix C.

In the following sections, the overall *OpenBRR Rating* of each technology implementation will be calculated, which defines a score from a minimum of 28 (unacceptable) to a maximum of 140 (excellent)<sup>3</sup>. An average score of each of the five important categories discussed above will be provided as a numeric value.

### Applying OpenBRR to Closed-Source Projects

At first glance, it seems possible to apply the OpenBRR metrics to closed-source projects. For example, metrics such as frequency of releases, types of documentation, and activity on mailing lists can all be derived without needing access to open-source artefacts or methodologies. However, other metrics such as number of bugs, the number of security vulnerabilities and the number of unique contributors cannot be derived from a closed-source development methodology. Consequently, closed-source projects are given an OpenBRR ranking of *n/a* in the following chapters.

## 6.2 Metamodelling Environments

A number of metamodelling environments exist in academia and industry to assist the quick development of model-driven platforms and technologies. It is preferable to reuse an existing metamodelling environment over developing an environment from scratch, as a metamodelling environment represents a significant amount of development effort, and it does not seem like a finished environment would be an improvement over existing environments.

There are many existing metamodelling environments as discussed by Fowler [102, pg. 140]<sup>4</sup>, such as MetaEdit+ [178], the Meta-Programming System (MPS) by JetBrains [170]<sup>5</sup>, and the Generic Modeling Environment [206]. However, a full evaluation of each of these metamodelling environments is outside the scope of this thesis. In this section, four popular metamodelling environments – selected based on their industry support, integration with other technologies within a model-driven tool stack, academic activity, and source code available through an open source license – will be investigated and evaluated according to a number of evaluation criteria.

<sup>3</sup>In OpenBRR, each criteria is given an integer ranking between 1 (unacceptable) to 5 (excellent) [263]. A future standard of OpenBRR may benefit in instead changing the scale to use 0–4 to simplify ranking comparisons.

<sup>4</sup>Fowler uses the term “language workbench” to describe a metamodelling environment.

<sup>5</sup>The first public release of JetBrains MPS was released in 2009, and as such was not an option when this research was started.

This investigation is particularly important since the metamodel forms the “core” of the resulting development environment, and the choice of metamodeling environment will affect other aspects of the implementation, such as code generation and verification technologies.

### Additional Evaluation Criteria

Along with the common comparison criteria discussed earlier in Section 6.1.1 and Section 6.1.2, each metamodeling environment will be evaluated on a number of additional criteria:

- *Visual Definition:* Can a metamodel be implemented graphically? For a metamodeling environment that is intended to support the definition of graphical modelling languages, this feature may be essentially provided “for free” by the environment itself.
- *Graphical Editor:* Does the metamodeling environment provide a graphical interface for interacting with a developed model instance, and is this interface the intended means of end-user interaction? Such an environment could be used for both the metamodeling and graphical modelling environments, but it is important that a graphical representation of a model instance should be kept separate from the underlying model instance representation.
- *Metamodel Extensibility:* If a developer creates an instance of a metamodel, can this metamodel later be extended by a third party? That is, can a third party add additional elements and constraints to an existing metamodel? Metamodel extensibility allows a third party to extend the modelling environment or adapt it to a particular domain, and is a driving force behind UML Profiles [114].
- *Meta-metamodel:* What is the underlying meta-metamodel of the developed metamodels? That is, what language or technology is used to represent instances of metamodels used in the modelling environment?
- *MDA-compliant M3:* Is the meta-metamodel compliant to the MDA requirement for a meta-metamodel [185, pg. 88]; that is, has the meta-metamodel been defined in terms of itself?
- *Metamodel Sources:* What kinds of sources may be used to develop a metamodel instance? Being able to import a metamodel from an existing source may simplify the development of the modelling environment.
- *Verification Languages:* Which model instance verification languages does the metamodeling environment natively support, if any? In many cases, the metamodeling environment may be extended with additional languages as discussed later in Section 3.3.

#### 6.2.1 Eclipse Modeling Framework

The *Eclipse Modeling Framework* (EMF) is a Java-based metamodeling suite for developing metamodels within the Eclipse framework [314]. It is well supported by the Eclipse foundation, and is used as the metamodeling technology behind a number of other Eclipse-based projects, such as GMF. EMF focuses on the serialisation and functionality of the metamodel, and does not provide a graphical editor for model instances natively; instead, the GEF and GMF components discussed later in Section 6.3 can be used to provide a graphical editor [136].

EMF exists as part of the extensive *Eclipse Modeling Project*, which (at the time of writing) consisted of eleven sub-projects and over 130 components<sup>6</sup>. These sub-projects include graphical modelling, model transformations and model verification. Furthermore, as part of the Eclipse project, integration with other Eclipse technologies – such as version control, development environments and project management – is fairly straightforward.

Metamodels may be loaded from a diverse range of sources [314], such as Ecore model instances created by the EMF model editor; annotated Java source code; XMI models [250] exported from UML CASE tools, such as Rational Rose or ArgoUML [330]; XML Schemas [339]; or metamodels created dynamically at runtime. Ecore model instances can also be created graphically through the *Ecore Tools* graphical editor. Verification is supported by the *EMF Validation framework*, and constraints can be defined using a number of languages, including OCL constraints [252], XML Schema constraints, and manually-implemented Java code generated through scaffolding [314].

A loaded metamodel can then be used to generate the necessary Java scaffolding source code of the corresponding model, using code generation written according to Java Emitter Templates (JET). The generated Java scaffolding may be modified manually, with the generator controlled by the presence of `@generated` annotations within the source code<sup>7</sup>.

Alternatively, the functionality of the metamodel can be provided through *Dynamic EMF*, which is a simple interpretive implementation of EMF designed for sharing simple objects [314, pg. 36–38]. EMF also supports the extension of existing metamodels by allowing additional metamodels (resources) to be loaded at runtime.

Once created, this scaffolding can be used to create new model instances as simply as creating new Java objects. The EMF framework deals with all the necessary logic behind the implementation, natively supporting serialisation to XMI [250]. EMF metamodels are usually executed within an Eclipse environment, but may be used in a standalone environment.

The underlying meta-metamodel for EMF metamodels is the Ecore metamodel, which is defined in terms of itself. This self-definition allows for Ecore to reside at the M3 layer of the metamodelling architecture of MDA. As discussed earlier in Section 3.1.5, the Eclipse Modeling Framework was built on EMOF [12] and model instances can be serialised directly to EMOF [314, pg. 40].

EMF models also support model annotations using the `EAnnotation` model element. These annotations are particularly helpful when providing metamodel documentation, as illustrated in Listing 4. If necessary, these annotations can be stored and accessed at runtime [314], and subsequently may be reused by the *ModelDoc* framework to generate metamodel documentation [366].

### 6.2.2 ArgoUML

ArgoUML is a free, open-source modelling tool written in Java which was originally developed as a UML diagram editor [286]. While it was originally designed to only support the development of UML diagrams, it has since been extended to support UML Profiles, and thus is a form of a metamodelling environment. It has been used to implement other model-based prototypes, such as the discontinued *ArgoUWE* editor for UWE [187], and *ARGOi* for UMLi [67, 68].

---

<sup>6</sup>This component count was calculated by getting all non-release components from Bugzilla at the following URL: [https://bugs.eclipse.org/bugs/query.cgi?classification=Modeling&query\\_format=advanced](https://bugs.eclipse.org/bugs/query.cgi?classification=Modeling&query_format=advanced).

<sup>7</sup>When an EMF-generated class is regenerated, only fields and methods with an unmodified `@generated` annotation may be modified or removed by the generator [314, pg. 305–308]; this approach follows the code generation principles proposed by Fowler [102, pg. 126].



```

<eClassifiers xsi:type="ecore:EClass" name="Value">
  <eAnnotations source="http://www.eclipse.org/emf/2002/GenModel">
    <details key="documentation" value="Represents a single {@model Value#type}d
      value, accessible and modifiable at runtime."/>
  </eAnnotations>
  [...]
</eClassifiers>

```

Listing 4: EAnnotation documentation in Ecore

The ArgoUML metamodel is based on all UML 1.4 diagrams [249]; and as such, software based on ArgoUML also needs to describe their metamodel in terms of UML 1.4 concepts. Internally, model instances are implemented as instances of JavaBeans [286], and can be serialised to and loaded from XMI. At the time of writing, ArgoUML had no plans for supporting recent revisions of UML such as UML 2.0 [254].

Custom metamodels are defined in ArgoUML by implementing UML Profiles [187], which may be implemented visually using the graphical editor of ArgoUML itself and loaded through XMI files. However this approach is extremely limited, and profiles are usually supplied as Java class instances instead<sup>8</sup>. These are then provided to ArgoUML at runtime through additional JAR files, known as *modules*. These are loaded at startup, with the architecture of these modules following the *Dynamic Linkage* design pattern [132]. As ArgoUML metamodels are Java class instances, it is not possible to extend these profiles arbitrarily, hindering metamodel extensibility.

One benefit of using ArgoUML is that it provides a free graphical CASE tool to any new models, which will be discussed later in Section 6.3.3; in fact, it is only possible to create model instances graphically in ArgoUML, as illustrated in Figure 6.1. Model instance verification is supported through the concept of *critics* – implemented as Java class instances – which are used to highlight potential problems in a model instance [286]. ArgoUML provides the Dresden OCL toolkit [72] to implement OCL constraints, but as profiles are designed to be implemented through Java, OCL constraints for a particular profile must be integrated manually.

ArgoUML is best suited for developing graphical editors for metamodels entirely defined within UML 1.4, as any functionality not directly supported by a UML Profile must be implemented manually. The fact that ArgoUML does not easily integrate with other model-driven technologies, such as code generation or model completion, suggests that developing a proof-of-concept implementation of IAML using ArgoUML would be very difficult and require significant development effort.

### 6.2.3 Whitehorse

In Visual Studio 2005, Microsoft introduced a model-driven environment for developing applications. This framework was originally codenamed *Whitehorse* [61] and released under the name *DSL Tools*, but is currently released under the name of the *Visualization and Modeling SDK* (VMSDK). The environment is implemented with, and generates to, .Net-based languages such as C#.

Whitehorse provides a graphical interface for developing the structure of domain-specific languages, as well as visualising the created model instances of the developed language. The generated

<sup>8</sup>UML Profiles in ArgoUML are represented as instances of the `org.argouml.profile.Profile` abstract class.

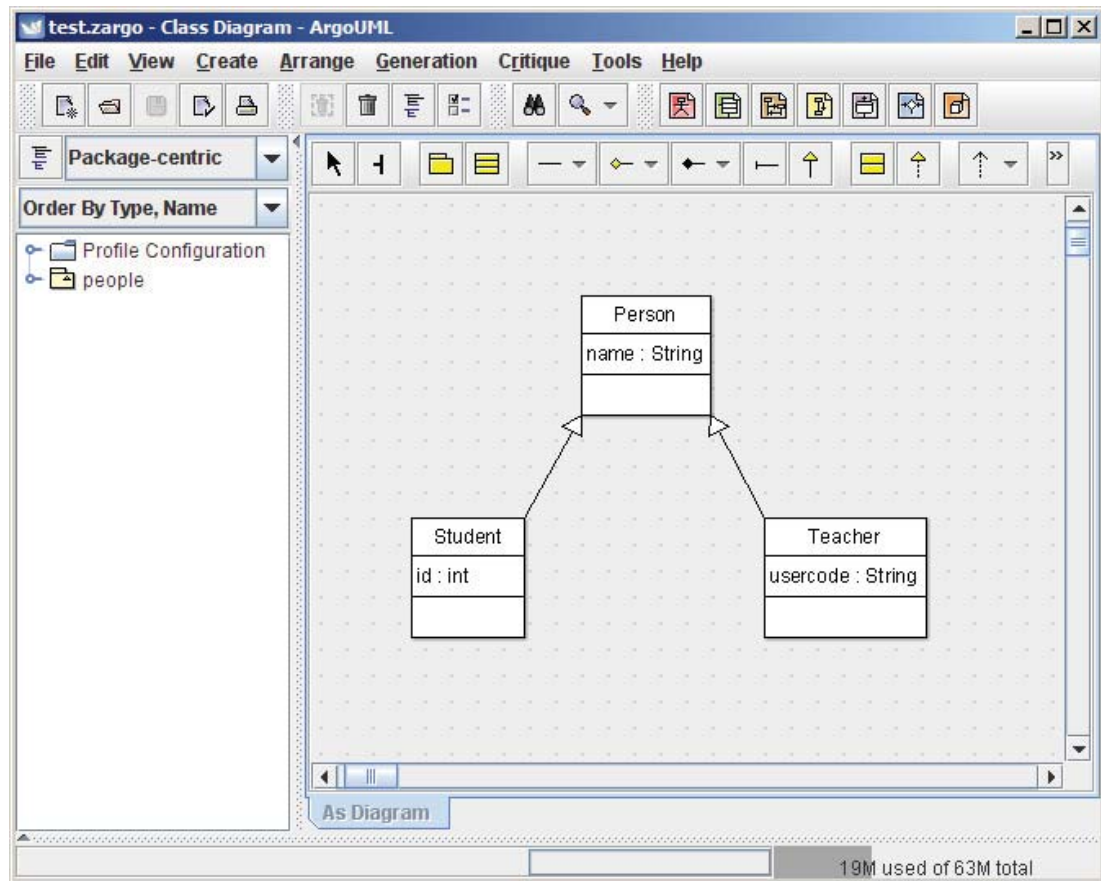


Figure 6.1: Using ArgoUML to define a UML class diagram

language supports model verification only in terms of a validation framework<sup>9</sup>, and constraints must be written in .Net; there is no support for OCL constraints.

In Whitehorse, a DSL can only be defined using the provided DSL editor. The *Managed Extensibility Framework* can be used to extend existing metamodels, however this extensibility is not provided automatically and there are a number of files that must be created manually [232].

The underlying meta-metamodel used to define metamodel instances within Whitehorse-developed is provided by the *Domain Model Framework* (DMF) [61] – a technology that fulfills a similar purpose as EMF, but does not reflect on its architectural role within the MDA. Similarly, the meta-metamodel behind DMF – which in this thesis is tentatively named the *DslDefinitionModel*<sup>10</sup> – does not appear to be defined in terms of itself. This step is necessary for a meta-metamodel to be a valid M3 layer of the Model Driven Architecture, as discussed earlier in Section 3.1.5.

Since Whitehorse was announced, there appears to have been very little academic or industry uptake of these technologies. However, the recent release of Visual Studio 2010 has reorganised the DSL software development kit into the *Visual Studio Visualization and Modeling SDK*. This new release supports the *ModelBus* model integration framework [35], and supports the serialisation of model instances into the XMI format. Previously, model instances could only be serialised using a proprietary XML-based format.

<sup>9</sup>Within Whitehorse, domain-specific language verification uses the *Microsoft.VisualStudio.Modeling.Validation* namespace.

<sup>10</sup>The XML namespace used for metamodel instances defined using the *Whitehorse* development tools is <http://schemas.microsoft.com/VisualStudio/2005/DslTools/DslDefinitionModel>.

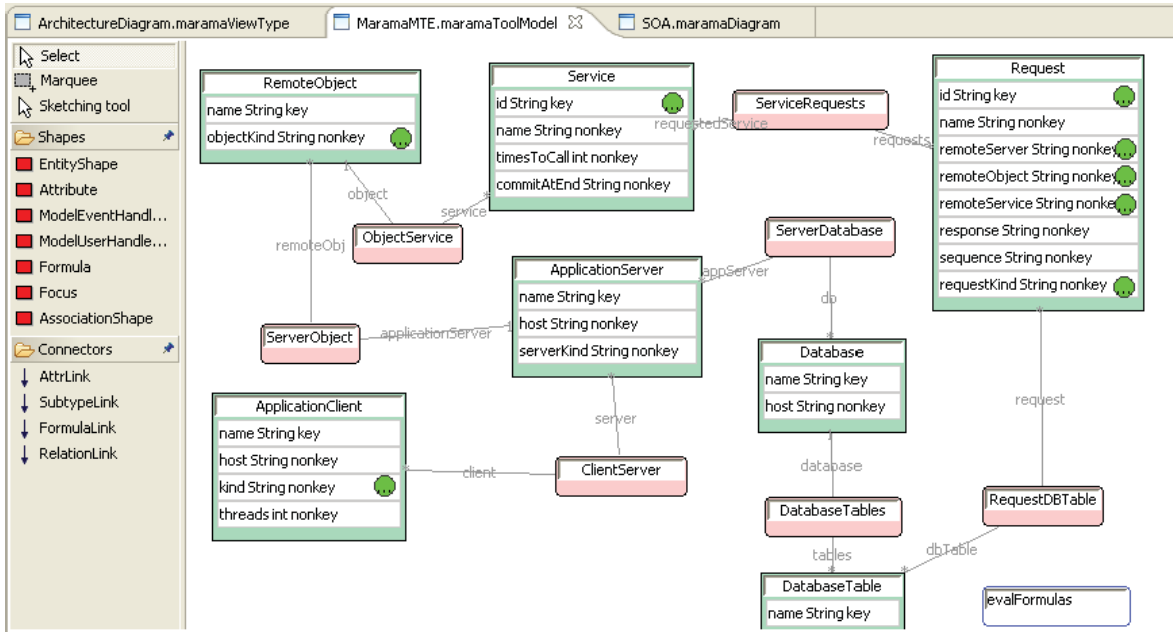


Figure 6.2: Defining a metamodel instance using Marama, adapted from Grundy et al. [139]

In this thesis, Whitehorse is the only metamodeling environment evaluated that is closed-source, and can only be used if a valid Visual Studio license is provided. A modelling environment generated through Whitehorse still requires the Visual Studio IDE to execute, and thus end users require a licensed copy of Visual Studio to host the environment and create model instances.

#### 6.2.4 Marama

The *Marama* project started off as an application to improve the quality of metamodeling tools generated by the *Pounamu* project [376]. *Pounamu* was originally a project to specify and generate independent visual modelling tools, and *Marama* was used to generate an Eclipse-based editor instead [137]; *Pounamu* now appears to be discontinued.

The *Marama* project is implemented within the Eclipse environment using the Java language, and generated modelling environments are hosted within the Eclipse environment. The project is based heavily on the EMF and GEF projects discussed in this chapter, and each modelling concept has an associated metamodel. Once a metamodel has been defined, model instances can be created visually using the graphical environment discussed later in Section 6.3.5.

*Marama* defines its own meta-metamodel for the definition of metamodels, which is referred to in this thesis as *MaramaModelProject* (MMProject), and is adopted from an extended Entity-Relationship paradigm [213]. However, this meta-metamodel is not defined in terms of itself; rather, the meta-metamodel is defined in terms of an Ecore model instance<sup>11</sup> [91]. There is no explicit support for the extensibility of existing metamodels defined within *Marama*.

Metamodel instances can be defined visually as in Figure 6.2. The *MaramaTorua* project [158] supports the definition of *Marama* metamodels from XML Schemas or the Microsoft *schema inference engine*, but this is not provided as part of the *Marama* project itself. Constraints (known as *Formulae* in *Marama*) can be supplied either as textual OCL constraints, or the OCL constraints can be constructed

<sup>11</sup>The *MaramaModelProject* meta-metamodel instance is provided in `project.ecore`.

Requirement	EMF	ArgoUML	Whitehorse	Marama
Language	Java	Java	.Net	Java
Runtime Environment	Standalone	ArgoUML	Visual Studio	Eclipse
Open Source	✓ (EPL)	✓ (New BSD)	✗	✓ (MPL)
Version	2.5	0.28.1	2010	20101022
Visual Definition	✓	✓	✓	✓
Graphical Editor	✗ (GMF)	✓	✓	✓
Metamodel Extensibility	✓	✗	✓	✗
Meta-metamodel	Ecore	UML	DMF	MMProject
MDA-compliant M3	✓	✓	✗	✗
Metamodel Sources	Ecore Annotations UML EMOF XML Schema Dynamic	UML Profiles JARs	DSL Editor	Marama XML Schema Microsoft WS
Verification Languages	Java OCL XML Schema	Java OCL	.Net	Java OCL
OpenBRR Rating	120	83	n/a	60
Average Quality	3.5	4.2		2.5
Average Support	4.5	1.0		1.0
Average Documentation	5.0	2.0		2.0
Average Adoption	4.0	2.0		3.0
Average Architecture	4.0	5.0		1.0

Table 6.1: Requirement comparisons between existing metamodeling environments

visually using *MaramaTatau* [214]; user-defined extensions for these constraints can also be provided in Java [213].

### 6.2.5 Summary

A summary of these modelling environments is provided in Table 6.1. From the evaluation of existing metamodeling environments, the Eclipse Modeling Framework was found to be the most suitable for the implementation of IAML, due to its openness, industry support, documentation, and easy integration with the Eclipse environment. Marama is very promising, but without the same level of industry support as the EMF project and high quality documentation it may be difficult to adapt the toolset to new environments and situations.

## 6.3 Graphical Modelling

Graphical modelling provides an environment where an underlying model instance, as described by a metamodel, may be represented as a visual model; this is achieved through the association of shapes and relationships to the metamodel element types, through some form of mapping. This section will

evaluate a range of existing graphical modelling environments, with the intent of reusing one of these environments rather than having to develop one from scratch. As in the previous section, these environments were selected based on their industry support, integration with other technologies within a model-driven tool stack, academic activity, and source code available through an open source license.

A common design pattern found with model-driven graphical modelling environments is that this graphical-to-metamodel mapping process is decomposed into smaller model instances which are later combined. For example, the design of shapes, and their association to model elements, can be described as a “gallery” metamodel to support common shapes across multiple editors [136, pg. 61].

### Additional Evaluation Criteria

Along with the common comparison criteria discussed earlier in Section 6.1.1 and Section 6.1.2, each graphical modelling environment will be evaluated on a number of additional criteria:

- *Diagram Technology*: What is the underlying technology used for *rendering* the graphical interface – that is, the diagram editor – of the generated tool?
- *Model Source*: What is the underlying technology used for representing the model instances that are being modified by the graphical editor?
- *Shape Metamodel*: Is there a metamodel defined for describing the shapes, or figures, of the graphical editor with an abstracted model? As discussed at the beginning of this section, model-driven graphical modelling environments will often have a separate metamodel for these shapes in order to support a reusable gallery of shapes.
- *Notation Metamodel*: When a model instance is created graphically, there needs to be some means of storing the diagrammatic elements of the visual representation, such as positions, sizes, fonts and colours. Is the graphical notation instance data stored as a model instance with a corresponding separate metamodel? That is, is this graphical environment implemented according to a model-driven development approach? If a separate metamodel is defined, the represented model instance does not have to be polluted with notation information.
- *Automatic Layout*: Does this environment provide built-in support for automatically laying out diagrams? As discussed by Yap et al. [370], automatic layout algorithms can vastly improve the usability and efficiency of creating visual model instances, but the implementation of these algorithms can be difficult and would ideally be provided as part of the graphical environment.
- *Printing Support*: Does this environment provide built-in support for printing diagrams via an external printer?
- *Export Diagrams*: Does this environment provide built-in support for exporting diagrams to image files? If the environment supports sending images to a printer, then implementation of this feature should be straightforward.
- *Extensible Generator*: Is this graphical environment intended to be extended by developers? Can the editor developer interact with or modify the logic and templates behind the generation of the final editor? This extensibility can allow an editor developer to add functionality without having to modify generated code.

### 6.3.1 Graphical Editor Framework

The *Graphical Editor Framework* (GEF) [136] is an Eclipse-based framework which aims to assist in the development of rich graphical editors and user interface views. It extends on its underlying technologies, Draw2D and SWT, to provide a higher-level interface using concepts such as shapes and connections, and also provides a “toolbox” interface that can be used to create new shapes. GEF acts as the controller between an underlying model and the interface, following the Model-View-Controller (MVC) approach [195].

GEF is designed to operate against some type of model, which does not need to be any particular metamodeling framework; in most cases, this model is a set of custom Java classes. GEF is therefore easy to integrate with EMF, as EMF can be used to generate custom Java classes [9]. The framework is used as the basis for a number of other graphical environments, such as GMF and Marama which are both discussed later in this section.

Shapes in GEF are instances of Java classes, and GEF does not provide an independent metamodel for describing these shapes; this is in contrast to the *gmfgraph* metamodel approach provided by GMF as discussed in the next section. Because the underlying model is simply instances of Java classes, there is also no notation metamodel.

GEF is therefore not used to generate graphical editors, in contrast to how many of the other technologies in this section are designed; rather, it exists as a runtime wrapper around existing Java software and different forms of model instances. This means that GEF’s extensibility relies on the developer manually extending the GEF framework directly, as opposed to supplying the framework with different configuration inputs.

### 6.3.2 Graphical Modeling Framework

The *Graphical Modeling Framework* (GMF) [136] aims to extend the functionality offered by EMF and GEF into a higher-level framework for the development of graphical editors. GMF abstracts the functionality of graphical editors into four metamodels – *gmftool* for the palette tooling definition, *gmfgraph* for the graphical shape definitions, *gmfmap* for the mapping of shapes to model elements, and *gmfgen* for the generation of the editor itself – and provides a higher-level view of a graphical editor than the underlying technologies.

The developer of a graphical editor is given the task of implementing model instances according to these metamodels, each of which are implemented and serialised using EMF. GMF can then compile these definitions into a finished graphical editor, with each editor provided as an Eclipse plugin. To assist in the development of these metamodels, GMF provides automated tools and wizards to simplify the development of these model instances. A screenshot of the *GMF dashboard wizard* is provided in Figure 6.3; this wizard highlights the steps necessary to develop a GMF-based graphical editor.

The translation of these graphical model instances into other model instances is achieved with JET, and the generation of the Java source code for the final diagram editors is implemented using Xpand; both of these technologies are discussed later in Section 6.4. These templates are extensible using *dynamic templates*, which allow the editor developer to customise the generated editors without modifying the GMF framework directly; the implementation of dynamic templates in an editor is discussed later in Section 7.4.2.

GMF uses the *GMF Notation* metamodel for annotating model instances with the information necessary to visually represent the model instance. By default, the model instance and the associated



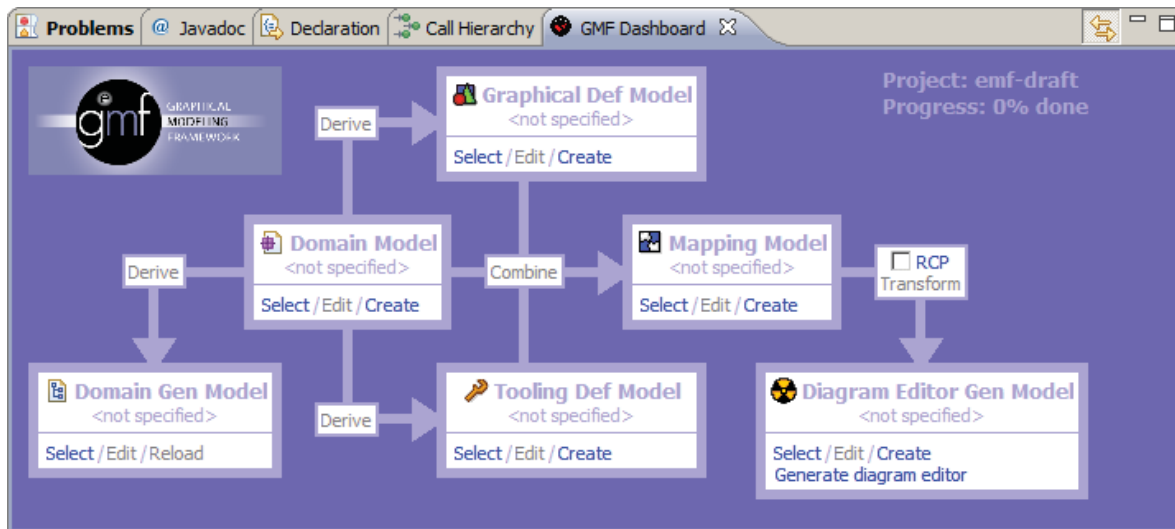


Figure 6.3: The GMF Framework Wizard in Eclipse, illustrating the steps necessary to create a graphical editor

visual representation model instance are stored in separate resources<sup>12</sup>. Each of these metamodels are implemented as an Ecore metamodel under the EMF framework, as discussed earlier in Section 6.2.1.

### 6.3.3 ArgoUML

As discussed earlier in Section 6.2.2, ArgoUML was originally developed as a UML diagram editor, which has since been used as a graphical editor for other UML-based models, such as ArgoUWE for UWE. However, since ArgoUML is designed to edit UML diagrams only, it is unlikely that it can be used to edit instances of metamodels that are not related to UML. The underlying diagram technology for ArgoUML is a custom-built *Graph Editing Framework* (GEF\*<sup>13</sup>) [286]. GEF\* is published as a standalone project under the open source BSD license.

Model element shapes in ArgoUML are not defined according to any metamodel, but are Java instances of GEF classes that are implemented manually<sup>14</sup>. The underlying model instance and the visual representation are still represented separately; the visual representation is serialised using the *Precision Graphics Markup Language* (PGML) [286], which itself is defined using a DTD [332].

As discussed earlier in Section 6.2.2, extensions of ArgoUML rely on defining *modules* and *plugins* [322]. There is no generation stage, and thus there is no graphical editor generator framework that can be extended. However, CASE tools that are based on ArgoUML automatically obtain the notation and layout functionality of the framework, and thus ArgoUML can be considered high-level.

<sup>12</sup>For example, a UML model instance could be stored in `instance.uml`, and the visual representation would be stored in `instance.uml_diagram`

<sup>13</sup>Since the Eclipse Graphical Editor Framework discussed earlier also uses the GEF initialism, the ArgoUML Graph Editing Framework will instead be referred to in this thesis as GEF\*.

<sup>14</sup>For example, the UML model element *FinalState* is implemented within ArgoUML by the Java class `org.argouml.uml.diagram.state.ui.FigFinalState`.



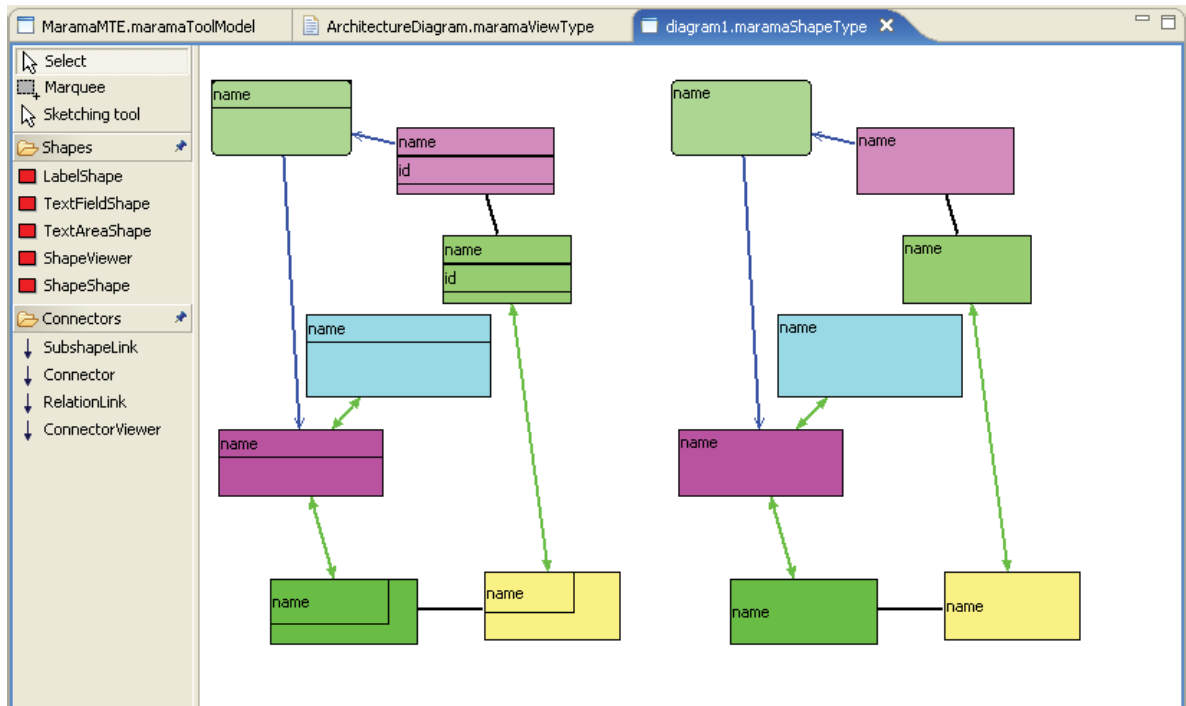


Figure 6.4: The *Shape Designer* of Marama showing the shape design and corresponding concrete view, from Grundy et al. [139]

### 6.3.4 Whitehorse

As discussed earlier in Section 6.2.3, the *Whitehorse* project was introduced as a way of developing graphical editors for domain-specific languages. The underlying environments of the Whitehorse DSL technologies are the Design Surface Framework (similar to GEF) and the Domain Model Framework (similar to EMF) [61]. Both of these technologies form part of the Visual Studio DSL SDK, and are not independent projects in the vein of GEF and EMF.

The shapes of model elements are represented according to the *DslDesignerDiagram* (DDiagram) metamodel<sup>15</sup>. The diagram model instance is *project-specific* and implies that there is no common shared notation metamodel between different Whitehorse projects; however the underlying model instance and the visual representation are still stored separately. These shapes directly associate with model elements in the underlying metamodel, so it is not possible to define a shared gallery of model element shapes.

When developing a graphical editor using Whitehorse technology, the developer does not need to concern themselves with concepts such as layout or printing, and thus can be considered a high-level framework. The graphical component of Whitehorse does not directly support extensible templates, and changes to the graphical editor must be performed manually in the .Net language.

### 6.3.5 Marama

Discussed earlier in Section 6.2.4, the *Marama* project is designed to provide all the aspects necessary to develop a graphical model-driven environment. The key difference between Marama and GMF is that the associations between the metamodel and the graphical environment, as well as the constraint

<sup>15</sup>No XML namespace is provided for this metamodel, but the .Net namespace for this metamodel is defined as `microsoft.VisualStudio.Modeling.DslDesigner.DslDesignerDiagram`.

Requirement	GEF	GMF	ArgoUML	Whitehorse	Marama
Language	Java	Java	Java	.Net	Java
Runtime Environment	Eclipse	Eclipse	ArgoUML	Visual Studio	Eclipse
Open Source	✓ (EPL)	✓ (EPL)	✓ (New BSD)	✗	✓ (MPL)
Version	3.5.2	2.2	0.28.1	2010	20101022
Diagram Technology	Draw2D	GEF	GEF*	DSF	GEF
Model Source	Java	EMF	JavaBeans	DMF	EMF
Shape Metamodel	-	gmfgraph	-	DDiagram	Pounamu
Notation Metamodel	n/a	Notation	PGML	Custom	MDiagram
Automatic Layout	✓	✓	✓	✓	✓[370]
Printing Support	✗	✓	✓	✓	✗
Export Diagrams	✗	✓	✓	✓	✓
Extensible Generator	✗	✓	✗	✗	✗
OpenBRR Rating	99	98	83	n/a	60
Average Quality	3.7	2.5	4.2		2.5
Average Support	3.5	4.0	1.0		1.0
Average Documentation	4.5	4.0	2.0		2.0
Average Adoption	4.0	3.5	2.0		3.0
Average Architecture	1.5	4.0	5.0		1.0

Table 6.2: Requirement comparisons between existing graphical modelling environments

definitions, are all intended to be implemented visually. For example, shapes in GMF are intended to be defined using a tree-based editor, whereas the Marama *shape designer* – illustrated here in Figure 6.4 – allows for the graphical definition of a shape. This shape can then be used by the Marama *view definer* to associate metamodel elements with defined shapes.

The Marama project was started before the GMF framework had been released, and the project had to manually implement much of the functionality provided by GMF, such as validation decorators, complex shapes and automatic layout algorithms [370]. At the time of writing, the primary source of Marama documentation was its user manual [213]. The underlying metamodel for defining shapes, tooling and views in Marama is Pounamu, and this metamodel is defined through a collection of DTD instances. The diagram notation metamodel *MaramaDiagram*<sup>16</sup> (MDiagram) associates shapes to model element instances for a given model instance, and is represented in terms of an Ecore model instance [91].

The implementation of Marama through a combination of DTD and Ecore model instances highlights a strength of model-driven approaches, in that many different model sources can be integrated together. However, it may be preferable in the future to migrate the shape metamodel to an Ecore or XML Schema representation, as DTDs are not M3-compliant in the MDA, as discussed earlier in Section 3.1.5; Ecore metamodels or XML Schemas can provide stronger structural constraints than DTDs [343, 314]; and Eclipse-based tooling support for Ecore is much stronger than those for DTDs.

### 6.3.6 Summary

A summary of these graphical environments is provided in Table 6.2, and the GMF framework was selected as the most suitable environment to implement a graphical editor for IAML. As GMF is both an Eclipse technology and built upon EMF, it is also expected that it should be straightforward to integrate these editors with the underlying EMF metamodeling environment selected in the previous section.

While Marama is very similar to GMF, there are two major factors that influenced the decision to use GMF instead: firstly, as an Eclipse Foundation project, GMF has more use in industry and a higher quality of documentation, and this is reflected in the OpenBRR evaluations. Secondly, GMF generated editors are designed to be extensible, which will simplify the implementation of custom extensions (such as model completion, discussed later in Section 6.5).

## 6.4 Code Generation

Another design goal of the proof-of-concept implementation of IAML is the implementation of a code generator to translate IAML model instances into executable web applications, forming an important part of the evaluation in Chapter 8. Code generation is a model-to-text transformation which translates a platform-independent model at a higher level of abstraction, into platform-specific models at lower levels of abstraction, as discussed earlier in Section 3.1.7.

As with the rest of this chapter, there are a number of existing technologies for implementing model-driven code generation, and this section will be used to discuss and evaluate these technologies for suitability. As our previous two technologies are both heavily dependent on the Eclipse environment, it must also be easy to integrate a selected code generation technology into an Eclipse environment. This means that other environments such as Visual Studio and ArgoUML will not be considered.

### Additional Evaluation Criteria

Along with the common comparison criteria discussed earlier in Section 6.1.1 and Section 6.1.2, each code generation technology will be evaluated on a number of additional criteria:

1. *Navigation Language:* Within a template, what language is used to navigate through a model instance? That is, what language is used to browse through the model instance graph?
2. *Query Language:* Within a template, what language is used to query a model instance within expressions? That is, what language is used to translate a model instance into an expression? If these two languages are different, each language may be simpler and easier to test, but there may be more of a learning curve for new developers.
3. *Underlying Metamodel:* What is the underlying metamodel for the code generation templates? The increased expressibility and enforced structure of a richer metamodel such as EMF may make it easier to write code generation templates, when compared to a lower-level metamodel such as Java.

---

<sup>16</sup>This name is derived from the root element in the Marama `diagram.ecore` metamodel.

```

<%@ jet imports="java.util.* org.openiaml.model.model.*" %>
<%@ include file="copyright.javajetinc" %>
<% InternetApplication root = (InternetApplication) argument; %>
<html><body>
<h1>List of Pages</h1>
<ul>
<% for (Iterator i = root.getScopes().iterator(); i.hasNext(); ) {
    Scope s = (Scope) i.next();
    if (s instanceof Frame) { %>
        <li><a href="<%=((Frame) s).getUrl()%>"><%=s.getName()%></a></li>
    }
} %>
</ul>
</body></html>

```

Listing 5: Internet Application code generation using a JET template

4. *Third party extensibility*: Does the technology allow third parties – that is, the model developers themselves – to extend the provided code generation templates? This can improve developer usability significantly, as the developer does not need to access the source code of the modelling environment to make changes, but could instead just add extensions to the given environment.

### 6.4.1 XSLT

As discussed earlier in Section 3.1.7, *Extensible Stylesheet Language Transformations* (XSLT) [341] can be used to translate XML instances according to some simple translation rules. Since EMF models are often represented as XML files, and can otherwise be exported into XMI, it would be possible to define XSLT files to translate a given IAML model instance into a web application. A model instance can be navigated using the XPath language [347], and queried using the XQuery language [349].

The XSLT approach may be executed according to an underlying metamodel if the source metamodel is first translated to an XML Schema instance – for example, translating an XMI-serialised model valid to an Ecore metamodel, into an XML model valid to an XML Schema. With regards to EMF-based metamodels, XML Schema is less expressive than Ecore; some metamodel information cannot be translated to XML Schema constructs, such as bidirectional references or target reference types [314, pg. 179–235].

XSLT instances are serialised using the general-purpose machine-readable language XML, and are consequently more verbose than domain-specific textual languages. This makes it difficult to develop stylesheet instances without adequate tool support. XML stylesheets do not directly support third-party extensibility, as all component stylesheets must be *explicitly* included and exist at the time the stylesheets are compiled.

Since XSLT is only a specification, there are many XML *template engine* implementations for a number of languages, and stylesheets can be executed within any runtime environment. Providing an OpenBRR evaluation against a range of these template engines is outside the scope of this thesis, and no such evaluation will be provided. As discussed by Gottlob et al., four popular existing engines include XALAN, Saxon, XT and IE6 [131].

### 6.4.2 Java Emitter Templates

Java Emitter Templates (JET) [136], part of the Eclipse Model to Text (M2T) project, are a template-based code generation technique. The syntax of JET templates are very similar to Java Server Pages (JSP) [159], as illustrated in Listing 5; in fact, the template engine was built using source code from the JSP implementation *Tomcat* [314, pg. 375]. JET may be used within an Eclipse environment, but this environment is not necessary; JET may be used in a standalone environment, as the only runtime requirement is EMF, which may also be provided in a standalone fashion.

JET templates are evaluated against instances of plain old Java objects (POJOs) [171] – allowing them to interact directly with Java code – and can also be used to iterate over XML data. Consequently, Java is both the navigation and query languages used in JET. JET templates also can import existing libraries of templates, known as *taglibs*, to extend the functionality of the code generator.

JET templates can also be extended by third parties through runtime `@include` directives. To generate Java source code from metamodels, the Eclipse Modeling Framework uses JET templates in this fashion, and these generated source codes may be extended by third parties [314, pg. 379]<sup>17</sup>.

### 6.4.3 openArchitectureWare

The *openArchitectureWare* project (OAW) began as an independent implementation of various model-driven technologies [87]. This included a code generator (*OAW Xpand*<sup>18</sup>), a textual metamodel definition language (*Xtext*), a model extensibility framework (*Xtend*) and an OCL-like suite of model checks (*Checks*). Each of these technologies are written in Java, and require the Eclipse environment at runtime. These technologies are internally based on EMF model instances, and allows code generation templates to be developed using an underlying metamodel.

OAW Xpand uses a custom language for defining templates, as in Listing 6; the *Xpand expressions* sub-language is “a syntactical mixture of Java and OCL” [87], which can also be used to navigate through the model instance separately from Xpand. OAW Xpand supports the concept of *dynamic templates*, which supports a form of aspect-oriented programming. In particular, an existing template can be overridden or extended by a new third party template.

Since this research was started, all development focus on openArchitectureWare has essentially ceased, and the four subprojects have moved into new Eclipse Foundation projects. Consequently, the OpenBRR evaluation of OAW Xpand is not as positive as the evaluations of the other code generation technologies, as there has been no recent development activity.

### 6.4.4 Xpand

As another part of the Eclipse M2T project, *Xpand* [136] started as an Eclipse-branded branch of the *OAW Xpand* component of openArchitectureWare. It appears that the branch was started to satisfy the model-to-text code generation requirement of GMF, as the GMF code generation templates to generate graphical editors could be extended by third parties to add additional functionality to the generated graphical editors.

<sup>17</sup>For example, JET templates are used within the proof-of-concept implementation of IAML to extend the Eclipse Modeling Framework to enable XSD references, as discussed by resolved issue 251: *XSD References cause GMF editors to crash with transaction exception*.

<sup>18</sup>In this thesis, the openArchitectureWare implementation of Xpand is named *OAW Xpand* in order to distinguish this implementation from the Eclipse implementation, which is simply named *Xpand* as in the next section.

```

«IMPORT iaml»
«EXTENSION template::GeneratorExtensions»

«DEFINE root FOR model::InternetApplication»
«EXPAND Copyright::copyrightTemplate»
<html><body>
<h1>List of Pages</h1>
<ul>
«FOREACH scopes.typeSelect(model::visual::Frame) AS frame»
  <li><a href="«frame.url»">«frame.name»</a></li>
«ENDFOREACH»
«ENDDEFINE»

```

Listing 6: Internet Application code generation using an OAW Xpand Template

Requirement	XSLT	JET	OAW Xpand	Xpand
Language	XML	Java	Java	Java
Runtime Environment	n/a	Standalone	Eclipse	Eclipse
Open Source	n/a	✓ (EPL)	✓ (EPL)	✓ (EPL)
Version	2.0	0.8.2	4.3.1	1.0.0
Navigation Language	XPath	Java	Xpand expr.	Xpand expr.
Query Language	XQuery	Java	Xpand expr.	Xpand expr.
Model-driven	✗	✗	✓	✓
Third party extensibility	✗	✓	✓	✓
OpenBRR Rating	n/a	105	94	104
Average Quality		4.5	3.7	3.0
Average Support		4.0	3.0	4.0
Average Documentation		4.0	2.5	5.0
Average Adoption		3.0	4.0	3.0
Average Architecture		4.0	4.0	4.0

Table 6.3: Requirement comparisons between existing code generation environments

Since its integration into Eclipse, Xpand has seen a number of improvements, particularly in terms of standardisation, functionality and performance. With the release of GMF 2.2, Xpand was extended to support QVTo model queries [53]. An improved code editor has also been released, along with significant performance improvements. The underlying expression language remains a mix of Java and OCL, as discussed in the previous section. As part of the Eclipse Foundation-supported ecosystem, Xpand is still under active development with a wide range of support options.

### 6.4.5 Summary

A summary of this investigation is provided in Table 6.3. As discussed earlier in Section 3.1.8, QVT and ATL are omitted as neither technology supports *model-to-text* transformations, which is a requirement for code generation. With respect to these technologies, the decision was made to use the Xpand framework to implement code generation within IAML. The proof-of-concept implementation uses the OAW Xpand project, as the Eclipse-branded Xpand project had not yet been announced when work on the proof-of-concept implementation began.

Finally, it must be noted that none of these evaluated approaches natively support any form of round-trip engineering of model transformations, as discussed earlier in Section 3.1.8. It may be possible to implement round-trip engineering support in a model-to-text transformation implementation<sup>19</sup>, and this research remains an important area of future work. This limitation will not impact on the proof-of-concept implementation of IAML however, as round-trip engineering is not a design goal of this thesis.

## 6.5 Model Completion

Discussed earlier in Section 3.2, *model completion* is the automated process of taking an incomplete base model and adding elements to this model to create an *intended model* according to sensible default rules. Model completion is particularly important for the implementation of *Wires* as discussed in Section 4.9.2, as this means the logic and semantics behind a *Wire* can be implemented within an additional model completion layer, and the underlying metamodel and resulting code generation layers can remain unchanged.

There are a number of different technologies that may be used to implement model completion rules. Almost any language that supports some form of rule-based inference can be used; this includes rule engines such as Drools [311], Jena [221] and Jess [112]. The Java specification request JSR-94 [306] covers the definition of a Java rule engine API, and most commercial rule engines are implementations of this standard. In this section, these three rule engines will be evaluated by implementing the *default checkbox rule* introduced earlier in Section 3.2.2.

As model completion is designed as a model transformation operating within the same source and target metamodels, model-to-model transformations may also be used to implement model completion. However this section will not deal with the implementation of model completion in these general-purpose transformation engines, as it is expected that a dedicated rule engine grammar and implementation will provide better expressibility and performance. For example, the three rule engines discussed here utilise the Rete algorithm [311] in order to improve performance, whereas this approach is novel in existing model transformation engines [29]. Similarly, rule engines may be used to implement model-to-model transformations, but this discussion is outside the scope of this thesis.

### Additional Evaluation Criteria

Along with the common comparison criteria discussed earlier in Section 6.1.1 and Section 6.1.2, each rule-based technology will be evaluated on a number of additional criteria:

1. *Uses POJO model*: Does the underlying model used by the rule engine support POJOs? That is, can the rule engine be inserted into an existing Java application without any additional translation steps necessary? Since EMF model instances are accessible as POJOs, this means that the rule engine can be integrated with an EMF-based modelling environment easily. Any translation steps necessary will reduce the performance of model completion, and impact the integration and traceability of the environments.
2. *Multiple interface inheritance*: Can the rules use multiple interface inheritance? The IAML metamodel has heavy usage of both multiple interface inheritance. If a rule engine cannot sup-

<sup>19</sup>For example, GMF already uses *trace models* to support traceability on its model-to-model transformations.



```
[exampleModelCompletion:
  (?P rdf:type iaml:BooleanProperty)
  (?C iaml:children ?P)
  noValue(?E iaml:for ?P)
  makeSkolem(?A, ?P, ?C)
  ->
  (?A rdf:type iaml:Checkbox)
  (?A iaml:id 'generated')
  (?A iaml:isGenerated 'true')
  (?A iaml:for ?P)
  (?C iaml:children ?A)
]
```

Listing 7: Example rule implementation in Jena

port multiple interface inheritance, a metamodel can be extended to single interface inheritance, but this indicates additional development effort that may not be necessary.

3. *R2ML Translator*: Does a translator exist to translate R2ML rules [125] into this language, and vice versa? An R2ML translator may permit the automatic translation of rules written in one rule engine into another rule engine, improving interoperability.
4. *Custom Rule DSL*: Does this rule engine directly support the definition of a domain-specific language for model completion rules? By providing a DSL for a given set of rules, rule developers can express rules more easily. This is not strictly necessary, as a domain-specific language can be written outside of the scope of the rule engine.

### 6.5.1 Jena

*Jena* is an open source Java framework for building semantic web applications by providing a number of RDF-based APIs. Knowledge is represented internally as an RDF graph according to the W3C recommendations for the Semantic Web [221]. These graphs may then be extended with components such as the Semantic Web query language RDQL [221, 304] and a hybrid forward/backwards chaining inference engine, to support queries and derivation on this knowledge.

Since Jena is focused on expressing and evaluating RDF triples, model instances must be translated into an RDF format in order to implement model completion rules using Jena. This may be performed manually, however the translation of a model instance into an RDF format will lose the semantic information of the underlying metamodel – such as subtype hierarchies and containment relationships – unless these metamodel constructs are emulated in the graph.

The Web Ontology Language (OWL) [351] is a standard for representing the higher-level knowledge of data, and in particular can be used to represent a metamodel of a model instance using RDF. The *emftriple* project can be used to translate Ecore models into OWL models [154], which can then be loaded as part of an RDF-based model completion process.

There are a number of constraints for the translation of an Ecore model into the OWL format using *emftriple*. The tool assumes that there are unique names for all classes, properties and enumerations in the metamodel; and struggles with external references to Ecore metamodels located within the

```

(deftemplate IamlFactoryImpl (declare (from-class IamlFactoryImpl)))
(deftemplate BooleanPropertyImpl (declare (from-class BooleanPropertyImpl)))
(deftemplate CheckboxImpl (declare (from-class CheckboxImpl)))
(deftemplate PageImpl (declare (from-class PageImpl)))

(defrule model-completion-example "Example rule of model completion."
  (BooleanPropertyImpl (OBJECT ?property))
  (PageImpl (OBJECT ?container))
  (IamlFactoryImpl (OBJECT ?factory))
  (test ((?container getChildren) contains ?property))
  (test (eq (?property getEditor) nil))
  =>
  (bind ?o (?factory createCheckbox))
  (?o setFor ?property)
  (?o setIsGenerated TRUE)
  ((?container getEditors) add ?o)
  (add ?o)
)

```

Listing 8: Example rule implementation in Jess

project<sup>20</sup>. These constraints may be resolved in the future to support automated Ecore translation.

Once the Ecore model has been translated into an OWL format, and a model instance represented in EMF is translated into an RDF format, the Jena inference engine can be used as in Listing 7. The *makeSkolem* builtin predicate allows for a new blank node to be generated based on the uniqueness of a particular set of parameters, and is essential to a Jena implementation of model completion.

However, there is no easy way to implement the insertion cache concept using Jena rules; that is, the new node created by *makeSkolem* is inserted into the model immediately. Since model completion rules are orderless – that is, the order in which they are executed must not matter – the lack of an insertion cache is not fatal for a Jena-based implementation. However, a Jena-based implementation of model completion may need to define a new *makeCachedSkolem* function in order to correctly satisfy the requirements of model completion.

The Jena framework supports the integration of additional rule engines in order to evaluate model instances, such as Pellet [310], Racer [142] and FaCT++ [325]. These engines can provide a limited amount of additional functionality and improved performance; however only the default Jena reasoner will be evaluated in this thesis.

### 6.5.2 Jess

Jess is a closed-source rule engine written in Java, which is designed to operate both on facts and POJOs. As discussed at the beginning of this chapter, a closed-source component would make the proof-of-concept implementation difficult to redistribute, and would also hinder the implementation of rule engine extensions, such as the *insertion cache*. Jess was a driving force behind the design of JSR-94, and is the reference implementation for the original specification [306]. An R2ML/Jess translator is also available [245].

The underlying model that Jess operates within is sets of facts, which can be inserted and retracted from within the execution environment. Java objects can also be inserted as facts into the rule engine,

<sup>20</sup>In particular, all references to the Eclipse XSD metamodel had to be removed before the ontology could be generated.

```
rule "Example rule"
when
  p : BooleanProperty( )
  not ( Editor ( for == p ))

then
  Checkbox c = handler.generatedCheckbox(p);
  handler.setFor(c, p);
  cache.add(c, drools);

end
```

Listing 9: Example rule implementation in Drools, adapted from Wright and Dietrich [369]

and if these objects are also POJOs, then the bean properties can also be accessed directly as facts. However, the structure of inserted POJOs must be defined manually as part of the rules, as illustrated in Listing 8. Consequently model completion rules can be difficult to implement in Jess if the underlying objects are based on EMF.

Critically, Jess does not currently support multiple interface inheritance [111], where a class can have more than a single supertype. This means that there are certain classes of rules and type hierarchies that cannot be described using Jess; and the sample model completion rule proposed by Wright and Dietrich [369] cannot be implemented in Jess without first flattening the inheritance hierarchy.

### 6.5.3 Drools

*Drools*, also known as *JBoss Rules*, is an open source business rule management system and inference rule engine implemented in Java [311]. Inference rules are evaluated using an enhanced implementation of the Rete algorithm [311]. Drools natively provides an expressive textual language for defining inference rules, but also supports the integration of a custom rule DSL to improve the productivity of defining rules within certain domains.

The underlying model that Drools operates within is simple POJOs, making it easy to integrate into an existing Java-based software system. Unlike Jess, the structure of inserted POJOs do not need to be defined as part of the rule base; this means that all metamodel properties and operations are always accessible to a Drools rule. Alternatively, metamodel reflection is possible using the EMF API.

Along with Jena, the design and implementation of R2ML was heavily supported by Drools [126], and thus the translation of R2ML rules to Drools rules and vice versa is well supported [245]. The Drools engine also supports an implementation of the JSR-94 specification.

A model completion rule is implemented in Drools using an inference rule, as described by Wright and Dietrich [369] and illustrated in Listing 9. In particular, the use of an *insertion cache* assists in the implementation of the stratification technique of model completion, where newly created model elements are inserted into a cache before being reinserted into the working memory.

### 6.5.4 Summary

A summary of this investigation is provided in Table 6.4. Each of these engines were found to be expressive enough to describe many different types of rules. The UServ Business Rules Model 2005

Requirement	Jena	Jess	Drools
Language	Java	Java	Java
Runtime Environment	Standalone	Standalone	Standalone
Open Source	✓(BSD)	✗	✓(Apache)
Version	2.6.4	7.1p2	4.0.7
Uses POJO model	✗	✓	✓
Multiple interface inheritance	✓	✗	✓
R2ML Translator	✓	✓	✓
Custom Rule DSL	✗	✗	✓
OpenBRR Rating	99	n/a	101
Average Quality	3.8		2.3
Average Support	2.0		4.5
Average Documentation	2.5		5.0
Average Adoption	3.0		4.0
Average Architecture	4.0		5.0

Table 6.4: Requirement comparisons between existing model completion rule engines

[126] provides a “common benchmark” set of rules which may be implemented to compare language expressiveness, and each of these evaluated rule engines satisfies these requirements [126, 245].

Even though there are a large number of rule engines in production, there are very few reliable benchmarks published on these engines, making it difficult to compare engines in terms of performance. This may be due to the fact that rule engine performance depends heavily on the type of rules, the size of the rules, and the size of the input data sets. The UServ benchmark is only concerned with expressiveness, and cannot be used to compare performance characteristics.

In IAML, the decision was made to use the Drools rule engine in order to implement the model completion framework. As the engine was open source, it was fairly straightforward to implement the concepts behind model completion (such as stratification through the insertion cache), as discussed by Wright and Dietrich [369]. This work evaluated performance metrics against a test suite of sample input models, and a wide range of model completion rules, and found that Drools was a suitable target platform to implement model completion.

## 6.6 Model Instance Verification

As discussed earlier in Section 3.3, *model instance verification* is the process of evaluating syntactically correct model instances against constraints to verify they are correct with respect to the intended function of the modelled domain. Each of these verification rules may be implemented in a different technology; for example, the structural definition of the language – which includes a type of model instance verification – is implemented as part of the metamodeling environment, whereas reference cycles can be detected using a different framework.

In this section, a simple example constraint will be used to *illustrate* the design and expressibility of each verification language. The constraint *infinitely redirects* is a constraint that identifies web application pages that, when visited, would continually redirect the user in a non-terminating loop. This example constraint is an instance of a property that is very difficult to identify through source code analysis, and is much easier to identify within a model of the system. This constraint may

be implemented within both functions- or relations-based verification languages, and may also be evaluated using a model checking approach.

### Additional Evaluation Criteria

Along with the common comparison criteria discussed earlier in Section 6.1.1 and Section 6.1.2, each model verification technology will be evaluated on a number of additional criteria:

1. *Uses POJO model:* Does the underlying model used by the language implementation support POJOs? That is, can the verification engine be inserted into an existing Java application without any additional translation steps necessary? The benefits of this approach are identical to those as discussed in the previous section.
2. *Supports subtypes:* Does the language support the concept of typed elements natively? If the language does not, then the type system of the metamodel will need to be supported through additional predicates, which may introduce development problems or impact performance. If the language cannot support inference, then the range of possible constraints that may be implemented will be heavily restricted.
3. *Derivation cause:* Can the verification engine provide any type of trace for a failed constraint? Traceability will allow for a more informative error message to be displayed to the model developer, improving usability.
4. *Implementation:* Is an implementation of the language provided? This requirement is critical for the technology to be considered as part of the proof-of-concept implementation in Chapter 7.
5. *Transient properties:* Is it possible to define the properties necessary for verification as transient? That is, can constraints define additional properties and predicates without having to modify the source metamodel? A language that does not support transient properties will need to use a separate verification metamodel at runtime, increasing development effort.
6.  *$L_{ver}$  expressiveness:* With respect to the categories defined earlier in Section 3.3.2, is the expressiveness of this verification language based on functions or relations? This may impact on the performance and decidability of constraints expressed within this language.
7. *Higher-order logic:* Does this language support the definition of additional functions or relations using higher-order logic? This may include support for transitive closure over relations, or aggregation over functions.
8. *Supports inference:* Can the language define additional properties or predicates through inference, or can a predicate only be defined within a single definition? Inference may allow the decomposition of a complex constraint into many smaller definitions, supporting constraint reuse.
9. *Recursive rules:* Can the language define recursive properties or predicates – that is, definitions that can recursively call themselves? A language that cannot support recursion may restrict the range of its expressible constraints. In some scenarios, recursion may be implemented using transitive closure instead [73].

10. *Model checking*: Can the language evaluate a constraint by systematically checking all of the possible states of the system, as described earlier in Section 3.3.2? Model checking allows for a much more detailed analysis of the behaviour of the system, but requires significantly more computing power in order to evaluate a constraint.

### 6.6.1 Drools

As discussed in the previous section, Drools was evaluated as a potential implementation of the model completion concept. This same approach can also be used to implement model instance verification, by using inference to infer constraint violations. Likewise with the implementation of Drools used in model completion, the constraints can directly evaluate the POJO objects of the model instance under evaluation. Drools does not natively support any form of transitive closure, so this must be implemented manually with the appropriate guards to prevent an infinite loop.

The implementation of *infinitely redirects* in Drools is provided in Listing 10. The first rule defines the inference of a transient property *NavigatesTo* with respect to two *Frames*; and the second rule defines that the property is transitive. The final rule defines how constraint verification occurs, and also provides information about the source of the violation. Drools does not support the creation of transient facts, so a separate verification metamodel needs to be created and defined; however, this metamodel can be entirely separate of the metamodel under evaluation.

The Drools rule engine does not natively support the derivation trace for a given model element, however the rule engine could be extended to support this scenario. As the implementation of Drools is provided as an Eclipse project, it would be relatively straightforward to implement Drools-based model verification within an Eclipse-based development environment.

### 6.6.2 Jena

Similarly to Drools, Jena was also evaluated for model completion in Section 6.5.1, and similarly can also be used to implement model instance verification. Jena's inference engine explicitly supports the definition of special model verification rules, which can be selectively enabled or disabled [284]. Likewise with the implementation of model completion, it is necessary to translate the metamodel into an OWL representation [351] using the previously-introduced *emftriple* project [154], and translate the model instance into an RDF implementation.

The implementation of *infinitely redirects* is provided in Listing 11. Similarly to the implementation of the constraint in Drools, the rule (*redirects1*) defines the inference of a transient property *redirectsTo*. The second rule (*redirects2*) defines that the property is transitive, as Jena does not directly support a higher-order transitivity operator on relations. The final rule enables the validation engine, and defines the constraint itself. Once a constraint violation is detected by Jena, it is possible to obtain a derivation trace to the source of the violation; however, since the evaluated models are RDF translations of their original representations, it would require some development effort to reunite the derivation trace with these original models.

### 6.6.3 OWL 2 Full

A verification constraint can also be represented using OWL 2 [351], which has native support for concepts such as transitive closure and inference, by using class equivalence to select constraint violations. To be more specific, the metamodel designer would define a new class, defined as *equivalent* to

```
rule "Frame navigating to another Frame on access"
when
    access : Event ( )
    p : Frame ( onAccess == access )
    p2 : Frame ( eval(p != p2) )
    navigate : ECARule ( from == access, to == p2 )
    not ( NavigatesTo ( from == p, to == p2 ))
then
    NavigatesTo navigatesTo = factory.createNavigatesTo();
    navigatesTo.setFrom(p);
    navigatesTo.setTo(p2);
    insert(navigatesTo);
end

rule "NavigatesTo is transitive"
when
    p1 : Frame ( )
    p2 : Frame ( eval(p1 != p2) )
    n : NavigatesTo( from == p1, to == p2 )
    p3 : Frame ( eval(p2 != p3) )
    n2 : NavigatesTo ( from == p2, to == p3 )
    not ( NavigatesTo ( from == p1, to == p3 ))
then
    NavigatesTo navigatesTo = factory.createNavigatesTo();
    navigatesTo.setFrom(p1);
    navigatesTo.setTo(p3);
    insert(navigatesTo);
end

rule "An infinitely redirecting loop"
when
    p : Frame ( )
    n : NavigatesTo ( from == p, to == p )
    not (Violation ( source == p ))
then
    Violation violation = factory.createViolation();
    violation.setSource(p);
    violation.setReason("An infinitely redirecting loop");
    verify.failed(violation);
    insert(violation);
end
```

Listing 10: Implementation of *infinitely redirects* in Drools



```

[redirects1:
  (?P rdf:type iaml.visual:Frame)
  (?E rdf:type iaml.Event)
  (?P iaml:onAccess ?E)
  (?P2 rdf:type iaml.visual:Frame)
  (?W rdf:type iaml:ECARule)
  (?W iaml:from ?E)
  (?W iaml:to ?P2)
  ->
  (?P eg:redirectsTo ?P2)
]

[redirects2:
  (?P rdf:type iaml.visual:Frame)
  (?P2 rdf:type iaml.visual:Frame)
  (?P3 rdf:type iaml.visual:Frame)
  (?P eg:redirectsTo ?P2)
  (?P2 eg:redirectsTo ?P3)
  ->
  (?P eg:redirectsTo ?P3)
]

[ruleInfiniteRedirect: (?v rb:validation on()) ->
  [ (?X rb:violation error('violation', 'infinite redirect', ?X)) <-
    (?X rdf:type iaml.visual:Frame)
    (?X eg:redirectsTo ?X)
  ]
]

```

Listing 11: Implementation of *infinitely redirects* in Jena

all violating instances of a certain constraint. These constraints could then be combined with the generated metamodel ontology, and the exported RDF graph of the model instance, and evaluated using an ontology reasoner to identify the constraint violations.

One possible implementation of the *infinitely redirects* constraint is provided in Listing 12. This constraint is defined through equivalence of a transitive property *instantRedirectTo*. However, since selecting this property requires a self restriction on a transitive property with a cardinality constraint, this falls outside the bounds of OWL 2 DL, and thus this ontology is an OWL 2 Full ontology.

Unfortunately, at the time of writing there were no implementations of an OWL 2 Full ontology reasoner; furthermore, OWL 2 Full ontologies has been shown to be generally undecidable [240], and it is possible that no implementations will ever exist in the future. Consequently it is not possible to use this technology as a verification engine for the proof-of-concept implementation of IAML.

#### 6.6.4 CrocoPat

CrocoPat is a relational reasoner written in C++ [32], developed and released under the LGPL. It uses Binary Decision Diagrams to efficiently infer information, and natively supports higher-order logic through existential **EX**, universal quantification **FA** and transitive closure **TC** operators. Properties can also be defined recursively, and the language itself has well-defined axiomatic semantics. CrocoPat was initially designed to evaluate software models for design patterns, and was found to be much

```

// namespaces omitted
Ontology(<http://openiaml.org/verification/2009/infiniteRedirect.owl>
Import(<http://openiaml.org/model0.6>)

// some classes omitted

EquivalentClasses(InfiniteRedirectFrame ObjectIntersectionOf(
  ObjectExistsSelf(instantRedirectTo visual:Frame))
SubClassOf(InfiniteRedirectFrame iaml:Scope)
SubClassOf(InfiniteRedirectFrame iaml:VisibleThing)
SubClassOf(InfiniteRedirectFrame visual:Frame)

EquivalentClasses(AccessEvent ObjectIntersectionOf(DataHasValue(
  iaml:name "access") iaml:Event))

// some properties omitted

TransitiveObjectProperty(instantRedirectTo)
ObjectPropertyDomain(instantRedirectTo visual:Frame)
ObjectPropertyRange(instantRedirectTo visual:Frame)

ObjectPropertyDomain(navigatesToFrame iaml:ECARule)
ObjectPropertyRange(navigatesToFrame visual:Frame)

SubObjectPropertyOf(SubObjectPropertyChain(iaml:listeners isECARule
  InverseObjectProperty(isECARule)) onAccessNavigate)

SubObjectPropertyOf(SubObjectPropertyChain(iaml:target isFrame
  InverseObjectProperty(isFrame)) navigatesToFrame)

SubObjectPropertyOf(SubObjectPropertyChain(iaml:listeners toFrame) eventToFrame)

SubObjectPropertyOf(SubObjectPropertyChain(iaml:target isFrame
  InverseObjectProperty(isFrame)) toFrame)

// defines the verification constraint
SubObjectPropertyOf(SubObjectPropertyChain(iaml:events isAccessEvent
  InverseObjectProperty(isAccessEvent) onAccessNavigate navigatesToFrame)
  instantRedirectTo)
)

```

Listing 12: Implementation of *infinitely redirects* in OWL 2 Full

```

RedirectsTo(a, b) :=
  Frame(a) & a != "null" &
  EX(e, Event(e) &
  onAccess(a, e) &
  Frame(b) & b != "null" &
  EX(w, ECARule(w) & trigger(w, e) & target(w, b)));

RedirectsTo(a, b) :=
  Frame(a) & Frame(b) & TC(RedirectsTo(a, b));

InfiniteRedirect(p) :=
  Frame(p) & p != "null" &
  RedirectsTo(p, p);

PRINT ["Infinite redirection"] InfiniteRedirect(p);

```

Listing 13: Implementation of *infinitely redirects* in CrocoPat

faster than alternative approaches, including Prolog and a commercial relational database [32].

Since CrocoPat is written in C++, the reasoner does not yet directly support the interaction with Java objects or POJOs; as a result, a Java object must first be serialised into a textual format according to the RSF/RML languages. This representation can then be concatenated with a rule file and evaluated by CrocoPat. There is no native support for typing, so a type system has to be manually implemented if it is necessary.

The implementation of *infinitely redirects* in CrocoPat is provided in Listing 13. The first definition defines the inference of a transient property of *RedirectsTo* with respect to two *Frames*; and the second rule defines that the property is transitive. The third rule defines the constraint violation through inference of this property, and finally the reasoner is instructed to find all instances of that violation.

CrocoPat does not natively support the derivation trace for a given property, but it is possible to add this through an extension of the reasoner. As the implementation of CrocoPat is provided as a C++ project, some development effort is necessary in order to implement CrocoPat-based verification within an Eclipse-based environment.

### 6.6.5 Alloy

Alloy [169] is a specification language for assisting in the design and specification of software and software models, implemented using Java and released under the MIT license. Its intended use is in the development of robust modelling language that conform to a given suite of assertions, by trying to derive model instances that are invalid according to those assertions (known as *counter-examples*). Alloy can also be used to discover constraint violations for a single *given* model instance, although this is not its original intent.

Since Alloy is designed for the specification of languages, it does not directly support other sources of models, such as POJOs; models and metamodels must first be translated into the Alloy specification language. The language does not support multiple inheritance, however the Variant design pattern [46] can be used to emulate this design with single inheritance. It is difficult to define primitive data values such as strings and integers; these have to be translated manually into singleton world facts. Finally, Alloy does not support recursive functions, however in many situations these can be emulated using

```

abstract sig Frame {
  redirectsTo : set Frame
}

one sig Frame_page1 extends Frame { }{ redirectsTo = Frame_page2 }
one sig Frame_page2 extends Frame { }{ redirectsTo = Frame_page3 }
// ...

assert test {
  no p : Frame | p in p.^redirectsTo
}
check test for 3

```

Listing 14: Implementation of *infinitely redirects* in Alloy

the transitive closure operator `.*` [73].

There are a number of ways in which the *infinitely redirects* constraint can be implemented in Alloy; a simple and efficient implementation is provided in Listing 14. In this approach, most of the components of the underlying IAML metamodel are ignored, and only the essential structure is defined. Some properties, such as *redirectsTo*, are pre-calculated by the translation step; this greatly simplifies the computational requirements of the Alloy verification step.

One powerful feature of Alloy is that it natively supports the interactive visualisation of a constraint violation. The integration of this visualisation into a model verification approach would be ideal to illustrate the reasoning behind a specific counter-example. Because Alloy is designed to search for possible counter-examples for a given range of constraints, Alloy can be considered a model checking language.

### 6.6.6 openArchitectureWare

openArchitectureWare (OAW) was previously discussed in Section 6.4.3 as an implementation of a code generation engine. The framework also supports the definition of constraints through the combination of the *Xtend* language for transient model extensions, and the *Checks* language for constraint definitions, which use an expression language similar to Java and OCL [87].

Likewise with the implementation of OAW used for code generation, the constraints can directly evaluate the POJO objects of the model instance under evaluation. OAW does not natively support any form of transitive closure, and has difficulty supporting recursive constraints; inferred extensions should be explicitly labelled as *cached*, and some recursive definitions will result in an infinite loop.

Nevertheless, the implementation of *infinitely redirects* is provided in Listings 15 and 16. The first model extension defines a property *redirectsTo* which lists the *Frames* that a given *Frame* may directly redirect to; the second model extension then defines a property *doesRedirectTo*, which is true if a given *Frame* will redirect to another *Frame*. The third model extension *infiniteRedirect* defines a property for which infinite redirection can be detected. Finally, the constraint definition in Checks is used to define the constraint error message.

The default implementation of OAW does not support the derivation trace of inferred properties. However, if the OAW framework is already integrated into a project in terms of a code generation framework, then it is trivial to reuse this framework to evaluate verification constraints as part of the code generation process.

```
// initial def
cached List[model::visual::Frame] redirectsTo(model::visual::Frame this) :
  onAccess.listeners.target.typeSelect(model::visual::Frame);

// we need to keep track of which nodes we have visited
cached Boolean doesRedirectTo(model::visual::Frame a,
  model::visual::Frame b, Set[model::visual::Frame] visited) :
  visited.contains(b) || a.redirectsTo().exists( c | c == b ||
    doesRedirectTo(c, b, (visited.add(a)).toSet()));

// definition
infiniteRedirect(model::visual::Frame this) :
  doesRedirectTo(this, this, {}.toSet());
```

Listing 15: Implementation of *infinitely redirects* in openArchitectureWare: Xtend

```
extension validation::InfiniteRedirect;

context model::visual::Frame ERROR "Infinite redirection":
  !infiniteRedirect();
```

Listing 16: Implementation of *infinitely redirects* in openArchitectureWare: Checks

### 6.6.7 OCL

The Object Constraint Language (OCL) [259] is a query and expression language for UML [185, pg. 33], and includes support for the definition of operations, constraints, and derived attributes and associations. Verification properties must be defined through a separate verification metamodel, or derivation-based extensions to the original metamodel must be provided, in order to support complex forms of model instance verification. The language itself is functions-based, and supports the higher-order logic of function aggregation through the *iterate* expression [252, pg. 211].

The OCL standard is platform-independent, and there are many different implementations of the language. Only two implementations will be considered in this section: the EMF Validation Framework, and the Dresden OCL toolkit. Since the OCL standard does not support property inference, the *infinitely redirects* constraint needs to be implemented using operations as listed in Listing 17. These constraints are derived from the implementation of the *acyclical class inheritance* constraint of UML class diagrams, as discussed earlier in Section 3.1.9.

#### EMF Validation Framework with OCL

The EMF Validation Framework [84] provides a number of methods to define constraints for a given EMF model. One approach is to define the constraints as part of the metamodel itself [314, pg. 549–565]; this is ideal for simple constraints such as properties, as the constraints are included in the generated metamodel code itself. However, this approach does not automatically translate the constraints into executable code; for example, an OCL constraint will not automatically call the Eclipse OCL implementation. Consequently, this approach will not be considered in this section.

Alternatively, the constraints can be provided by a constraint *adapter*, allowing the model developer or other tools to selectively execute the constraints. This is ideal for more advanced constraints

```

context Frame::allRedirectsTo() : Set(Frame) body:
  allRedirectsTo = self.redirectsTo()->union(
    self.redirectsTo()->collect(p | p.allRedirectsTo())

context Frame::redirectsTo() : Set(Frame) body:
  redirectsTo = if self.onAccess->oclIsUndefined() then OrderedSet{}
  else self.onAccess.listeners.target->collect(p | p.oclIsKindOf(Frame))
  endif

context Frame inv:
  not self.allRedirectsTo()->includes(self)

```

Listing 17: Implementation of *infinitely redirects* in OCL

that may be more computationally intensive. This adapter can explicitly define that a given constraint is written in OCL, and such a constraint will then be executed by the Eclipse OCL implementation.

However, constraints defined in this way are not fully functional OCL specifications; they can only be used to check invariant-based constraints that are not recursive, and cannot be used to specify operation bodies. The recursive OCL constraints introduced earlier in Listing 17 therefore cannot be implemented using this *adapter* approach. Since the EMF Validation framework is a member of the Eclipse ecosystem, it is well-supported with rich documentation.

## Dresden OCL

The Dresden OCL2 toolkit [72] implements the OCL 2.2 language [259] and may be executed within a standalone environment, or integrated into another modelling environment. OCL specifications are defined separately from the definition of the metamodel, allowing for constraint verification to occur independently of the model instance.

The latest release of the toolkit – formerly named *Dresden OCL for Eclipse* – includes integration within the Eclipse environment and allows OCL constraints to execute on a variety of models, including EMF-based model instances and POJOs. The Dresden OCL toolkit also forms the basis of ArgoUML’s support for verifying specified OCL constraints [322].

### 6.6.8 NuSMV

NuSMV is a symbolic model checker written in C [56] which supports the evaluation of specifications written in the CTL and LTL specification languages [160]. As opposed to both CrocoPat and Alloy, NuSMV specifications are defined as a series of states; NuSMV then exhaustively navigates through these states to try and find counter-examples. Once a counter-example has been identified, a trace of state changes to the invalid state is provided as a derivation trace. To verify the asynchronous nature of RIAs, threading is supported in NuSMV through *modules* of states, which can be triggered in any order according to a fairness algorithm.

Since a NuSMV specification is based around the definition of states and their transitions, a considerable amount of effort must first be expended to translate a given model instance into a system specification. This often involves the definition of system functionality; for example, a web application may define the set of available web pages, buttons that may be clicked, current browser location, state-based functionality of operations and so on. In the domain of web applications, the functionality

```

LTLSPEC
G ((!(navigation_running = 1 -> !(F navigation_finished = 1)))
  U navigation_running = 0)

```

Listing 18: Implementation of *infinitely redirects* in LTL

of the web browser itself would have to be emulated. As a result, these specifications are unlike any of the other approaches mentioned in this chapter, as NuSMV does not interact with types or model instances, and cannot infer any information.

The implementation of the specification for the *infinitely redirects* constraint in LTL, which NuSMV may then evaluate, is provided in Listing 18. This specification states that in all situations (**G**), we must never (**!**) have a navigation that never finishes (**!F**), as long as we can only navigate one page at a time (**U**). This example does not include the considerable amount of definitions necessary to model the system; these definitions are included in Appendix D.

This specification focuses on the abstract concept of “page navigation” to identify an infinitely redirecting page. This means that any form of page navigation in an IAML model instance must be implemented according to these concepts, but also provides the model developer with a much stronger system. Other concepts such as the behaviour of predicates and operations can be defined entirely within this state-based system, allowing for the specification of more advanced constraints. Such constraints could include that all operations executed within a system must terminate, or that no variable may overflow its specified range (such as integer overflow).

The NuSMV project is released under the open source LGPL license; however development does not occur in public, so it is not possible to obtain accurate measurements for many of the OpenBRR quality measurements. In particular, there is no public bug tracker, so it is not known if there are any critical or security issues in any given release. The OpenBRR evaluation therefore assumes the worst-case values for these metrics.

### 6.6.9 Summary

A summary of this investigation is provided in Tables 6.5 and 6.6. This evaluation identified that none of these technologies satisfy all of the desirable properties of a universal verification engine, and each technology has at least one feature that it does not currently support, with respect to properties important for model instance verification engines.

Model instance verification will be implemented within the three categories of constraint expressiveness discussed earlier in Section 3.3.3, and each category implemented using a technology discussed here. The desirable properties for each of these categories must be discussed to inform this decision. A summary of this evaluation against each criteria is provided in Table 6.7.

1. *Functions-Based Language*: The most important criteria for a function-based verification language are ease of use and speed, since these will be evaluated frequently. Therefore, this technology needs to provide an implementation, and should also support POJOs; support native subtypes; and support the inference of transient properties. Since OAW Checks satisfies all of these criteria and is executed within the Eclipse runtime environment, this technology is the ideal candidate for the verification of model instance properties.



Requirement	Jena	OWL	Drools	CrocoPat	OAW
Language	Java	n/a	Java	C++	Java
Runtime environment	Standalone	n/a	Standalone	Standalone	Eclipse
Open source	✓ (BSD)	n/a	✓ (Apache)	✓ (LGPL)	✓ (EPL)
Version	2.6.4	2.0	4.0.7	2.1.4	4.3.1
Uses POJO model	✗	✗	✓	✗	✓
Supports subtypes	✓	✓	✓	✗	✓
Derivation cause	✓	✓	Manual	Manual	✗
Implementation	✓	✗	✓	✓	✓
Transient properties	✓	✓	✗	✓	✓
$L_{ver}$ expressiveness	Relations	Relations	Functions	Relations	Functions
Higher-order logic	✗	✓	✗	✓	✗
Supports inference	✓	✓	✓	✓	✓
Recursive rules	✓	✓	✓	✓	Manual
Model checking	✗	✗	✗	✗	✗
OpenBRR rating	99	n/a	101	77	94
Average Quality	3.8		2.3	3.0	3.7
Average Support	2.0		4.5	1.0	3.0
Average Documentation	2.5		5.0	3.0	2.5
Average Adoption	3.0		4.0	2.0	4.0
Average Architecture	4.0		5.0	1.0	4.0

Table 6.5: Requirement comparisons between existing model verification environments (1)

2. *Relation-Based Language with Higher-order Logic*: Relation-based languages may incur a higher performance penalty, but should support recursive rules and transitive closure. It is also important that a derivation cause can be provided to the user, as a recurse constraint violation may be more difficult for a model developer to understand. According to these evaluations, no technology provides all of these features; the CrocoPat engine is however selected as it is preferable to have recursive rules over model checking support.
3. *Model Checking Language*: Finally, an ideal technology for model checking must not only support model checking, but provide a derivation trace for constraint violations. The key difference between Alloy and NuSMV is that Alloy tries to find invalid structures, whereas NuSMV tries to find invalid *behaviour* through states. This difference allows NuSMV to be much more expressive for describing complex constraints and more thorough when finding violations, and is therefore selected for the proof-of-concept implementation.

## 6.7 Conclusion

In this chapter, a range of different technologies have been discussed, each which may be used in the development of a model-driven environment for IAML model instances. For each type of technology, one has been selected as the basis of the first proof-of-concept implementation of IAML, and the details of implementing this modelling environment will be discussed in the next chapter.

Requirement	Alloy	EMFV OCL	Dresden OCL	NuSMV
Language	Java	Java	Java	C
Runtime environment	Standalone	Eclipse	Standalone	Standalone
Open source	✓ (MIT)	✓ (EPL)	✓ (LGPL)	✓ (LGPL)
Version	4.1.10	1.3.1	3.1.0	2.4.3
Uses POJO model	✗	✗	✓	✗
Supports subtypes	✗	✓	✓	✗
Derivation cause	✓	✗	✗	✓
Implementation	✓	✓	✓	✓
Transient properties	✗	✗	✗	✗
$L_{ver}$ expressiveness	Relations	Functions	Functions	n/a
Higher-order logic	✓	✓	✓	✗
Supports inference	✓	✗	Partial	✗
Recursive rules	✗	✓	✓	✗
Model checking	✓	✗	✗	✓
OpenBRR rating	91	90	72	63
Average Quality	3.7	3.3	3.0	1.7
Average Support	1.0	3.0	1.0	1.0
Average Documentation	2.5	5.0	2.5	2.5
Average Adoption	3.5	2.0	2.0	3.0
Average Architecture	4.0	3.0	3.0	2.5

Table 6.6: Requirement comparisons between existing model verification environments (2)

Technology	Function Language	Relation Language + HOL	Model Checking
EMFV OCL	#	✗	✗
Dresden OCL	✓	✗	✗
Drools	✓	#	✗
OAW	✓✓	#	✗
Jena	✓	✓	✗
CrocoPat	✓	✓✓	✗
Alloy	#	✓	✓
NuSMV	#	✗	✓✓

Marker	Description
✓✓	Best approach
✓	Satisfies most requirements
#	Requires workarounds
✗	Not supported

Table 6.7: A summary of the suitability of verification languages for addressing each verification approach category



## Chapter 7

# Proof-of-Concept Implementation

In the previous chapter, a range of model-driven technologies were reviewed and selected to form the basis of the proof-of-concept implementation of IAML. This chapter will fully discuss the effort and details required in actually translating the design of the modelling language into a functional graphical editor. This proof-of-concept implementation is supplied on the media attached to this thesis, as described in Appendix E.

The development history of this implementation is available online at the IAML project page of <http://code.google.com/p/iaml/>, and the public Subversion [58] source code repository at <http://iaml.googlecode.com/svn/trunk/>. Outstanding development tasks<sup>1</sup> on this implementation are detailed here as *issues*, and are discussed in further detail on the online issue tracker at <http://code.google.com/p/iaml/issues/list>.

### 7.1 Introduction

The implementation of this proof-of-concept graphical editor involves the development and integration of a large number of components, such as model completion; code generation; instance verification; and the underlying model instances and IAML metamodel itself. An overall summary of these components and their integration is provided here as a UML component diagram in Figure 7.1, with each technology viewed as a black-box component connected through external interfaces; the rest of this chapter will decompose each of these components into a white-box view [254, pg. 152].

The overall runtime environment of the proof-of-concept implementation of IAML is within the Eclipse framework [117], which provides a rich plugin environment based on OSGi bundles [318]. Each of these implementation components may therefore be provided as an OSGi bundle, where each connecting interface is specified through exporting and importing packages.

For each component in the proof-of-concept implementation of IAML, the corresponding *OSGi bundle ID* for its implementation will be provided using this visual syntax.

#### 7.1.1 Implementation License

As discussed earlier in Section 5.1.5, one of the design goals of the IAML language was to provide the language and proof-of-concept implementation under an open source license. It is undesirable

---

<sup>1</sup>For example, the outstanding Issue 281: *Refactor model elements into documented packages*.

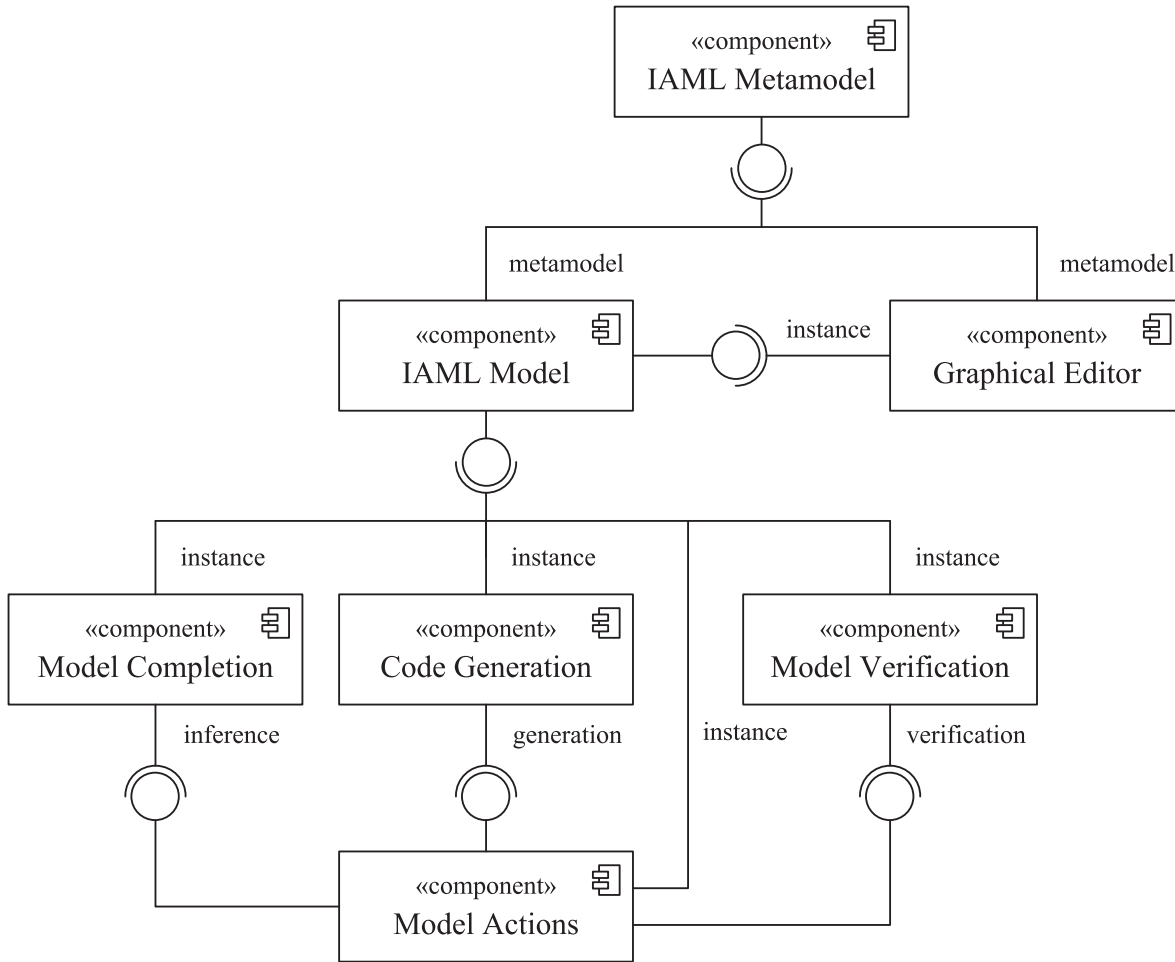


Figure 7.1: Overall UML component diagram of the Proof-of-concept Implementation of IAML

to define a new open source license, as a wide range of licenses have already been defined, and the proliferation of different licenses is a major problem [109].

As mentioned earlier in Section 2.7.4, a full discussion on the integration of different open source licenses is well outside the scope of this thesis, and the interested reader is referred to Michaelson [231]. Other than simply making the source code available, other intentions of licensing this implementation under an open source license were to ensure the project could be integrated into other platforms; that the license is an open source license according to the Open Source Definition [262]; that the license supports the development of *free software* [231]; and that third-party modifications should also be released under the same license to encourage community development.

In this implementation, the decision was made to use the *Eclipse Public License* (EPL) [82], as it satisfies all of these license requirements. It is also trivial for EPL-licensed components to legally work together, as many of the dependencies of the IAML implementation are already licensed under the EPL. Finally, all EPL-licensed code can be reintegrated into the Eclipse project [83], meaning that any extensions or fixes developed as part of this implementation can directly be contributed back into the relevant EPL-licensed component.

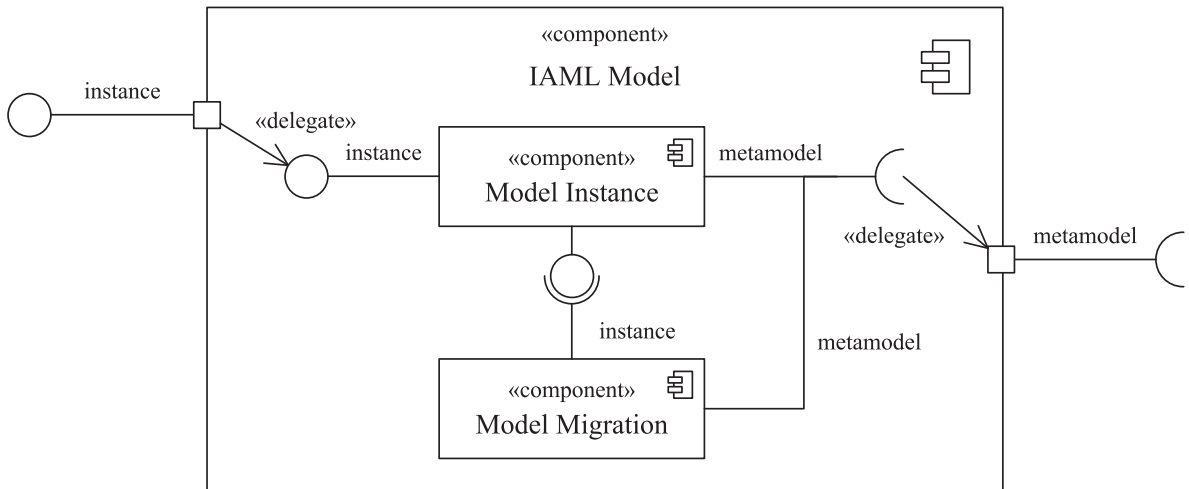


Figure 7.2: UML component diagram of the *IAML Model* Component Decomposition

## 7.2 IAML Metamodel

The *IAML Metamodel* component is implemented as a single component, and is implemented in EMF 2.5 [314]. This component internally depends on two external components to provide the metamodels for XSD and Ecore. In particular, both the XSD and Ecore metamodels are provided through existing implementations by the Eclipse Foundation [314, 85].

The corresponding OSGi bundle for the implementation of the *IAML Metamodel (EMF)* component is `org.openiaml.model`. This bundle internally depends on the `org.eclipse.xsd` and `org.eclipse.emf.ecore` bundles, implemented and supplied by the Eclipse Foundation.

## 7.3 IAML Model

The decomposition of the *IAML Model* component is provided in Figure 7.2. This component is made up of two sub-components: a *Model Instance* component; and a *Model Migration* component. The former component provides the functionality to store, interact and serialise a given IAML model instance, with respect to the IAML metamodel; and the latter component provides the functionality to migrate model instances between different metamodel versions.

### 7.3.1 Model Instance

The *Model Instance* component is the most critical component of the entire implementation, as all of the other components in the system interact with IAML model instances. The implementation of this component, however, is simply the generated source code from EMF from the given Ecore-based metamodel. As described by Steinberg et al. [314], this involves automatically creating a `genmodel` model for a given metamodel source – which, in this case, is an Ecore file – and then using this `genmodel` model instance to generate the relevant source codes. IAML model instances are therefore stored as XMI representations within `iaml` files, as illustrated throughout this thesis.

Only one extension needs to be used in the generation of the model instance source code. To simplify the reference of model element instances in IAML model instances, all elements are given an

ID; that is, an `EAttribute` with the ID property set to `true` [314, pg. 110]. However, these IDs are not generated automatically; the relevant *Factory* for the creation of the element must create its own ID. To implement this, a *dynamic template* is provided to the JET engine<sup>2</sup>.

The corresponding OSGi bundle for the implementation of the *Model Instance* component is `org.openiaml.model`. This bundle ID is the same as the *IAML Metamodel (EMF)* component, which is a common scenario of generated EMF models [314, pg. 71].

### 7.3.2 Model Migration

As IAML was developed using an iterative evolutionary process as discussed in Section 5.1.3, existing model instances used for testing the implementation had to be kept valid throughout the development lifecycle with respect to changes to the underlying IAML metamodel. In this thesis, this process is defined as *model migration*, which is “the execution of [an algorithm] on existing domain models to transform them into domain models that are correct in the evolved domain” [313, pg. 6]; this is equivalent to a model transformation where the source and target metamodels are different versions of the same metamodelled domain.

The evolution of metamodels, and the subsequent migration of model instances to updated metamodels, is a significant area of academic research [287] and an important aspect of developing a domain-specific language [102, pg. 65]. As IAML was designed and implemented in an incremental fashion, the model migration strategy used in the proof-of-concept implementation is an *incremental migration strategy*, as discussed by Fowler [102, pg. 65]. That is, a suite of migrators are developed over the course of the metamodel’s evolution, allowing for changes to build upon existing migrators.

The actual implementation of these migrators followed a naïve approach without references to a model-driven environment. In particular, the migrators are written in Java, and operate on the XMI representations of a model instance within `iaml` files. This approach was used as it was the simplest to implement; there was no need to create a repository of old IAML metamodels, which would have been of significant size<sup>3</sup>; and model migration is not a major focus of this research. Future model migrators should instead follow one of the other migration approaches in order to obtain their benefits.

The corresponding OSGi bundle for the implementation of the *Model Migration* component is `org.openiaml.model.actions`.

## 7.4 Graphical Editor

The decomposition of the *Graphical Editor* component is provided in Figure 7.3. This component is decomposed into five separate sub-components: *Model Edit*, a component which provides a basic interface to edit model instances; *Diagram Editors*, the set of diagram editors that actually provide the graphical interface; *Diagram Definitions*, the model instances that define the diagram editors, and are used to generate the editors themselves; *Diagram Extensions*, which provides graphical extensions

<sup>2</sup>The relevant dynamic template is `org.openiaml.model/templates-emf/model/FactoryClass.javajet`.

<sup>3</sup>At the time of writing, the IAML metamodel definition file `iaml.ecore` had been modified 240 times within the version control repository.



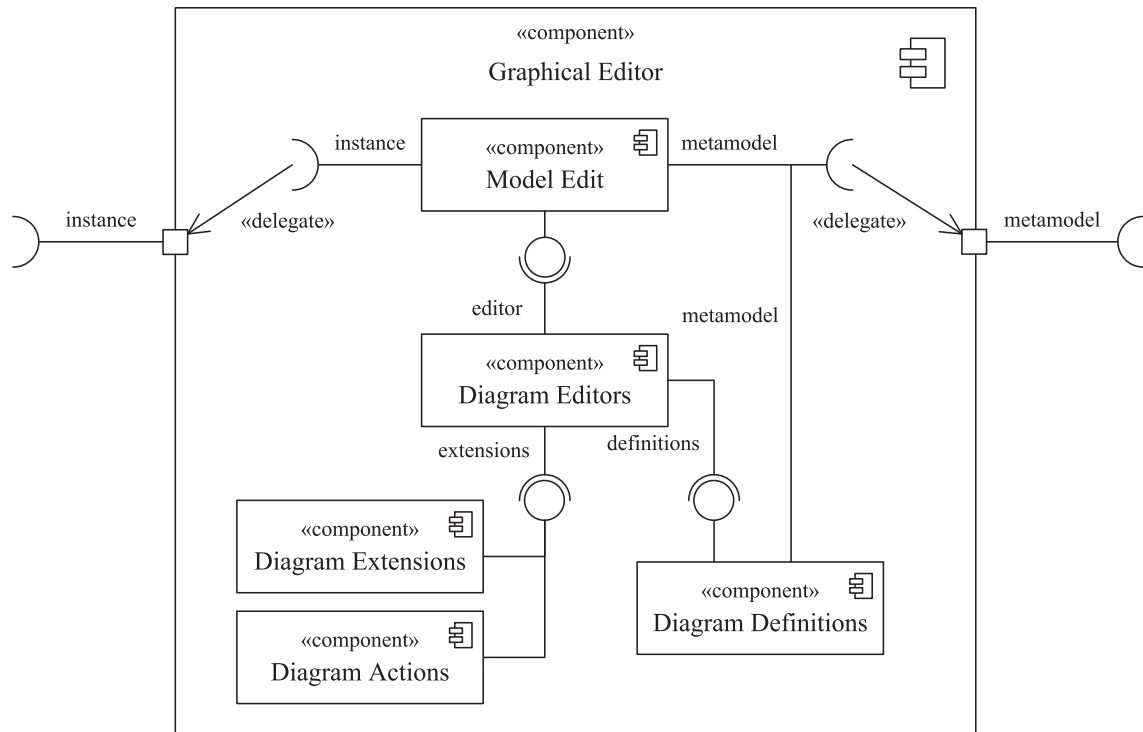


Figure 7.3: UML component diagram of the *Graphical Editor* Component Decomposition

to the diagram editors, without having to modify the editors directly; and *Diagram Actions*, which provide additional actions to the diagram editors themselves.

#### 7.4.1 *Model Edit*

Part of the implementation of EMF is *EMF.Edit Support*, which provides a basic user interface to editing model instances, by combining the generated model plugin with the Eclipse UI Framework *JFace* [314, pg. 45–46]. Similarly to the generated model plugin as discussed in Section 7.3.1, an Edit plugin may be generated automatically from the `genmodel` model instance.

This generated plugin includes a tree-based viewer of a model instance derived automatically from the metamodel structure, and a properties-based element viewer allowing a model instance developer to modify the attributes and references of selected model elements. The generated implementation of this component is fairly crude, but is a necessary requirement for the diagram editors, as discussed in the next section.

The corresponding OSGi bundle for the implementation of the *Model Edit* component is `org.openiaml.model.edit`. This bundle ID follows the standard naming pattern for generated EMF.Edit plugins [314, pg. 649].

#### 7.4.2 *Diagram Editors*

The *Diagram Editors* component represents the actual graphical editors for IAML model instances. As discussed earlier in Section 6.3, these editors are defined according to the Graphical Modeling Framework (GMF). In particular, the graphical editor was implemented using GMF 2.2.

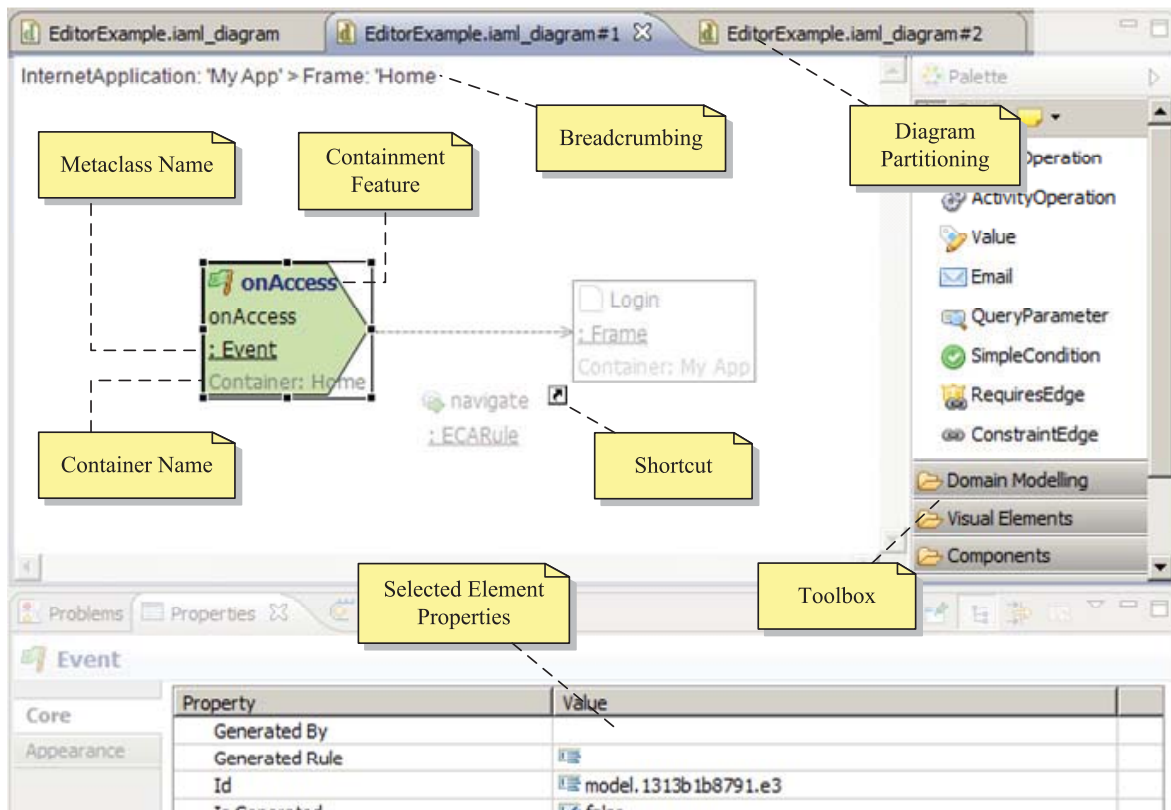


Figure 7.4: Implementation of a graphical editor for IAML model instances using the Graphical Modeling Framework

The source code for the diagram editors are generated according to the process described earlier in Section 6.3.2; in particular, the editors are defined according to GMF metamodels, and are provided by the *Diagram Definitions* component. An annotated screenshot of the resulting diagram editor is provided in Figure 7.4, which also illustrates some of the diagram extensions provided by the *Diagram Extensions* component.

The visual representation of a model instance is stored independently from the underlying model instance within a `iaml_diagram` file. GMF editors also provide the action “**initialise diagram file**” to initialise a new visual representation of a model instance, which uses the automatic layout logic provided by GMF. Along with the IAML implementation of the *Diagram Editors* component, this allows any `iaml` model instance to be transformed into a visual representation.

## Hierarchical Modelling

As discussed earlier in Section 5.2.3, model instances are intended to be edited following a hierarchical modelling approach, in order to reduce the complexity of model instance development and to support a number of different visual metaphors. The implementation of hierarchical modelling using GMF was straightforward, but required some diagram extensions to improve usability.

The basic implementation of hierarchical modelling was provided by GMF’s built-in support for *diagram partitioning*; this approach allows for different model elements to appear as the current “root”, and model instance developers can navigate through the model instance by opening and closing elements and tabs, as illustrated in Figure 7.4. For example, by double clicking on the representation of a particular `Frame` instance, a new `Frame` diagram editor window will open, visually representing the

Top-Level Container	OSGi Bundle ID
Internet Application	org.openiaml.model.diagram
Access Control Handler	org.openiaml.model.diagram.access_handler
Activity Predicate	org.openiaml.model.diagram.condition
Email	org.openiaml.model.diagram.email
Frame	org.openiaml.model.diagram.frame
Domain Instance	org.openiaml.model.diagram.instance
Domain Iterator	org.openiaml.model.diagram.iterator
Login Handler	org.openiaml.model.diagram.login_handler
Activity Operation	org.openiaml.model.diagram.operation
Domain Type	org.openiaml.model.diagram.schema
Session	org.openiaml.model.diagram.session
Visible Thing	org.openiaml.model.diagram.visual

Table 7.1: Generated IAML diagram editors and their associated OSGi bundle IDs

directly contained contents of that [Frame](#) element instance; a number of diagram extensions such as breadcrumbing and shortcuts extend this functionality, as discussed later in Section 7.4.4.

Only model elements that are intended to contain other elements should have an associated diagram editor; this significantly reduces the number of different diagram editors that are necessary. A list of all of the diagram editor instances, along with their associated OSGi bundle IDs and the type of their root model element – known in GMF as the *top-level container*<sup>4</sup> – are listed in Table 7.1.

### 7.4.3 Diagram Definitions

As discussed earlier, the *Diagram Definitions* component defines the abstract representations of the intended diagram editors according to the GMF metamodels, and these diagram definitions may then be translated into instances of diagram editors as through the process discussed earlier in Section 6.3.2. In particular, four metamodels (*gmftool*, *gmfgraph*, *gmfmap* and *gmfgen*) are defined by GMF, and the GMF framework provides code generation templates implemented in Xpand to translate instances of *gmfgen* models into Java source code.

Only one instance of the *gmfgraph* metamodel is necessary, as there should only be one visual representation of each IAML metamodel element. These graphical definitions are shared across all of the editors as described in Table 7.1. For each of these editors, an associated *gmftool*, *gmfmap* must manually be defined, as each diagram editor has a different set of elements that can be displayed (*gmfmap*), and a different set of elements that can be created (*gmftool*).

By combining these three model instances together using the GMF framework, an instance of the *gmfgen* metamodel can be created for each of the diagram editors, and this instance can then be used to generate the source code of the actual diagram editor. For example, the [Session](#) diagram editor is generated by combining the graphical definitions in the `iaml.gmfgraph` file; the palette

<sup>4</sup>A diagram editor defined GMF with an abstract root element type as a top-level container will generate a warning message of “Top-level diagram container must be concrete” when the editor is generated. This warning is raised as it is not possible for a model developer to create an instance of an abstract element, and occurs when generating the [Visible Thing](#) diagram editor, but this warning may be ignored in hierarchical approaches for elements contained directly or indirectly by the concrete root element.

tooling definition in the `session.gmftool` file; and the shape and element mapping definitions in the `session.gmfmap` file, into the editor generation definitions into the `session.gmfgen` file.

The corresponding OSGi bundle for the implementation of the *Diagram Definitions* component is `org.openiaml.model`, as the diagram definitions are stored in the same location as the *IAML Metamodel* component to simplify development.

## SimpleGMF

It was recognised during the development of the *Diagram Definitions* component that many of the GMF model instances shared similar features. For example, the “open diagram behaviours” needed to be consistent across all *gmfgen* instances; mappings in *gmfmap* instances needed to be consistent with the IAML metamodel structure; and tool definitions in the *gmftool* instances needed to be consistent with these *gmfmap* mappings.

In response to these common patterns, a domain-specific modelling language named *SimpleGMF* was developed, which also uses model transformations in openArchitectureWare to simplify the development of these many GMF model instances. This language allows for many common GMF definitions to be defined in a single graphical definition file, and generates all of the model instances of the four metamodels (*gmftool*, *gmfgraph*, *gmfmap* and *gmfgen*) necessary. The full description of SimpleGMF is well outside the scope of this thesis; the interested reader is instead referred to the project homepage at <http://openiaml.org/simplegmf/>.

## Visual Notation Cognitive Effectiveness

As discussed earlier in Section 3.4.4, the design of the visual syntax for IAML model instances needs consider the usability of the resulting syntax, and to satisfy the goal of having a one-to-one mapping between elements and notations. With respect to the cognitive effectiveness guidelines proposed by Moody [236], the visual syntax has the following design:

1. **Construct deficit**, where a model concept does not have a corresponding visual notation, is permitted for attributes and references that are rarely used. This is acceptable as all attributes and references are still accessible to the developer through an additional dialog – the *Selected Element Properties* frame, as in Figure 7.4.
2. **Construct redundancy**, where multiple visual notations can represent a single model concept, is not accepted with two exceptions: displaying the metaclass name of elements, and displaying the name of element parents, as discussed in the next section.
3. **Construct overload**, where a visual notation can represent multiple model concepts, is not permitted.
4. **Construct excess**, where a visual notation does not represent any model concepts, is also not permitted.

## Shape Design

It is necessary to describe the design of the visual notations of IAML model elements, along with the rationale behind each of their designs. This process involves seven parameters: shape type; shape





Shape	Model elements represented
	The default shape for IAML model elements. This includes visual elements such as <a href="#">Visible Things</a> ; domain modelling elements such as <a href="#">Domain Types</a> and <a href="#">Domain Sources</a> ; and value instances such as <a href="#">Values</a> .
	This shape is used for <a href="#">Operations</a> .
	This shape is used for <a href="#">Functions</a> and <a href="#">Permissions</a> .
	This shape is used for <a href="#">Events</a> , and is based on the visual notation for UML <i>SendSignalAction</i> elements [254, pg. 284].

Table 7.2: Shape styles for the visual representation of IAML model elements

style; icons; colours; line weights; line patterns; and font styles. The design decisions for each of these parameters are partially inspired by the guidelines discussed earlier in Section 3.4.4 by Moody [236] and Rumbaugh [291].

1. **Shape type** refers to whether a model element is displayed as a node or an edge. All model elements will be represented as nodes, with the references between them as edges; however, certain IAML model elements (such as [Wires](#)) represent relationships themselves, and should therefore be displayed as edges to simplify the visual syntax.

GMF editors are designed to work best when there is a one-to-one mapping between graphical elements and element instances in the underlying metamodel. GMF may represent references between element instances using nodes or edges, however this requires the definition of additional *gmfmap* constraints. Additional types of derived nodes and edges may also be displayed, however requires the definition of both additional *gmfmap* constraints and OCL constraints, which had only recently been supported in GMF 2.3<sup>5</sup>. Due to the resource constraints of this research, it was therefore decided to only support the representation of element instances as nodes or edges, and not the representation of references or attributes.

All model elements within IAML are displayed as a node, with the exception of elements that represent relationships or flows. That is, edges are used as the syntax for all [ECA Rules](#), [Conditions](#), [Parameters](#), [Wires](#) and [Constraint Edges](#). Edges are also used for the operation modelling elements of [Data Flow Edges](#), [Execution Edges](#) and [External Value Edges](#); and for the domain modelling elements of [Provides Edges](#), [Requires Edges](#), [Schema Edges](#) and [Select Edges](#). References and attributes for a particular model element instance are not represented as nodes or edges, but as labels attached to the shape type of the element itself.

2. **Shape style** only applies to node shape types, and refers to the actual style of the node shape; for example, whether the node is displayed as a rectangle, an ellipse, or another shape. The list

<sup>5</sup>Eclipse bug 256461: *Use ParsingOptions.IMPLICIT\_ROOT\_CLASS for implicit access to the features of EObject in all OCL queries in GMF*

Colour	Model elements represented	Sample
White	Default shape background color.	
Red	Indicates components, such as <a href="#">Login Handlers</a> and <a href="#">Access Control Handlers</a> .	
Orange	Indicates <a href="#">Functions</a> and <a href="#">Conditions</a> .	
Yellow	Indicates scopes, such as <a href="#">Sessions</a> .	
Green	Indicates <a href="#">Events</a> , <a href="#">Gates</a> , or other sources of <a href="#">Actions</a> .	
Blue	Indicates operations and their contents, such as <a href="#">Operations</a> and <a href="#">Activity Nodes</a> .	
Brown	Used for domain modelling and user modelling, such as <a href="#">Domain Types</a> , <a href="#">Domain Iterators</a> and <a href="#">Permissions</a> .	
Gray	Indicates instances of data, such as <a href="#">Values</a> and <a href="#">Temporary Variables</a> .	

Table 7.3: Background colours for the visual representation of IAML model elements

of visual shapes currently used for various model elements is illustrated in Table 7.2. Where appropriate, visual designs are inspired by similar UML 2.0 visual stereotypes [254].

3. **Icons** are used as an abstract representation of the node or relationship itself. An icon may be visible in three locations, as illustrated in Figure 7.4: the tree-view model instance editor created by EMF [314]; the element creation palette; and on the element itself in the graphical editor.

Every model element in IAML is provided with a unique icon, partially satisfying the requirement that each model element must have a unique notation. Moody [236] surprisingly finds that icons are rare in software engineering visual notations, and that most rely on geometrical shapes; however, icons can easily be too subtle, and do not satisfy the guideline proposed by Rumbaugh that instances must be easy to draw by hand [291].

4. **Colours** refer to the colours used in the nodes and edges. For colourblind users or for printing in black and white, the choice of colours can impact on the accessibility of the environment; however, the colour of a visually represented model element does not impact on the behaviour or functionality of that element in any fashion, as elements are distinguished through their meta-class names. There are three categories of colours used:

- (a) **Edges:** One of the guidelines proposed by Rumbaugh [291] is that diagrams “must fax and copy well using monochrome images”. Subsequently, all lines and edges used in the IAML visual syntax are coloured black.
- (b) **Nodes:** These colours refer to the overall colours of the node backgrounds. As our implementation is within the Eclipse environment, it is appropriate to reuse the style and design guidelines provided for Eclipse plugins [199]. Seven colours – the maximum number of colours that a user can distinguish [236, pg. 770] – are selected from the Eclipse palette<sup>6</sup>, and applied to the background colour of model elements as illustrated in Table 7.3.
- (c) **Text:** Following the same logic behind the decision of edge colours, all text in the IAML visual syntax is either coloured black, gray, or dark blue. Gray is only used for labels that are read-only, such as includes container names (Section 7.4.4). Dark blue is only used to distinguish the *containment feature* of particular elements, such as [Events](#), as discussed later in the next section.

<sup>6</sup>According to the style and design guidelines of the Eclipse user interface [199], the predominant colours used in the palette are Blue, Yellow, Green, Red, Brown, Purple, and Beige.



5. **Line weights** for all nodes and edges within IAML are rendered at the default line weight. As per the guideline proposed by Rumbaugh [291] – “distinctions are not too subtle” – no other line weights are used, as it can be difficult to distinguish between line weights when a diagram is printed or viewed in a different screen resolution.
6. **Line patterns** refer to the pattern used on an edge or border; in the visual syntax of IAML, most edges and borders are rendered as a solid line. A single form of dashed line is used for elements involving some level of interactivity or behaviour, to ensure distinctions between line patterns are not subtle. In particular, a dashed line is used to represent all [ECA Rule](#), [Wire](#), [Data Flow Edge](#) and [Constraint Edge](#) instances.
7. **Font styles** refer to the font faces, sizes, weights, and decorations used for text on model elements.
  - (a) **Font face:** It is not desirable for an interface to use too many different font faces. Only one font face is therefore used across all IAML model elements.
  - (b) **Font size:** Similarly, it is not desirable for an interface to use too many different font sizes, and only one font size is used across all IAML model elements.
  - (c) **Font weight:** Font weight refers to the measure in which a font is emboldened. In IAML, the two font weights of normal and bold are used, and most IAML model elements are represented using the normal weight. The only two exceptions are the name of the *containment feature* and to highlight overridden elements, as discussed in the next section.
  - (d) **Font decorations:** Font decorations refer to whether a font is in italic, or underline, or both. All labels in the IAML visual syntax are represented as undecorated labels, with one exception; the metaclass name of a model element is displayed with an underline as per the UML notation [253, pg. 201], as discussed in the next section.

For each model element in the IAML language, a value for each of these parameters is selected according to the design decisions behind the parameter, and encoded as a diagram definition in a *gmfgraph* model instance. A comprehensive reference to the visual syntax used to represent each modelling element is provided in Appendix I.

#### 7.4.4 Diagram Extensions

While the GMF framework is powerful, it is still necessary to extend the generated IAML graphical editors to provide additional domain-specific functionality. For example, breadcrumbing and metaclass name extensions would be very difficult or impossible to achieve strictly within the GMF runtime, and these extensions should be provided to the editors in an automated way.

There are three ways in which diagram editors generated through GMF may be extended. One method is by providing extensions through Eclipse extension points, defined by GMF<sup>7</sup>. Another is by modifying the generated source code manually; the GMF framework adheres to the semantics of the `@generated` tag as from EMF, as discussed in Section 6.2.1.

Finally, *dynamic templates* may be provided to the generation framework. These templates use the functionality of dynamic templates provided by Xpand, as discussed in Section 6.4.3. In particular,

<sup>7</sup>One such extension point is the `org.eclipse.gmf.runtime.diagram.ui.layoutProviders` extension point, which is used to define layout providers for diagram editors.



a template file with the same name and the same location as an existing Xpand template will be used to extend or replace the original template. This is achieved with the **AROUND** operator and the `targetDef.proceed()` command.

Manual modifications to the diagram editors are not used in IAML. This means that no diagram editor code needs to be modified manually, and thus the source code does not need to be committed to the version control repository. Extension points would be an ideal way of modifying diagram editors; extension points allow extensions to be added and removed silently, improving performance and scalability, and reducing errors by defining well-designed interfaces [117]. However, the extension points available at the time could not be used to implement some of the necessary diagram extensions.

The generated diagram editors in IAML were therefore extended through both extension points and dynamic templates. The *Diagram Actions* component represents instances of extensions through extension points, and is discussed later in Section 7.4.5; this section will instead discuss the use of dynamic templates for diagram extensions. Five diagram editor extensions will be discussed in this section: *breadcrumbing*; *shortcuts*; *container names*; *metaclass names*; and *generated element notations*. These dynamic templates are supported by a library of common code to improve performance and reduce dependencies, which is stored in a separate OSGi bundle.

The dynamic templates for the *Diagram Extensions* component are stored within the `org.openiaml.model` OSGi bundle, along with the dynamic templates used for the *Model Instance* component. The library of common code used by the *Diagram Extensions* component is provided by the `org.openiaml.model.diagram.helpers` OSGi bundle.


## Breadcrumbing

As discussed earlier in Section 4.1.2, it is important for a hierarchical modelling approach to provide appropriate context. For example, a developer may “zoom into” one of two unrelated **Frames**, and each may both contain an **Input Text Field** labelled “name”; context is necessary to inform the developer of which **Frame** instance they are currently viewing.

*Breadcrumbs* are a technique used in web applications to provide context information to the user about their current location within a web site, which may improve user satisfaction and the efficiency of site navigation [36]. This technique is adapted to the diagram editors of IAML; these breadcrumbs highlight the navigation path from the root model element (the **Internet Application**) to the currently viewed model element via the implicit *container* references of each model element, as discussed earlier in Section 5.4.2. This breadcrumb is calculated automatically and does not persist as part of the diagram; an example of breadcrumbs is illustrated earlier in Figure 7.4.

## Shortcuts

While breadcrumbing provides context for a particular view of the contents of a model element, it does not provide information about references to model elements outside of the current model element contents. For example, when a **Button** is intended to navigate to another **Frame** when the *onClick* event is triggered, this external **Frame** reference should be visible.

GMF supports this concept through *shortcuts*, where external element references are displayed as normal elements, but with an additional shortcut annotation (). These model elements must have a corresponding shape/element mapping in the *gmfmap* definition of the current editor, but does not

necessarily have to have a corresponding palette tooling entry in the *gmftool* definition. Additionally, these shortcuts are only displayed for node types, and not for edge types such as [Wires](#).

It was necessary to extend the default GMF implementation of shortcuts. With the default implementation, only elements that can be directly contained in the view element could be added as shortcuts; edges between elements would disappear if they are not directly contained; and edges between shortcut elements did not persist across views. Some of these issues are acknowledged GMF bugs, whereas others are expected functionality.

To resolve these issues, a custom shortcut controller *GetShortcuts* was implemented, which provides a list of relevant nodes and edges that should be rendered as shortcuts contained within a particular model element instance. This controller is then integrated into the generated GMF editors through the use of dynamic templates.

### Container Names

While shortcuts can be used to display a model element outside of the currently viewed container, the shortcut icon is a binary annotation that does not provide context to where the model element is actually contained. To resolve this issue, a reference to the *container* of the particular model element is displayed for every model element. Currently this reference is a label documenting the containers' name, as illustrated in Figure 7.4.

Not all model elements need a reference to their parent displayed; for example, [Domain Types](#) and [Predicates](#) are global throughout the entire application, and such a parent reference would clutter up the visual representation. Parent labels are therefore only displayed for certain element types<sup>8</sup>.

### Metaclass Names

As discussed earlier in Section 5.4.4, UML generally uses differences in visual notation to distinguish between the instances of different metaclasses. That is, an [Input Form](#) should have a different visual notation to an [Input Text Field](#). However, developing unique visual notations for every model element represents a significant amount of effort. A simple interim solution was to follow the UML syntax for defining instances of classes – as illustrated in the UML infrastructure specification [253, pg. 201] – where a model element may also display the name of the defining metaclass of that element.

An example of how metaclass names are displayed on every model element is illustrated in Figure 7.4. These element instance names are derived automatically from the model element instance class at the time that the model element is displayed, and these labels do not persist in the visual notation. The textual format of metaclass names also means that every different model element type may be distinguished, even if the representation is printed without any colours.

### Containment Feature

In terms of EMF, a *containment feature* is a reference that may contain other elements. The IAML metamodel was designed to reduce containment feature redundancy, where all model elements should only be contained via a single containment feature<sup>9</sup>.

<sup>8</sup>At the time of writing, these element types were [Domain Attribute](#); [Domain Attribute Instance](#); [Event](#); [Operation](#); [Function](#); [Value](#); [Frame](#); [Gate](#); and [Visible Thing](#).

<sup>9</sup>For example, if a model element could contain any number of [Functions](#), and a subtype of this model element could additionally contain any number of [Predicates](#), then an instance of a [Boolean Property](#) could be contained within either of these containment features by an instance of this subtype.

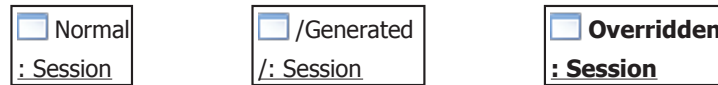


Figure 7.5: Illustrating generated and overridden elements in the IAML editor

However, the IAML metamodel does support containment feature redundancy for certain elements; for example, a [Changeable](#) element defines the *onAccess* and *onChange* events through containment features earlier in Figure 5.16, and each of these references must hold an [Event](#) instance. The functionality of each [Event](#) depends on the containment feature that contains it; however within GMF, a shape is defined by the element type itself, and does *not* consider the containing feature<sup>10</sup>.

To distinguish between the different containment features which may contain an element, one option is to define a range of element shapes, one for each containment feature; for example, the *onAccess* event could have a slightly modified shape, or the *onChange* event could have a different background colour. However, it is preferable to keep the shape styles consistent, as it is only the containment feature that dictates the differences in functionality.

A *containment feature label* is therefore defined for elements that may be contained through many containment features<sup>11</sup>. These labels are rendered in a dark blue and bold font, and displays the name of the containment feature. An example of this containment feature label is illustrated in Figure 7.4; this label is also provided automatically and does not persist as part of the visual notation.

## Generated Element Notations

As a novel concept that directly interacts with the intent of a model developer, the usability of the implementation of model completion depends strongly on the support of the development environment. For example, a visual notation is necessary for distinguishing the difference between developer-created elements, and elements created through model completion. Based on the UML concept of *derived* properties with a name prefix of a forward slash “/”, all generated elements are prefixed with this slash, as illustrated in Figure 7.5.

Similarly, a notation is necessary to distinguish elements that have been manually overridden by the developer, as discussed later in Section 7.5.2. One approach to solving this problem is derived from the use of stereotypes or classifiers in UML visual syntax, where textual stereotype labels such as «generated» or «overridden» may be rendered on generated or overridden model element instances.

Another approach to solving this problem is illustrated within Mozilla Firefox; overridden configuration values are highlighted in bold, allowing users to quickly identify overridden values. This approach is used in IAML diagram definitions, where every label for an overridden elements is rendered in bold, as illustrated in Figure 7.5. In this proof-of-concept implementation, this approach is preferred over using textual stereotype labels as it is simpler to implement, and reduces the visual complexity of the resulting diagrams.

<sup>10</sup>A more detailed discussion on GMF view mapping mismatches between the underlying metamodel and the resulting representation is discussed later in Section 9.3.3.

<sup>11</sup>At the time of writing, these elements were [Gate](#); [Event](#); [Value](#) and [Builtin Property](#).

### 7.4.5 Diagram Actions

Some of the extension points provided by GMF are used to define additional action-based extensions to the diagram editors. Two actions have been provided to operate on the overall serialisation of the visual representations of IAML model instances – i.e. `iaml_diagram` files – as follows:

1. **Export to PNG:** GMF provides a builtin command to export a single diagram view to a range of image formats, such as GIF, PNG and SVG. This action extends this functionality to also export *all* of the views of this diagram; that is, every hierarchical level of the diagram, from the root element to every child.
2. **Export to HTML:** This action extends the previous action to also generate a set of HTML files, which provide `imagemaps` [333] on the exported PNGs to allow a user to navigate through the model instance graphically. This action is used by the *ModelDoc* framework [366] to export navigable example models.

Three actions were also defined to operate on individual model elements within a particular diagram, and can also be applied to collections of individual model elements.

1. **Infer only contained elements:** This action applies model completion to only the selected element, in order to create all of the elements that would be contained by this particular element if model completion were to occur. That is:
  - (a) First the entire model is completed, as in the *Infer all elements* action discussed later in Section 7.8.
  - (b) The contents of the model are iterated over, and all *generated* elements – that is, `Generated Elements` where `is generated` is `true` – that are not contained (directly or indirectly through children) by the target element are removed.
  - (c) All *generated* elements that refer to deleted elements are then recursively removed. This process is repeated until the model has stabilised.

It is important to note that this action is not guaranteed to be safe according to the properties of model completion. In particular, subsequent model completions on this partially inferred model instance may create a different model instance, i.e.  $C(X) \neq C(C(X) - Y)$ . However, in practice this restriction has not yet proven problematic, and it helps the model developer in overriding the contents of individual model elements.

2. **Remove contained generated elements:** This command similarly operates on a single element, and attempts to remove all *generated* elements that are contained within the selected element. If there are other non-generated elements that refer to the deleted generated element (and its contained children), then the developer is warned that the resulting model may be inconsistent.
3. **Move into separate model:** This refactoring action moves the selected diagram elements into a separate IAML model instance, allowing for complex model instances to be split into smaller model instances. This action simply reuses existing EMF functionality [314, pg. 33].

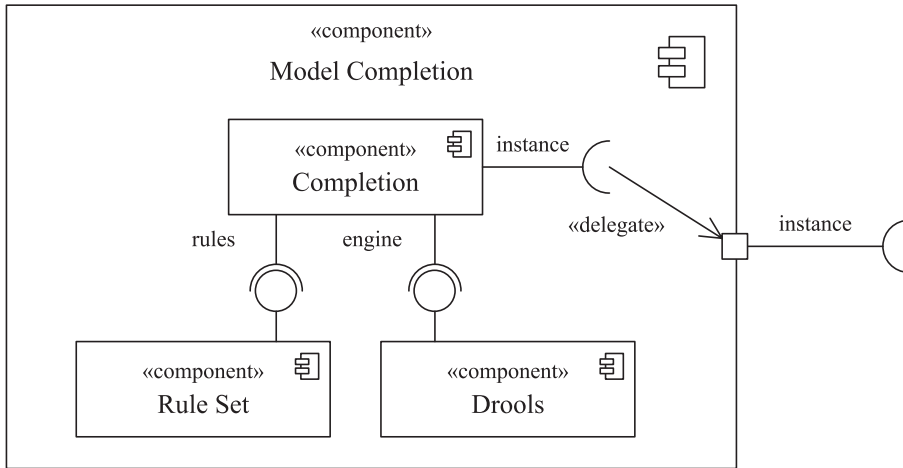


Figure 7.6: UML component diagram of the *Model Completion* Component Decomposition

The corresponding OSGi bundle for the implementation of the *Diagram Actions* component is `org.openiaml.model.diagram.custom`.

## 7.5 Model Completion

As discussed throughout this thesis, model completion may be used to complete additional functionality on a model instance. This includes the implementation of *Wires* such as *Sync Wires* and *Autocomplete Wires* (Section 5.9); completing the *Domain Feature Instances* within *Domain Instances* based on their defining *Domain Types* (Section 5.6.4); or the implementation of client-side input validation based on the primitive types of *Visible Things* (Section 5.5.4).

The *Model Completion* component defines the implementation of the model completion process, as proposed by Wright and Dietrich [369] and discussed earlier in Section 3.2. The underlying rule engine behind the implementation of model completion in IAML is Drools [311], as selected through the evaluation process of the previous chapter in Section 6.5.

The decomposition of this component is provided in Figure 7.6<sup>12</sup>. This component is made up of three sub-components: the *Drools* component, which provides the implementation of the Drools rule engine; the *Rule Set* component, which encodes the logic and intent behind IAML model completion into rules into a format for the rule engine; and the *Completion* component, which connects the rules, engine, and model instance together, and provides the interface necessary to execute model completion on model instances.

The corresponding OSGi bundle for the implementation of the *Model Completion* component is `org.openiaml.model.drools`, and all sub-components provided within this same bundle.

<sup>12</sup>This decomposition represents an ideal-world situation, where standards such as RuleML [353] should allow the transparent exchange of different rule engines without loss in functionality; realistically, Drools rules include functionality specific to the engine itself.

### 7.5.1 Completion

The *Completion* component integrates the set of model completion rules (the *Rule Set* component) into an instance of the rule engine (the *Drools* component), and also extends the rule engine with the requirements necessary to implement model completion according to the requirements proposed by Wright and Dietrich [369]. In particular, the implementation of the *insertion cache* [369] is provided by this component<sup>13</sup>.

### 7.5.2 Rule Set

The model completion rules necessary to implement the semantics of the modelling language in Appendix I are represented here as the *Rule Set* component, and are implemented within 263 instances of Drools rules. To simplify development, these rules are separated into 21 separate *rule packages*, each within a single file. Each rule package implements one aspect of model completion, and a summary of the model completion rules used in IAML is provided in Table 7.4. The inclusion of the source code of each model completion rule used in the implementation of IAML is well outside the scope of this thesis.

As discussed earlier in Section 3.2.1, an important aspect of model completion is ensuring that the rules are well-documented, so that the developer can anticipate the intent of the process. This documentation requirement is addressed by the *ModelDoc* framework [366], where the `@inference` tag can be used to describe the intent of a model completion rule. A sample implementation of one of these rules is provided in Listing 19; this inline documentation is used later to populate the documentation of the *Sync Wire* element itself in Section I.94.

One key idea behind model completion is that depending on the intent of the model developer, generated elements may be removed, or model completion may be selectively disabled [369]. The IAML proof-of-concept implementation supports this concept by selectively overriding the generation of elements, using the `overridden` and `overridden names` properties of *Generated Elements*.

---

<sup>13</sup>The implementation of the insertion cache is provided by `DroolsInsertionQueue.java`.

```
/**
 * @inference SyncWire When a {@model Changeable} is connected to an
 *   {@model ContainsOperations} by a {@model SyncWire}, the source
 *   element will {@model ECARule call} the 'update' {@model Operation} on
 *   the target when the source {@model Changeable#onChange} changes.
 */
rule "Run instance wire from edit to update (onChange)"
  when
    sw : SyncWire( overriddenNames not contains "run" )
    source : Changeable( )
    target : ContainsOperations( )
    eval( functions.connects(sw, source, target) )

    event : Event( source.onChange == event )
    operation : Operation( eContainer == target, name=="update" )

    not ( ECARule( trigger == event, target == operation, name == "run",
                  eContainer == sw ) )

    eval( handler.veto(sw) )

  then
    ECARule rw = handler.generatedECARule(sw, sw, event, operation);
    handler.setName(rw, "run");
    queue.add(rw, drools);

end
```

Listing 19: One Drools rule used in the model completion implementation of the [Sync Wire](#) model element



Rule File	Rules	Description
autocomplete-wires.drl	13	Completes the functionality of <a href="#">Autocomplete Wires</a> .
base.drl	6	Creates 'field value' <a href="#">Values</a> and the related 'is set' <a href="#">Predicate</a> .
casting.drl	6	Rules related to <a href="#">types</a> and casting, and the 'can cast?' <a href="#">Predicate</a> .
conditions.drl	8	Recursively connects <a href="#">Simple Conditions</a> and <a href="#">Parameters</a> with respect to model completion on <a href="#">Wires</a> .
detail-wires.drl	9	Completes the functionality of <a href="#">Detail Wires</a> .
domain.drl	3	Completes the content of <a href="#">Domain Types</a> with respect to type <a href="#">inheritance</a> .
emails.drl	7	Completes the default <a href="#">Values</a> of <a href="#">Emails</a> .
events.drl	2	Create the default <a href="#">Events</a> for <a href="#">Accessible</a> and <a href="#">Changeable</a> elements.
file-domain-object.drl	2	Completes functionality necessary to use <code>.properties</code> files as <a href="#">Domain Sources</a> .
gate.drl	16	Completes the functionality and logic of <a href="#">Gates</a> within <a href="#">Frames</a> and <a href="#">Scopes</a> .
instance.drl	12	Creates the default <a href="#">Operations</a> and <a href="#">Predicates</a> for <a href="#">Domain Iterators</a> .
login-handler.drl	60	Secures <a href="#">Scopes</a> based on <a href="#">Login Handlers</a> and <a href="#">Access Control Handlers</a> contained within that <a href="#">Scope</a> .
new-instance.drl	4	Creates the <a href="#">Domain Instances</a> within <a href="#">Domain Iterators</a> and refreshes <a href="#">Domain Attribute Instance</a> mappings with respect to the connecting <a href="#">Domain Type</a> of the iterator.
operations.drl	6	Creates the default <a href="#">Operations</a> for <a href="#">Changeable</a> elements, and the implementation of certain <a href="#">Activity Operations</a> .
paginate.drl	19	Creates the interface necessary to permit pagination through <a href="#">Domain Iterator</a> results when connected to an <a href="#">Input Form</a> .
sessions.drl	2	Initialises the necessary contents <a href="#">Sessions</a> , including the onInit <a href="#">Event</a> .
set-wires.drl	34	Completes the functionality of <a href="#">Set Wires</a> .
sync-wires.drl	21	Completes the functionality of <a href="#">Sync Wires</a> .
users.drl	22	Completes the functionality of <a href="#">Roles</a> when connected to <a href="#">Login Handlers</a> or <a href="#">Access Control Handlers</a> , including role <a href="#">inheritance</a> and initialisation of the default 'User' <a href="#">Role</a> .
validate.drl	4	Rules related to runtime input validation, including type checking.
visible.drl	7	Creates the default <a href="#">Events</a> , <a href="#">Operations</a> and <a href="#">Predicates</a> of <a href="#">Visible Things</a> .

Table 7.4: Summary of the rules used for model completion on IAML model instances

The behaviour of the *overridden* property is simple. As part of the creation of new modelling elements through model completion, each modelling element is referenced as *generated by* a particular source element<sup>14</sup>; if this source element is specified as *overridden*, then this element will be ignored by the model completion framework. The *overridden names* property specifies a list of element names that should not be generated as *children* of a particular model element, while allowing model completion to proceed normally.

As these two properties are defined as normal attribute within the metamodel, they may be modified by interacting with the standard properties dialog displayed for any model element instance. As described earlier in Section 7.4.4, all overridden elements are represented in the graphical editor in a bold font, to illustrate that the element is overridden.

### 7.5.3 Drools

The final component necessary to implement model completion is the implementation of the Drools rule engine, as described by Proctor et al. [279]. At the time of writing, version 4.0.7 of the Drools engine was used, which is provided as a collection of JARs.

## 7.6 Code Generation

The *Code Generation* component implements the model-to-text transformation step necessary to translate an IAML model instance into an executable web application. The underlying code generation engine behind this implementation is openArchitectureWare (OAW) [87], as selected through the evaluation process of the previous chapter in Section 6.4. This component relies on a number of sub-components, and the decomposition of the *Code Generation* component is provided in Figure 7.7<sup>15</sup>.

This component is made up of six sub-components: the *openArchitectureWare* component, which provides the OAW implementation; the *Templates* component, which provides the code generation templates themselves; the *Runtime Library* component, which provides additional libraries necessary for runtime applications; the *Platform Configuration* component, which provides platform-specific information to the platform-independent IAML model instance; the *Output Formatter* component, which reformats generated source code into a more readable form; and the *Generator* component, which connects all of the components together into a single workflow.

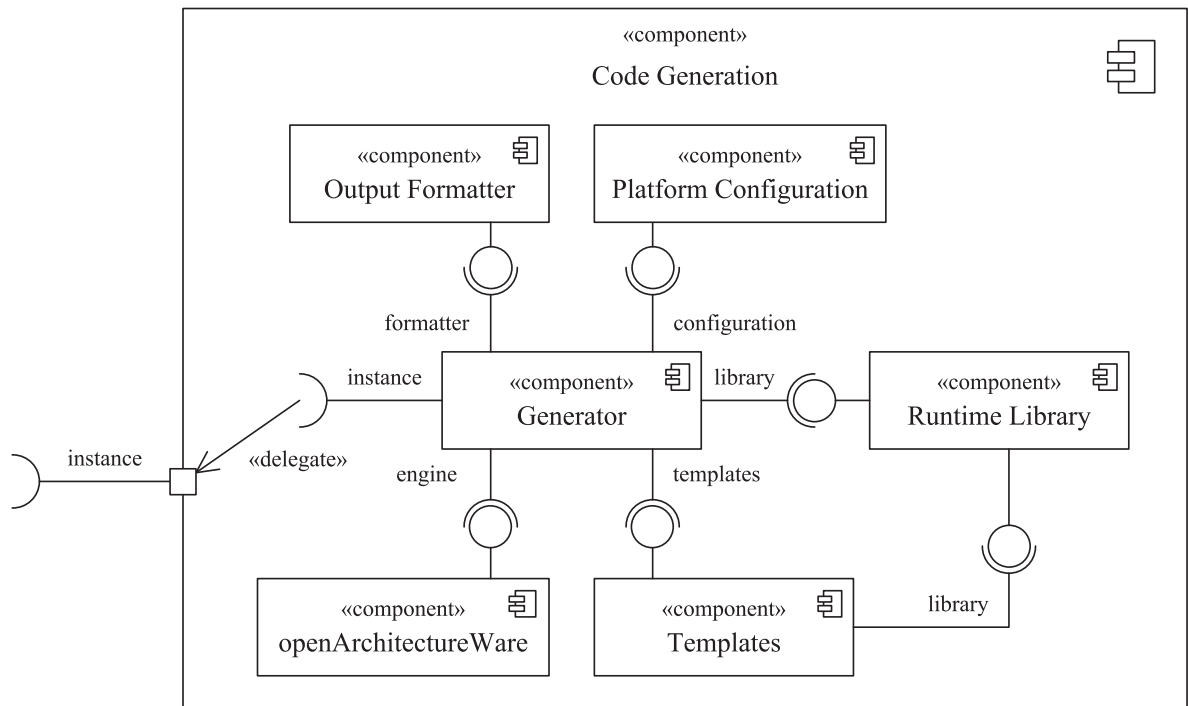
### 7.6.1 openArchitectureWare

This component simply provides the implementation of the OAW template engine, as described by Efftinge et al. [87]. The code generation templates were implemented with the openArchitectureWare 4.3.1 framework. As discussed earlier in Section 6.4.4, it is desirable to migrate these templates to the recently released Xpand framework, and this remains future work<sup>16</sup>. The OAW library is implemented as a set of Eclipse plugins, and thus is available as a set of OSGi bundles.

<sup>14</sup>This also means that for an element to be used in the generation of another element, this element must be a subtype of the *Generates Elements* interface.

<sup>15</sup>Similarly to the decomposition of the *Model Completion* component, this represents an ideal-world situation where code generation templates may be executed using different code generation engines without any loss in functionality.

<sup>16</sup>Issue 71: *Migrate OAW Xpand to Eclipse Xpand*.

Figure 7.7: UML component diagram of the *Code Generation* Component Decomposition

The corresponding OSGi bundles for the implementation of the *openArchitectureWare* template engine component are `org.openarchitectureware.dependencies` and `org.openarchitectureware.workflow`, provided by `openArchitectureWare.org`.

### 7.6.2 Templates

The collection of code generation templates necessary to implement the behaviour of IAML model elements are represented as the *Templates* component, and are implemented using 160 template files. The *openArchitectureWare* Xpand implementation of one of these templates – `runFrameEvents`, which implements the triggering behaviour of the *onInit* and *onAccess* Events within a *Frame* – is illustrated in Listing 20.

The target languages of the proof-of-concept code generation implementation is a combination of PHP 5.3 [207]; Javascript 1.6 [99]; HTML 4.01 [333]; CSS level 2 [344]; and the relational database *sqlite* [266]. At the time that these templates were implemented, the development of HTML 5 had not yet reached the recommendation stage. PHP was selected as the underlying server-side language as it was already well-known, and a PHP-based implementation would strongly assist in the proof that the IAML metamodel is platform-independent.

These templates include some model instance extension definitions, using the Xtend language [87]. For example, the extension `containingScope` implements the semantics of the *containing scope* definition earlier described in Section 5.11; the source code for this metamodel extension is illustrated in Listing 21. The collection of templates also require the runtime library, as discussed in the next section.

```

/**
 * @implementation Action
 *   {@model Action}s are run in order of descending priority; that is,
 *   a higher priority {@model Action} will execute first.
 */
«DEFINE runFrameEvents(String eventName) FOR model::Event-»
  // Frame EventTrigger «this»
  «IF eventName == "init" || eventName == "access"-»
    // Actions sorted by priority
    «EXPAND template::operations::OperationCall::callOperation(true, false)
      FOREACH listeners.sortBy(e|-e.priority)-»
    «ELSE-»
      «throwException("I don't know what to do with frame event " + name)»
    «ENDIF-»
  «ENDDEFINE»

```

Listing 20: Implementation of the runFrameEvents code generation template in Xpand

```

/**
 * Get the containing {@model Scope} of the given element,
 * or <code>null</code> if none.
 */
model::Scope containingScope(emf::EObject this) :
  null;

model::Scope containingScope(model::NamedElement this) :
  containingScope(eContainer);

model::Scope containingScope(model::Scope this) :
  this;

model::Scope containingScope(model::InternetApplication this) :
  null;

```

Listing 21: Implementation of the containingScope model extension in Xtend

The corresponding OSGi bundle for the implementation of the *Templates* component are `org.openiaml.model.codegen.php`.

### 7.6.3 Runtime Library

The *Runtime Library* component provides common libraries necessary for the generated web applications, in order to obtain a number of maintainability-related benefits. In particular, these libraries can be tested independently of the code generation templates; improvements in the libraries can be published independently of the metamodel; code generation templates can be simpler, and generated source code will consequently be simpler; and it is easier to include third-party library components as part of an included library.

Much of the runtime library is the implementation of a single library in two different implementation languages. For example, the typing system of IAML has to be implemented both in the server-side

PHP language, and in the client-side Javascript language. For functionality that cannot be executed on the client-side – for example, sending e-mails – callback interfaces must be implemented. Similarly, builtin XPath functions such as `fn:contains` must be implemented in both PHP and Javascript as part of the implementation of [XQuery Functions](#).

The runtime library also includes two third-party components: *PHPMailer* [8], which provides a rich interface to send e-mails and is used in the implementation of [Email](#); and the *Prototype Javascript Framework* (prototype.js) [280], which may be used to simplify the development of client-side Javascript within RIAs. The development of this library also included the development of an additional component *html2text*, which provides functionality to transform an arbitrary block of HTML-formatted text into a text-only format, which looks similar to the corresponding HTML representation.

The corresponding OSGi bundle for the implementation of the *Runtime Library* component are `org.openiaml.model.runtime`, and the third-party PHPMailer library is provided through the OSGi bundle `org.openiaml.model.runtime.phpmailer`.

#### 7.6.4 Platform Configuration

There are a number of platform-specific configuration elements that are necessary for the generation of a web application, such as proxy information and API keys. As one of the design goals of the IAML metamodel was to provide a platform-independent metamodel, it is not possible to place this configuration information in these model instances. These platform-specific configuration values are instead defined within a *platform-specific metamodel*, and these model instances are instead provided to the code generation framework.

Currently, the platform-specific configuration attributes of web applications generated through IAML are provided as a set of key/value pairs to the *Code Generator* component. A summary of these properties are provided in Table 7.5. However, this model instance could be represented as an instance of an EMF metamodel, which would afford the resulting platform-specific model instances all of the benefits of a model-driven approach, as described earlier in Section 3.1.3. This approach remains future work<sup>17</sup>.

The implementation of the *Platform Configuration* component is included as part of the *Generator* component within the `org.openiaml.model.codegen.php` OSGi bundle.

#### 7.6.5 Output Formatter

If a code generation framework is generating source code for a language that uses a freeform syntax<sup>18</sup>, the resulting source code is often poorly structured in terms of readability, as the code generation templates do not need to understand the syntax of the target language. While this unreadable source code is still valid program code, it may be desirable to transform this source code into a readable format, to assist in debugging and third-party extensibility.

A common approach to solve this problem is to provide a component to reformat the generated source code; for example, the EMF framework uses Eclipse's Java Development Toolkit (JDT) project

<sup>17</sup>Issue 87: *Develop platform-specific metamodel for project properties*.

<sup>18</sup>For example, languages such as Python and Haskell use whitespace and indentation as part of the language syntax, and are *not* freeform.

Variable	Description
<i>config_runtime</i>	A file reference to the location of the server-side libraries of the <i>Runtime Library</i> component.
<i>config_web</i>	A file reference to the location of the client-side libraries of the <i>Runtime Library</i> component.
<i>debug</i>	If set to <code>true</code> , debugging information will be rendered as part of the generated code.
<i>email_handler</i>	The handler used to send <a href="#">Emails</a> . Only two values are supported: <code>php-mailer</code> and <code>file</code> .
<i>email_handler_phpmailer_include</i>	For e-mails handled by <code>phpmailer</code> , a file reference to the location of the PHPMailer libraries.
<i>email_handler_file_destination</i>	For e-mails handled by <code>file</code> , a file reference to where sent e-mails will be output.
<i>map_handler</i>	The handler used to render <a href="#">Maps</a> . Only two values are supported: <code>googlemaps</code> and <code>mock</code> .
<i>google_maps_api_key</i>	For maps rendered using <code>googlemaps</code> , the Google Maps API requires an API key to be provided.
<i>proxy_host</i>	If set, this proxy host will be used when accessing remote data.
<i>proxy_port</i>	If set, this proxy port will be used when accessing remote data.
<i>proxy_userpass</i>	If set, this proxy username and password will be used when accessing remote data.

Table 7.5: The set of platform-specific configuration properties used by the IAML *Code Generator* component

to reformat the generated metamodel source code [314, pg. 358]. In openArchitectureWare terms, this formatter is known as a *postprocessor*, which operates on file instances as they are created through the workflow [87].

Because the web applications generated by the IAML code generator are a mix of PHP, Javascript, CSS and HTML, it is difficult to format the source code correctly using a single AST-based parser. Source code of these languages may occur at any time at any arbitrary location; for example, a HTML page may contain a Javascript script, which then contains a PHP instruction within a variable identifier. At the time of writing, no existing parser supported simultaneous formatting for these four languages.

As part of this proof-of-concept implementation of IAML, the *iacleaner* project was developed to apply code formatting to the diverse mix of web application languages used within IAML. This project uses a custom parser which provides the ability to “jump out” between languages, but does not construct an AST. The *iacleaner* project is released as an Eclipse plugin under the Eclipse Public License, and is available online at <http://code.google.com/p/iacleaner/>.

The corresponding OSGi bundle for the implementation of the *Output Formatter* component is `org.openiaml.iacleaner`.

### 7.6.6 Generator

The final component necessary to implement the code generation component of IAML is the *Generator* component, which orchestrates all of the sub-components together. This component includes

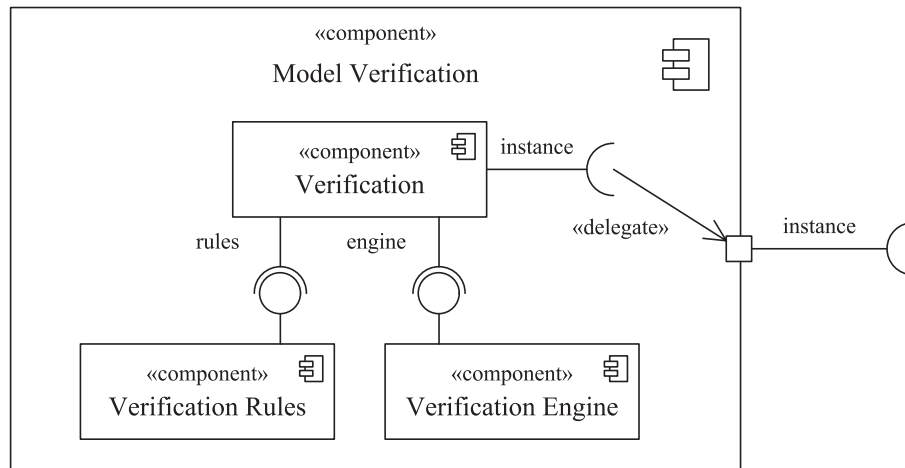


Figure 7.8: UML component diagram of the *Model Verification* Component Decomposition

an instance of an openArchitectureWare *workflow* [87], which is executed by the template engine to generate the web application source code. This component also includes the user interface actions necessary to execute the code generation on a particular model instance.

The corresponding OSGi bundle for the implementation of the *Generator* component is `org.openiaml.model.codegen.php`.

## 7.7 Model Verification

The *Model Verification* component defines the implementation of model instance verifiers. As discussed in Section 6.6, there are many different types of verification that may be performed on a model instance, each with differing requirements and necessary resources; therefore, it is beneficial to have an overall framework for model instance verification.

This component provides this model verification framework, which is implemented as the combination of four sub-components as illustrated in Figure 7.8<sup>19</sup>. Essentially, each type of verification is implemented as a set of verification rules (the *Verification Rules* component), executing on a particular verification engine (the *Verification Engine* component), that operates on a particular model instance (the *Verification* component).

GMF diagram editors integrate constraint violations registered through the EMF Validation framework into the corresponding visual representation of a model instance. When a constraint violation is detected on a model element, a problem is registered in the Eclipse problems view [117], and a graphical annotation is displayed on its visual representation. For example, an invalid *Sync Wire* which violates the constraint specified in Listing 22 will be displayed to the model developer as illustrated in Figure 7.9.

In this section, the implementation of the four selected verification engines as discussed earlier in Section 6.6 – openArchitectureWare Checks, OCL through the EMF Validation Framework, CrocoPat

<sup>19</sup>Similarly to the decomposition of the *Model Completion* component, this also represents an ideal-world situation where verification rules may be executed using different verification engines without any loss in functionality, which are one of the aims of the JSR-94 [306] and SBVR [256] specifications.



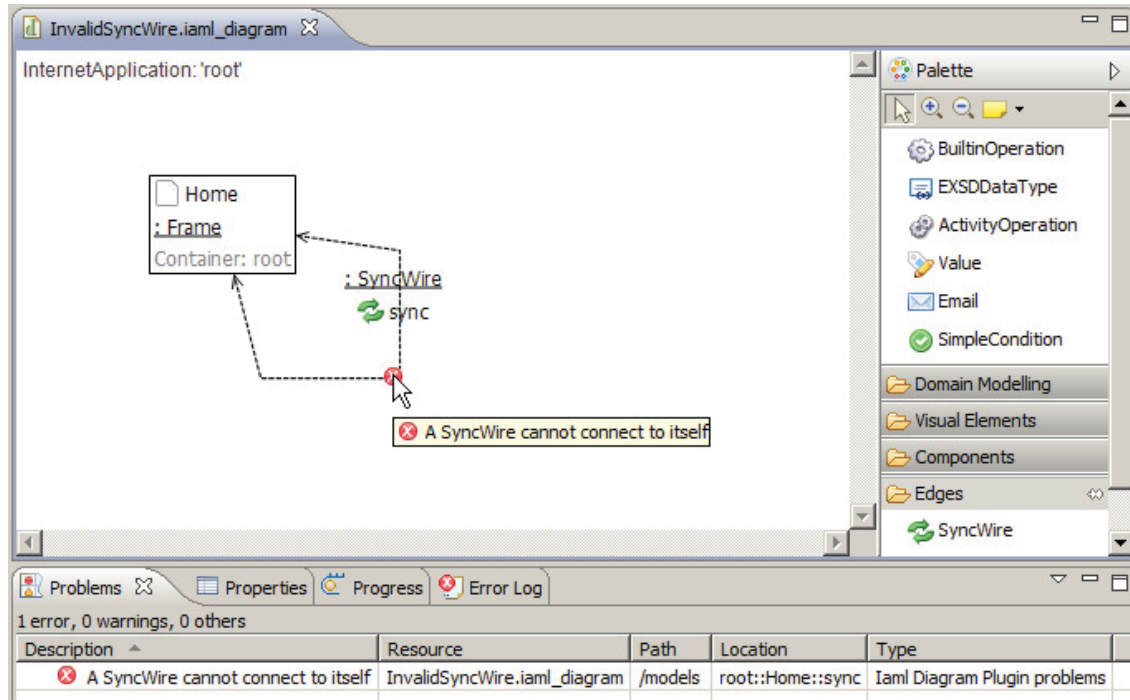


Figure 7.9: Visualisation of model instance constraint violations within a diagram editor

```
context model::wires::SyncWire ERROR "A SyncWire cannot connect to itself":
  from != to;
```

Listing 22: Implementation of a Checks constraint for Sync Wires

and NuSMV – will be discussed. With the exception of Checks, each of these components are implemented as a separate set of OSGi bundles, allowing for verification to occur independently of the development process as configured by a model developer.

### 7.7.1 Model Instance Verification with Checks

The Checks language forms part of the openArchitectureWare platform [87], and provides a model instance verification syntax that is a hybrid of Java and OCL. Due to the ease of implementation and framework functionality of openArchitectureWare, this technology was selected in Section 6.6.9 as the ideal technology to implement constraints within a functions-based verification language.

Consequently, the Checks-based *Verification Engine* component is provided by the openArchitectureWare 4.3.1 framework, and the Checks-based *Verification Rules* component is a collection of 70 constraints. These constraints are provided through the code generation plugin, as model instance verifier is always executed as a prerequisite of code generation. All of the constraints placed on the IAML metamodel throughout Appendix I were implemented in terms of Checks constraints. For example, a constraint that specifies that a *Sync Wire* cannot be used to synchronise an element with itself is illustrated in Listing 22.

For Checks model instance verification, the *Verification Engine* component is provided by the `org.openarchitectureware.dependencies` and `org.openarchitectureware.workflow`

```

<constraint statusCode="1" severity="ERROR" lang="OCL"
  name="Value constraint 1" id="ocl1">
  <message>Values must be type XSDSimpleType or EXSDDataType</message>
  <target class="Value"/>
  <!-- the OCL expression -->
  <![CDATA[
    self.type->isEmpty()
    or self.type.oclIsKindOf(xsd::XSDSimpleTypeDefinition)
    or self.type.oclIsKindOf(model::EXSDDataType)
  ]]>
</constraint>

```

Listing 23: Implementation of an OCL constraint for *Values* in the EMF Verification Framework

OSGi bundles; and both the *Verification Rules* and *Verification* components are provided by the `org.openiaml.model.codegen.php` OSGi bundle.

### 7.7.2 Model Instance Verification with OCL in the EMF Validation Framework

While most metamodel constraints are defined and verified using the `openArchitectureWare Checks` verification component, there are a number of metamodel constraints defined earlier in Chapter 5 which are defined in terms of OCL. It is therefore desirable to reuse these OCL constraints, to improve the robustness of the resulting model instances and prove these constraints are correctly implemented. As discussed earlier in Section 6.6.7, OCL is a functions-based verification language that also supports a form of higher-order logic.

As these constraints are also not defined in a recursive fashion – one of the considerations of the verification technology evaluation discussed earlier in Section 6.6.7 – these constraints may be implemented using the EMF Validation Framework. It is preferable to use this OCL implementation over the Dresden OCL2 toolkit as this framework is a member of the Eclipse ecosystem and well-integrated into EMF-based metamodels. OCL constraints are implemented as part of the `plugin.xml` plugin manifest file of an Eclipse plugin [117].

All of the constraints placed on the IAML metamodel throughout Chapter 5 were therefore implemented using the EMF Validation framework. For example, one of the constraints defined in Figure 5.8 – where a *Value* must define a *type* of either type *XSD Simple Type* or *EXSD Data Type* – is implemented within the EMF Validation framework as an OCL constraint in Listing 23.

For OCL model instance verification, the *Verification Engine* component is provided by the `org.eclipse.emf.validation` and `org.eclipse.emf.validation.oc1` OSGi bundles; and both the *Verification Rules* and *Verification* components are provided by the `org.openiaml.validation.oc1` OSGi bundle.

### 7.7.3 Model Instance Verification with CrocoPat

The CrocoPat framework [32] discussed earlier in Section 6.6.4 was selected as the ideal technology to implement constraints using a relations-based verification language, as CrocoPat supports the

higher-order operator **TC** of transitive closure on relations. As the CrocoPat engine is written in the C language, the component is not normally provided as an OSGi bundle; but for ease of implementation and component reuse, it has been manually wrapped into a bundle. Due to resource constraints, only the *infinitely redirects* constraint has been implemented in CrocoPat; this constraint has been discussed earlier in Section 6.6.4 and illustrated in Listing 13.

For CrocoPat model instance verification, the *Verification Engine* component is provided by the `org.sosy_lab.crocopat.cli` OSGi bundle; and both the *Verification Rules* and *Verification* components are provided by the `org.openiaml.verificatio.crocopat` OSGi bundle.

#### 7.7.4 Model Instance Verification with NuSMV

Finally, the NuSMV framework [56] discussed earlier in Section 6.6.8 was selected as the ideal technology to implement model checking. Similarly to the CrocoPat engine, the NuSMV engine is not provided as an OSGi bundle, but is manually wrapped into a bundle.

Due to resource constraints, only two constraints have been implemented using NuSMV; the *infinitely redirects* constraint as implemented earlier in Listing 18, and a constraint that tries to identify *infinite execution loops* expressed within an *Activity Operation*. Since the IAML model instance must first be translated into a textual NuSMV input format, a model-to-text transformation must be executed; these transformation templates are also implemented in the openArchitectureWare language.

Brambilla et al. [41] uses a similar approach to implement model checking of design properties of WebML model instances by specifying properties in the LTL language. This approach also uses model transformations to translate a system into a representation suitable for evaluation, and uses a combination of QVT and XSLT to implement these transformations.

For NuSMV model instance verification, the *Verification Engine* component is provided by the `it.itc.first.nusmv.cli` OSGi bundle; and both the *Verification Rules* and *Verification* components are provided by the `org.openiaml.verificatio.nusmv` OSGi bundle.

### 7.8 Model Actions

A separate *Model Actions* component integrates the components discussed throughout this chapter, and provides a user interface to access their functionality. In a similar fashion to the user interface provided by the *Diagram Actions* component, this user interface is implemented through context-sensitive menus through the Eclipse framework. Seven actions are defined to operate on model instances, i.e. `iaml` files.

1. **Infer all elements:** This action executes the model completion rules of the *Model Completion* component on the selected model instance, replacing the original model instance.
2. **Generate code:** This action first evaluates model completion against the given model instance, and then executes the code generation templates in the *Code Generation* component to generate an executable web application. By default, this generated code is placed into a directory named `output` within the current Eclipse project.

3. **Generate code and load in browser:** This action extends the *generate code* action to also load the generated application using the system web browser, if one is available. The URL of the generated application is taken from the `runtimeUrl` property of the *Internet Application*.
4. **Migrate model to latest version:** This action evaluates the model migrators of the *Model Migration* component on the given model instance. If the current model instance can be successfully migrated to the latest version of the metamodel<sup>20</sup>, then the original model instance is replaced with the migrated instance.
5. **Remove phantom edges:** During the development process of a model instance, instances of *Wires* or *ECA Rules* may lose their `from` and `to` references, causing constraint violations since they cannot be visualised. This action searches a model instance for instances of these *phantom edges* and removes them, since they will have no behavioural effect on the generated application.
6. **Rewrite generated element IDs:** All *Generated Elements* must define a unique element ID, which is initialised at creation time into a system-unique value<sup>21</sup>; however, these initial values are fairly verbose, such as `model.126aaefae53.29`. This action relabels element IDs to a more human-readable format; for example, an *Input Text Field* instance may be renamed to `text1`. Future work includes reimplementing the unique ID algorithm to render this operation unnecessary<sup>22</sup>.
7. **Export to DOT:** If the elements in a model instance are represented as nodes, and the relationships between the elements as edges, then the model instance can be considered a directed graph. This action exports the selected model instance into an instance of the DOT graph description language, as illustrated by Wright and Dietrich [369].

A single action is defined to operate on folders of model instances – or within Eclipse terminology, on instances of *IContainer* workspace elements.

1. **Migrate all IAML models:** This action recursively searches through the selected container to identify IAML model instances, and attempts to migrate each model instance according to the *Model Migration* component. Any problems identified are aggregated into a single issue, and presented to the developer at the end of the migration process. This action is particularly useful for migrating suites of example models, and was very useful in the development of the test suite of model instances as discussed in Section 7.9.

The corresponding OSGi bundle for the implementation of the *Model Actions* component is `org.openiaml.model.actions`.

## 7.9 Tests

A design goal discussed earlier in this thesis in Section 5.1.3 was to develop the proof-of-concept implementation of IAML simultaneously with the production of a suite of automated tests. As de-

<sup>20</sup>A “successfully migrated model instance” is one which can be loaded through EMF without any warnings or errors.

<sup>21</sup>This unique value is a combination of the package name, the time of package initialisation, and a package-specific counter; this generally guarantees uniqueness with  $O(1)$  computational complexity, since the model instance does not need to iterate through the model.

<sup>22</sup>Issue 25: *More descriptive generated IDs*.

scribed in Section 2.7.3, a well-designed suite of test cases can be used to improve the quality of the developed software. Consequently, each of the components discussed in this chapter have been implemented according to a range of automated JUnit test cases, within individual OSGi bundles.

In particular, the suite includes tests for the model completion framework and rules; the code generation framework and templates; metamodel and diagram model consistency; release quality tests; and general integration tests. A separate diagram test suite is also defined for testing aspects of the diagram editors themselves, such as the breadcrumbing, container names, and diagram partitioning aspects of the diagram editors.

In order to test the model-driven aspect of the implementation, it was necessary to develop test models that could be evaluated. Consequently, this test suite included the creation of a large suite of test model instances, which at the time of writing included at least 244 model instances. This suite of test models has also been used to evaluate the performance of model completion using the Drools engine, and also to demonstrate the non-trivial nature of model completion [369].

The corresponding OSGi bundles for the implementation of the *Tests* component are `org.openiaml.model.tests` and `org.openiaml.model.tests.diagram`.

### 7.9.1 Model-driven Code Coverage

During the development of the proof-of-concept implementation of IAML, a brief experiment was performed where code coverage techniques were applied throughout the code generation process. Because a source IAML model instance is transformed many times – through model completion, code generation, runtime libraries, and executed on any number of target platforms – the concept of *code coverage* can be extrapolated into the concept of *model-driven code coverage*, which applies common code coverage metrics to each individual step of the model transformation process.

As discussed by Nagappan and Ball [244, pg. 417], *code coverage* is “an important metric by which the extent of testing is often quantified,” based on the premise that errors cannot be detected within software unless the error is tested. Conversely, code coverage may also be used to identify portions of the source code that is never executed, and to an extent this unused source code may be removed safely in order to improve the simplicity and quality of the software itself.

Code coverage metrics need to be aggregated across a number of different runs of the instrumented software, with each run provided a different set of valid input values. The suite of test cases and test models introduced in this section are designed to exhaustively test the complete implementation of IAML, and therefore form an ideal suite of test data for this process. These code coverage metrics may therefore be automatically captured while these tests are evaluated.

For example, the code generation templates in Section 7.6 may be annotated with four different types of coverage through this aggregation, as illustrated in Figure 7.10: templates that are never used; templates that are generated, but never executed; templates that are executed by the web server; and templates that are executed by the client (web browser). This figure shows that the source code generated by this particular template is executed on both the client and the server; parts of the generated code are never executed; and parts of the template are never used to generate any source code.

The code coverage approach has also been used to evaluate the suite of model completion rules used as part of the *Rule Set* component in Section 7.5.2. This is achieved by extending the behaviour of the *insertion queue* component of model completion to also keep track of the source rule that inserted each element. Model completion rules that are never used across any test case may therefore



Figure 7.10: Illustrating code coverage of code generation templates by annotating client-side and server-side functionality

be candidates for removal from this suite; conversely, this evaluation has shown that all of the current model completion rules summarised in Table 7.4 are necessary to implement the intended behaviour of the IAML metamodel.

## 7.10 Conclusion

In this chapter, the proof-of-concept implementation of a modelling environment for the development, code generation and verification of IAML model instances has been discussed. In particular, the implementation has been designed as the combination of a number of independent components, which are supplied as OSGi bundles. This implementation will be used to evaluate both the modelling environment and the IAML modelling language itself, as discussed in the next chapter.





## Chapter 8

# Evaluation

Throughout this thesis the design, description and proof-of-concept implementation of the Rich Internet Application modelling language IAML has been discussed. This chapter will focus on the evaluation of this research and these contributions in terms of the five evaluation criteria discussed earlier in Section 5.1.4. This includes an implementation of the benchmarking application *Ticket 2.0* using IAML in Section 8.3.2, represented using XMI in Appendix F; this implementation is subsequently compared against an equivalent Symfony implementation using source code metrics in Appendix G.

### 8.1 Feature Comparison

As discussed in Section 2.3, one of the first steps of this research involved the identification of features that a RIA modelling language should possess. These features were then used to evaluate the range of existing modelling languages for web applications, as published by Wright and Dietrich [368]. This feature comparison will now be re-evaluated on the current IAML implementation, in order to compare this language with existing web application modelling languages, and illustrated in Table 8.2.

- **Events:** The concept of an event-condition-action rule is a fundamental concept of the IAML metamodel, represented as the associations between [Events](#), [Conditions](#) and [Actions](#) and expressed as an [ECA Rule](#). Arbitrary events cannot be defined by the model developer, as this would require the inclusion of an event modelling language as discussed earlier in Section 5.7. IAML does not currently support any time-based events.
- **Browser Interaction:** IAML does not implement any form of browser interaction, such as navigation, cookies, identifying user agents or opening new windows. However, many of these requirements have been conceptually designed to fit into IAML as discussed in the next section, and this effort remains future work.
- **Lifecycle Management:** Lifecycle support in IAML is supported by the definition of [Events](#) in [Scopes](#) – such as the *onInit* and *onAccess* events of [Sessions](#). There is little support for events at the end of a lifecycle – for example, an [Email](#) provides an *onFailure* event – and future work may include defining events such as *onDelete* or *onTimeout*.
- **Users:** As discussed in Section 5.10, IAML natively supports the definition of role-based access control mechanisms through [Roles](#) providing [Permissions](#). However, there is no modelling support for the interactive collaboration between users.

Modelling Concept	Existing Metamodel	Level of Reuse
Primitive Types	XML Schema [339]	Complete
Domain Types	EMF Ecore [314]	Partial
Operations	UML activity diagram [254]	Adapted
Users and Access Control	RBAC [293]	Adapted

Table 8.1: Reuse of existing metamodels in the IAML metamodel

- **Access Control:** As discussed in Section 5.10, access to [Scopes](#) can be restricted using combinations of [Gates](#), [Access Control Handlers](#) and [Login Handlers](#). Future work includes the development of model verification techniques to formally verify the security of these approaches.
- **Databases:** IAML supports the definition of primitive types using the XML Schema datatype metamodel, and complex domain types using the EMF Ecore metamodel. Both of these metamodels are platform-independent, allowing for domain-specific data to be represented across any type of database. However, there is not yet any support for common database concepts such as views, joins or sub-selects. There is also no support for modelling offline data, or for uploading files.
- **Messaging:** Messaging is supported through RSS feeds and [Emails](#). As discussed earlier in Section 8.2 support for other types of web services has been designed, but these modelling concept have not yet been implemented.
- **UI Modelling:** A limited range of user interface components, such as [Input Text Fields](#) and [Maps](#), are natively supported in the metamodel, as illustrated earlier in Section 5.13. However, there is no way to define new types of user interface components, except through the EMF-based extension of the IAML metamodel. Complex user interface functionality can be simplified into design patterns represented as [Wires](#). Low-level visual properties such as fonts and colours are not part of the IAML metamodel, and generated applications are instead supplied with additional templates, scripts and stylesheets.
- **Platform Independence:** As an important design goal of IAML, the language is designed to be platform-independent with distinct components. Both the metamodel and the model instances are serialised in the XMI format, encouraging integration with other model-driven approaches. The proof-of-concept implementation of the code generation component only supports a single set of target platforms, but future work includes focus on additional platforms<sup>1</sup>.
- **Standards:** Standardised technologies and formats have been used wherever possible in the development of the IAML metamodel and implementation, and a summary of these reused metamodels is provided in Table 8.1. The implementation of each component is provided as OSGi bundles, and reuse existing implementations wherever possible.
- **Use of metamodels:** The IAML metamodel satisfies both the metamodeling architecture and viewpoint architecture of the MDA, as discussed in Section 3.1.5.
- **Verification:** The proof-of-concept implementation of IAML implements model instance verification using four different technologies, as discussed earlier in Section 7.7. Each technology

<sup>1</sup>This future work is discussed in further detail later in Section 9.3.6.

Feature Category	WebML	UWE	W2000	IAML
Events	Some	-	Poor	Good
Browser Interaction	Poor	-	-	-
Lifecycle Management	Poor	Good	Poor	Some
Users	Good	Poor	Poor	Some
Access Control	Some	Some	Poor	Good
Database Support	Good	Some	Poor	Poor
Messaging	Good	Poor	Some	Some
UI Modelling	Poor	Some	Some	Some
Platform Independence	Excellent	Excellent	Good	Excellent
Standards Support	Poor	Excellent	Excellent	Good
Use of Metamodels	Poor	Excellent	Excellent	Excellent
Verification	Some	Some	-	Excellent
Software Support	Good	Some	Poor	Good

Table 8.2: A comparison of IAML against existing modelling language support for the general feature categories of modelling Rich Internet Applications, adapted from Wright and Dietrich [368]

uses a different verification language with different performance characteristics and resource requirements, allowing a model instance developer to selectively evaluate certain classes of constraints as necessary throughout the development lifecycle.

- **Software Support:** The IAML metamodel has been implemented in Chapter 7 using a wide range of technologies into a proof-of-concept implementation, and this environment can be used to develop web applications. This implementation is provided as free software under an open-source license.

These evaluation results are reflected in Table 8.2, adapted from Table 2.3 earlier in Section 2.4.7. It is important to note that the other languages have not been re-evaluated since it was first published; for example, WebML has recently proposed additional support for more client-side interactivity and user interface components [40, 359].

The results in this comparison table suggest that the IAML metamodel and its consequent proof-of-concept implementation satisfy more categories of the features necessary to model Rich Internet Applications than any other existing language evaluated earlier in this thesis. This comparison table also illustrates that there are a number of features, such as database support and standards support, in which IAML support is less than excellent, and these areas should be the focus of future research.

## 8.2 Modelling Requirements

To improve on the accuracy of the previous evaluation, each of the detailed requirements of RIAs used in the design of the metamodel [367] can be individually evaluated against the current implementation of IAML. These requirements represent the 46 requirements of *Basic RIAs*, and the results of evaluating these requirements against the proof-of-concept implementation of IAML is illustrated in Tables 8.3 and 8.4.

#	Requirement	D	M	I	V	IAML Support
<b>Data</b>						
D1	Static Pages	✓	✓	✓	✓	Frame
D2	View Data	✓	✓	✓	✓	Domain Iterator
D3	Update Data	✓	✓	✓	✓	Domain Iterator
D4	Pagination	✓	✓	✓		Domain Iterator
D5	Provide Data Feed	✓	✓	✓	✓	Frame (RSS only)
D6	Use Web Services	✓	✓	✓		Frame (RSS only)
D9	Web Service Provider	✓				Frame
D10	Uploading Files	✓				Datatypes
D11	Access Server Data	✓	✓	✓	✓	Value, Domain Attribute Instance
D12	Local Variables/Data	✓	✓	✓	✓	Value, Temporary Variable
D13	Cookies	✓				A Value within a new Scope
<b>Events</b>						
E1	Scheduled Events	✓				A timed Event
E2	Client Timer Support	✓				A timed Event
E3	Server Timer Support	✓				A timed Event
E4	Async Form Validation	✓	✓	✓	✓	Datatype validation
E5	Client Form Validation	✓	✓	✓	✓	Datatype validation
E6	Server Form Validation	✓	✓	✓	✓	Datatype validation
E8	Browser-Based Chat	✓				Implemented manually
<b>Users and Access Control</b>						
S1	User Authorisation	✓	✓	✓	✓	Access Control Handler
S2	Session Support	✓	✓	✓	✓	Session
S3	User Logout	✓	✓	✓	✓	Login Handler
S4	Automatic User Auth	✓				Login Handler extension (see D13)
S5	User Access Control	✓	✓	✓		Permission
S6	Group Access Control	✓	✓	✓	✓	Role
S7	Access Control Levels	✓	✓	✓	✓	Role-based Access Control
S8	Single Sign-In Solutions	✓	✓	✓		iam10penIDURL type through Gate
S9	Personalisation	✓	✓	✓	✓	Domain Attribute
<b>User Agents</b>						
A1	Browser Identification	✓				Values within a new “browser” Scope
A2	User Redirection	✓	✓	✓	✓	ECA Rule to a Frame
A3	Multiple Browser Support	✓	✓			Additional code generation templates
A4	Multiple Outputs	✓	✓	✓	✓	Frame: HTML, RSS
A5	Client-Side Application	✓	✓	✓		Visible Thing
A7	Back Button Control	✓				Event
A10	Navigation Control	✓				Hash Fragment (like Query Parameter)

Table 8.3: Evaluation of the modelling requirements of *Basic RIAs* against IAML (1), adapted from Wright and Dietrich [367]

#	Requirement	D	M	I	V	IAML Support
<b>Interaction</b>						
T1	E-Mailing Users	✓	✓	✓	✓	Email
T2	E-Mail Unsubscription	✓				Email with library support
T3	Mobile Phone Comm.	✓				Similar to Email
<b>User Interface</b>						
U2	Client-Side Scripting	✓				New client-side Events
U3	Drag And Drop	✓				Event or Wire
U4	Loading Time Support	✓				Event
U5	Keyboard Shortcuts	✓				Event
U6	Opening New Windows	✓				ECA Rule attribute
U7	Pop-Up Dialog Boxes	✓	✓	✓		alert Operation
U8	Runtime Interface Updates	✓	✓	✓	✓	Only after callbacks
U10	Modal Dialogs	✓				Visible Thing attribute
U12	Provide External Libraries	✓	✓	✓	✓	IAML runtime library

Table 8.4: Evaluation of the modelling requirements of *Basic RIAs* against IAML (2), adapted from Wright and Dietrich [367]

Key	Requirement	Count
<b>D</b>	Designed: Introduced in Chapter 4	46 (100%)
<b>M</b>	Modelled: Implemented in the EMF metamodel	27 (59%)
<b>I</b>	Implemented: Tested with test cases and examples	26 (57%)
<b>V</b>	Validated: Successfully implemented in the Ticket 2.0 application	20 (43%)

Table 8.5: Legend to the modelling requirements evaluation of IAML

The implementation of a particular requirement is decomposed into four activities, as illustrated in the legend of Table 8.5: the design of the requirement (**D**); the implementation of the requirement in the IAML metamodel (**M**); the implementation of the requirement with code generation templates, visual syntax and test cases (**I**); and the validation of the requirement using the *Ticket 2.0* benchmarking application (**V**). These four activities represent the steps necessary in the hybrid modelling language software process model, discussed earlier in Section 5.1.3; that is, a requirement must be designed before it can be implemented, and implemented before it can be evaluated.

This detailed evaluation illustrates that all of the detailed modelling requirements of *Basic RIAs* have been considered in the design of the IAML metamodel, but due to time constraints they have not all been fully implemented and validated. Their implementation remains future work, and it is expected that the implementation of the remainder of these modelling requirements would improve each element in the feature comparison evaluation of IAML in Table 8.2 to “excellent”.

### 8.3 Benchmarking Application Implementation

As discussed in Section 2.5, the benchmarking application *Ticket 2.0* was developed in order to evaluate the functionality of a RIA modelling language within a single application. This application combines all of the detailed feature requirements of a *Full RIA* into a single application specification [367]; and can be used to benchmark different programming and web modelling languages, by comparing different implementations of the same application.

Such a comparison forms one of the evaluation criteria of the IAML proof-of-concept implementation against a web application implemented manually – that is, one developed with conventional web programming languages such as PHP and Javascript, without using any model-driven technologies. The system metrics discussed earlier in Section 2.6.3 will be evaluated against each application to compare them; this is necessary as Symfony applications are implemented in a variety of different scripting and configuration languages, whereas IAML applications are implemented as XMI model instances with additional templates.

The development metrics will also be split based on generated code, library code, and manually-written code in order to highlight the manual effort necessary for each application, and illustrate how much functionality is generated or provided by the framework. In Appendix G, these metrics are re-evaluated against each of the different languages used in each implementation.

### 8.3.1 Implementation in Symfony

To develop the manual implementation of Ticket 2.0, the Symfony framework [274] was selected as a basis of the approach, as this framework provides a clean approach to web application design through the use of modules, plugins, actions and views. Using a framework to implement the application is appropriate, as complex web applications are rarely implemented from scratch.

The Symfony framework is implemented in the PHP general-purpose language, and provides web applications using HTML, Javascript and CSS. Symfony uses the YAML language to simplify the configuration of framework functionality, as YAML provides a simple human-readable text-based representation of tree-based data [27]. Symfony also utilises a significant number of other open-source frameworks, such as Propel or Doctrine for database access [274]; PHPUnit for unit testing [372]; and Prototype or jQuery for a client-side Javascript framework [280, 208].

The implementation of Ticket 2.0 in Symfony 1.0.12 as *Ticketsf* was fairly straightforward, and released under an open-source license online<sup>2</sup>. All of the requirements of Ticket 2.0 [367] were successfully implemented in this application. A number of issues were discovered during the implementation of the language, and these issues were used as inspiration for portions of the IAML design, and also considered in the reimplementations of *Ticketsf-mini* later in Section 8.3.3. These issues included:

1. Particular components, such as login over SSL, PDF creation and internationalisation (“i18n”), were supported natively or as *Symfony plugins*. This meant that much of the development effort focused on integrating plugins, rather than implementing components from scratch.
2. Some application requirements such as the Flash-based MP3 player were not available as Symfony plugins, but the development of new plugins to satisfy these goals was fairly straightforward.
3. When designing the ticketing client-side application, an identical server-side application had to be implemented simultaneously. Consequently, some portions of the business logic had to be reimplemented in both PHP and Javascript, because there was no easy way to translate between the two. Web application testing frameworks such as PHPUnit [372] and JWebUnit [152] may be used to verify the consistency of these logics, but the effort necessary to keep these two implementations of logic identical directly impacts on the maintainability of the application.

---

<sup>2</sup>The source code of *Ticketsf* is available online under the Eclipse Public License [82] through Sourceforge at <http://sourceforge.net/projects/iaml>.

4. Each of the different secured parts of the application – for example, the public site, the user site, and the administrator site – are each implemented as a separate “application” according to Symfony. However, each application is intended to work independently of the other, making it difficult to integrate these sites together.
5. Users defined through the Symfony framework do not natively support complex authentication mechanisms such as OpenID. This functionality had to be integrated manually by integrating the third-party *sfOpenIDAuth* plugin.

### 8.3.2 Implementation in IAML

The implementation of Ticket 2.0 in IAML as *Ticketiaml* was performed once the IAML metamodel had been completed as discussed in this thesis. The implementation of *Ticketiaml* is similarly released under an open-source license, with the source code available online<sup>3</sup>. The IAML model instance for Ticket 2.0 is also provided using an XMI representation in Appendix F.

In this section, the design and implementation of each implemented Ticket 2.0 requirement will be discussed, and illustrated with a visual representation of that part of the IAML model instance. Each figure represents all of the children contained by a single element, and with the exception of the root [Internet Application](#), this container element is contained by another element within this model instance. The path from the root element to the currently displayed element is displayed as a *breadcrumb*<sup>4</sup> within the title of each figure.

#### Root Internet Application

The root [Internet Application](#) of *Ticketiaml* defines the [EXSD Data Types](#) used in the application; the definitions of the domain-specific [Roles](#) and [Domain Types](#) of the application; and the [Sessions](#) and [Frames](#) used in the application. The “Home” [Frame](#) is defined as the homepage of the application itself.

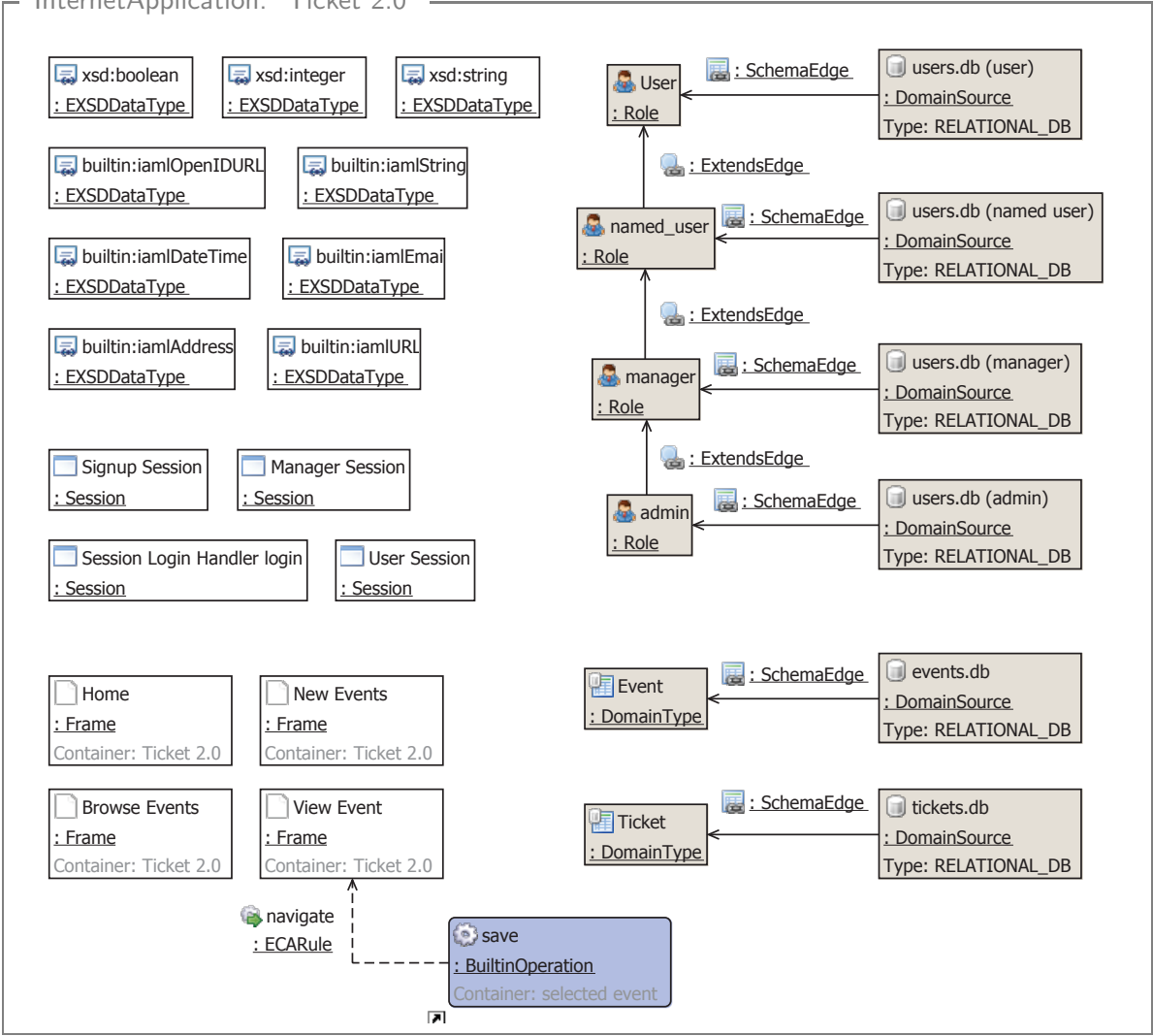
---

<sup>3</sup>The source code of *Ticketiaml* is hosted within the IAML project, and is available under the Eclipse Public License [82] at <http://openiaml.org/ticket20/>.

<sup>4</sup>As discussed earlier in Section 7.4.4, breadcrumbing is adapted within this approach to highlight the navigation path from the root model element to the currently displayed element, via the implicit *container* references of each model element.

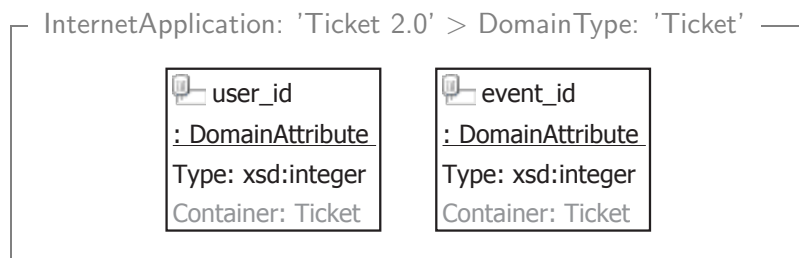
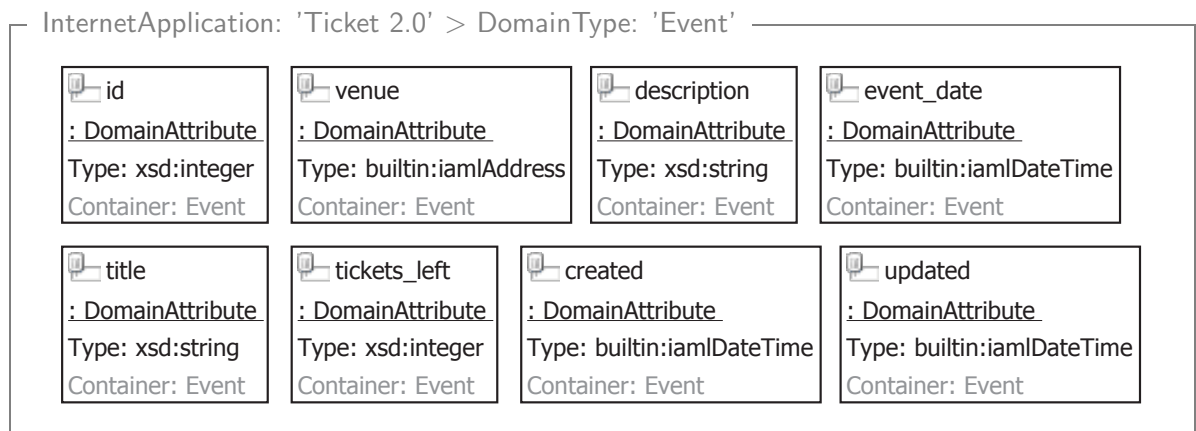
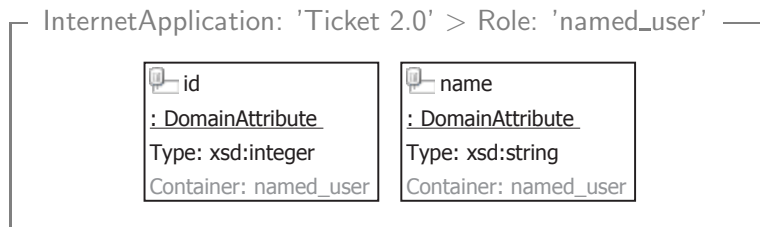


InternetApplication: 'Ticket 2.0'



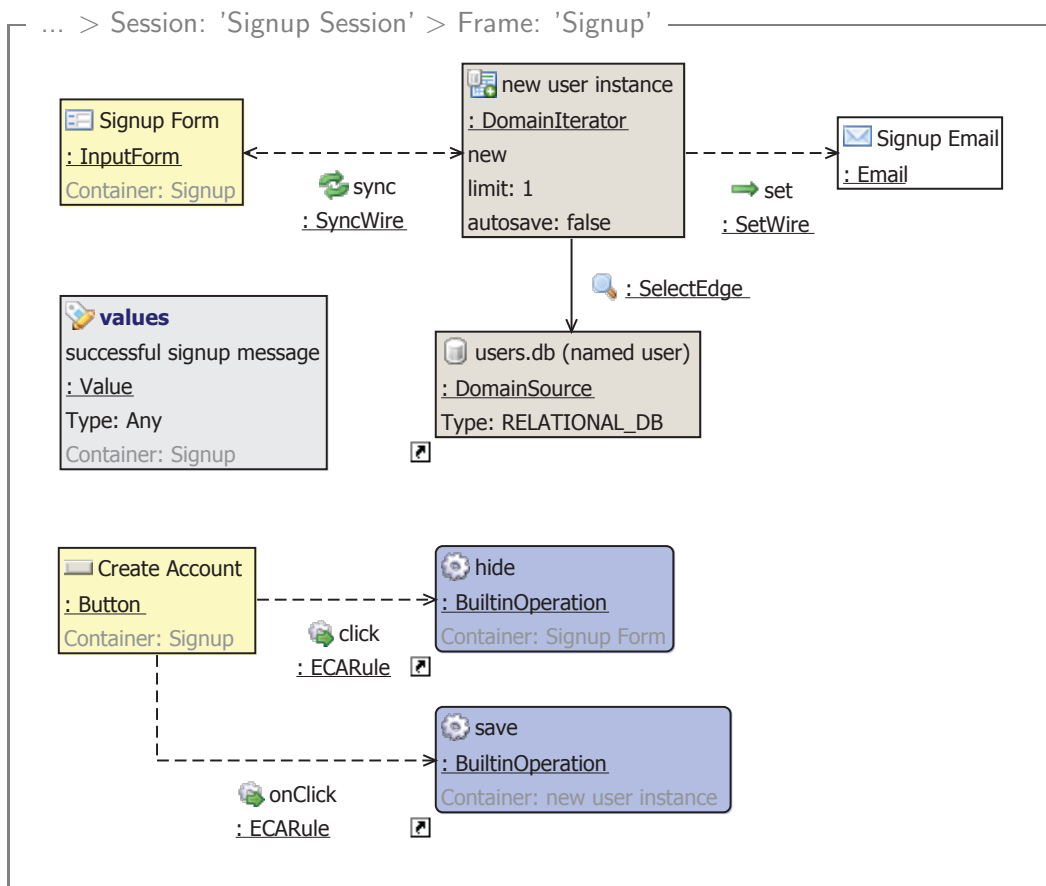
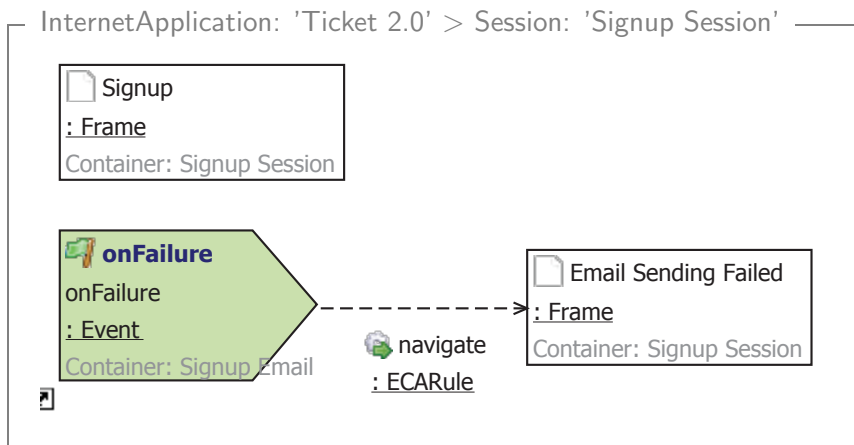
## Types

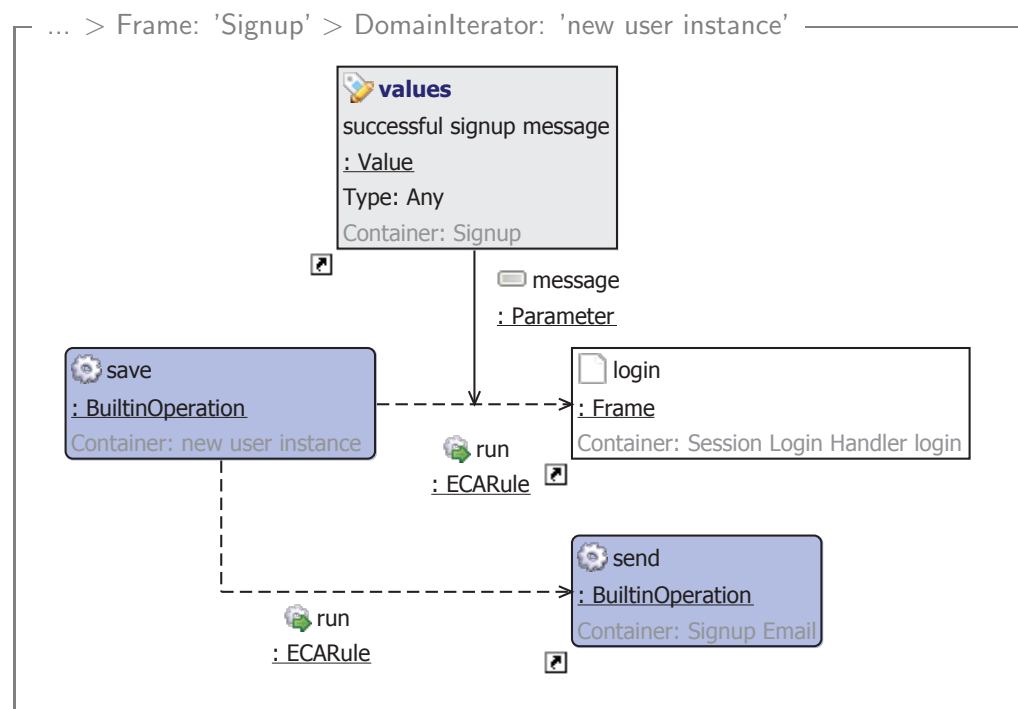
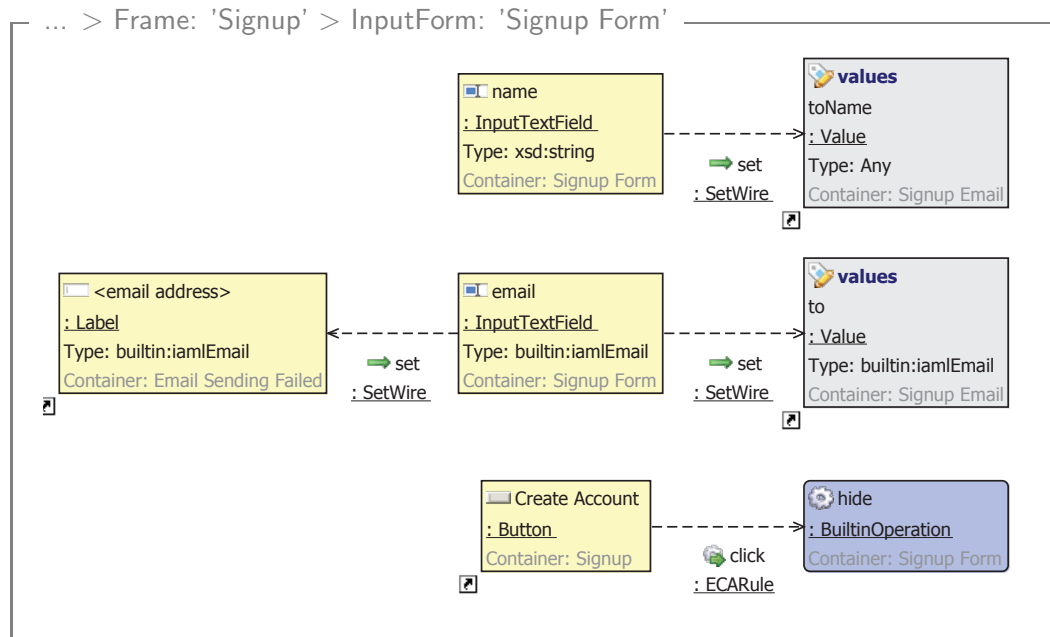
Ticketiaml defines four [Roles](#), defined within a role hierarchy; however, only the *named\_user* Role specifies additional [Domain Attributes](#), so the contents of these other three Roles are initially empty. Through model completion, the *User* Role is automatically populated with `email` and `password` [Domain Attributes](#), as discussed in Section I.84.

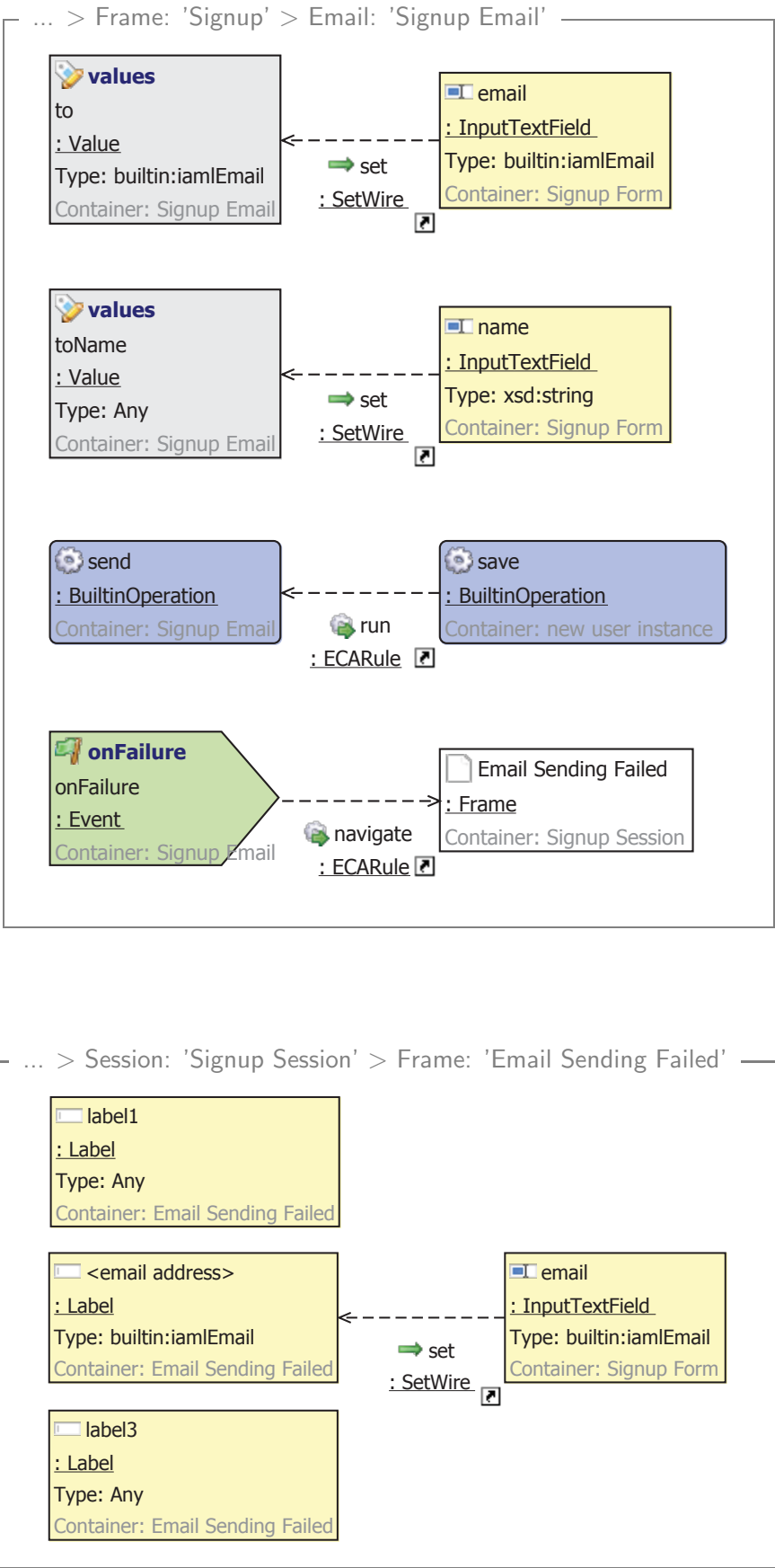


## Signup Session

The *signup session* is a [Session](#) that allows a user to create a new user account. If the signup process was not wrapped within a [Session](#), then other users could hijack the user data entered in by another user. The signup process also specifies that when a new user account is successfully created (through the [Builtin Operation](#) named [save](#)), a new [Email](#) will also be sent to the new user.



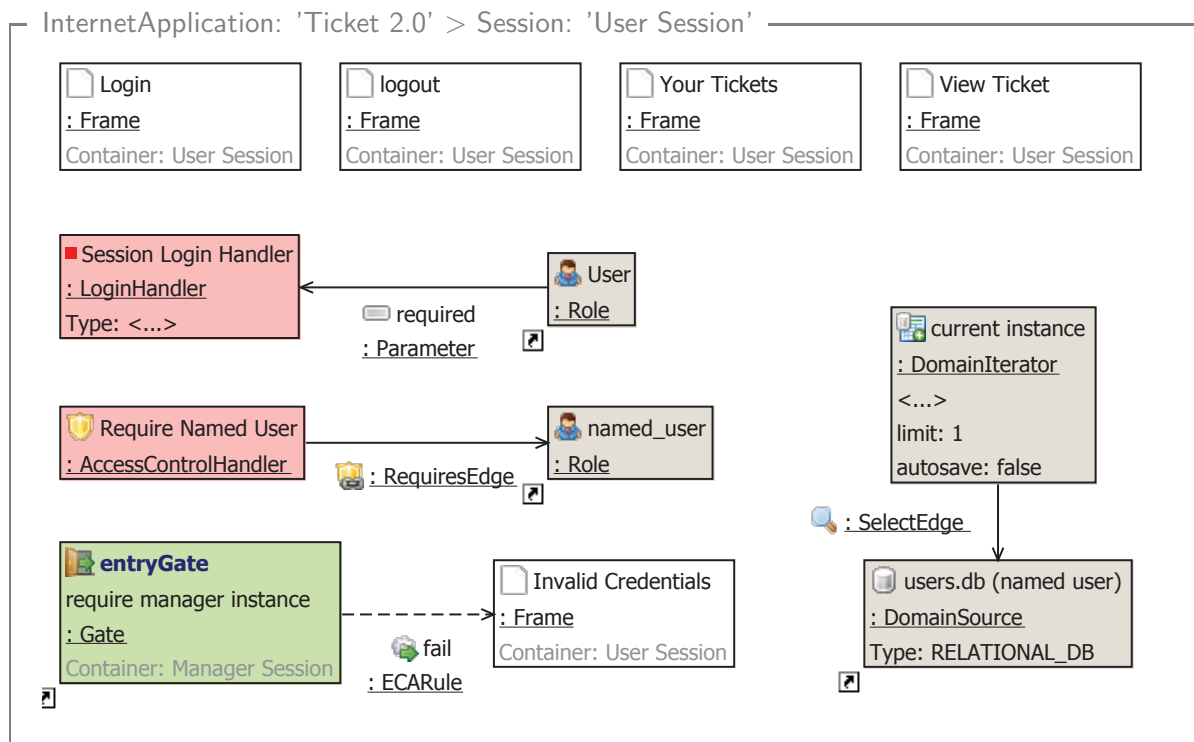


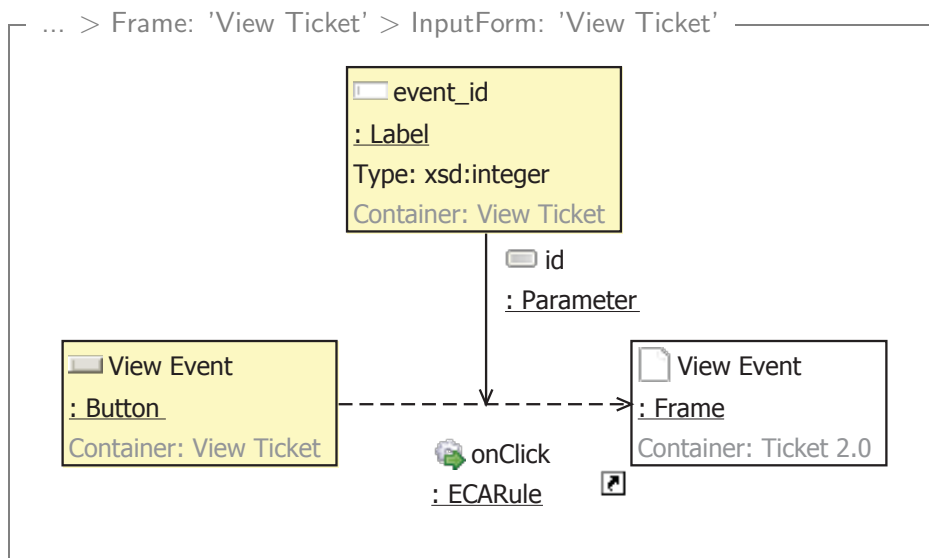
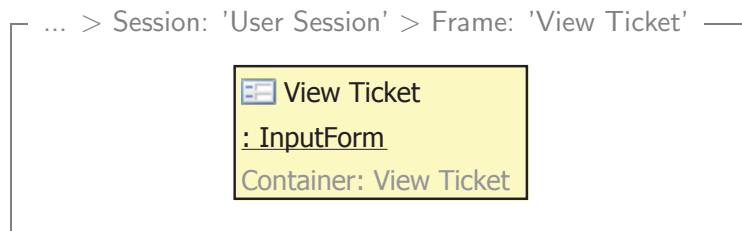
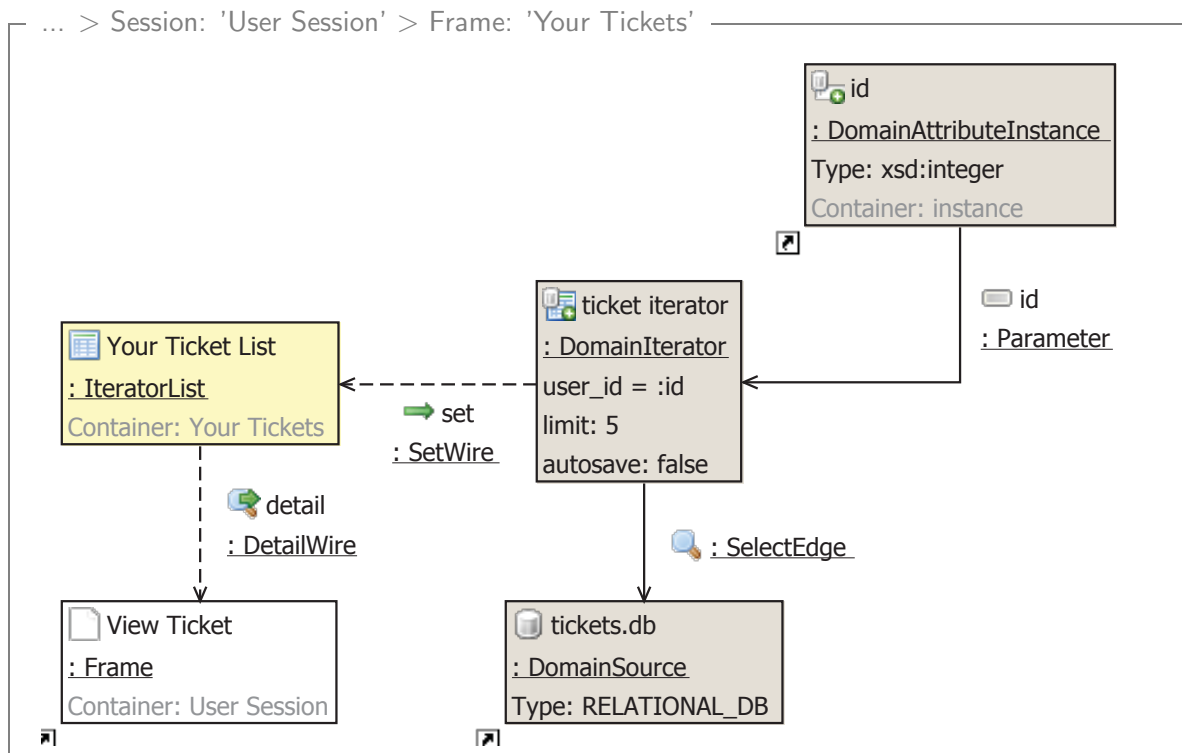


## User Session

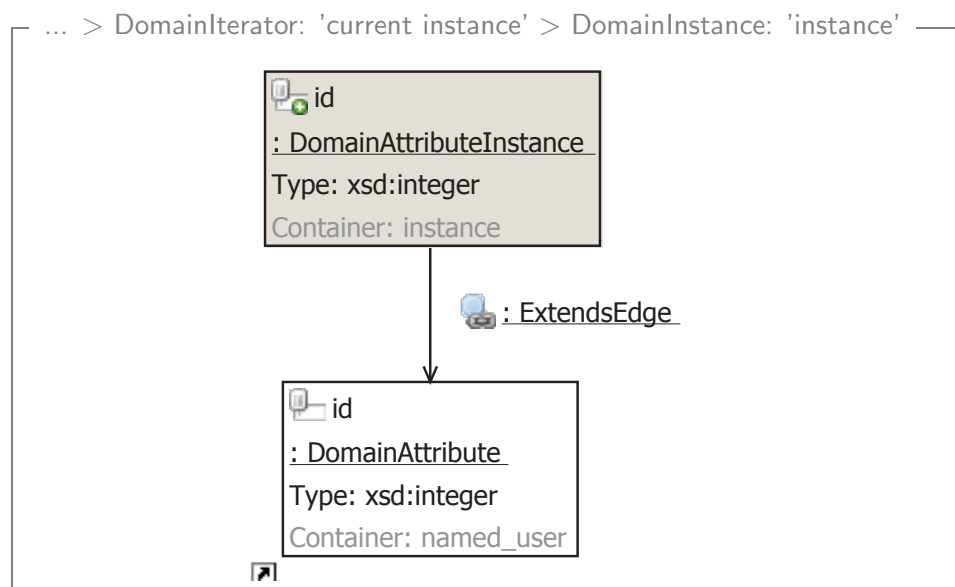
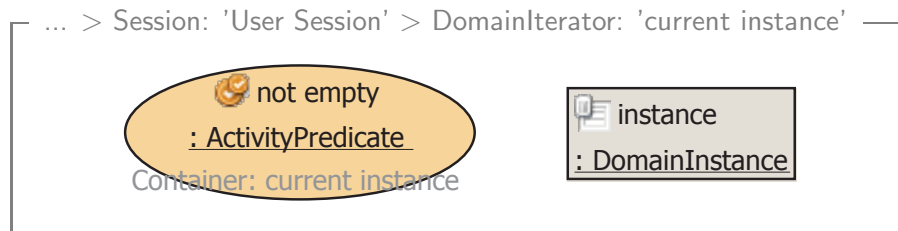
The *user session* is a [Session](#) that allows a registered user on the site to login, logout, and interact with their user-specific details. Presently this only includes the *Your Tickets* and *View Ticket Frames*, allowing the user to view details of purchased tickets. This [Session](#) also includes a [Domain Iterator](#) named “current instance”, allowing the application to access the user-specific profile details of the currently logged-in user through [Domain Attribute Instances](#) (as completed through model completion).

This session is protected by both a [Login Handler](#) and an [Access Control Handler](#); the [Login Handler](#) ensures that a *User* has logged on, and the [Access Control Handler](#) ensures that the *User* also possesses the *named\_user Role*.

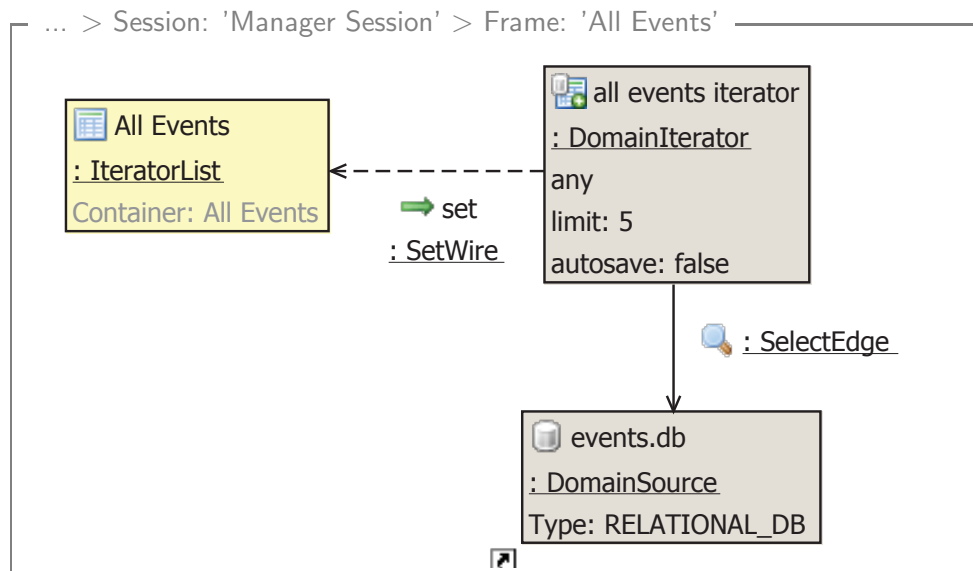
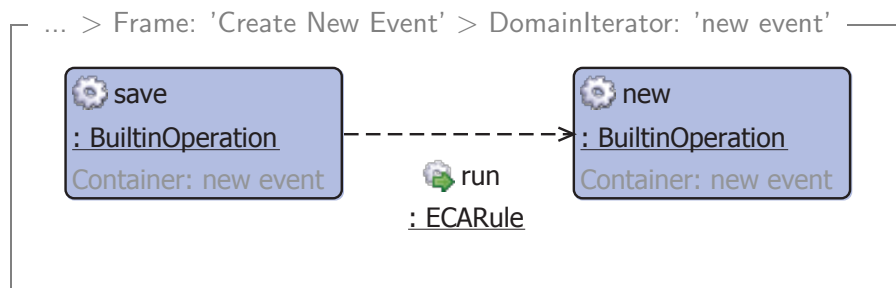
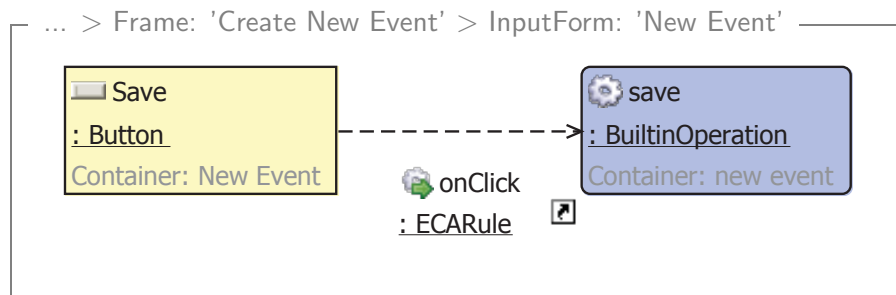
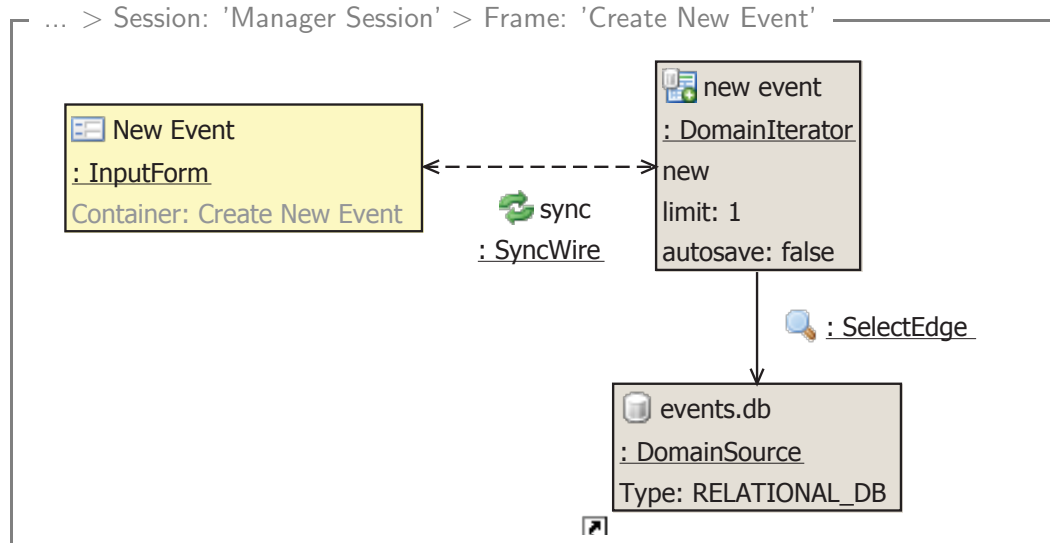


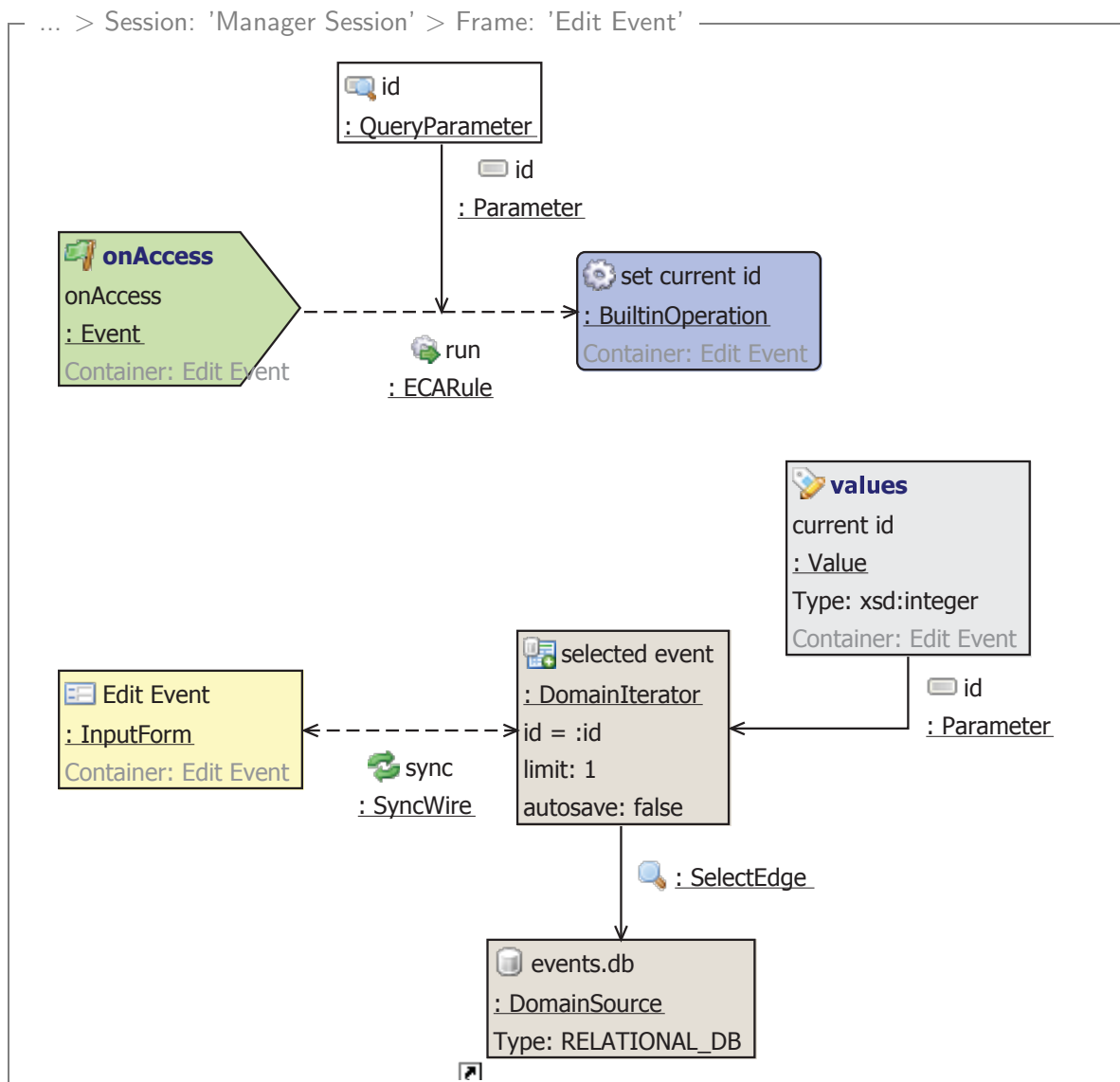
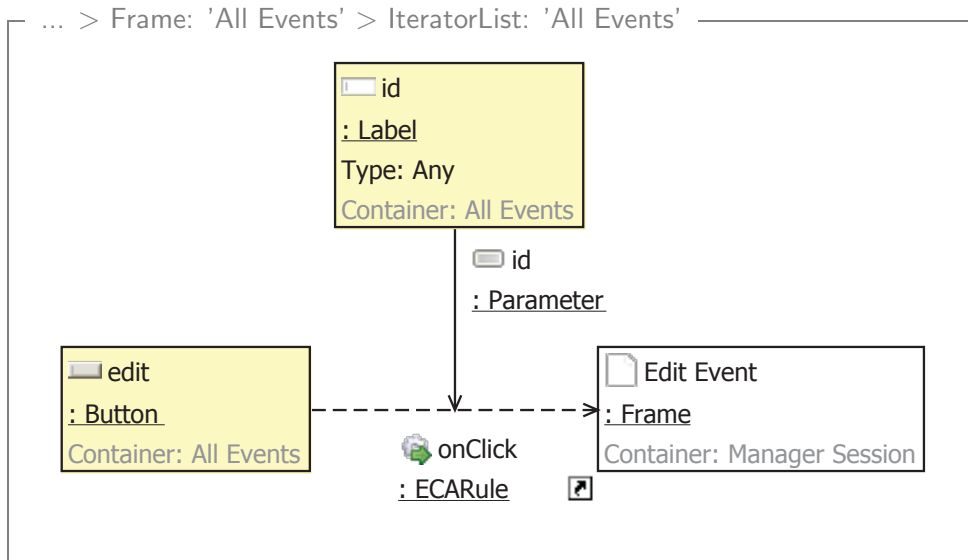


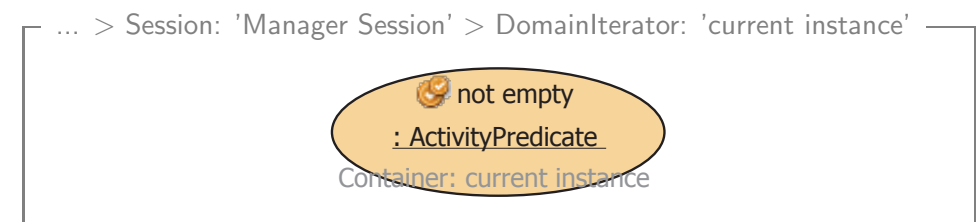
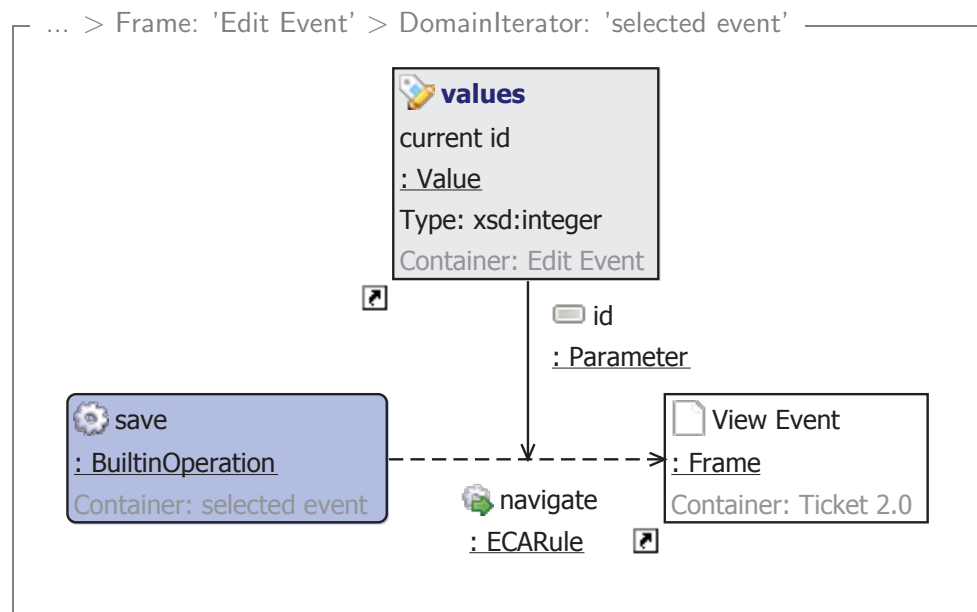
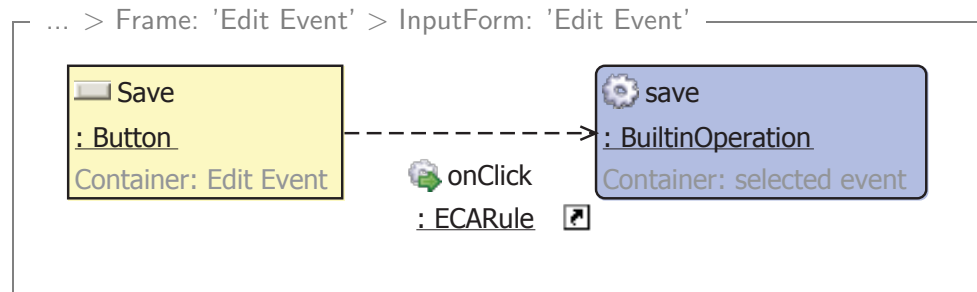








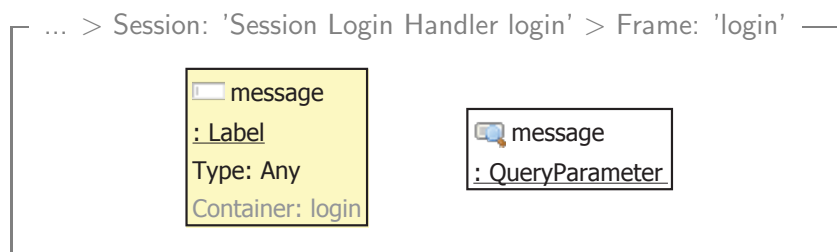
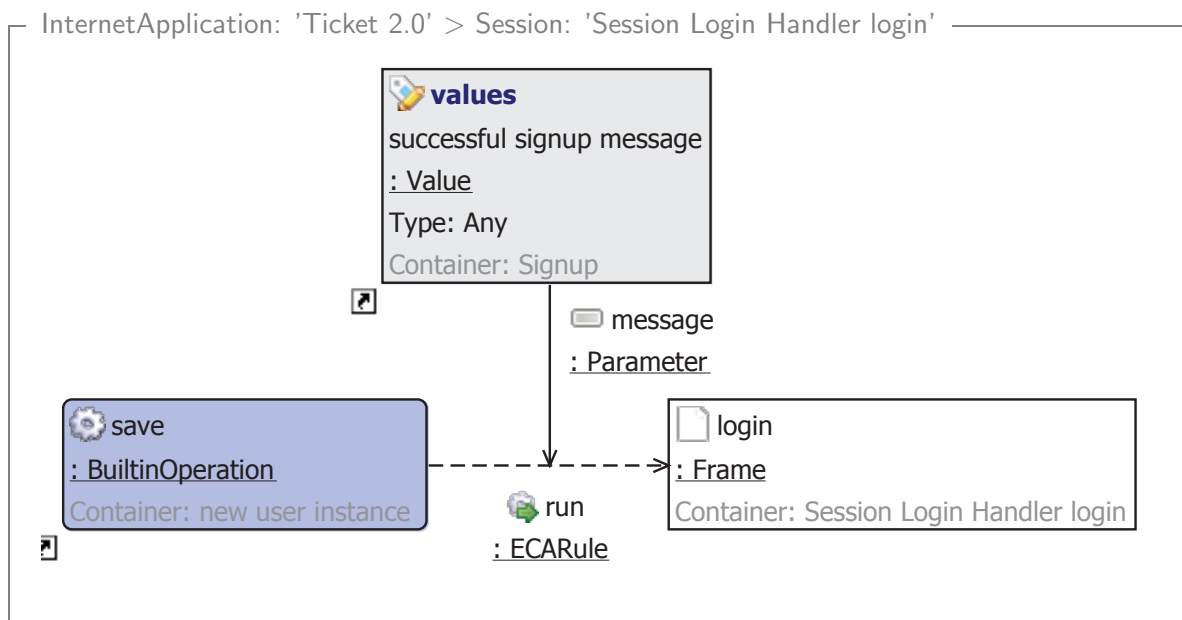


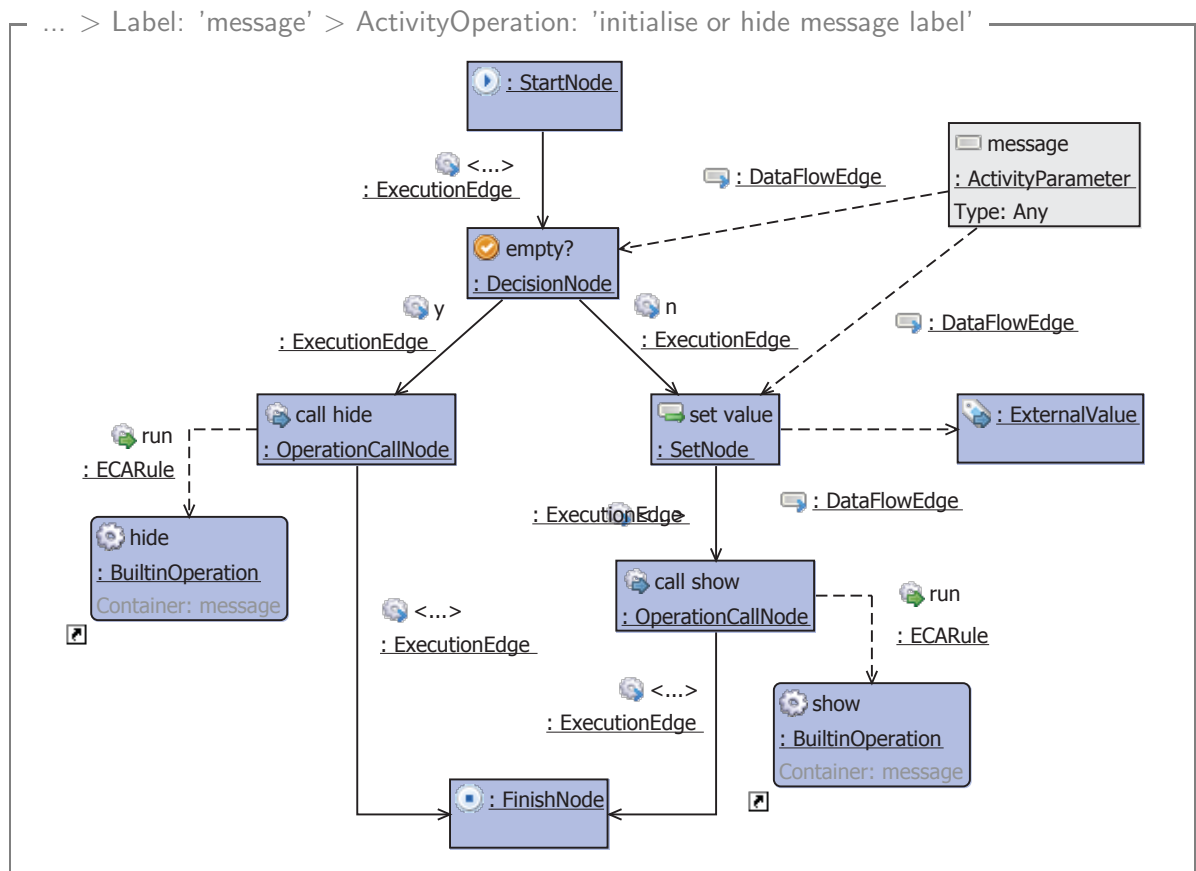
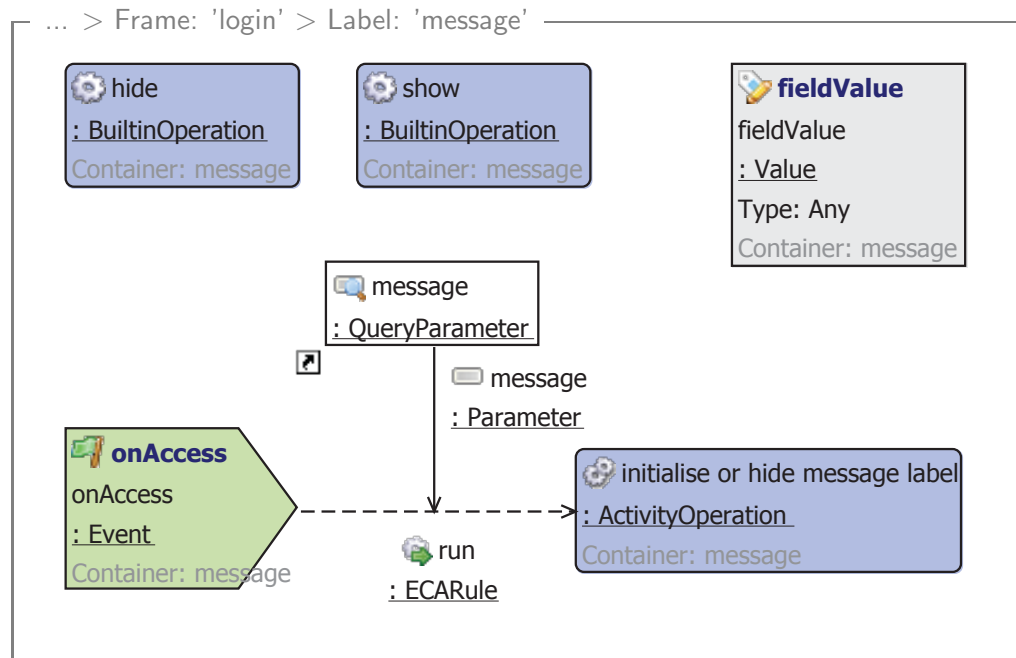


## Login Handler Session

Finally, the *login handler session* [Session](#) deals with the login and logout of *Users* within the web application. By default, this session is automatically generated through model completion, as discussed in Section I.64.

Ticketiaml extends these default conventions to add an additional [Label message](#), which allows a message to be displayed on the login [Frame](#). This message is provided as a [Parameter](#) as in the *new user instance* [Domain Iterator](#); the message is subsequently obtained through a [Query Parameter](#) within the *login* [Frame](#). The label itself is initialised or hidden through the [Activity Operation](#) named *initialise or hide message label*.

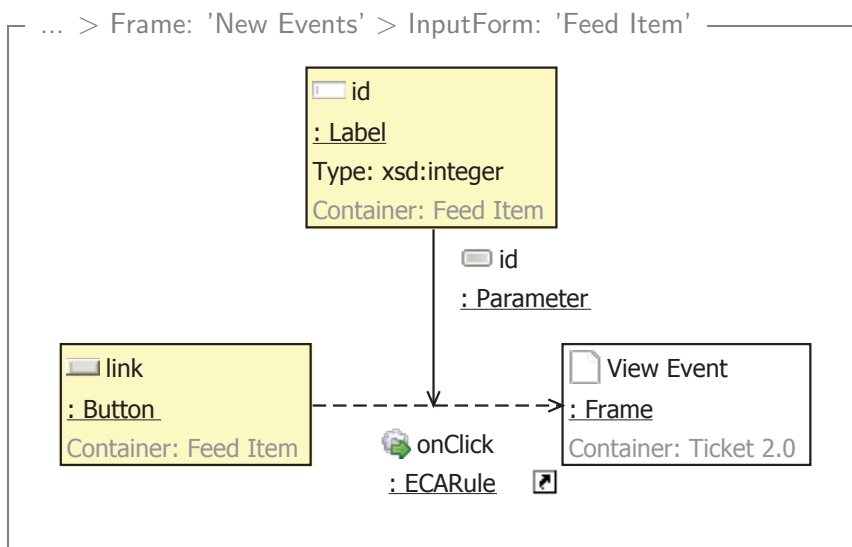
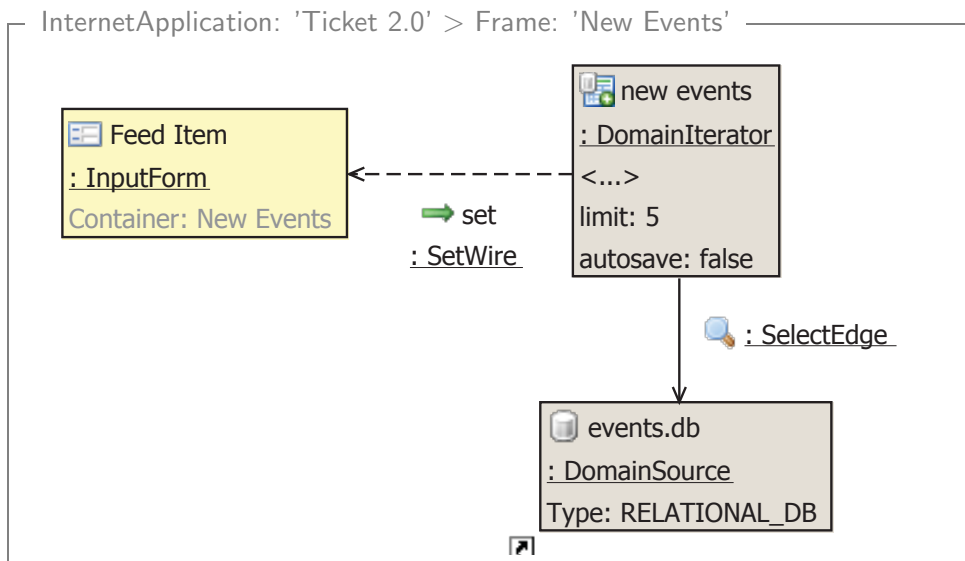


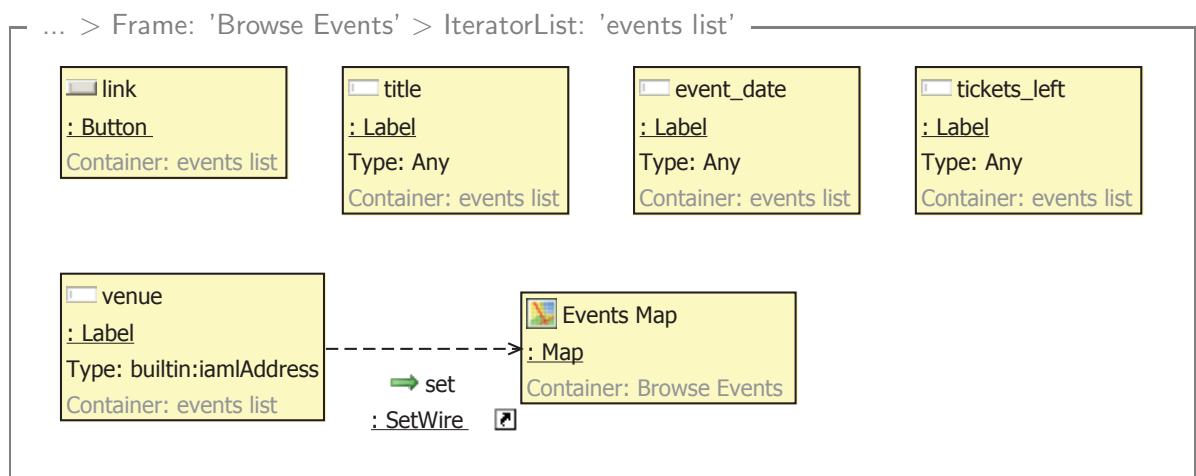
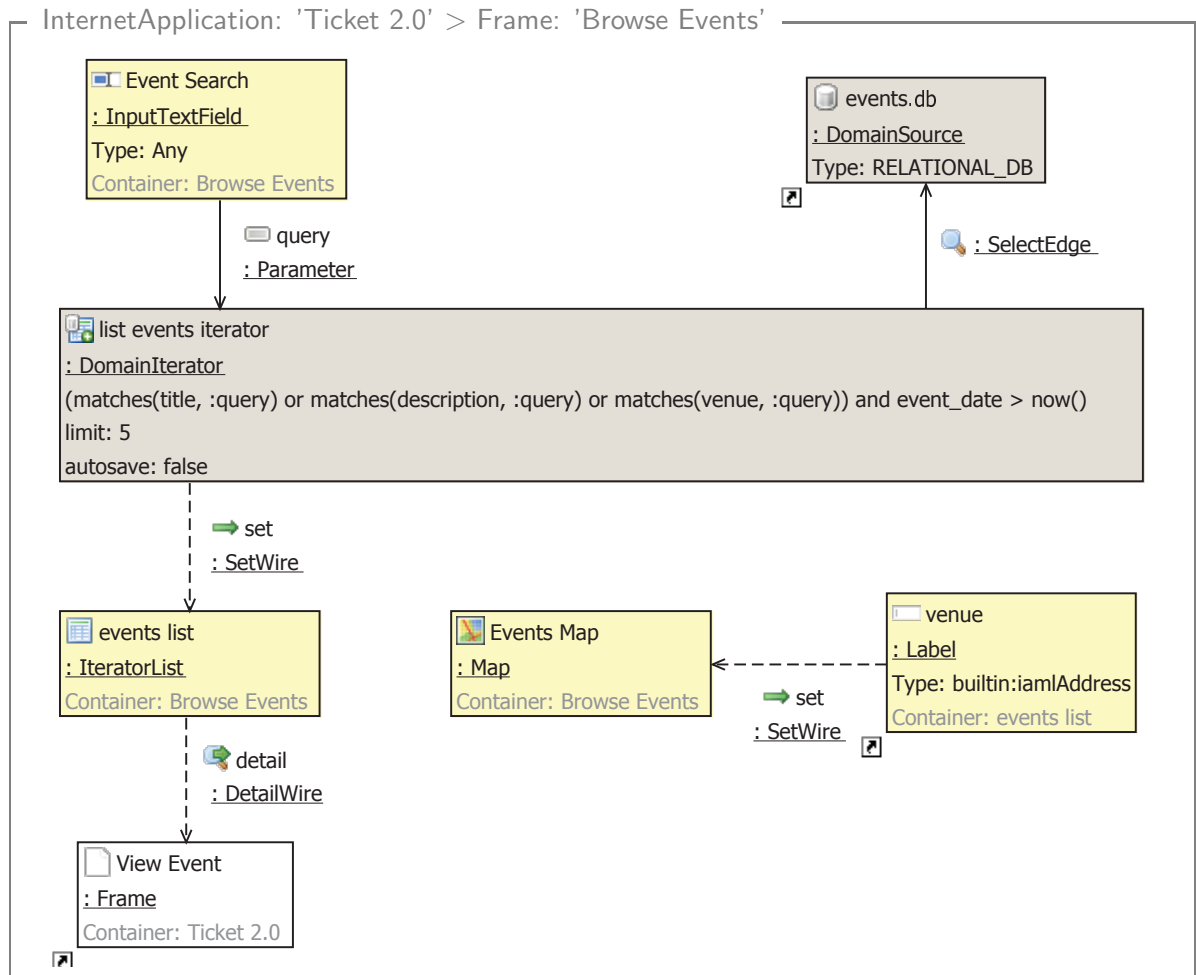


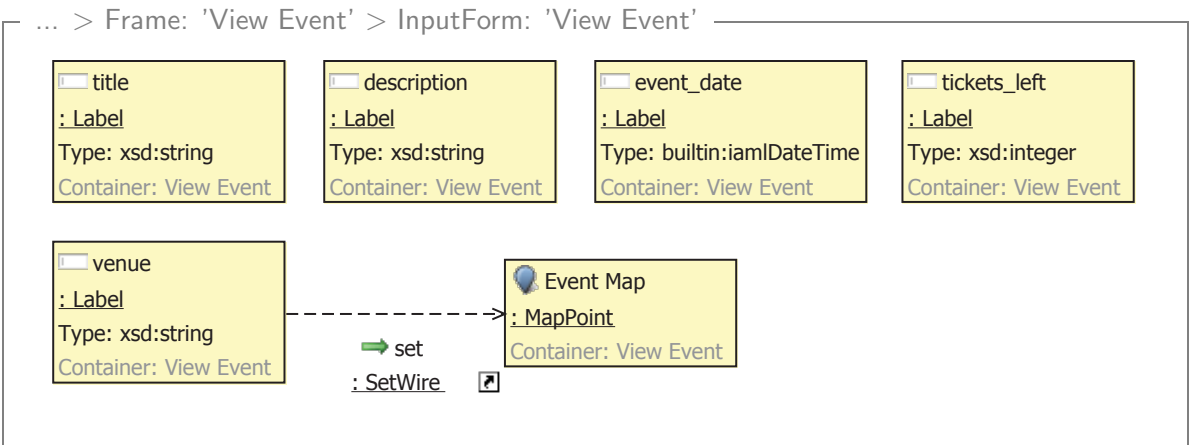
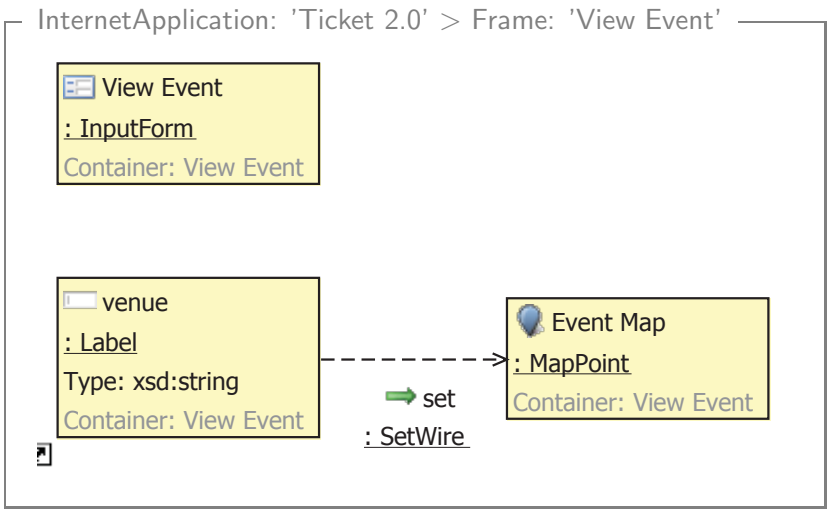


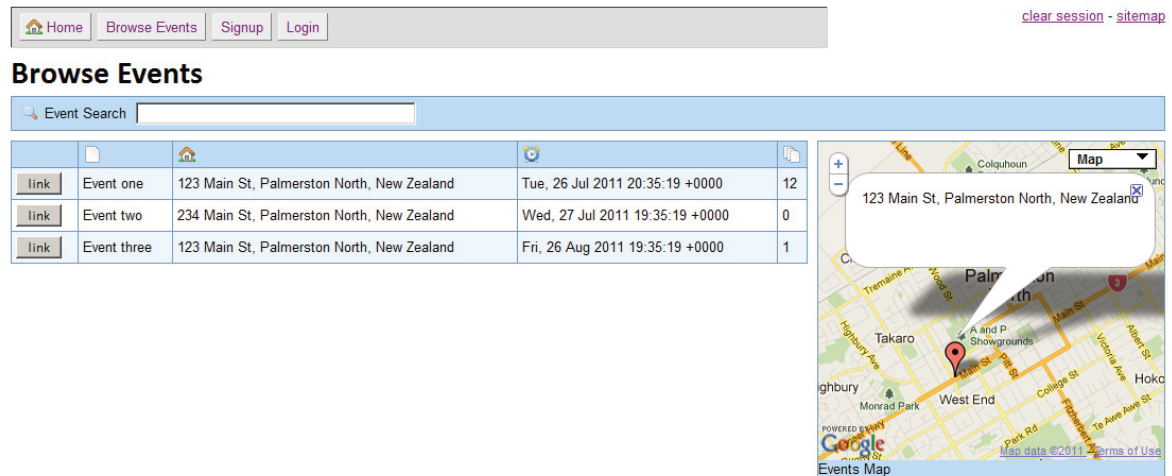
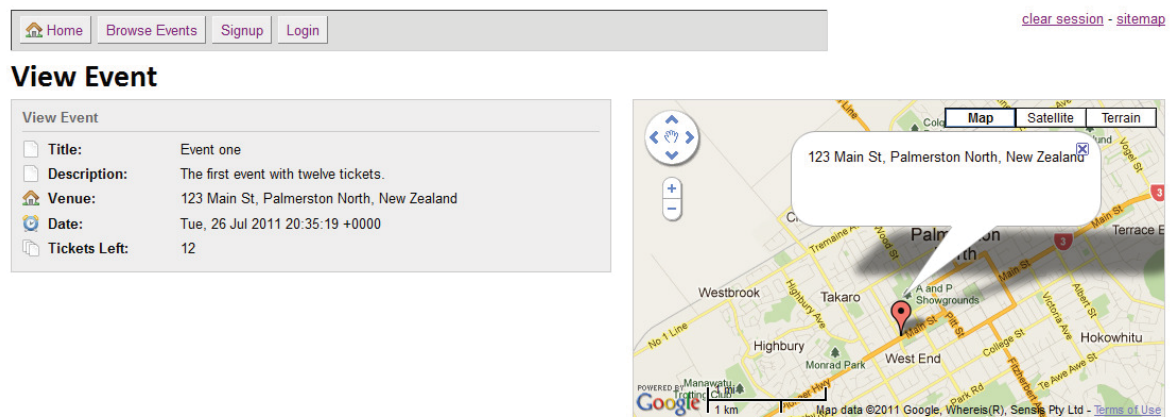
## Root Frames

There are a number of public **Frames** that should be accessible to all users, regardless of whether they are presently logged in. These **Frames** are placed directly in the root **Internet Application**, and include the *New Events* RSS feed and *Browse Events* **Frames**.







Figure 8.1: A screenshot of the *Browse Events* page implemented in TicketiamlFigure 8.2: A screenshot of the *View Event* page implemented in Ticketiaml

## Discussion

A screenshot of the Ticketiaml implementation of the page *Browse Events* is illustrated in Figure 8.1. On this page, a list of all events in the system are displayed, along with an interactive map showing their locations. If one of these listed events is selected, the *View Event* page in Figure 8.2 is displayed, which displays more information about the particular event along with another map.

As the design of IAML includes support for common reusable patterns of web applications – such as access control, data access, mashups and visual element design – the implementation of these features in Ticketiaml were straightforward. However, other concepts that are not yet implemented as discussed in Section 8.2 – such as populating foreign keys, or overriding or extending the model completion process – were difficult, and their implementation required more effort.

Model completion was used heavily in the implementation of Ticketiaml, such as automatically completing the contents of *Input Forms* through *Sync Wires*, and utilising the generated functionality of *Login Handlers* to protect *Scopes*. However, in two situations the intent of model completion rules was not clear, and the developer had to read the source code of the rules themselves.

Unfortunately, the full requirements of the Ticket 2.0 application could not be completed using the current IAML implementation, and only 22 of the original 123 requirements of the application could be implemented. A number of system resource limits were hit during the implementation of the model instance<sup>5</sup>, and no resources were available to optimise the implementation of the language further.

The underlying technologies for model completion and code generation routinely deal with model instances larger than those generated through IAML [211], and there are no known theoretical or design limits that would prevent the full implementation of Ticket 2.0 using IAML. This suggests that these limits were not a result of using any particular technology, but may instead be a configuration error in their current integration, or a bug introduced during the development of the proof-of-concept implementation.

Additional development on the IAML implementation is therefore necessary, and performance has already been a development focus<sup>6</sup>. It is important to note that performance is not a focus of this research, however there are a number of approaches that may be used to resolve these problems, as discussed later in Section 9.3.7.

### 8.3.3 Re-implementation in Symfony

As it would not be possible to evaluate the Symfony-based implementation of Ticket 2.0 with the incomplete IAML-based implementation in any meaningful way, the decision was made to reimplement Ticket 2.0 in Symfony to match the implemented features of *Ticketiaml*. This smaller implementation of Ticket 2.0 was implemented in Symfony 1.4.8 as *Ticketsf-mini*, and exactly matches the features of *Ticketiaml*. The implementation of *Ticketsf-mini* is similarly released under an open-source license, with the source code available online<sup>7</sup>.

This reimplement was fairly straightforward and once again, highlighted the same issues discovered during the implementation of *Ticketsf*; namely, client-side and server-side logic must be implemented in two different languages (PHP and Javascript) and kept consistent throughout the development process. Screenshots of the *Ticketsf-mini* implementation of the pages *Browse Events* and *View Event* are illustrated here in Figures 8.3 and 8.4.

---

<sup>5</sup>For example, the architecture-specific maximum heap spaces for the Sun Java virtual machine were repeatedly reached when performing model completion and code generation [181].

<sup>6</sup>For example, issues 184, 210 and 261 have focused on the performance of model completion, and issues 117, 169 and 181 have focused on the performance of code generation.

<sup>7</sup>The source code of *Ticketsf-mini* is hosted within the IAML project, and is available online under the Eclipse Public License [82] at <http://openiaml.org/ticket20/>.

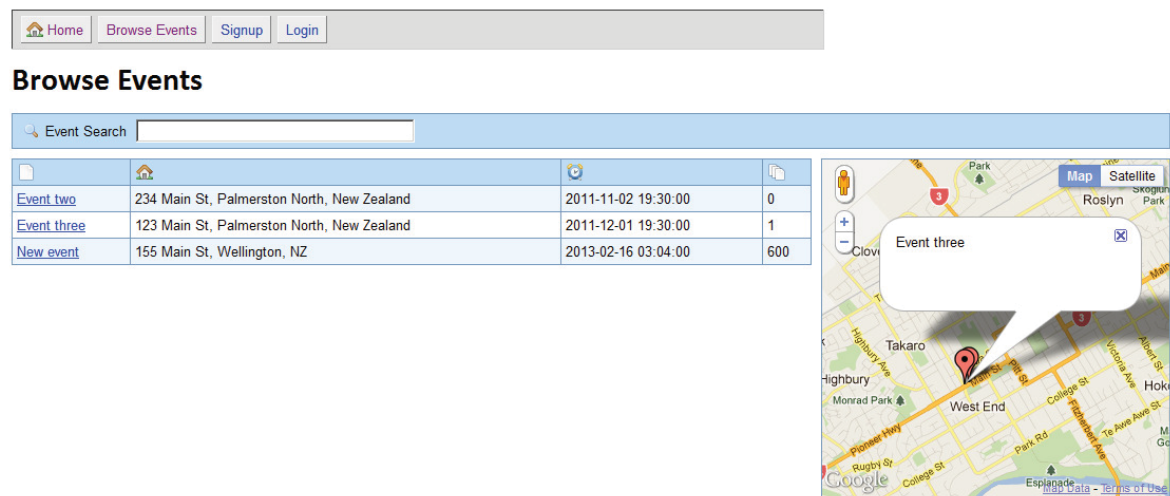


Figure 8.3: A screenshot of the *Browse Events* page implemented in Ticketsf-mini

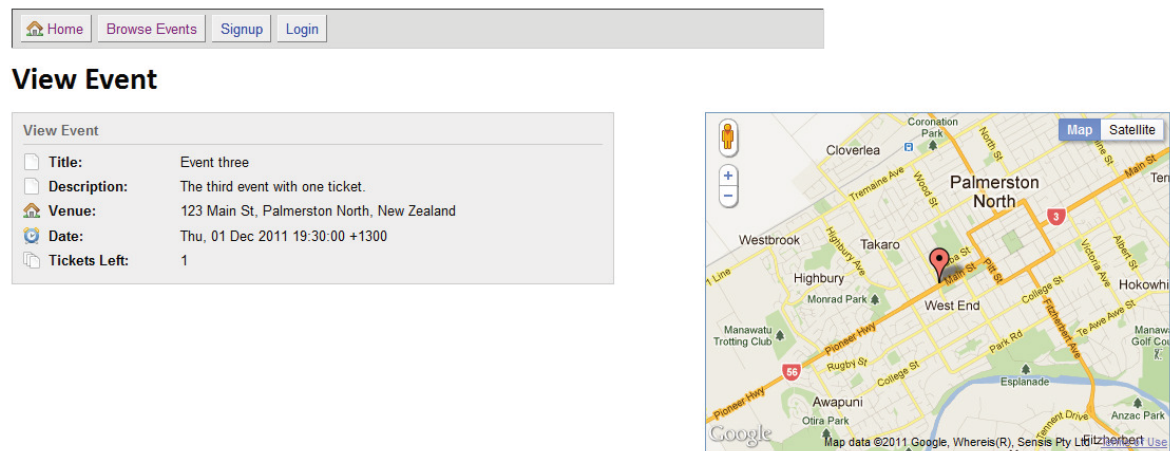


Figure 8.4: A screenshot of the *View Event* page implemented in Ticketsf-mini

Metric	Description	Ticketsf-mini	Ticketiaml	Difference
<i>Tasks</i>	Tasks implemented	22 (18%)	22 (18%)	-
<i>Time</i>	Development time (weeks)	1	1	-
<i>NDev</i>	Number of developers	1	1	-
<i>FC</i>	Number of file changes	229	43	-81%
<i>Rev</i>	Number of SVN revisions	21	19	-10%
<i>DTech</i>	Different technologies used	9	7	-22%
<i>DMedia</i>	Different media types used	1	1	-

Table 8.6: Comparing two Ticket 2.0 implementations using system metrics: overall development effort

Metric	Description	Ticketsf-mini	Ticketiaml	Difference
<i>Files</i>	Number of files	45	7	-84%
<i>Size</i>	Size of all files (bytes)	49,390	46,998	-5%
<i>NCLOC</i>	Physical lines of code	1,010	606	-40%
<i>ALOC</i>	Average NCLOC per file	22.44	86.57	286%

Table 8.7: Comparing two Ticket 2.0 implementations using system metrics: manual effort

### 8.3.4 System Metrics Evaluation

These two different implementations of the same benchmarking application can be compared through the system size metrics described earlier in Section 2.6.3, and these metrics are provided here in Table 8.6. The file-specific metrics can be broken down based on the proportion of manual development effort<sup>8</sup>; the results of these metrics against the manual implementation effort is provided in Table 8.7.

These values can be used to make a number of observations on each systems' implementation. These metrics suggest that an IAML-based web application requires less manual development effort with fewer implementation technologies than a Symfony-based web application, and the reduction in implementation technologies are illustrated in Appendix G. However, as only one benchmarking application has been evaluated, additional benchmark-based evaluations need to be performed in order to strengthen this observation.

Compared to the Symfony implementation, the IAML implementation required significantly fewer file changes (*FC*). This is due to the different architectures of each implementation; IAML source code is designed to be provided as a single model instance with additional platform-specific templates and includes, whereas Symfony source code is designed to be provided across a range of subtypes on generated framework code and classes. This is also reflected in the difference in the number of files (*Files*) between the two implementations, highlighted in the file-based metrics of Table 8.7.

The IAML implementation also required fewer lines of code (*NCLOC*) than the Symfony implementation; this may be due to the differences between PHP scripts and XMI instances; the former is designed to be human-editable, whereas the latter is designed to be machine-readable. This difference is also reflected in the average lines of code per file (*ALOC*). These results are also a consequence of the architecture of the IAML implementation; EMF allows for a model instance to span many XMI

<sup>8</sup>In this evaluation, *manual effort* is defined as code which requires manual development effort in order to implement a requirement. This includes new source code files and manual changes to generated code, but does not include generated source code that does not require any modification, nor runtime libraries provided by the application framework.



files if necessary [314, pg. 404–415], which if applied would reduce the value of this metric.

While these results can suggest RIAs may be implemented using less development effort using the proof-of-concept implementation of IAML than using the Symfony framework, it is important to consider that this is based on the comparison against a single benchmarking application. While *Ticket 2.0* is currently the only known implementation of a benchmarking application in the RIA domain, it would be preferable to perform similar evaluations against other RIAs within different domains, and these may affect the results of these metrics.

For example, consider the metrics which measure the final size of the system (*Size*) and the average lines of code per file (*ALOC*). Since IAML does not yet support the textual definition of expression logic (proposed later in Section 9.3.4 as future work), the implementation of a web application with significant amounts of custom business logic may proportionally increase the values of these metrics against a Symfony-based implementation of the same application. Conversely, since IAML supports the concept of *model completion* to automatically provide scaffolding for common application concepts, the implementation of a web application with much less custom logic may proportionally decrease the values of these metrics.

The implementation of Ticket 2.0 against different existing web application frameworks may also produce different values for each of these metrics, and it is not known whether the Symfony framework is the most efficient existing framework for the development of RIAs. However, as RIAs by definition include client-side scripting technologies, it is expected that the metric measuring the number of programming technologies used (*DTech*) will consistently be lower in an IAML-based implementation, than in an implementation with existing web application development frameworks. As IAML abstracts away the concept of client-side and server-side logic, an IAML model developer does not have to concern themselves with maintaining the logic against two (or more) different technologies.

## 8.4 Metamodelling Metrics

To evaluate the quality of the underlying metamodel, the suite of metamodel metrics defined earlier in Section 2.6.2 – reproduced here in Table 8.9 – will be evaluated against both the IAML metamodel and against other metamodels related to model-driven development. The results of this evaluation are provided in Table 8.8<sup>9</sup>, where each metric has been applied against the most recently published Ecore representation of each metamodel.

These metamodel metrics are also applied to Ecore-based implementations of the UWE and WebML metamodels. The UWE metamodel used in this evaluation is the most recent Ecore version of the UWE metamodel as provided through the *UWE4JSF* project proposed by Kroiss et al. [197]. Since the UWE metamodel is defined as an extension to the UML metamodel, the UWE metamodel includes all of the packages, classes and references of the UML metamodel; for example, the UWE metamodel defines the class *PresentationElement* as a subtype of the UML class *Class*.

At the time of writing, the implementation of the WebML metamodel in the latest version of WebRatio (6.1.0) was provided using a proprietary MDA-incompatible architecture (DTDs and XMLs of units) which is not unified into an overall metamodel. However, the AspectWebML project proposed by Schauerhuber [295] extracted an Ecore-based metamodel from these DTDs, based on work by

<sup>9</sup>In this table, the results produced by Monperrus et al. [235] could not be reproduced, as they had modified the original Ecore metamodel. Consequently, the metrics provided here have been regenerated from the most recent version of the Ecore metamodel.

Metamodel	<i>TNP</i>	<i>NoC</i>	<i>NoAC</i>	<i>TNoR</i>	<i>TNoA</i>	<i>NoD</i>	<i>NoE</i>	<i>Nav</i>	<i>Cont</i>	<i>Dat</i>
UML	1	247	48	480	106	17	13	0.33	0.38	0.18
GMF Notation	1	69	6	31	68	17	13	0.13	0.52	0.69
XSD	1	57	22	125	98	25	20	0	0.32	0.44
OCL	1	52	9	4	1	0	0	0	0.25	0.20
ModelDoc	1	33	4	62	55	1	1	0.55	0.44	0.47
Ecore	1	20	5	48	33	32	0	0.33	0.38	0.41
IAML	9	103	43	126	39	8	7	0.41	0.52	0.24
UWE [197]	10	319	59	533	155	19	15	0.36	0.34	0.23
WebML [295]	17	122	21	200	205	12	12	0	0.66	0.51

Table 8.8: Using metamodel metrics to evaluate the IAML metamodel against other similar metamodels implemented using the Eclipse Modeling Framework, adapted from Monperrus et al. [235] and Vépa et al. [331]

Metric	Description
<i>TNP</i>	Number of packages, i.e. <i>EPackages</i> .
<i>NoC</i>	Number of classes, i.e. <i>EClasses</i> .
<i>NoAC</i>	Number of abstract classes, i.e. <i>EClasses</i> that are specified as <b>abstract</b> .
<i>TNoR</i>	Total number of references, i.e. <i>EReferences</i> .
<i>TNoA</i>	Total number of attributes, i.e. <i>EAttributes</i> .
<i>NoD</i>	Number of primitive datatypes, i.e. <i>EDataTypes</i> .
<i>NoE</i>	Number of enumerations, i.e. <i>EEnums</i> .
<i>Nav</i>	Navigability: The proportion of references with a defined <i>eOpposite</i> .
<i>Cont</i>	Containment: The proportion of containment references to all references.
<i>Dat</i>	Data quantity: The proportion of attributes to overall structural features.

Table 8.9: Selected metamodeling metrics for metamodels within the Eclipse Modeling Framework, adapted from Section 2.6.2

Schauerhuber et al. [296]. The WebML metamodel uses a lot of attributes, since they do not distinguish between a PIM and PSM; for example, concepts such as server port numbers are defined as attributes on model elements, whereas they should be defined as platform-specific configuration attributes provided to the code generation templates, or as arguments to the runtime environment.

Metamodeling metrics for the number of classes (*NoC*) and abstract classes (*NoAC*) in each metamodel are also illustrated in Figure 8.5. These metrics show that the IAML metamodel relies heavily on abstract classes, with a higher proportion of abstract classes than other metamodels. In particular, the IAML metamodel heavily uses model concepts to represent the source and targets of relationships; for example, the interfaces of [Requires Edges Source](#) and [Requires Edge Destination](#) are used as the source and target of a [Requires Edge](#), as Ecore metamodels do not support the concept of type unions.

In Figure 8.6, the different design architectures of each modelling language are illustrated through metamodeling metrics measuring the total number of references (*TNoR*) and attributes (*TNoA*) for each modelling language. This illustrates that the WebML metamodel is heavily oriented around attributes, whereas the majority of other languages are oriented around references, and the IAML metamodel follows a similar architecture. This is also illustrated by the *Dat* metric, which represents the proportion of attributes to structural features in the metamodel.

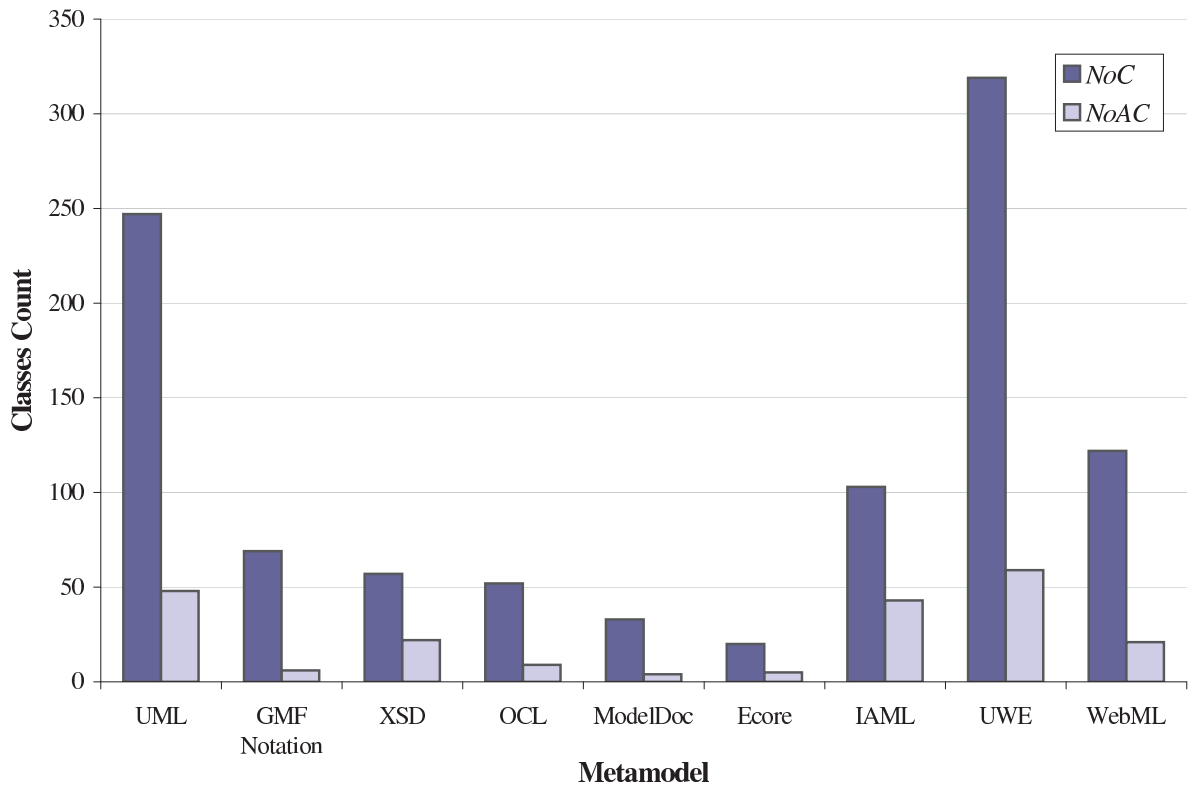


Figure 8.5: Comparison of the number of classes (*NoC*) and abstract classes (*NoAC*) of the IAML metamodel against other similar metamodels

As discussed by Monperrus et al. [235], the metrics for *Nav*, *Cont* and *Dat* are believed to give information on the modelled domain and the modelling styles and practices, but a balance between these values has not been determined. The *Nav* metric represents the opposite navigability of references through *eOpposite*, simplifying development of model-driven tools such as code generators. The *Cont* metric represents the proportion of references that are containment references, in that one element can “contain” another element, and may illustrate the modelling architecture used. However, it is not desirable for this metric to be at either of the extremes, suggesting that a modelling language is too unstructured, or conversely too structured.

Importantly, the metamodeling metrics evaluated in this section highlight that the IAML metamodel uses less classes, abstract classes, references and attributes than the existing web application modelling languages of WebML and UWE. This may suggest that the IAML metamodel is simpler than these existing metamodels, but it is difficult to compare these languages without first verifying that each language can support the same functionality requirements. This functionality-based requirements evaluation is proposed as future work in Section 9.3.1, and this evaluation could verify that the simplicity of the IAML metamodel does not negatively impact the functionality of the language.

## 8.5 Visual Model Evaluation

As discussed earlier in Section 5.1.4, the visual notation of the IAML metamodel will be evaluated according to the two evaluation techniques discussed earlier in Section 3.4.4, in order to judge the effectiveness of this proposed visual syntax. While the topic of researching and developing an effective visual notation for representing IAML model instances was outside the scope of this research, the

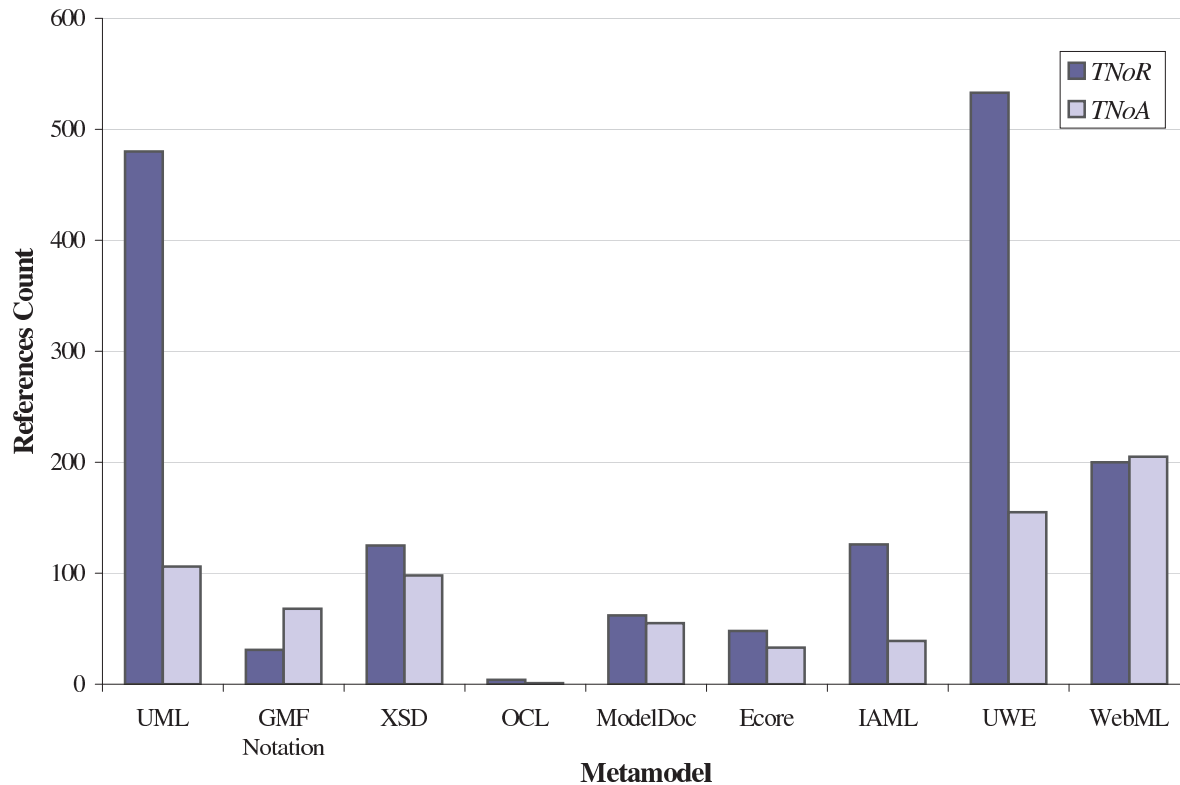


Figure 8.6: Comparison of the total number of references (*TNoR*) and attributes (*TNoA*) of the IAML metamodel against other similar metamodels

results of these evaluations can be used to show that basic guidelines for developing effective user interfaces were considered during the development of the existing proof-of-concept implementation.

### 8.5.1 Notation Information Capacity

In Table 8.10, the visual syntax of IAML is evaluated for cognitive effectiveness against the information capacity of each visual syntax variable, proposed by Moody [236] and discussed earlier in Section 3.4.4. The results of this evaluation illustrates that the visual notation of IAML does not exceed any of the maximum information capacities for each of these syntax variables. For some of these variables, the definition of the visual notation was constrained by the limits defined by the guidelines themselves, as discussed in Section 7.4.3.

For example, only five shapes are currently used to distinguish between model shapes; only two levels of brightness are used for text (black and gray); no textures are used as part of the notation; and seven colours from the Eclipse palette [199] are used for the basis of default colours. However, the GMF framework permits the user to override the predefined visual notations for each model element, such as assigning specific elements a new background colour, or changing the font size or font face on specific elements.

The IAML visual notation does not explicitly discuss horizontal or vertical positions, and the user may place diagram elements anywhere within a very large drawing plane. This is intentional, as a modelling element may contain any number of children elements; there should not be a limit on the number of elements displayed in a single editor. However, such limits could be encouraged by implementing automatic layout managers for each editor in the future.

This evaluation also highlights that the IAML visual syntax currently uses only five different

Variable	Power	Capacity	IAML Maximum
Horizontal position (x)	Interval	10–15	n/a
Vertical position (y)	Interval	10–15	n/a
Size	Interval	20	n/a
Brightness	Ordinal	6–7	2
Colour	Nominal	7–10	7
Texture	Nominal	2–5	0
Shape	Nominal	Unlimited	5
Orientation	Nominal	4	1

Table 8.10: Evaluation of the information encoding capacity of IAML visual notation, adapted from Moody [236]

shapes, due to resource constraints on the development of the proof-of-concept implementation. Moody however argues that an unlimited number of shapes can be used within a visual syntax in order to improve usability [236]. In the future, the visual notation for IAML should be extended to support new shapes, particularly shapes derived from existing visual notations<sup>10</sup>; for example, [Decision Nodes](#) could be represented using a diamond shape [254, pg. 359–361], or [Domain Sources](#) represented using a database shape (pg. 10).

### 8.5.2 Cognitive Dimensions Evaluation

The *cognitive dimensions* framework [133] may be used to subjectively evaluate a visual notation, as discussed earlier in Section 3.4.4 and illustrated by Grundy et al. [138]. The evaluation of each cognitive dimension against the proof-of-concept implementation of IAML is omitted here due to lack of space, but is included in this thesis as Appendix H.

This evaluation has shown that the visual representation of IAML model instances broadly satisfies each of the thirteen dimensions of this framework. Many of these dimensions – such as *secondary notation* and *visibility* – are automatically addressed to an extent through the GMF framework, and through the hierarchical design of the modelling language. The design and implementation of the IAML metamodel is also an important factor in many of these dimensions; for example, the smallest valid IAML model instance consists of only two elements – a single empty [Frame](#) contained by a root [Internet Application](#) – and this assists in the *progressive evaluation* cognitive dimension.

## 8.6 Conclusion

In this chapter, the design and implementation of IAML has been evaluated within five evaluation criteria. The results of these evaluations can be summarised as follows:

1. The IAML metamodel and its proof-of-concept implementation implement more of the features necessary to model Rich Internet Applications than any existing modelling language for web applications. In particular, IAML has strong support for event modelling; platform-independence; reusing metamodels; and the verification of model instances.

<sup>10</sup>Issue 231: *Add more shape styles for the visual syntax of model elements.*

2. All of the *Basic RIA* modelling requirements have been used in the design of the IAML meta-model. However due to resource and time constraints only 59% of these original requirements have been implemented in the proof-of-concept modelling environment, and only 43% have been validated as part of the Ticket 2.0 benchmarking application.
3. Since the IAML metamodel does not fully implement the requirements of a *Full RIA*, Ticket 2.0 cannot be fully implemented; resource constraints also prevented the implementation of this application. However, preliminary results from this evaluation suggest that an IAML-based implementation is comparable to a Symphony-based implementation, and reduces the number of development technologies necessary for the implementation of a Rich Internet Application.
4. In terms of metamodeling metrics, the IAML metamodel can be considered simpler than the existing metamodels of UML, UWE and WebML, but more complex than simpler metamodels such as Ecore and OCL. These metrics also suggest that the overall architecture of the IAML metamodel is similar to the architecture of other metamodels.
5. The proposed visual syntax of the IAML metamodel can be considered effective, through evaluating the maximum information capacity of the variables of its visual syntax, and evaluating the overall implementation subjectively against the Cognitive Dimensions framework.

## Chapter 9

# Conclusions and Future Research

In the thesis, the requirement capture, design, development and proof-of-concept implementation of IAML – a new modelling language for the development of Rich Internet Applications – has been discussed in detail. In this chapter, the main contributions of this research to the fields of software engineering and model-driven development will be summarised, along with a brief discussion on some remaining research questions and future work for this research.

### 9.1 Research Contributions and Conclusions

#### 9.1.1 Requirements for Modelling Rich Internet Applications

As discussed in Section 2.3 (pg. 14), the first step for this research was to identify the requirements that an RIA modelling language should satisfy. The existing literature on evaluating web application modelling languages did not consider the interactive requirements for RIAs, therefore seven representative existing RIAs were investigated and thirteen categories of RIA features proposed.

These categories were used to re-evaluate existing web application modelling languages in terms of their suitability for modelling the requirements of RIAs. This evaluation found that no existing web application modelling language was expressive enough to support all of the requirements identified to adequately model RIAs, and that the level of support for modelling fundamental RIA concepts such as events and user interfaces was particularly poor [368].

These same seven representative RIAs were then analysed to identify the specific requirements of a RIA modelling language, in order to objectively evaluate the functionality and expressiveness of different web application modelling languages. This investigation against the thirteen feature categories resulted in the development of 69 use cases, published in this thesis in Appendix A.

These use cases were then analysed and translated into a suite of 59 fundamental requirements of RIAs, as published by Wright and Dietrich [367]. This list of requirements forms an important contribution by providing formal definitions of RIA-specific functionality. These requirements were used in the design of the IAML metamodel, and in the future could be used to evaluate existing modelling languages with a greater level of detail than the previously discussed feature comparison.

#### 9.1.2 A Benchmarking Application for Rich Internet Applications

The benchmarking application named *Ticket 2.0* [367] was then engineered to use all of these 59 requirements of RIAs within a single design specification. This forms an important contribution for



the development of RIA modelling languages, by proposing a standardised benchmarking application that may be used to evaluate the implementations of RIA requirements within different languages.

To illustrate that this hypothetical web application represented a realistic RIA for benchmarking purposes, this application was implemented using the web application framework Symfony [274] as described in Section 8.3.1 (pg. 232). This application successfully implemented all of the requirements of *Ticket 2.0*, and the subsequent evaluation found that one of the key problems in the conventional implementation of RIAs is in keeping client-side and server-side logic synchronised throughout its development and maintenance in two different languages (PHP and Javascript).

*Ticket 2.0* was also implemented using the proof-of-concept implementation of IAML, as described in Section 8.3.2 (pg. 233), in order to identify the strengths and weaknesses of the modelling language and its implementation. By using design concepts of IAML such as wires and users, it was found that most of the necessary scaffolding for the application could be generated automatically through model completion, which suggests that model completion is beneficial within model-driven development of RIAs.

Through the proposal of system metrics earlier in Section 2.6.3 (pg. 29), the development expressiveness and productivity of these two approaches could then be evaluated. This subsequent evaluation illustrated that fewer development technologies are necessary for the implementation of an RIA, thereby reducing the complexity involved in developing and maintaining RIAs. This evaluation also illustrated that less manual development effort is necessary to implement an interactive web application using IAML than a similar implementation using Symfony, increasing developer productivity.

### 9.1.3 A Modelling Language for Rich Internet Applications

The *Internet Application Modelling Language* (IAML) represents a significant contribution as a modelling language for the model-driven development of RIAs. As evaluated in Section 8.1 (pg. 227), IAML satisfies the design goal of platform-independence; reuses existing standards where appropriate; and satisfies both the metamodeling architecture and viewpoint architecture of the MDA [243, 185]. This language has been designed to support all of the concepts of modelling Basic RIAs, and the full description of this language forms Chapter 5 and Appendix I in this thesis.

The design of IAML includes a number of novel modelling concepts, and where appropriate also adapts concepts from existing modelling approaches. Each concept described in Chapter 5 represent an important contribution, but a number of these are particularly noteworthy:

1. **Logic-based Core:** The core of the language is derived from first-order logic concepts, and supports [Functions](#), [Conditions](#), [Predicates](#) and [Complex Terms](#) (pg. 97). By defining these concepts in terms of an established mathematical notation, this metamodel core provides a strong foundation for the rest of the language.
2. **Type System:** IAML defines a rich underlying type system for model instances, composed of primitive type modelling adapted from XML Schema datatypes [339] (pg. 106), and complex domain modelling adapted from EMF Ecore [314] (pg. 112). New primitive types may be defined through the derivation of existing types, and the domain modelling approach of IAML supports multiple inheritance.

The type system of IAML is statically checked; objects may only have a single type, allowing for the type validity of a model instance to be checked during development. Types within IAML

are also weakly checked; typed objects may be implicitly cast to another type at runtime, and objects may be typed to a *default type* (pg. 109).

3. **Hierarchical Design:** The IAML metamodel is designed in a hierarchical fashion, allowing a complex system to be progressively decomposed into more manageable smaller models [236]; increases the usability of the visual notation, as discussed in Section 8.5.2; and simplifies the integration of model completion within the language implementation.
4. **Event-Condition-Action (ECA) Rules:** The evaluation of existing modelling languages by Wright and Dietrich [368] found that existing web modelling languages rarely support events as a first-level citizen of the language, even though events are a fundamental nature of RIAs. ECA rules are supported by IAML to capture this common design, and represented as an [ECA Rule](#) in Section 5.7 (pg. 122). However, IAML does not support developers defining their own arbitrary events, as this would require the inclusion of an event modelling language.
5. **Wires:** Inspired by the *connection* concept in VisualAge for Smalltalk [210], the concept of a [Wire](#) was introduced in Section 5.9 (pg. 131) to support reusing common development patterns for RIAs, such as keeping the values of two model elements synchronised. To support model developers overriding the default behaviour of [Wires](#), this functionality is implemented by evaluating model completion rules against a model instance.
6. **Scopes:** As discussed in Section 5.11 (pg. 140), [Scopes](#) permit the model developer to utilise lifecycle events, such as when a [Session](#) is initialised or accessed. [Scopes](#) also provides a natural way to define how different elements of data can be differently scoped, along with the associated access and storage semantics of the immediate and indirect contents of each [Scope](#).
7. **Users and Access Control:** IAML allows model developers to define the potential users of RIAs through the Role-Based Access Control model [293], represented as [Roles](#) and [Permissions](#) in Section 5.10 (pg. 136). Common functionality such as login and authentication is provided through [Login Handlers](#) to reduce the manual development effort necessary for these common RIA use cases, and implemented using model completion.

The expressiveness and complexity of the IAML metamodel has also been evaluated against a suite of existing modelling languages using the metamodeling metrics defined earlier in Section 2.6.2 (pg. 28). As discussed in Section 8.4, this evaluation illustrated that the IAML metamodel is simpler than the existing metamodels of WebML and UWE, and the design of the language follows a similar architecture to many existing languages.

#### 9.1.4 Model Completion

As discussed by Wright and Dietrich [369], a key challenge in the design of a modelling language is in balancing the level of detail expressible in its design; that is, its level of abstractness to its level of flexibility. Software frameworks approach this problem by supporting programming by convention, where documented conventions allow much of the scaffolding to be provided automatically.

The concept of *model completion* adapts this concept to the model-driven development domain and forms a significant contribution to the field of model-driven development, by representing documented conventions as *non-monotonic* inference rules. To illustrate the real-world implications of model completion in a modelling environment, these model completion rules may be implemented

using a commercial rule engine. Wright and Dietrich [369] discuss the implementation of model completion using Drools [311], showing model completion is both consistent and scalable, and Drools is used to implement model completion for IAML as discussed in Section 7.5 (pg. 210).

Importantly, model completion does not permit any original information in the original model to be removed, ensuring that developer effort can never be overridden. This process may be manually overridden by the developer, and is implemented within IAML by using the [Generated Element](#) abstract type as a supertype of all IAML model elements. Model completion is used to complete all of the necessary scaffolding for reusable patterns such as [Wires](#), [Login Handlers](#), and to support the inheritance of [Domain Types](#).

### 9.1.5 Model Instance Verification

As discussed in Section 3.3 (pg. 54), model instance verification is the process of identifying invalid models from syntactically correct models based on desired correctness constraints. In many cases it is preferable to identify errors within a system model than in its implementation, as the relative cost of fixing and error increases over time [278, pg. 197].

In this thesis, model instance verification was decomposed into three different categories, based on the expressiveness of different verification languages: functions-based languages; relations-based languages; and through model checking, with each category possessing different performance characteristics and resource requirements. By implementing model instance verification constraints within each of these different categories, a model developer may selectively evaluate correctness according to their available resources.

Within the proof-of-concept implementation of IAML, model instance verification has been implemented using four different technologies as discussed in Section 7.7 (pg. 219). Simple constraints are evaluated using the EMF Validation framework and Checks; more complex constraints are evaluated using CrocoPat; and the exhaustive evaluation of application states are evaluated by model checking in NuSMV.

### 9.1.6 Evaluation of Model-Driven Technologies

An important design goal of IAML was to provide a proof-of-concept implementation of the modelling language, and such an implementation involves the integration of a number of different technologies. Each of these technologies – such as model completion, code generation, model instance verification, and the metamodeling and graphical environments themselves – may be provided by a number of different technology implementations.

The evaluation of many different implementation technologies for each of these model-driven technologies forms an important contribution of this thesis, and is discussed in detail within Chapter 6. Each technology was evaluated according to functionality and expressiveness requirements; the simplicity of integrating these technologies together; and implementation quality evaluated using the OpenBRR open source evaluation framework [263] (pg. 160).

The results of this evaluation were used to select a suite of implementation technologies used for the proof-of-concept implementation of IAML. These technologies included the Eclipse Modeling Framework (EMF) for a metamodeling environment; the Graphical Modeling Framework (GMF) for a graphical modelling environment; the openArchitectureWare Xpand language for a code generation framework; the Drools rule engine for the implementation of model completion; and a selection of

four different languages to support different forms of model instance verification.

As one of the design goals of IAML, a visual syntax was also designed in Section 7.4.3 (pg. 202) in order to support the graphical definition of IAML model instances. The effectiveness and simplicity of this visual syntax was evaluated in Section 8.5 (pg. 257) against both its notation information capacity, and against the *cognitive dimensions* framework [133].

### 9.1.7 Other Contributions

Through the process of designing, implementing and subsequent verification of a modelling language for RIAs, a number of smaller contributions throughout this research have also been recognised, and will be briefly discussed here.

An important consideration for the development of model completion rules is in providing adequate and reliable documentation for these conventions [369]. The *ModelDoc* framework described by Wright [366] was developed to allow this documentation to be automatically loaded from the model completion rules themselves; this documentation can then be combined with other sources of documentation into an authoritative documentation source. In particular, this documentation source forms the basis of the IAML metamodel reference in Appendix I.

During the development of the IAML visual modelling environment using GMF, it was recognised that many of the GMF model instances used to generate these editors shared common features, yet had to be implemented manually. The *SimpleGMF* modelling language was developed in Section 7.4.3 (pg. 202) to reduce the development effort necessary to maintain these graphical editors, by supporting the automatic generation of GMF model instances.

In order to reduce the complexity of the implementation of each model-driven technology within the proof-of-concept implementation, a model-driven code coverage technique was developed in Section 7.9.1 (pg. 224). By obtaining these metrics through the evaluation of the exhaustive test suite for IAML, any unused source code – including code generation templates and model completion rules – could be removed to reduce the complexity of the implementation. These code coverage metrics could also be used to improve the test suite itself, or to identify elements of the code generation templates that are never executed within a particular platform.

## 9.2 Answers to Research Questions

In Section 1.3, the underlying statement of this thesis was broken down into six research questions. These questions have been systematically answered throughout this thesis, as summarised below.

*What unique challenges do Rich Internet Applications introduce into the process of their development and maintenance?*

A brief history of Rich Internet Applications was discussed in Section 2.1 (pg. 9), which culminated in identifying some of the features of existing RIAs in Section 2.3.1 (pg. 14) and published by Wright and Dietrich [368] in 2008. By developing a comprehensive suite of use cases in Appendix A, these general features were expanded into a list of 59 detailed modelling requirements in Section 2.3.2 (pg. 17). These requirements were then combined together into a single benchmarking application named *Ticket 2.0* – published by Wright and Dietrich [367] in 2008 – and discussed here in Section 2.5 (pg. 24).

The practical challenges of implementing a Rich Internet Application using conventional technologies was highlighted in the implementation of this benchmarking application using the Symfony framework in Section 8.3.1 (pg. 232). For example, the business logic behind a client-side application must be implemented identically on both client and server technologies, and this represents a significant maintenance issue. The implementation of external authentication providers such as OpenID was also not natively supported by the Symfony framework, and had to be integrated manually.

*Are there shortcomings in using existing modelling languages to model Rich Internet Applications?*

The modelling requirements of RIAs introduced earlier in Section 2.3.1 (pg. 14) were evaluated against a number of existing web application modelling languages, such as WebML, UML and UWE. The results of this evaluation in Section 2.4 (pg. 20), published by Wright and Dietrich [368] in 2008, found that no existing modelling language supported the fundamental requirements for modelling RIAs.

*If existing modelling languages cannot be used to develop Rich Internet Applications, can a modelling language be developed to address these challenges, either from the extension of an existing language or the development of a new language?*

As existing languages could not describe fundamental RIA concepts without requiring significant changes, a new modelling language named the *Internet Application Modelling Language* (IAML) has been designed in Chapter 5 and implemented in Chapter 7. As evaluated in Section 8.2 (pg. 229), this language has been designed to support all of the requirements of modelling *Basic RIAs*, and partially validated through the implementation of the *Ticket 2.0* benchmarking application.

*Are there techniques and technologies that may be used to improve the maintainability of a modelled Rich Internet Application?*

The *model completion* concept, introduced in Section 3.2 (pg. 51) and published by Wright and Dietrich [369] in 2010, was developed to improve the maintainability of model-driven applications by automatically completing model instances into intended models through documented conventions, and is an important feature of the IAML modelling environment. The usability and understandability of this concept has not yet been evaluated – in particular, identifying the most effective method in documenting model completion rules, and presenting them to the user [369, pg. 9] – and in the following section is discussed as future work.

Investigations into the use of round-trip engineering and reverse engineering were also performed in Section 3.1.8 (pg. 47); however, information loss due to the non-injective nature of most model transformations makes the reverse engineering of arbitrary model transformations very difficult [153, pg. 33]. It was also noted that none of the code generation frameworks evaluated in Section 6.4.5 (pg. 176) natively supported any form of round-trip engineering.

The modelling language itself provides a single interface for specifying client-side and server-side logic, meaning that the logic behind a client-side application does not need to be reimplemented using a different server-side technology. This reduces both the development effort necessary, and the resulting maintenance task that would otherwise be present. A discussion on the impact of this concept is provided in Section 8.3.4 (pg. 254).



*Can it be shown that an Rich Internet Application modelling language increases the security of the underlying system?*

As discussed in Section 5.2.2 (pg. 90), this question has been addressed by designing the modelling language to not support insecure application design, preventing the inadvertent introduction of vulnerabilities into a modelled application. For example, database access through a [Domain Iterator](#) may only occur through a parameterised API, preventing SQL injection vulnerabilities (pg. 117).

The development of a suite of model instance verification tools, as discussed in Section 7.7 (pg. 219), also allows a model developer to evaluate their modelled application against different design and correctness properties. The complexity of certain constraints may significantly impact the performance of model verification, such as exhaustively searching the navigation space of a web application in Appendix D. By decomposing constraints into four increasingly complex verification technologies, IAML model instances may be verified without significantly impacting the usability of the modelling environment.

*Can it be shown that such a modelling language improves the development process of these web applications, in terms of speed, simplicity and consistency?*

In Section 8.3 (pg. 231), the *Ticket 2.0* benchmarking application was implemented within both the conventional web application framework Symfony, and the modelling environment provided by the proof-of-concept implementation of IAML. The resulting evaluation suggests that the implementation of an RIA in IAML was performed *faster* than an equivalent implementation within Symfony, and is *simpler* by reducing the number of technologies involved in its development.

The quality of the IAML metamodel has also been evaluated through the metamodelling metrics in Section 8.4 (pg. 255), illustrating that the IAML metamodel is *simpler* than the metamodels of existing web application modelling languages. The metamodel has also been designed – as per the *consistency* language design principle – to reuse existing standards and visual metaphors where appropriate<sup>1</sup>, and to reduce the occurrence of model developers inadvertently introducing security vulnerabilities into their modelled applications.

## 9.3 Future Research

The research involved in this thesis has included contributions from a wide variety of research areas, and has uncovered a number of unanswered research questions which remain as future work. In this section, some of these areas of future work will be briefly discussed. The proof-of-concept implementation itself also includes a number of outstanding issues<sup>2</sup>; some of these issues will also be briefly discussed here.

### 9.3.1 Further Evaluation of Existing Web Modelling Languages

As discussed by Wright and Dietrich [367], existing web modelling languages have not been re-evaluated against the list of 59 detailed RIA modelling requirements in Section 2.3.2 (pg. 17), with the exception of the evaluation against IAML in Section 8.2 (pg. 229). Many of these languages have continued to evolve and, in some cases, incorporate limited functionality for RIA concepts. For

<sup>1</sup>These design goals are discussed in Section 5.1.2 (pg. 86) and Section 5.15 (pg. 151), respectively.

<sup>2</sup>The list of outstanding issues is available online at <http://code.google.com/p/iaml/issues/list>.

example, recent work for the actively developed WebML environment includes improved support for modelling client-side data and a new event model, as discussed by Fraternali et al. [105]. A re-evaluation of these web application modelling languages would highlight the strengths and weaknesses of the IAML metamodel with respect to recent developments.

While the development and publication of the *Ticket 2.0* benchmarking application was an important original contribution of this thesis, this application specification only describes the design requirements of *Ticket 2.0*, and does not specify the actual functionality of the benchmark. A web application testing framework such as JWebUnit [152] could be used to develop integration test cases that would independently evaluate the functionality of a certain implementation of *Ticket 2.0*, and guarantee the completeness of each implementation; it would also significantly impact on developer flexibility, as such a test suite would enforce many more constraints on an implementation. The test suite could also be used to accurately profile the implemented applications for performance, and different implementations compared to identify potential areas for future optimisation work.

### 9.3.2 Modelling Full RIAs

The current design and implementation of IAML only supports the modelling requirements of *Basic RIAs*. As discussed in Section 5.1.1 (pg. 85), this feature set restriction was necessary due to the time and resource constraints of this research project. There do not seem to be any design issues that would prevent the extension of IAML to support modelling *Full RIAs*, and this may be achieved through the definition of an extension metamodel. For example, some of the enterprise requirements of Full RIAs – such as offline storage and internationalisation – could be defined through annotations on the existing model elements *Scope* and *Visible Thing*, rather than the redefinition of their underlying concepts. In Section 8.3.2, the evaluation of the implementation of *Ticket 2.0* using IAML also highlighted that additional optimisation effort on the proof-of-concept implementation of the language is necessary.

### 9.3.3 Improved Graphical Editor View Mappings

During the development of the proof-of-concept graphical editor of IAML using GMF, an unexpected issue was discovered resulting from the level of abstraction of the interaction between the generated editor and the underlying metamodel. As discussed on Section 1 (pg. 203), GMF-based editor definitions work best when there is a one-to-one mapping between graphical elements and underlying metamodel elements. This means that without extending the generated graphical editor, one cannot easily represent the aggregation or derivation of elements as a single graphical element.

Consider Figure 9.1a, where a *Complex Term* (in this case, a *Simple Condition*) is provided the field value of a *Changeable* element as a *Parameter*. This field value is defined as the default *Value* for that element, and it would be preferable to represent this as Figure 9.1b, where the *Parameter* instead connects the *Changeable* element directly. For each of these visual representations, the underlying model instance can be exactly the same – the only difference is in the simplification of the syntax.

By default, the generated graphical editor cannot support reducing Figure 9.1a into Figure 9.1b, because the underlying one-to-one mapping model specifies that a *Parameter* must connect to a *Parameter Value* (and consequently a *Value*). Consequently, without the manual reconfiguration of the generated graphical editor and the templates that generate it, it is easier to specify that *Changeable* is a subtype of *Value* as syntactic sugar as discussed in Section 5.4.6, enforcing additional complexity on the underlying metamodel.



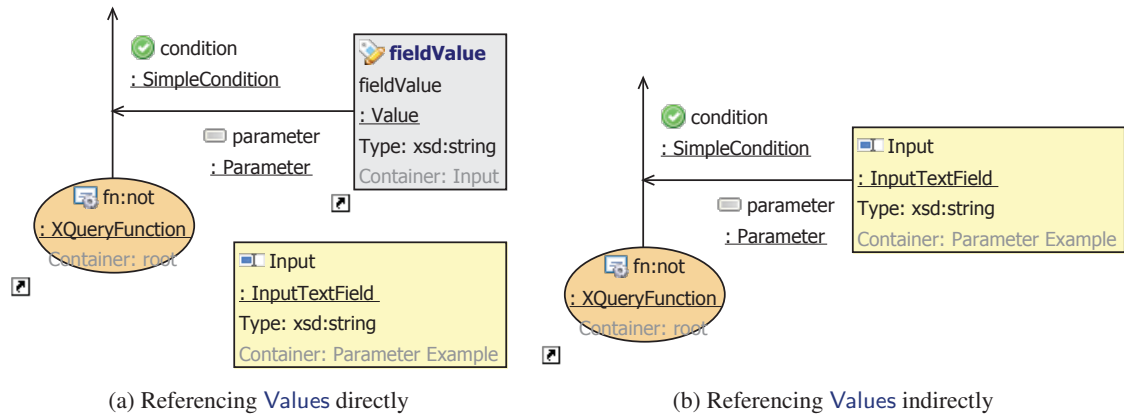


Figure 9.1: Two approaches in defining a **Complex Term** that references a **Value** contained within a **Changeable** as a **Parameter**

As the artefacts necessary to generate GMF graphical editors are also EMF metamodel instances [136], it may be possible to define a new intermediary graphical mapping metamodel – a *view mapping model* – between these artefacts to formally define this syntactic sugar. Model transformations may then be used to transform the underlying metamodel and view mapping model into the necessary GMF model instances. As SimpleGMF already follows a similar architecture, it may be possible to extend this language into this desired view mapping metamodel.

### 9.3.4 Integrating Textual Expression Languages in the Visual Editor

The proof-of-concept implementation of the IAML metamodel has been designed to support a visual representation for all of the underlying model instances. However, as shown earlier in Figures 5.25 and 5.27 (pg. 126), this approach can transform simple textual expressions into complex visual representations of the same expression. In languages such as UML, this problem is resolved by allowing constraints to be defined textually; in some cases expressions can be defined in terms of OCL [254], where the main issue is in defining the mapping between textual syntax elements and the underlying model instance.

Since this research was started, the EMFText project was proposed [150] which provides a textual interface to model instances represented within EMF; the resulting syntax is similar to YAML representations of configuration data [27]. The openArchitectureWare framework includes a similar textual interface to EMF model instances called *Xtext*, however the authors of EMFText argue that *Xtext* cannot be used to interface with existing metamodels [150]. An investigation into integrating such a textual interface into the visual editor therefore remains future work; such a textual interface should not involve any modifications to the underlying metamodel, as it would be a different representation of the same underlying model.

### 9.3.5 Extraction of Reusable Components

As illustrated in Chapter 7, the development of the proof-of-concept implementation of IAML required the development of a large number of components. Some of these components, such as the *IAML Metamodel* and *Runtime Library* components, are specific to the RIA domain; others such as the *Model Completion* and *Model Verification* components may be reusable in other modelling domains.

This model-driven development environment generalisation has already been successfully used in the development of the openArchitectureWare framework [87]. As discussed in Section 6.4.3 (pg. 175), this framework integrated a number of existing technologies into a single architecture connected by a workflow. This integration significantly simplified the accessibility of model-driven technologies to other developers, and this architecture is now being integrated into the Eclipse ecosystem. Integrating model completion and model verification into this model-driven architecture could therefore be a very beneficial area of future research and development.

### 9.3.6 Code Generation Templates for Additional Platforms

While one of the design goals of IAML was to develop a platform-independent modelling language, the code generation templates of the *Templates* component in Section 7.6.2 (pg. 215) only support a single combination of target platforms; in particular, the combination of PHP, Javascript, HTML, CSS and sqlite. To prove that the IAML metamodel is truly platform-independent, it would be desirable to independently reimplement these templates to support additional browser platforms. For example, as the J2ME standard is designed to be device-independent [323], an implementation of the code generation templates of IAML into a J2ME format would verify that the design of the IAML metamodel is platform-independent and also device-independent.

### 9.3.7 Incremental Transformations

One of the issues identified during the implementation of the *Ticket 2.0* benchmarking application in Section 8.3.2 is that the entire process is relatively slow and consumes significant system resources. This is because both model completion and code generation currently operate as batch processes, meaning that small changes require the complete re-evaluation of each model transformation. This slow feedback cycle discourages incremental design processes or those that follow test-driven development principles [23].

*Incremental generation* is a code compilation technique that allows source code to be incrementally transformed into executable code as necessary [229, pg. 1144–1149], and is an important feature of the Eclipse development environment to reduce development time [117, pg. 146]. Giese and Wagner [123] investigated the development of applying incremental generation to model transformations, proposing bidirectional transformations to support the synchronisation of two models, and found that the corresponding speedup may allow the development of larger model instances.

At the very least, incremental model transformations could be applied to both the model completion and code generation aspects of the IAML proof-of-concept implementation, and would ideally be implemented through the Eclipse plugin framework to support an automated build process within Eclipse itself. For example, Drools supports the assertion and retraction of facts on a running model instance [279], and these techniques may support incremental model completion. Similarly, the Eclipse release of Xpand 1.0 has recently announced support for incremental code generation which benefits the transformation of large model instances [163].

### 9.3.8 Extensibility of the Proof-of-Concept Implementation of IAML

Many of the implementation technologies evaluated in Chapter 6 included an evaluation of the extensibility of each technology, in order to support third-party extensibility of model-driven environments.

For example, both EMF, GMF, Drools and openArchitectureWare all support varying levels of extensibility; and the proof-of-concept implementation of IAML has been implemented through OSGi bundles that use existing Eclipse extension points.

As discussed earlier in Section 3.1.3, integration with other models and environments is an important benefit of model-driven environments. Consequently, an important future research question is on identifying the aspects of which these technologies may be extended, and how the extensibility of these technologies may be impacted by their integration into a single editor. It would also be interesting to investigate how the extensibility of these selected technologies compare to another extensible metamodeling environment such as Marama [137].

### 9.3.9 User Evaluations on the Proof-of-Concept Implementation of IAML

In this thesis, the interactions between the modelling environment and the model developers themselves have not been evaluated as this investigation fell outside the scope of this research. Without this user evaluation, it is difficult to measure the productivity of the modelling environment, and the simplicity of the modelling language itself. Performing end-user evaluation could quantify the learning curve of IAML and its underlying concepts, and the learning effort necessary to become productive in the modelling environment. Accurate evaluations would however need users that are very experienced in the development of RIAs – both through conventional web application development practices, and through the IAML environment – and are therefore very expensive to perform.

Many of the techniques themselves involved in the proof-of-concept implementation would also benefit from some form of user evaluation. For example, the benefits and difficulties of the model completion concept could be evaluated in terms of their usability and understandability, and would identify how much documentation is necessary for end-users to understand the concept and the individual rules [366]. Similarly, the benefits and difficulties of using the many different types of model verification approaches could be investigated – is it acceptable to simultaneously provide three different approaches? What is the best way to highlight constraint violations identified using model checking? The results from these user evaluations could be used to inform best practices for developing modelling language within particular domains.

### 9.3.10 Identifying Metamodel Refactoring Patterns

A number of common activities and patterns emerged as part of the incremental design and development of IAML. For example, something as simple as renaming a class name involved modifications to source code, model completion rules, code generation rules, test cases, test models, documentation and design documents. Over time, a number of development patterns emerged, where similar types of metamodel changes would result in similar amounts of subsequent refactoring effort.

Similarly to the concept of “refactoring” within the software domain [100], similar types of changes applied to a metamodel may be termed *metamodel refactoring*, and the resulting changes necessary for the implementation of that given metamodel termed *coincidental refactoring*. Some of these metamodel refactorings identified through this research are discussed below, grouped into four categories of increasing subjective difficulty.

- **Easy:** Adding a new attribute, reference, interface or class; creating a new package.

Generally, adding something to the metamodel does not require a lot of coincidental refactoring. This is due to the nature of object-oriented software; for an extension, only the extension needs

to be considered, and all the underlying code should remain the same.

Within IAML, none of these refactorings required any direct changes to other components. In particular, the SimpleGMF framework provides all new classes with a default representation within each graphical editor.

- **Moderate:** *Renaming an attribute, reference, interface or class; changing a metamodel namespace; moving a class to another package; removing an empty package.*

When renaming something in the metamodel, the coincidental changes are fairly straightforward. Generally, existing references simply need to be refactored; there is no change in the internal logic. Automated refactoring tools can often assist in the renaming process – although EMF does not yet directly integrate with the Eclipse refactoring tools.

Within IAML, these refactorings would often require the graphical editors references to be renamed, and regenerated; code generation templates and model completion rules to be updated; the model migrator updated; and existing model instances migrated. Renaming interfaces was generally the easiest, as since you cannot have an instance of an interface, existing model instances did not need to be migrated.

- **Difficult:** *Merging two attributes or references; removing an interface or class; splitting an interface into two interfaces.*

These metamodel refactorings require a much more in-depth refactoring, as the developer will have to manually modify the dependent components.

Within IAML, these changes required extensive development effort on the graphical editors, as graphical elements needed to be removed or merged. Model completion rules would have to be merged, and code generation templates would either be merged or require additional logic. The model migrator would have to be updated and all existing model instances migrated.

- **Very difficult:** *Splitting a reference into two references; removing an attribute or reference; splitting a class into two classes.*

Within IAML, these changes would require changes on all of the implementation components. Because these metamodel changes are changing the *semantics* of a metamodel element, these refactorings may not trigger compile-time errors. The test suite discussed in Section 7.9 (pg. 223) was extremely beneficial in identifying necessary functional changes that would otherwise have gone unnoticed.

As in software development, a modelling language developed incrementally would likely benefit from the existence of automated refactoring tools to assist in applying similar changes. For example, something as simple as renaming a model element will normally result in thousands of necessary source code changes, including Java source code, model completion rules, documentation and design documents, and test models; with the right development environment architecture, many of these changes could be performed automatically or with little developer input. Further research into this area could include the development of new metamodel refactoring tools, or in new documentation or development methods to increase the robustness of model-driven environments.

## 9.4 Summary

Model-driven development can be utilised to simplify the development of complex software applications across many platforms, but this research has found that no existing web application modelling language could be used to satisfactorily model or develop Rich Internet Applications. The Internet Application Modelling Language (IAML) has been designed and proposed in this thesis as a platform-independent RIA modelling language. This language reuses existing modelling languages where appropriate; introduces a number of novel modelling concepts, such as model completion; and also defines a visual syntax to support the graphical definition of IAML model instances.

The design of this language was evaluated by the development of a proof-of-concept implementation in the Eclipse framework, to verify that the design concepts of the language benefit the implementation of real-world RIAs. This implementation included the integration of a number of different model-driven technologies – such as a visual modelling interface, code generation templates, model completion rules, and different types of constraints for model verification – in order to improve the usability of developing robust IAML model instances.

The IAML metamodel supports many features not found in other web application modelling languages, such as ECA rules, the expression of reusable patterns through [Wires](#), and a metamodel core based on first-order logic. Through the implementation of the RIA benchmarking application *Ticket 2.0*, these concepts have been shown to simplify the development of real-world RIAs when compared to a conventional implementation through the Symphony framework. This research has also raised a number of interesting research questions that, when answered, may simplify the development of model-driven environments in the future.



# Appendix A

## Use Cases

As part of the research in defining modelling languages for RIAs, an important step was to identify the functional requirements that a modelling language would need to support. This involved the evaluation of a suite of seven existing RIAs to identify common concepts and functionality, in particular those enabled by new technologies such as AJAX [121] and HTML 5 [343], and those that embraced this new functionality to improve the interactivity and usability of their interfaces.

This is similar to previous work in evaluating the expressiveness of existing modelling languages in the domains of hypermedia [276], conventional web applications [104], and implementations of a conference system [301]. As discussed in Section 2.3.2, these use cases have been summarised into a suite of 59 individual functionality requirements, as published by Wright and Dietrich [367], that may be used to design a modelling language for RIAs.

### A.1 Actors

Without going into too much detail, the actors involved in these use cases are defined as follows:

- *Visitor*: An anonymous visitor to the application.
- *User*: A logged in/authorised *Visitor*.
- *Developer*: The developer of the site. Does not have to be a *User*.
- *Administrator*: A site administrator of the application, who is also considered a *Developer*.
- *Client*: Also known as a *web browser* [175, pg. 2]; the user interface running on the *Visitor*'s machine, such as *Mozilla Firefox* [241] or *Google Chrome* [128].
- *Server*: The application host, such as the *Apache HTTP Server* [234].
- *Remote Server*: Another application host, separate from the local *Server*.
- *Software*: A piece of application software running on the same machine as the *Client* that is executed independently of the *Client*, and is not commonly used to browse the Internet.
- *Device*: A separate piece of hardware (with respect to the hardware currently hosting the *Client*) that may be used to browse the Internet, such as a mobile phone.



## A.2 List of Use Cases

The use cases in this appendix were derived from a suite of seven existing web applications, as discussed in the paper [367]. These applications were selected as some the most popular instances of RIAs at the time, and those that introduced new RIA functionality; for example, autocomplete was one of the first well-known features of RIAs, as illustrated by the Gmail web application (Section 5.9.4).

1. **Gmail:** A web-based e-mail client, including an integrated chat-box, POP3/SMTP integration and rich text capabilities.  
<http://www.gmail.com>
2. **Google Calendar:** A collaborative online calendar, featuring mobile phone notifications and the use of external calendar data sources.  
<http://calendar.google.com>
3. **Google Reader:** A *feed reader* that retrieves and presents data feeds of content from other sites, such as RSS feeds [290], and features offline support through Google Gears [129].  
<http://reader.google.com>
4. **Google Docs:** A collaborative web-based word document editor, allowing multiple users to edit a document at once.  
<http://docs.google.com>
5. **Last.fm:** A social networking site integrated with music, allowing users to share their music tastes and listen to personalised Internet radio.  
<http://www.last.fm>
6. **Google Pages:** An online web publishing suite for producing and publishing static web sites<sup>1</sup>.  
<http://pages.google.com>
7. **Facebook:** A social networking site providing a public API which applications can use to harness the power of the social network.  
<http://www.facebook.com>

These use cases are presented in a standard use case format, describing the actors, sequence, pre- and post-conditions, exceptions, related use cases and other interesting comments [278]. These use cases could be loosely grouped into subject areas (e.g. *database-driven*, *events*, *user interface*, *access control* and others), and this categorisation forms the basis of the 59 individual functionality requirements subsequently defined in the next appendix.

---

<sup>1</sup>Google Pages is now known as *Google Sites*.

UC-01	<b>View Data</b>
<b>Description</b>	A website visitor can view a list of products on a website. This information is stored in a relational database.
<b>Preconditions</b>	A database exists; a list of products exists in the database.
<b>Actors</b>	Visitor, Server
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. The Visitor requests a list of products in the database.</li> <li>2. The Server connects to the database and lists all available products on one page.</li> </ol>
<b>Postconditions</b>	
<b>Exceptions</b>	<ol style="list-style-type: none"> <li>1. There are no products in the database; either an error is thrown or an empty list is displayed.</li> <li>2. The database cannot be accessed; an error message is displayed.</li> </ol>
<b>Comments</b>	Well supported in existing systems; a basic requirement of database-driven websites.

UC-02	<b>Update Data</b>
<b>Description</b>	A user can update their user account details on a website. Their account details are persistently stored in a relational database.
<b>Preconditions</b>	A User has a user account on the website/database and is logged in (see <i>User Authorisation</i> ).
<b>Actors</b>	User, Server
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. The User requests the Edit User Account page.</li> <li>2. The Server presents the Edit User Account page, which contains “name” and “password” fields.</li> <li>3. The User changes the name displayed in the “name” field.</li> <li>4. The User submits the page back to the server.</li> <li>5. The Server changes the “name” of the current User to the value provided by the User.</li> <li>6. The User is redirected to the home page.</li> </ol>
<b>Postconditions</b>	
<b>Exceptions</b>	<ol style="list-style-type: none"> <li>1. The User does not have the permissions to edit their user account (e.g. the account is blocked); an error message is displayed.</li> <li>2. The database cannot be accessed; an error message is displayed.</li> </ol>
<b>Comments</b>	See <i>View Data</i>

UC-03	<b>Pagination</b>
<b>Description</b>	A website visitor can view a list of products on the website. If there are more than 10 products, they are displayed in a pages, with first, previous, next, last buttons.
<b>Preconditions</b>	Preconditions in <i>View Data</i> .
<b>Actors</b>	Visitor, Server
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. The Visitor requests a list of products in the database.</li> <li>2. The Server connects to the database.</li> <li>3. There is more than 10 items; the Server displays the first 10 items, along with buttons to navigate forwards and backwards.</li> <li>4. The Visitor selects the navigation button to go forwards or backwards: <ol style="list-style-type: none"> <li>(a) The Server changes the range of products currently displayed.</li> <li>(b) The Server displays the new list to the Visitor, with the navigation buttons updated to show the new position.</li> </ol> </li> </ol>
<b>Postconditions</b>	
<b>Exceptions</b>	<ol style="list-style-type: none"> <li>1. Exceptions in <i>View Data</i>.</li> <li>2. The Visitor tries to navigate outside of the valid range; the result is restricted to the nearest bound page (i.e. first page or last page).</li> <li>3. The list of products change whilst the user is browsing the pages; the Server ignores the change.</li> </ol>
<b>Comments</b>	<ol style="list-style-type: none"> <li>1. Web applications may refresh the list and navigation buttons using AJAX technology [121].</li> <li>2. If the list of products may change, the Server may wish to inform the user, store the initial results in memory, or display the updated list anyway.</li> </ol>

UC-04	<b>User Action Auditing</b>
<b>Description</b>	For auditing purposes, any edit or delete a user makes to a persistent data store or object will be logged to a logging table.
<b>Preconditions</b>	<ol style="list-style-type: none"> <li>1. A persistent data store or object is marked as auditable.</li> <li>2. There is a logging data store to populate with auditing information.</li> <li>3. The User is authenticated.</li> </ol>
<b>Actors</b>	User, Server
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. The User performs an action.</li> <li>2. The Server requests to edit or delete a value in the data store or object.</li> <li>3. The Server makes a copy of the change in the database in the logging data store, along with the current User details and the time.</li> <li>4. The original data store or object is edited or deleted, and execution resumes.</li> </ol>
<b>Postconditions</b>	
<b>Exceptions</b>	
<b>Comments</b>	<ol style="list-style-type: none"> <li>1. The “copy of the change” may be an entire copy, or only the delta change (<i>diff</i>).</li> <li>2. Auditing the actions of anonymous users (Visitors) could also be supported.</li> <li>3. Systems tend to lack support for modelling cross cutting concerns like these; this could be implemented through an aspect-oriented approach on the design level with stereotypes/naming patterns, or through an implementation-level solution such as Java Servlet filters.</li> </ol>

UC-05	<b>Debug Mode</b>
<b>Description</b>	For debugging purposes, complex operations may have support for debug statements interspersed in the code.
<b>Preconditions</b>	<ol style="list-style-type: none"> <li>1. Debugging statements are dispersed throughout the application.</li> <li>2. Administrators can turn on/off debug mode.</li> <li>3. Administrators can change the current level of debug messages recorded.</li> <li>4. A debugging log exists.</li> </ol>
<b>Actors</b>	Administrator, Server
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. The Administrator turns on the Debug mode.</li> <li>2. For every debugging/logging statement in the code executed by the Server, <ol style="list-style-type: none"> <li>(a) If the debug statement is covered by the current debug level,</li> <li>(b) The statement is logged into the debug log by the Server along with the current time.</li> </ol> </li> <li>3. This continues until the Administrator turns off the Debug mode.</li> </ol>
<b>Postconditions</b>	
<b>Exceptions</b>	<ol style="list-style-type: none"> <li>1. The debugging table does not exist; it is created or an error message is sent to the Administrator and debugging mode is temporarily turned off.</li> <li>2. the debugging table cannot be written to; an error message is displayed to the Administrator and debugging mode is temporarily turned off.</li> </ol>
<b>Comments</b>	<ol style="list-style-type: none"> <li>1. Debugging may slow down execution.</li> <li>2. May present lifecycle issues: How to toggle debug mode without restarting the application?</li> <li>3. See <i>User Action Auditing</i> for an alternative debugging method.</li> </ol>

UC-06	<b>Server Transaction Support</b>
<b>Description</b>	A bank customer uses a web application to transfer money. The transaction should only succeed if every aspect of the transfer succeeded; otherwise the transaction should be rolled back.
<b>Preconditions</b>	The system is in a consistent state; the web application represents a banking application.
<b>Actors</b>	User, Server
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. The User submits a transfer request.</li> <li>2. The Server starts the transaction.</li> <li>3. The Server deducts the amount from the source account.</li> <li>4. The Server adds the amount to the destination account.</li> <li>5. The operation is logged to a logging table (optional).</li> <li>6. The Server checks to make sure all operations succeeded.</li> <li>7. The Server commits the transaction.</li> </ol>
<b>Postconditions</b>	The system is in a consistent state.
<b>Exceptions</b>	<ol style="list-style-type: none"> <li>1. Step 3, 4 or 5 fail; the transaction is rolled back.</li> <li>2. The operation does not reach to Step 7; the transaction is rolled back after a specified timeout time.</li> </ol>
<b>Comments</b>	Many web applications are front-ends for transactional systems.

UC-07	<b>Local Data Storage</b>
<b>Description</b>	A shopping cart is implemented on a User's client, with the shopping cart data submitted to the Server only at checkout.
<b>Preconditions</b>	The User is browsing a list of products (see <i>View Data</i> ); the Client can store data locally.
<b>Actors</b>	User, Client, Server
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>For each product the User wants to purchase, <ol style="list-style-type: none"> <li>The User requests to add the product to their shopping cart.</li> <li>The Client stores the product addition to a local copy of the shopping cart. This data is stored throughout the entire session and not transmitted to the Server.</li> <li>The User can continue to purchase more products.</li> </ol> </li> <li>The User requests to checkout their shopping cart.</li> <li>The Client submits the stored shopping cart data to the Server.</li> <li>The Server takes the data stored from this shopping cart and charges it to their account.</li> </ol>
<b>Postconditions</b>	
<b>Exceptions</b>	<ol style="list-style-type: none"> <li>The application loses network connectivity; the shopping cart will likely be lost. See also <i>Persistent Client Data</i>.</li> </ol>
<b>Comments</b>	<ol style="list-style-type: none"> <li>For client-side storage, technologies such as Javascript memory, cookies, Java applets, URL rewriting and browser filesystem storage could be used. This should be transparent to all actors involved.</li> <li>This could be modelled with stereotypes such as "server", "client", "cacheable" etc.</li> <li>Important if client has unreliable connection (e.g. ad hoc/wireless network).</li> <li>Important if high level of system availability is required (e.g., companies outsourcing office applications to web based providers).</li> <li>Technology wise, there are some development aiming at this feature, e.g. Dojo Toolkit [81], Google Gears [129].</li> <li>One potential implementation of this use case is discussed in <i>Store Data in Local Database</i>.</li> </ol>

UC-08	<b>Server Data Access</b>
<b>Description</b>	The User can download an e-mail data object (like an e-mail draft) from the server to their client machine to work on.
<b>Preconditions</b>	There is an existing e-mail data object to work on, that exists in a database.
<b>Actors</b>	User, Server
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. The User opens an editable page for the e-mail.</li> <li>2. The Client downloads the e-mail object from the Server.</li> <li>3. The User edits the e-mail without recontacting the Server; this is achieved by editing the object in Client memory.</li> <li>4. The User is finished editing the e-mail, so submits it back to the Server.</li> <li>5. The Client submits the changed object data to the Server.</li> <li>6. The Server reconstructs the object based on the Client data and saves it back to the database.</li> </ol>
<b>Postconditions</b>	The e-mail object is changed and saved back to the database.
<b>Exceptions</b>	<ol style="list-style-type: none"> <li>1. The browser or active Internet connection goes offline while the user is editing the e-mail; the User should be informed.</li> </ol>
<b>Comments</b>	<ol style="list-style-type: none"> <li>1. This is different to a User working on data stored exclusively on their local machine, as in <i>Persistent Client Data</i>.</li> <li>2. Data accessed locally can be automatically saved through <i>Client Timer Support</i>.</li> <li>3. See also <i>Local Data Storage</i>.</li> </ol>

UC-09	<b>Persistent Client Data</b>
<b>Description</b>	An e-mail can be worked on, over many sessions, with the data staying persistent on the client's device.
<b>Preconditions</b>	
<b>Actors</b>	User, Client, Server
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. The User starts writing an e-mail on their Client.</li> <li>2. The User closes the editing session.</li> <li>3. The Client saves a copy of the e-mail to some storage device.</li> <li>4. After some time, the User re-opens the e-mail editing session.</li> <li>5. The Client opens the saved copy and resumes editing.</li> <li>6. The User finishes their changes and submits the final copy to the Server for delivery.</li> <li>7. The local copy of the e-mail is removed.</li> </ol>
<b>Postconditions</b>	
<b>Exceptions</b>	<ol style="list-style-type: none"> <li>1. After an extended period of time (e.g. a year) the data may be lost.</li> <li>2. Stored data may be deleted by the User themselves (e.g. clearing cookies).</li> </ol>
<b>Comments</b>	<ol style="list-style-type: none"> <li>1. Like in <i>Store Data in Local Database</i>, this should be transparent to the actors involved.</li> <li>2. Could be implemented through technologies mentioned in <i>Store Data in Local Database</i>.</li> <li>3. See <i>Local Data Storage</i> for client storage comments.</li> <li>4. E-mail delivery is covered in <i>E-mailing Users</i>.</li> </ol>

UC-10	<b>Temporary Server Data</b>
<b>Description</b>	A user is working on an image editing client; the image needs to be stored on the server, but only temporarily during the editing session.
<b>Preconditions</b>	The User can upload an image to the Server.
<b>Actors</b>	User, Server
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. The User uploads an image to work on.</li> <li>2. The Server saves this image to a temporary storage device.</li> <li>3. The User performs editing operations on the image (e.g. rotate, flip, add text, resize), which may take a long time and transit over many pages.</li> <li>4. The Server keeps the changes and original images in temporary storage.</li> <li>5. Once the User is finished, the final image is sent via e-mail.</li> <li>6. Once the session is over, or a new image is uploaded, the temporary data is deleted or marked for deletion.</li> </ol>
<b>Postconditions</b>	The image is not stored on the Server.
<b>Exceptions</b>	<ol style="list-style-type: none"> <li>1. The editing session is disconnected; the User may permanently lose the temporary image, or the session could be automatically restored (<i>Restore Server Session</i>).</li> </ol>
<b>Comments</b>	<ol style="list-style-type: none"> <li>1. The temporary server data could be stored either in files, or in a database.</li> <li>2. The temporary data should be able to persist over server reboots, though this may incur a performance penalty.</li> <li>3. E-mail delivery is covered in <i>E-mailing Users</i>.</li> </ol>



UC-11	<b>Uploading Files</b>
<b>Description</b>	To send a file via e-mail, the user can upload a file to the server, which is temporarily stored before being sent.
<b>Preconditions</b>	The User can upload a file to the Server.
<b>Actors</b>	User, Server
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. The User is creating an e-mail to send.</li> <li>2. The User attaches a file to the e-mail, by uploading a file to the Server.</li> <li>3. The Server downloads the file from the User and saves it in a temporary location.</li> <li>4. The Server informs the User the file has been attached and is ready to send.</li> <li>5. The User submits the completed e-mail to the Server.</li> <li>6. The Server attaches the file(s) to the e-mail and sends the e-mail.</li> </ol>
<b>Postconditions</b>	
<b>Exceptions</b>	<ol style="list-style-type: none"> <li>1. The temporarily uploaded file is removed unexpectedly; the System displays an error message to the User, and has to re-upload their file before they can continue sending the e-mail.</li> <li>2. The uploaded file cannot be stored; the System displays an error message to the User, and sends an error message to the Administrator.</li> <li>3. The uploaded file is of an invalid or unexpected type; the System displays an error message to the User, and the temporary file is removed.</li> <li>4. The uploaded file is too big; the System displays an error message to the User, and the temporary upload data is discarded.</li> </ol>
<b>Comments</b>	<ol style="list-style-type: none"> <li>1. The file must be able to persist over server reboots in this situation; this addresses fault tolerance and availability.</li> <li>2. It may be useful to add a progress dialog to the file upload; this could be accomplished by another service, Ajax + Javascript, etc</li> <li>3. E-mail delivery is covered in <i>E-mailing Users</i>.</li> </ol>

UC-12	<b>Restore Server Session</b>
<b>Description</b>	While working on an image on the server, the User's session is disconnected; once the User returns, it is possible to continue the previous session.
<b>Preconditions</b>	The User is authorised.
<b>Actors</b>	User, Server
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. The User is working on an image manipulation program (see <i>Temporary Server Data</i>).</li> <li>2. The session is disconnected prematurely, for example a power failure on the User's machine.</li> <li>3. The User returns to the website and re-authorises themselves.</li> <li>4.</li> <li>5. The User requests to resume the existing session.</li> <li>6. The User continues using the application with the old session data.</li> </ol>
<b>Postconditions</b>	The User is still authorised, and is using the same session data.
<b>Exceptions</b>	<ol style="list-style-type: none"> <li>1. The User requests to save the session data; the old session data is saved into permanent storage for access later, and a new session is started.</li> <li>2. The User requests the session data is destroyed; the old session is removed and a new session is started.</li> <li>3. The old session is destroyed before the session can be resumed; the session data will be lost.</li> <li>4. The old session is corrupted or loading the session would result in an inconsistent system state; the User is informed that they cannot reload it and must start a new one.</li> </ol>
<b>Comments</b>	<ol style="list-style-type: none"> <li>1. Step 4 could be enhanced with a "preview existing sessions" step, allowing the User to specify which sessions to load or remove. This process would be similar to Microsoft Word's document recovery process.</li> <li>2. There would probably have to be a time limit to how long sessions could be retained. Lost sessions could persist over server reboots, if required.</li> <li>3. Some session data may not be suitable for recovery, for example, temporary authorisation with a bank transfer.</li> <li>4. (This use case is different to Mozilla Firefox's "Restore Session" functionality; Firefox does not actually restore the sessions, only the pages that were in use. This use case is server-based.)</li> <li>5. This would be a way to load and save data from sessions active on multiple machines.</li> </ol>

UC-13	<b>User Authorisation</b>
<b>Description</b>	A Visitor can log in with certain credentials to become an authorised User.
<b>Preconditions</b>	A Visitor has a session (see <i>Session Support</i> ); there is a list of existing Users in the application.
<b>Actors</b>	Visitor, User, Server
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. A Visitor visits a website, and goes to the “login” page.</li> <li>2. The Visitor enters in a username and password, and submits the form to the Server.</li> <li>3. The Server checks their username and password against the list of existing user accounts, and verifies that the account exists.</li> <li>4. The Server updates the Visitor’s session to indicate that they are an authorised User.</li> <li>5. The Visitor becomes a User, and is redirected to a User-only page.</li> </ol>
<b>Postconditions</b>	
<b>Exceptions</b>	<ol style="list-style-type: none"> <li>1. the entered credentials are invalid; the Visitor is informed and is prevented from continuing.</li> <li>2. The User account found is blocked from authorising; an error message is displayed to the Visitor.</li> <li>3. The Visitor loses its session while authorising; a new Session is created, or an error message is displayed to the Visitor, informing them they cannot currently login.</li> </ol>
<b>Comments</b>	<ol style="list-style-type: none"> <li>1. This is a basic requirement of database-driven websites.</li> <li>2. If the user cannot login, the Visitor can either register a new account, or retrieve their lost information (e.g. <i>Password Reset</i>).</li> <li>3. This is usually not directly modelled in web applications but provided as service.</li> <li>4. However, the access rights controlling other resources must be defined.</li> <li>5. The registration of new User accounts is covered in <i>Account Registration</i>.</li> <li>6. User authentication can also occur through <i>Basic</i> or <i>Digest</i> HTTP authentication.</li> </ol>

UC-14	<b>Password Reset</b>
<b>Description</b>	A user cannot login with their given credentials; they know their username, but not their password. They choose to reset their password.
<b>Preconditions</b>	The user account has an associated e-mail address.
<b>Actors</b>	Visitor, Server
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. The Visitor asks the Server to reset their password for a given username.</li> <li>2. The Server resets the User's password to a random password.</li> <li>3. The Server e-mails this random password to the User's e-mail account.</li> <li>4. The Visitor receives this e-mail, and uses this new password to login as a User, using <i>User Authorisation</i>.</li> <li>5. The User is asked to change their password before they can continue with their session.</li> </ol>
<b>Postconditions</b>	
<b>Exceptions</b>	<ol style="list-style-type: none"> <li>1. The password e-mail cannot be sent; an error message is displayed to the User, and the Administrator is notified of the problem.</li> <li>2. The User does not change their password in Step 5; this step will repeat before <i>User Authorisation</i> can complete.</li> </ol>
<b>Comments</b>	<ol style="list-style-type: none"> <li>1. The "random password" may be implemented through a secret link which does not affect the user account's password at all.</li> <li>2. In Step 2, the Server can request for additional account information, e.g. asking a secret question stored with the account.</li> </ol>

UC-15	<b>Session Support</b>
<b>Description</b>	Visitors to the website are tracked over the stateless HTTP protocol with an implementation of sessions.
<b>Preconditions</b>	
<b>Actors</b>	Visitor, User, Server
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. The Visitor visits the web application for the first time.</li> <li>2. The Server creates a new session for the Visitor, and links this session to the current Visitor.</li> <li>3. After a period of inactivity, the session is lost.</li> </ol>
<b>Postconditions</b>	
<b>Exceptions</b>	<ol style="list-style-type: none"> <li>1. The Visitor cannot be tracked using existing session technologies; an error message is displayed to the Visitor and cannot continue using the web application.</li> <li>2. The Visitor is authenticated as a User when the session is lost (Step 3); the Visitor will have to re-authenticate as a User (<i>User Authorisation</i>).</li> </ol>
<b>Comments</b>	Sessions can be implemented through many technologies, such as cookies, IP tracking, or URL rewriting. The choice of technology should be transparent to the actors.

UC-16	<b>Account Registration</b>
<b>Description</b>	To use a shopping account properly, a Visitor needs to register their account details with the website to create a User account.
<b>Preconditions</b>	<ol style="list-style-type: none"> <li>1. Visitors can create new User accounts.</li> <li>2. User accounts can be stored in a data store.</li> <li>3. The current Visitor is not authenticated as a User (<i>User Authorisation</i>).</li> </ol>
<b>Actors</b>	Visitor, User, Server
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. The Visitor requests to create a new User account.</li> <li>2. The Server responds with an application form, which includes fields for e-mail address, username, and password.</li> <li>3. The Visitor enters in their details and submits it to the Server.</li> <li>4. The Server creates the new User account, and marks it to disabled. The User is e-mailed instructions on how to activate their account.</li> <li>5. The Visitor receives the e-mail instructions, and follows the instructions to activate their account.</li> <li>6. The Server marks the account as enabled, and redirects the Visitor to the login page (<i>User Authorisation</i>).</li> </ol>
<b>Postconditions</b>	A new User account has been created.
<b>Exceptions</b>	<ol style="list-style-type: none"> <li>1. Invalid email address is submitted; the Visitor is informed of the error and must correct the application form.</li> <li>2. Email cannot be delivered; the Visitor is either notified immediately, or the registration is ignored.</li> <li>3. The given username (or email address) already exists; the Visitor is informed of the problem, and is given the option to choose a different username, or attempt to login with the username (in case they had forgotten they already had an account).</li> <li>4. The user does not follow through on the account activation; the account is automatically deleted after a period of inactivity.</li> <li>5. Erroneous account details are submitted, e.g. the wrong e-mail address; the delivered e-mail includes instructions on how to unsubscribe or cancel the account.</li> </ol>
<b>Comments</b>	<ol style="list-style-type: none"> <li>1. It is standard practice to encrypt the user password with a one-way hash with a hidden salt.</li> <li>2. It is becoming acceptable for e-mail addresses to fulfill the role of usernames.</li> <li>3. It is becoming less acceptable to force registration on users for trivial actions like downloading and submitting orders.</li> <li>4. For some websites, forced account activation (steps 4-6) may not be necessary; this is generally only used to reduce spam or fake accounts.</li> </ol>

UC-17	<b>Automatic User Authorisation</b>
<b>Description</b>	Instead of having to enter in their user details on every website visit, they are automatically logged in.
<b>Preconditions</b>	Preconditions in <i>User Authorisation</i> .
<b>Actors</b>	User, Client, Server
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. As the User is authenticating (<i>User Authorisation</i>), they select an option to automatically log in (“remember me”).</li> <li>2. The Server stores re-authentication data on the Client.</li> <li>3. The User leaves the website; the session is closed, but the stored data remains.</li> <li>4. The User returns to any page on the website.</li> <li>5. The Server automatically detects that the User has re-authorisation data; this information is verified on the Server, and if successful, the user is automatically re-authorised.</li> </ol>
<b>Postconditions</b>	Postconditions in <i>User Authorisation</i> .
<b>Exceptions</b>	<ol style="list-style-type: none"> <li>1. The re-authorisation data is incorrect; the user is not logged in, the re-authorisation data is removed, and an error message is displayed to the Visitor.</li> <li>2. The re-authorisation data cannot be stored on the Client; an error message is displayed to the Visitor.</li> </ol>
<b>Comments</b>	<ol style="list-style-type: none"> <li>1. Re-authorisation data is usually stored in cookies.</li> <li>2. When the user is automatically logged in, a message is displayed once to the User, informing they have automatically been logged in.</li> <li>3. A more modern method of storing this data is not to store the username and password on the server; but instead, store a “login key” on both the server and the client, which is used to verify the login. This also has the added benefit of surviving password changes, but has the problem of reduced security of access from compromised machines, unless existing login keys can be revoked.</li> <li>4. Generally sites using automatic authorisation also re-request for the user’s password before embarking on critical operations, such as changing user account details or deleting their account.</li> <li>5. Some sites, such as banking sites, should never have automatic authorisation enabled.</li> </ol>

UC-18	<b>Static Views (HTML)</b>
<b>Description</b>	A calendar web application wishes to publish an HTML-only version of a calendar.
<b>Preconditions</b>	The calendar page can be represented purely in HTML.
<b>Actors</b>	Visitor, Server
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. A Visitor requests the HTML-only version of a calendar page.</li> <li>2. The Server constructs the page by removing all scripting from the page.</li> <li>3. The Visitor receives the HTML-only page.</li> <li>4. For a User to add events to the calendar, <ol style="list-style-type: none"> <li>(a) The User clicks the Add Event button, which loads the event entry page.</li> <li>(b) The User fills in the form, with no help from Javascript or dynamic technologies.</li> <li>(c) The User submits the form, and the Server saves the new event.</li> </ol> </li> </ol>
<b>Postconditions</b>	
<b>Exceptions</b>	
<b>Comments</b>	<ol style="list-style-type: none"> <li>1. Useful for users which do not have Javascript enabled, or old/slow/incompatible browsers.</li> <li>2. It may be probably possible to convert any rich features to HTML?</li> <li>3. If necessary, the Server could emulate a user session on the server itself. An emulated session could assist in development of features that are not globally supported, e.g. ActiveX features emulated for Firefox users, or HTML 5 Canvas features [343] emulated for older versions of Internet Explorer.</li> <li>4. May be related to <i>Multiple Browser Support</i>.</li> <li>5. See also <i>Backwards-Compatible Scripting</i>.</li> </ol>



UC-19	<b>Asynchronous Form Validation</b>
<b>Description</b>	Form validation can occur via asynchronous communication with the server; see <i>Client Form Validation</i> for a Client-only version.
<b>Preconditions</b>	Preconditions in <i>Client Form Validation</i> .
<b>Actors</b>	Visitor, Client, Server
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. The Visitor is filling out an entry field in a form (e.g. an e-mail address in a registration form).</li> <li>2. Once the field is filled out, the Client submits the value to the Server for validation.</li> <li>3. If the Server finds the input is invalid (e.g. the address is malformed), it returns the result to the Client.</li> <li>4. The Client graphically highlights the field, and displays a message next to it informing the Visitor the input is invalid, why it is invalid, and how to fix it (“this e-mail address already exists in our database; would you like to login instead?”).</li> <li>5. The Visitor corrects the error (e.g. a valid email address).</li> <li>6. The input is submitted again to the Server, and it is now correct; the field highlight and error message is removed.</li> </ol>
<b>Postconditions</b>	Postconditions in <i>Client Form Validation</i> .
<b>Exceptions</b>	<ol style="list-style-type: none"> <li>1. The corrected value (Step 6) is incorrect; the process goes back to Step 3.</li> <li>2. The Client cannot contact the Server for validation; a message may be displayed to the Visitor, but validation should also occur on the Server after submission.</li> </ol>
<b>Comments</b>	<ol style="list-style-type: none"> <li>1. See the comments for the <i>Client Form Validation</i> use case.</li> <li>2. Care must be taken to ensure that the connection is not saturated with validation requests (e.g. validating on every key stroke).</li> <li>3. In some cases (e.g. unique usernames), the server result could be cached locally, so multiple inputs of the same data does not re-poll the server.</li> <li>4. Sometimes, this depends on the capability of the client, and it is desirable to decide where to validate a form at request time.</li> <li>5. Even though validation may occur on the client, it still <b>must</b> occur on the server before any data is retained.</li> </ol>

UC-20	<b>Client Form Validation</b>
<b>Description</b>	Fields can be validated on the Client via Javascript.
<b>Preconditions</b>	There is a field that needs to have a valid value.
<b>Actors</b>	Visitor, Client
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. The Visitor is filling out a destination e-mail address form (sending an e-mail).</li> <li>2. Once the field is filled out, the e-mail address is parsed on the Client for validity (perhaps through regular expressions).</li> <li>3. If the address is invalid: <ol style="list-style-type: none"> <li>(a) The field is highlighted and an error message is displayed, and possibly a way to fix the problem ("this e-mail address is invalid; there is no @ character").</li> <li>(b) The Visitor goes back to the invalid field and corrects the problem.</li> <li>(c) The Client re-checks the field for validity, and it is valid.</li> <li>(d) The highlight around the field is removed, and the error message is removed.</li> </ol> </li> </ol>
<b>Postconditions</b>	The field has a valid value.
<b>Exceptions</b>	<ol style="list-style-type: none"> <li>1. The Client does not have Javascript enabled, or validation cannot occur; an error message is displayed to the User, and validation must occur on the Server (<i>Server Form Validation</i>).</li> </ol>
<b>Comments</b>	<ol style="list-style-type: none"> <li>1. See the comments for the <i>Server Form Validation</i> use case.</li> <li>2. All Client-side operations and validation must be treated as insecure, and all validation must be re-executed on the Server.</li> <li>3. In some cases, it may be appropriate to disable sending the form until all attributes in the form are valid.</li> <li>4. The client and server form validations could be integrated together into one set of rules.</li> <li>5. The key difference between this use case and <i>Client Form Validation</i> is that this use case only occurs once the entire form is completed.</li> </ol>

UC-21	<b>Server Form Validation</b>
<b>Description</b>	When a form is submitted to the Server, the Server can assess the form for validity; if it is invalid, the form is redisplayed to the Visitor with the errors highlighted.
<b>Preconditions</b>	There is a field in a form that needs to have a valid value.
<b>Actors</b>	Visitor, Server
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. The Visitor is presented with a user registration form.</li> <li>2. The Visitor fills out the form with incorrect data and submits it to the server.</li> <li>3. The Server validates all the inputs of the form, and recognises that some of the inputs are invalid (e.g. malformed e-mail address, illegal username).</li> <li>4. The Server redirects the Visitor back to the form entry page, but highlights all the incorrect fields, and displays a list of problems with the submission.</li> <li>5. The Visitor corrects all the values on the form, and resubmits the form to the Server.</li> <li>6. The Server re-validates all the inputs of the form, and recognises that they are all valid. The registration is saved to the database.</li> </ol>
<b>Postconditions</b>	The field has a valid value.
<b>Exceptions</b>	<ol style="list-style-type: none"> <li>1. The Server cannot carry out the validation; an error message is displayed to the User and Administrator, and execution is not allowed to continue.</li> </ol>
<b>Comments</b>	Instead of simply saying “this input is invalid”, the Server could respond with a list of possibilities (“did you mean X?”).

UC-22	<b>Multiple Browser Support</b>
<b>Description</b>	The application can be rendered identically to multiple browsers, or take advantage of browser-specific features.
<b>Preconditions</b>	
<b>Actors</b>	Client, Server
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. The Client requests a page.</li> <li>2. The Server identifies the browser from the user request, in this case Internet Explorer 6: <ol style="list-style-type: none"> <li>(a) The Server constructs a page response that is standard to all target browsers.</li> <li>(b) The Server applies fixes to solve the differences in rendering techniques, for example Internet Explorer 6's incorrect CSS <i>float</i> model.</li> <li>(c) The Server sends this result to the Client.</li> </ol> </li> </ol>
<b>Postconditions</b>	
<b>Exceptions</b>	<ol style="list-style-type: none"> <li>1. No user agent can be identified; it sends the standard page response (Step 2a).</li> <li>2. The user agent is incorrectly identified by the Server; there should be a way for the Visitor to change the target browser, but it should be as transparent as possible.</li> </ol>
<b>Comments</b>	<ol style="list-style-type: none"> <li>1. The current user agent can be identified using the standard HTTP headers, or inline Javascript/CSS code.</li> <li>2. This could be cleanly implemented with a conditional processing instructions [194].</li> <li>3. This could also lend itself to multiple output formats, such as XML + XSL (client-side), the different flavours of XHTML [345, 346] and HTML [333, 343], and others.</li> <li>4. This technique can be useful to handle different mobile phone clients (<i>Mobile Phone Support</i>).</li> </ol>

UC-23	<b>Mobile Phone Support</b>
<b>Description</b>	Website visitors using a mobile phone can automatically be redirected to a design specifically created for mobile users.
<b>Preconditions</b>	The Visitor is visiting a normal web application with a mobile phone; a web application designed for mobile phones exist.
<b>Actors</b>	Visitor, Server
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. The Visitor requests a page on the website.</li> <li>2. The Server realises the user is on a mobile phone.</li> <li>3. The Server redirects them to the mobile phone version of the same web application.</li> </ol>
<b>Postconditions</b>	The Visitor is browsing the web application designed for mobile phones.
<b>Exceptions</b>	
<b>Comments</b>	<ol style="list-style-type: none"> <li>1. Identification of mobile phones can be achieved in the same way as <i>Multiple Browser Support</i>.</li> <li>2. Mobile pages tend to be smaller with less content than PC equivalents.</li> <li>3. Mobile pages also tend to have less graphics and limited Javascript support.</li> <li>4. The Visitor should be able to disable this automatic redirection if desired.</li> <li>5. The use case <i>Multiple Browser Support</i> can help rendering the same mobile site to multiple implementations of mobile platforms.</li> </ol>

UC-24	<b>Remote Data Source</b>
<b>Description</b>	A calendar application has been asked to import another calendar (in iCal format) from an external web server.
<b>Preconditions</b>	The application can import iCal calendar sources; calendar events can be stored in a local database.
<b>Actors</b>	User, Server, Remote Server
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. The User has requested to import an external iCal file.</li> <li>2. The Server contacts the Remote Server and requests the iCal file.</li> <li>3. The Server downloads and parses the iCal file.</li> <li>4. For each event identified, <ol style="list-style-type: none"> <li>(a) The Server checks to see if the event already exists in the local database.</li> <li>(b) If the event does not exist, the Server adds the event to the local database.</li> </ol> </li> </ol>
<b>Postconditions</b>	The remote calendar has been loaded into the local database.
<b>Exceptions</b>	<ol style="list-style-type: none"> <li>1. The Server cannot access the Remote Server; an error message is displayed to the User.</li> <li>2. The iCal file cannot be parsed; an error message is displayed to the User.</li> <li>3. A single iCal event cannot be parsed; the event is ignored, and a message is displayed to the User.</li> <li>4. An event already exists in the database; it can either be ignored, or updated with the new event data.</li> </ol>
<b>Comments</b>	<ol style="list-style-type: none"> <li>1. This does not cover removing events that no longer exist in a remote source.</li> <li>2. This process occurs only once; for accessing a remote data source regularly, see <i>Active Remote Data Source</i>.</li> </ol>

UC-25	<b>Active Remote Data Source</b>
<b>Description</b>	A calendar application has been asked to continuously import iCal events from an external web server, actively keeping the local calendar updated at all times.
<b>Preconditions</b>	<ol style="list-style-type: none"> <li>1. Preconditions in <i>Remote Data Source</i>.</li> <li>2. The Server has a registered remote data source for a given calendar.</li> <li>3. The Server has calendar data which has not been updated for more than 24 hours.</li> </ol>
<b>Actors</b>	Server, Remote Server
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. The Server locates the remote data source.</li> <li>2. The Server uses <i>Remote Data Source</i> to update the data.</li> <li>3. The Server marks the current time as the last calendar update.</li> </ol>
<b>Postconditions</b>	Postconditions in <i>Remote Data Source</i> .
<b>Exceptions</b>	1. Exceptions in <i>Remote Data Source</i> .
<b>Comments</b>	<ol style="list-style-type: none"> <li>1. See <i>Remote Data Source</i>.</li> <li>2. In some cases it can be useful to force a data refresh outside of the regular interval.</li> <li>3. For data which rarely changes, the Server could automatically reduce its access interval.</li> <li>4. Multiple requests to the same data source could be cached locally.</li> <li>5. This process could be aggregated into one automated process/server.</li> </ol>

UC-26	<b>Data Feeds</b>
<b>Description</b>	The application can provide external data feeds about new blog entries to external clients/servers via RSS.
<b>Preconditions</b>	Blog entries can be fed through an RSS feed [290]
<b>Actors</b>	Server, External Server
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. An External Server requests an RSS feed of the most recent blog entries.</li> <li>2. The Server compiles an RSS feed of the most recent blog entries and returns it to the External Server.</li> </ol>
<b>Postconditions</b>	
<b>Exceptions</b>	<ol style="list-style-type: none"> <li>1. The RSS feed is requested too many times by one client; the client is automatically denied access for a few hours, and the administrator is notified, to prevent denial of service attacks.</li> </ol>
<b>Comments</b>	<ol style="list-style-type: none"> <li>1. This could be generalised into a specific type of <i>Web Service</i> request.</li> <li>2. Some data feeds could use emulated user permissions, see below.</li> <li>3. Modelling may support RSS as general mechanism, i.e. to have modelling elements that can extract RSS summaries from resources, and to provide an abstraction from concrete RSS dialects (RSS 1.0, RSS 2.0, ATOM). This needs to be customisable, w.r.t. display, access control, etc.</li> </ol>

UC-27	<b>Web Service</b>
<b>Description</b>	The application can provide external functionality support to its services via web service communication protocols such as SOAP [340], such as converting currency.
<b>Preconditions</b>	A web service API is published for the local web application; the web application has exchange rates between US\$ and Yen.
<b>Actors</b>	Server, External Server
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. An External Server requests a web service function call via SOAP, to convert US \$50 to Yen.</li> <li>2. The Server retrieves its exchange rate for US\$ and Yen, and converts this into a number.</li> <li>3. The Server responds to the External Server with the result using the SOAP framework.</li> </ol>
<b>Postconditions</b>	
<b>Exceptions</b>	<ol style="list-style-type: none"> <li>1. The External Server does not have the permissions to call this web service; the request is denied.</li> </ol>
<b>Comments</b>	<ol style="list-style-type: none"> <li>1. If this web service also calls a web service itself, we must make sure to prevent infinite loops of web service calls.</li> <li>2. Alternative web service communication protocols includes JSON [65] and the XML-RPC protocol.</li> </ol>

UC-28	<b>Back/Forwards Button Control</b>
<b>Description</b>	The application can control where the back and forward buttons go, even on applications which do not change the browser history by default.
<b>Preconditions</b>	The User is using an e-mail application which is entirely rendered on the client.
<b>Actors</b>	User, Client
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. The User is in the e-mail inbox page.</li> <li>2. The User clicks on an e-mail, which replaces the view with the e-mail contents.</li> <li>3. The Client adds the e-mail inbox page to the browser history.</li> <li>4. The User clicks on the reply link, which adds a reply box to the bottom of the e-mail, and redirects focus to the bottom of the email.</li> <li>5. The Client adds the e-mail page to the browser history.</li> <li>6. The User enters in a reply.</li> <li>7. The User clicks the inbox link.</li> <li>8. The User redirected to the e-mail inbox page.</li> </ol>
<b>Postconditions</b>	
<b>Exceptions</b>	<ol style="list-style-type: none"> <li>1. Step 4, the user clicks Back; the User is taken to the e-mail inbox page, and the e-mail page is the next page.</li> <li>2. Step 6, the user clicks Back; the User is taken to the e-mail page, and the reply box is shown in the next page.</li> <li>3. Step 8, the user clicks Back; the User is taken to the e-mail page, with the last reply still in the content field. The e-mail inbox is shown in the next page, but the user is asked if they want to lose their changes if they go forwards.</li> </ol>
<b>Comments</b>	<ol style="list-style-type: none"> <li>1. If the User ended up sending the e-mail (after Step 6), going back to the reply box would have presented an empty reply.</li> <li>2. An application should always have back button support; disabling this button causes great user stress.</li> <li>3. Sometimes going back may require either the old view to be displayed (writing a reply), or the view refreshed (e-mail inbox), depending on the situation.</li> </ol>

UC-29	<b>Opening New Windows</b>
<b>Description</b>	The User can open links in new windows, even if the application is solely rendered via the Client.
<b>Preconditions</b>	
<b>Actors</b>	User, Client
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. The User is in the e-mail inbox page.</li> <li>2. The User clicks on an e-mail, but asks for it to be opened in a new tab/window.</li> <li>3. The e-mail is displayed in the new tab/window by the Client, with no previous browser history.</li> </ol>
<b>Postconditions</b>	
<b>Exceptions</b>	<ol style="list-style-type: none"> <li>1. The browser cannot open a page in a new window or tab; the page is opened in the current window instead, and the browser history is affected accordingly (see <i>Back/Forwards Button Control</i>).</li> </ol>
<b>Comments</b>	<ol style="list-style-type: none"> <li>1. Some links should not be allowed to open in new windows (e.g. actions), but these should not be rendered as text links, but as buttons.</li> <li>2. Also refer to <i>Pop-up Window Support</i>.</li> </ol>



UC-30	<b>Client-Side Application</b>
<b>Description</b>	The entire application user interface can be rendered by Javascript and XML by the Client, instead of HTML from the Server.
<b>Preconditions</b>	
<b>Actors</b>	User, Client, Server
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. The User requests the e-mail website.</li> <li>2. The Client downloads the Javascript to render the page.</li> <li>3. The Client executes the Javascript, which downloads an XML file of inbox e-mails.</li> <li>4. The Client draws the user interface, using the information from the XML.</li> <li>5. The User can interact with this user interface like any normal website, except most operations are run on the Client, not on the Server.</li> </ol>
<b>Postconditions</b>	
<b>Exceptions</b>	<ol style="list-style-type: none"> <li>1. The Javascript cannot be executed, or the user interface cannot be rendered in this browser; the User is redirected to a non-Javascript page.</li> <li>2. The XML cannot be loaded; the User is redirected to a non-Javascript page.</li> <li>3. The loading process takes too long; the User is redirected to a non-Javascript page.</li> </ol>
<b>Comments</b>	<ol style="list-style-type: none"> <li>1. This is not an example of a browser displaying XML+XSL; it is the use of Javascript and XML data islands to construct an HTML DOM in memory at run time on the client.</li> <li>2. Requires support for the <i>Back/Forwards Button Control</i> use case.</li> <li>3. This naturally tends towards storing some data on the Client side, and operating on it there; see <i>Local Data Storage</i>.</li> </ol>

UC-31	<b>Communication with Software</b>
<b>Description</b>	A web application can send messages to external software applications.
<b>Preconditions</b>	An e-mail site has a chat service; an external software application is also connected to this chat service.
<b>Actors</b>	User, Server, Software
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. A User is logged into a chat program, with the desktop Software.</li> <li>2. On the Server, the User receives a new e-mail message.</li> <li>3. The Server sends a notification to the desktop Software about the new e-mail, which renders as a pop-up box.</li> <li>4. The User clicks on the notification.</li> <li>5. The User is sent to a special URL using their Client, which opens in a new browser window.</li> <li>6. The Server displays the e-mail.</li> </ol>
<b>Postconditions</b>	The User is viewing the received e-mail.
<b>Exceptions</b>	<ol style="list-style-type: none"> <li>1. The Software cannot open a new browser window; an error message is displayed to the User.</li> <li>2. The authentication between the User and the Software expires before the Server can re-authenticate the User; the process is stopped at Step 6, and the User is asked to re-authenticate themselves.</li> </ol>
<b>Comments</b>	<ol style="list-style-type: none"> <li>1. Usually when the notification is clicked, the User is sent to a special URL, which is constructed to automatically authenticate the user (<i>User Authorisation</i>) if they are not already authenticated.</li> <li>2. Very common in chat programs integrated with e-mail clients, e.g. Google Talk and Gmail, or MSN Messenger and Hotmail.</li> <li>3. The notification (Step 3) could either be passive (polling) or active (an open connection is sent notifications).</li> <li>4. Passive polling can be achieved using a <i>Web Service</i> by the Software.</li> </ol>

UC-32	<b>Mobile Phone Communication</b>
<b>Description</b>	A user can get notifications via mobile phone (SMS).
<b>Preconditions</b>	<ol style="list-style-type: none"> <li>1. The web application receives e-mails.</li> <li>2. The User has a registered mobile phone number, and the number is contactable from the Server.</li> </ol>
<b>Actors</b>	User, Server, Device (mobile phone)
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. On the Server, the User receives a new e-mail.</li> <li>2. The Server sends a notification message to the User via a mobile network.</li> <li>3. The mobile network transports the notification to the User's mobile phone.</li> <li>4. The Device (mobile phone) receives the notification and displays the message to the User.</li> </ol>
<b>Postconditions</b>	
<b>Exceptions</b>	<ol style="list-style-type: none"> <li>1. The mobile network is down, or cannot be contacted; the User does not receive the notification until the network is back up, or the notification is lost.</li> <li>2. The mobile number no longer exists, or does not support messaging; the Server removes the association with the User and informs the User the next time they log in.</li> <li>3. The mobile phone is offline; the User will receive the message when they next turn on their phone.</li> </ol>
<b>Comments</b>	<ol style="list-style-type: none"> <li>1. On some networks, it may cost to send messages; the application may need to maintain some sort of user account balance.</li> <li>2. The notification message sent could include a URL, similar to <i>Communication with Software</i>, which the User can open and browse with their mobile phone.</li> </ol>

UC-33	<b>E-mailing Users</b>
<b>Description</b>	The User can be sent e-mails.
<b>Preconditions</b>	The web application supports internal messages; the User has requested to be e-mailed about new messages sent internally.
<b>Actors</b>	User, Server
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. In the web application, the User is sent a new internal message.</li> <li>2. The Server composes an e-mail to be sent to the User, including the internal message.</li> <li>3. The Server sends the e-mail.</li> <li>4. The User receives the e-mail.</li> <li>5. The User reads the internal message included.</li> </ol>
<b>Postconditions</b>	
<b>Exceptions</b>	<ol style="list-style-type: none"> <li>1. The e-mail bounces immediately (invalid address); the notification service is disabled, and the User is notified the next time they log in.</li> <li>2. The e-mail bounces after a while (cannot contact server); the message is discarded or added to a queue to retry later.</li> <li>3. The User never receives or downloads the e-mail; nothing happens.</li> </ol>
<b>Comments</b>	<ol style="list-style-type: none"> <li>1. This is a basic requirement of database-driven websites.</li> <li>2. One important aspect of sending e-mail messages is managing bouncebacks, and the lifecycle of e-mail accounts; this needs to be considered in any serious web application.</li> <li>3. Also related to <i>Mobile Phone Communication</i>, as emails are another example of an event delivery mechanism.</li> </ol>

UC-34	<b>E-mail Unsubscription</b>
<b>Description</b>	Any Visitor can unsubscribe their e-mail address from the web application, to prevent any further e-mails.
<b>Preconditions</b>	The web application can send e-mails ( <i>E-mailing Users</i> ).
<b>Actors</b>	Visitor, Server
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. The Visitor requests to be unsubscribed from all future e-mails to a particular address.</li> <li>2. The e-mail address is sent a confirmation e-mail, along with instructions on how to revert a successful unsubscription process.</li> <li>3. The Visitor clicks the link in the confirmation e-mail to disable all future e-mails.</li> <li>4. The Server adds this e-mail address to a “do not contact” list.</li> <li>5. Any future time an e-mail is being sent out: <ol style="list-style-type: none"> <li>(a) The destination address is checked against the list of people to not contact.</li> <li>(b) If the address exists in the invalid list, an error message is displayed, and the message is discarded.</li> </ol> </li> </ol>
<b>Postconditions</b>	The Visitor is never e-mailed again by this Server.
<b>Exceptions</b>	<ol style="list-style-type: none"> <li>1. The e-mail address has never been contacted before; the e-mail address is still added to the “do not contact” list.</li> <li>2. There are some e-mails in the e-mail queue for the user; these e-mails are deleted once confirmation has been given by the user, and if possible, error messages will be displayed to the Administrator.</li> </ol>
<b>Comments</b>	<ol style="list-style-type: none"> <li>1. The ability to permanently unsubscribe from web services is important to gain user trust, and also satisfy privacy regulations.</li> <li>2. Any further e-mails to this user must be sent manually, and cannot be sent by the system.</li> <li>3. The Visitor may, in the future, remove themselves from the “do not contact” list, by following the instructions in the e-mail in Step 2, or contacting the administrators.</li> <li>4. Steps 2 and 3 might not be possible, depending on privacy requirements.</li> </ol>

UC-35	<b>Persistent Errors</b>
<b>Description</b>	If an error occurs during an operation (such as sending an e-mail to an invalid address) when the source of the operation cannot be contacted, the error message is stored until later.
<b>Preconditions</b>	The system can send e-mails ( <i>E-mailing Users</i> ); error messages can be stored by the Server for particular Users.
<b>Actors</b>	User, Server
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. The User tries to send an e-mail to an address that doesn't exist.</li> <li>2. The e-mail is put into a local queue on the Server while it is trying to be sent.</li> <li>3. The User leaves the web application.</li> <li>4. The e-mail bounces back to the Server as undeliverable; the Server stores an error message for the User.</li> <li>5. The User returns to the web application.</li> <li>6. The error message is displayed from the Server along with the original message.</li> </ol>
<b>Postconditions</b>	The error message is removed from the Server once displayed.
<b>Exceptions</b>	1. The User never returns to the web application; nothing happens.
<b>Comments</b>	The error message could also be delivered via e-mail to the User.

UC-36	<b>User Content Access Control</b>
<b>Description</b>	Content can be shared between users, with specific permissions.
<b>Preconditions</b>	<ol style="list-style-type: none"> <li>1. There exist three Users: User 1, User 2, User 3.</li> <li>2. A User 1 has created and published a blog entry.</li> <li>3. User 2 and User 3 cannot usually access the blog entry.</li> </ol>
<b>Actors</b>	User 1, User 2, User 3, Server
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. User 1 assigns reading rights to User 2.</li> <li>2. User 2 tries to view the blog entry.</li> <li>3. The Server checks the entry permissions and allows User 2 to view the entry.</li> <li>4. User 3 tries to view the blog entry.</li> <li>5. The Server checks the entry permissions, but displays an error message to User 3 and does not let the User continue.</li> </ol>
<b>Postconditions</b>	<ol style="list-style-type: none"> <li>1. User 2 can view the entry.</li> <li>2. User 3 cannot view the entry.</li> </ol>
<b>Exceptions</b>	1. User 2 views the blog entry, but then the rights are revoked; User 2 can continue to view the entry until the page is closed.
<b>Comments</b>	Assigned permissions should be accessible and modifiable by User 1.

UC-37	<b>Private User Content Access Control</b>
<b>Description</b>	A User can share private feeds to another Visitor, which emulates the first User's account to achieve the desired rights.
<b>Preconditions</b>	Preconditions in <i>User Content Access Control</i> .
<b>Actors</b>	User, Visitor, Server
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. The User shares a private URL with the Visitor to view an RSS feed of all their blog entries.</li> <li>2. The Visitor visits this private URL without being authenticated.</li> <li>3. The Server temporarily assigns the Visitor the rights of the User, to allow them to view the request.</li> <li>4. The Server checks the entry permissions and allows the Visitor to view the entry.</li> </ol>
<b>Postconditions</b>	Postconditions in <i>User Content Access Control</i> .
<b>Exceptions</b>	<ol style="list-style-type: none"> <li>1. Exceptions in <i>User Content Access Control</i>.</li> <li>2. The User revokes or regenerates the private URL; the original URL will no longer provide access.</li> </ol>
<b>Comments</b>	<ol style="list-style-type: none"> <li>1. Often this is achieved by adding a special authentication key to a request.</li> <li>2. If a third party discovers this private URL, they can gain unauthorised access to this private data.</li> <li>3. Another common example is sharing a private calendar with a friend, which creates a temporary connection with very specific rights.</li> <li>4. Used in applications to share private data with other users, or to integrate with external feed readers without requiring separate authentication.</li> <li>5. This differs from <i>User Content Access Control</i> as that it does not assign permissions to any particular registered User.</li> </ol>

UC-38	<b>User Collaboration</b>
<b>Description</b>	Different users can work on the same calendar.
<b>Preconditions</b>	<ol style="list-style-type: none"> <li>1. An editable calendar is shared between two users, User 1 and User 2.</li> <li>2. Calendar data can be stored locally (<i>Server Data Access</i>).</li> </ol>
<b>Actors</b>	User 1, User 2, Server
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. User 1 goes to the shared calendar.</li> <li>2. User 2 goes to the shared calendar.</li> <li>3. User 1 adds a new event to the calendar.</li> <li>4. The new event is submitted to the Server.</li> <li>5. The Server notifies User 2 of the new event.</li> <li>6. User 2's Client adds the new event to its model.</li> <li>7. User 2 edits the new event, changing the date.</li> <li>8. User 2 submits the change to the Server.</li> <li>9. The Server notifies User 1 of the new event.</li> <li>10. User 1's Client updates the existing event in its model with the changes.</li> </ol>
<b>Postconditions</b>	The system state is consistent.
<b>Exceptions</b>	<ol style="list-style-type: none"> <li>1. User 1 and User 2 make a change at the same time on different events; both users are notified of each others change.</li> <li>2. User 1 and User 2 make a change at the same time on the same event; the event is changed to the last update, and both users are notified of the change.</li> </ol>
<b>Comments</b>	<ol style="list-style-type: none"> <li>1. Steps 5 and 9 do not have to be instant; they could only occur every 60 seconds, for example; but the core concept of this is that it is interactive and realtime.</li> <li>2. Some real-world examples of this use case include Google Pages, and spreadsheets in Google Docs.</li> </ol>



UC-39	<b>Interactive Map</b>
<b>Description</b>	A visitor is given an interactive graphical map of the world, which they can move around using their mouse.
<b>Preconditions</b>	A map can be rendered as an image; this image is broken up into many smaller, separate pieces.
<b>Actors</b>	Visitor, Client, Server
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. The Visitor is displayed an image of part of a map.</li> <li>2. The Visitor uses the mouse to drag the map to the right.</li> <li>3. The Client moves the map image according to the mouse movement.</li> <li>4. If the newly-revealed parts of the larger map image have not been loaded yet: <ol style="list-style-type: none"> <li>(a) The Client requests the Server for the map parts.</li> <li>(b) The Server returns the image parts.</li> <li>(c) The Client displays the new images by replacing older images in the map.</li> </ol> </li> </ol>
<b>Postconditions</b>	
<b>Exceptions</b>	<ol style="list-style-type: none"> <li>1. The Client cannot contact the Server; an error message is displayed, or a “missing image” is displayed.</li> <li>2. The Server does not have map imagery for the missing coordinates; a “no image exists; zoom out” image is returned instead.</li> </ol>
<b>Comments</b>	<ol style="list-style-type: none"> <li>1. This allows for Clients to browse a large image by only downloading the parts relevant to the current query.</li> <li>2. There must be some way to jump to specific co-ordinates in the larger map, without having to browse the map first.</li> <li>3. One obvious example of this use case is Google Maps.</li> </ol>

UC-40	<b>Drag and Drop</b>
<b>Description</b>	The user can use drag and drop to intuitively move e-mails from one folder to another.
<b>Preconditions</b>	
<b>Actors</b>	User, Client, Server
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. The User is displayed a list of e-mails in their inbox.</li> <li>2. The User uses the mouse to drag and drop an e-mail from the inbox to the trash icon.</li> <li>3. While being dragged, a highlight of the e-mail is displayed under the mouse cursor.</li> <li>4. When dropped, the Client: <ol style="list-style-type: none"> <li>(a) Contacts the Server to move the selected e-mail to the trash.</li> <li>(b) Removes the selected e-mail from the inbox, and updates the trash messages count.</li> </ol> </li> <li>5. The Server moves the selected e-mail to the trash.</li> </ol>
<b>Postconditions</b>	
<b>Exceptions</b>	<ol style="list-style-type: none"> <li>1. The escape key is pressed while dragging; the operation is cancelled.</li> <li>2. The e-mail is dragged to an invalid area on the screen; nothing happens.</li> <li>3. The e-mail is dragged outside the Client window; nothing happens.</li> </ol>
<b>Comments</b>	

UC-41	<b>Client Timer Support</b>
<b>Description</b>	The Client can automatically save a draft of the e-mail every ten minutes.
<b>Preconditions</b>	The Client is working on a local e-mail ( <i>Server Data Access</i> ); the Client can access the Server.
<b>Actors</b>	User, Client, Server
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. The User is working on an e-mail on the Client.</li> <li>2. Every ten minutes: <ol style="list-style-type: none"> <li>(a) The Client submits a copy of the e-mail to the Server.</li> <li>(b) The Server saves the e-mail as a draft copy.</li> <li>(c) The Client displays “Auto-saved at (time)”.</li> </ol> </li> </ol>
<b>Postconditions</b>	The e-mail object is automatically saved every ten minutes.
<b>Exceptions</b>	<ol style="list-style-type: none"> <li>1. The e-mail has not changed in the last ten minutes; the Server is not contacted.</li> <li>2. The Server cannot be contacted for any reason; the e-mail draft save is ignored, and the auto-saved message changes to “Could not auto-save”.</li> <li>3. The e-mail no longer exists on the Server; an error message is displayed to the User, and the User is asked if they would like to save the current draft.</li> </ol>
<b>Comments</b>	This use case is often implemented with the Javascript command <code>setTimeout()</code> on the current browser window.

UC-42	<b>Server Timer Support</b>
<b>Description</b>	The Server can automatically e-mail users new products every four hours.
<b>Preconditions</b>	<ol style="list-style-type: none"> <li>1. New products are added to a database on the Server.</li> <li>2. A User has requested that they are e-mailed every four hours of new product listings.</li> <li>3. This process has not occurred for at least four hours.</li> </ol>
<b>Actors</b>	User, Server
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. The Server gets a list of all the products added in the last four hours.</li> <li>2. The Server composes an e-mail listing all of these new products.</li> <li>3. The Server sends the e-mail to the User (<i>E-mailing Users</i>).</li> <li>4. The current time is saved as the time this process was last executed.</li> </ol>
<b>Postconditions</b>	The User has received an e-mail of new products.
<b>Exceptions</b>	<ol style="list-style-type: none"> <li>1. The e-mail cannot be sent; see exceptions in <i>E-mailing Users</i>.</li> <li>2. There are no new products added in the last four hours; Steps 2 and 3 are skipped.</li> <li>3. The process does not complete; an error message is displayed to the Administrator.</li> </ol>
<b>Comments</b>	<ol style="list-style-type: none"> <li>1. Other examples include new listings on auction sites, or digesting the replies from private messages.</li> <li>2. This is different from <i>Scheduled Events</i> as this use case is executed every 4 hours; <i>Scheduled Events</i> occurs at the same specified time every day.</li> </ol>

UC-43	<b>Page Caching</b>
<b>Description</b>	A frequently-requested home page can be cached to reduce unnecessary processing requirements on the Server.
<b>Preconditions</b>	The home page has some sort of caching setting (five minutes).
<b>Actors</b>	Visitor, Server
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. A Visitor requests the home page.</li> <li>2. The Server checks the cache to see if a cached version exists, and its age.</li> <li>3. If the page is too old, the page is recomputed in this request and stored in the cache.</li> <li>4. The cached result is supplied to the Visitor.</li> </ol>
<b>Postconditions</b>	
<b>Exceptions</b>	<ol style="list-style-type: none"> <li>1. No cached version of the current page exists; it is recompile from scratch (as in Step 3).</li> <li>2. The current Visitor is an Administrator; the cache is ignored and the page is re-constructed every time.</li> </ol>
<b>Comments</b>	<ol style="list-style-type: none"> <li>1. This can also be extended to handle page components; e.g. one section of the page is user account info, another is the home page.</li> <li>2. Cached pages can integrate with other technologies such as <i>memcached</i> or <i>Squid</i>, or other Content Delivery Networks (CDNs).</li> <li>3. Page caching should not occur when the application is in a <i>Debug Mode</i>.</li> </ol>

UC-44	<b>Offline Application Support</b>
<b>Description</b>	A Client-based application can be taken offline and continued to be worked on, but no changes can be saved until the Client is reconnected to the Internet.
<b>Preconditions</b>	<ol style="list-style-type: none"> <li>1. Preconditions in <i>Server Data Access</i>.</li> <li>2. The User is writing an e-mail.</li> </ol>
<b>Actors</b>	User, Client, Server
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. Access to the web application is taken offline, e.g. a network failure.</li> <li>2. The Client displays a message that the application is offline, but the current e-mail can still be edited.</li> <li>3. The User continues to work on the e-mail, without being able to save it or send it.</li> <li>4. The application is taken back online; the connection is restored with the Server.</li> </ol>
<b>Postconditions</b>	<ol style="list-style-type: none"> <li>1. The User can continue to write the e-mail, and save or send it.</li> <li>2. Postconditions in <i>Server Data Access</i>.</li> </ol>
<b>Exceptions</b>	<ol style="list-style-type: none"> <li>1. The User abandon the e-mail while the application is offline; a warning message is shown and the inevitable data loss is confirmed.</li> <li>2. The User tries to send the e-mail whilst offline; an error message is displayed to the User informing they cannot send the e-mail until online again.</li> </ol>
<b>Comments</b>	<ol style="list-style-type: none"> <li>1. This use case covers the functionality of an offline application, not the storage of data locally.</li> <li>2. A limited amount of information could be stored through the technologies mentioned in <i>Persistent Client Data</i>.</li> <li>3. In some cases, e-mails could still be sent by the client, except they would be queued up until network connectivity is restored.</li> <li>4. A number of technologies exist to implement this functionality, such as Google Gears [129] and HTML 5 [343].</li> </ol>

UC-45	<b>Loading Time Support</b>
<b>Description</b>	While an application is loading, the Server should inform the the User of the progress through application loading. If the application will take too long to load, the Server should request the User switches to a different version.
<b>Preconditions</b>	The Client can recognise or experience that the application needs to be loaded.
<b>Actors</b>	Visitor, Client, Server
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. A Visitor tries to download a large application script.</li> <li>2. The Client starts the download timer.</li> <li>3. After 60 seconds, the Client suggests that the Internet connection is too slow.</li> <li>4. A message is displayed informing the User about the situation, allowing them to continue waiting (the default), or providing them with a link to a simpler site.</li> </ol>
<b>Postconditions</b>	The web application is loaded, or the User has been redirected to a simpler site.
<b>Exceptions</b>	<ol style="list-style-type: none"> <li>1. The application never loads, or takes more than five minutes; the loading process is abandoned and the User is taken to the simpler site by default.</li> </ol>
<b>Comments</b>	<ol style="list-style-type: none"> <li>1. The <i>Static Views (HTML)</i> use case permits applications to serve static HTML pages as an alternative interaction method.</li> <li>2. See also <i>Multiple Browser Support</i>.</li> </ol>

UC-46	<b>Flash MP3 Support</b>
<b>Description</b>	An e-mail with an MP3 attachment can be streamed and played in the webmail application using a custom Flash component.
<b>Preconditions</b>	The User has an e-mail with an MP3 attachment; the Client can render Flash components.
<b>Actors</b>	User, Client, Server
<b>Normal Sequence</b>	<ol style="list-style-type: none"><li>1. The User opens an e-mail with an MP3 attached.</li><li>2. The Server returns the e-mail page, along with a Flash Object.</li><li>3. The Flash component is loaded by the Client and displays a play button.</li><li>4. Once the play button is pressed, the MP3 is streamed from the Server by the Flash component, and played back to the User.</li></ol>
<b>Postconditions</b>	
<b>Exceptions</b>	<ol style="list-style-type: none"><li>1. The User does not have Flash installed; the Server or Client detects the lack of Flash and instead displays a link to download the MP3 itself.</li><li>2. The Flash Object fails to load; a link to download the MP3 file is displayed instead.</li><li>3. The Server fails to stream the MP3; the Flash component is replaced with a link to download the MP3 file, or try contacting the Server again.</li></ol>
<b>Comments</b>	<ol style="list-style-type: none"><li>1. If Flash is not installed in the Client, the Client can ask the User if they would like to install Flash.</li><li>2. Other options for MP3 support include Java applets and the Quicktime plugin.</li><li>3. The upcoming HTML5 standard [343] includes a native component for playing media files.</li></ol>

UC-47	<b>Flash Communication Support</b>
<b>Description</b>	A web page wants to display a slider control for number of e-mails to display per page, using a custom Flash component.
<b>Preconditions</b>	
<b>Actors</b>	User, Client, Server
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. The User requests the list of e-mails.</li> <li>2. The Server returns the list of e-mails, along with a slider Flash Object.</li> <li>3. The Client loads the Flash component along with the current setting as a parameter (50 e-mails per page).</li> <li>4. The User drags the slider up to 75.</li> <li>5. The Flash component sends a message to the Client via Javascript of the new e-mails value.</li> <li>6. The Client requests the Server for the 25 previously undisplayed e-mails.</li> <li>7. The Server responds with the 25 e-mails.</li> <li>8. The Client adds the 25 e-mails to the bottom of the e-mail list.</li> </ol>
<b>Postconditions</b>	
<b>Exceptions</b>	<ol style="list-style-type: none"> <li>1. The Client cannot load the hidden e-mails; the display remains unchanged, and the slider is reset.</li> <li>2. The Client cannot load a Flash component; a normal slider, or text entry field, is displayed instead.</li> </ol>
<b>Comments</b>	<ol style="list-style-type: none"> <li>1. The Client should also be able to send messages back to the Flash component, possibly through Javascript.</li> <li>2. If Flash is not installed in the Client, the Client can ask the User if they would like to install Flash.</li> <li>3. This is a generalised use case of <i>Communication with Plugins</i>.</li> </ol>

UC-48	<b>Internationalisation Support</b>
<b>Description</b>	A web application can have support for multiple locales.
<b>Preconditions</b>	A web application is translated into at least two locales.
<b>Actors</b>	Visitor, Server
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. The Visitor visits the web application.</li> <li>2. The Server decides the most likely locale for the User, either from request location or HTTP request header.</li> <li>3. The Server checks to see which supported locale is the closest match.</li> <li>4. The Server presents the page with the translated components.</li> </ol>
<b>Postconditions</b>	
<b>Exceptions</b>	<ol style="list-style-type: none"> <li>1. The automatically selected locale is incorrect; the Visitor can select a different locale.</li> </ol>
<b>Comments</b>	<ol style="list-style-type: none"> <li>1. Different locales can be provided by different Servers or represented with different URLs.</li> <li>2. The chosen locale should be stored, probably with a cookie, for future requests.</li> </ol>

UC-49	<b>Logout Control</b>
<b>Description</b>	If a User logs out from a website, it should not be possible to repeat an activity as that User.
<b>Preconditions</b>	The User is authenticated.
<b>Actors</b>	User, Server
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. The User composes a message, and sends it through the web application.</li> <li>2. The User logs out and becomes a Visitor.</li> <li>3. The Visitor presses the back button on their browser, and tries to resubmit the message.</li> <li>4. The Server realises the Visitor has logged out, and instead displays an error message, asking the Visitor to re-authenticate.</li> </ol>
<b>Postconditions</b>	The Visitor cannot perform any User actions.
<b>Exceptions</b>	
<b>Comments</b>	<ol style="list-style-type: none"> <li>1. If the Client could recognise the Visitor has logged out, the Client could display an error message instead of allowing the Visitor to browse back in the navigation history.</li> <li>2. This is an important access control and security-related use case.</li> </ol>

UC-50	<b>Single Sign-In Solutions</b>
<b>Description</b>	An application can use a single centralised sign-on service (SSO) for authentication support.
<b>Preconditions</b>	A single sign-on provider exists; the User has an account with this service.
<b>Actors</b>	Visitor, User, Server
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. The Visitor authenticates with a web application.</li> <li>2. The application redirects the User to the SSO service.</li> <li>3. The User authenticates with their SSO details.</li> <li>4. The SSO page informs the Server of the login success.</li> <li>5. The Visitor is redirected to the Server, and is authenticated as a User.</li> </ol>
<b>Postconditions</b>	The Visitor is authenticated as a User.
<b>Exceptions</b>	<ol style="list-style-type: none"> <li>1. The User does not have an account with the SSO service; the SSO service can let the User create a new account.</li> <li>2. The given SSO ID has been blocked on the local web application; an error message is displayed to the Visitor, and authentication is blocked.</li> <li>3. The User cannot authenticate with the SSO service; the SSO service should help with re-establishing authorisation (e.g. lost password).</li> </ol>
<b>Comments</b>	<ol style="list-style-type: none"> <li>1. Two common SSO services are Google Accounts and Microsoft Passport.</li> <li>2. A decentralised form of these SSO services may be through OpenID [281] and OAuth [145], for identity and trust authentication respectively.</li> </ol>

UC-51	<b>User Redirection</b>
<b>Description</b>	A visitor can search a database of sites with a search string, and be redirected to a given site.
<b>Preconditions</b>	The Visitor can search a database of sites.
<b>Actors</b>	Visitor, Server
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. The Visitor searches for the search string “iaml”</li> <li>2. The Server returns a list of all the results for “iaml” in its database.</li> <li>3. The Visitor clicks the first search result, which leads to a click-through URL on the Server.</li> <li>4. The Server notes that the Visitor clicked the first search result for “iaml”.</li> <li>5. The Server sends the Visitor a redirection response to the actual search result URL, <code>openiaml.org</code>.</li> <li>6. The Visitor is redirected to the destination URL, <code>openiaml.org</code>.</li> </ol>
<b>Postconditions</b>	The Visitor is now on the third party website <code>openiaml.org</code> .
<b>Exceptions</b>	
<b>Comments</b>	<ol style="list-style-type: none"> <li>1. The ability to redirect the user is a basic requirement of database-driven websites.</li> <li>2. Redirects can either be internal (to the current web application) or external (to third party web applications).</li> <li>3. User redirection can be chained multiple times; care must be taken to prevent infinite redirections.</li> </ol>



UC-52	<b>Keyboard Shortcuts</b>
<b>Description</b>	A web application can implement keyboard shortcuts to help power users.
<b>Preconditions</b>	The User has a browser which is capable of intercepting key strokes.
<b>Actors</b>	User, Client
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. The User logs into their e-mail inbox.</li> <li>2. The User presses the “T” key to move to the Trash folder.</li> <li>3. The Client picks up the “T” key being pressed and “clicks” the Trash link.</li> <li>4. The User is redirected to the Trash folder.</li> </ol>
<b>Postconditions</b>	The User is viewing the Trash folder.
<b>Exceptions</b>	1. The Client cannot intercept the key stroke (e.g. no Javascript support); nothing happens.
<b>Comments</b>	<ol style="list-style-type: none"> <li>1. Keyboard shortcuts are useful for power users.</li> <li>2. Keyboard shortcuts should not be too intrusive, and must not occur when trying to enter in actual data.</li> </ol>

UC-53	<b>Undo/Redo Support</b>
<b>Description</b>	Whilst working on an online document, the user can undo and redo actions.
<b>Preconditions</b>	The User is editing a document locally (see <i>Server Data Access</i> ), with “Bold”, “Undo” and “Redo” buttons.
<b>Actors</b>	User, Client, Server
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. The User sets a selection of text to be bolded, and presses the “Bold” button.</li> <li>2. The Client informs the Server, and the Server bolds the text on the document.</li> <li>3. The Client renders the bolded text.</li> <li>4. The User presses the “Undo” button.</li> <li>5. The Client requests an Undo from the Server.</li> <li>6. The Server unbolds the text on the document, and informs the Client of the change.</li> <li>7. The Client renders the un-bolded text.</li> <li>8. The User presses the “Redo” button.</li> <li>9. The Client requests a Redo from the Server.</li> <li>10. The Server bolds the text on the document, and informs the Client of the change.</li> <li>11. The Client renders the bolded text.</li> </ol>
<b>Postconditions</b>	The User is editing the same document.
<b>Exceptions</b>	<ol style="list-style-type: none"> <li>1. The previous action cannot be undone; the Undo button is disabled.</li> <li>2. An undone action cannot be redone; the Redo button is disabled.</li> </ol>
<b>Comments</b>	<ol style="list-style-type: none"> <li>1. Could be implemented through the <i>Command</i> design pattern [118].</li> <li>2. Combining <i>Undo/Redo Support</i> with <i>User Collaboration</i> may introduce some complex synchronisation problems.</li> </ol>

UC-54	<b>Browser-Based Chat</b>
<b>Description</b>	Users can open up a browser-based chat window with other users, connected through the server.
<b>Preconditions</b>	<ol style="list-style-type: none"> <li>1. There are two Users in the same chat room, opened with two Clients.</li> <li>2. Each Client has a connection to the Server.</li> <li>3. User 2 is already connected to the Server.</li> </ol>
<b>Actors</b>	User 1, User 2, Client 1, Client 2, Server
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. User 1 connects to the chat room.</li> <li>2. Client 1 opens a connection with the Server.</li> <li>3. The Server sends the Client 1 a list of previous chat messages, which are displayed to User 1.</li> <li>4. The Server sends a message to Client 2, informing them that User 1 has connected.</li> <li>5. User 2 sends a chat message.</li> <li>6. Client 2 sends the chat message to the Server.</li> <li>7. The Server sends the message to all connected Clients, displaying them to their Users.</li> </ol>
<b>Postconditions</b>	
<b>Exceptions</b>	<ol style="list-style-type: none"> <li>1. The connection to the Server cannot be opened; an error message is displayed to the User, and the chat window is closed.</li> <li>2. The connection to the Server is lost; the Client attempts to reconnect the connection; once connected, the Server resends the Client a list of any activity whilst disconnected.</li> <li>3. The connection to the Server is lost, and cannot be reconnected; an error message is displayed to the User, and the chat window is closed.</li> </ol>
<b>Comments</b>	<ol style="list-style-type: none"> <li>1. This is one implementation of a browser-based chat session, with the messages sent over the open connection and interpreted once received.</li> <li>2. A different implementation (more common) uses active polls on open connections to keep synchronised with server activity.</li> <li>3. This could be implemented without client-side Javascripts (regular page refreshes).</li> </ol>

UC-55	<b>Pop-up Window Support</b>
<b>Description</b>	An application can open pop-up windows to assist users in adding images to e-mails.
<b>Preconditions</b>	A User is composing an e-mail (see <i>Server Data Access</i> ).
<b>Actors</b>	User, Client, Server
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. The User clicks a button to add an image to the e-mail.</li> <li>2. The Client opens a pop-up window displaying a list of available images.</li> <li>3. The User clicks an image to add to the e-mail.</li> <li>4. The Client closes the pop-up window and adds the image to the e-mail composition.</li> </ol>
<b>Postconditions</b>	The image is added to the e-mail display.
<b>Exceptions</b>	<ol style="list-style-type: none"> <li>1. The User does not select an Image, and closes the window; the Client does not add any image to the e-mail.</li> <li>2. The User ignores the pop-up window, and only interacts with it once the e-mail application has been closed; any further action on the window either does nothing, or displays an error message and closes the window.</li> </ol>
<b>Comments</b>	This can be implemented with any number of technologies, including a normal browser popup; a modal popup; a Java application; a Flash component; or a Javascript message dialog.

UC-56	<b>Incompatible Client Warning</b>
<b>Description</b>	If the Visitor visits the website with a browser/client that cannot fully support all the features the site absolutely requires, the Visitor is given a warning.
<b>Preconditions</b>	The web application has some elements that cannot be rendered in the Client.
<b>Actors</b>	Visitor, Client, Server
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. The Visitor visits the website with their incompatible Client.</li> <li>2. The Server recognises the Client, compares it to the list of features needed for the website to operate correctly.</li> <li>3. The Server identifies the Client is incompatible or misses some features.</li> <li>4. The Server displays an initial warning message, instructing the User how to upgrade their Client, and allowing them to proceed anyway at their own risk, or access an alternative site (e.g. <i>Static Views (HTML)</i>).</li> </ol>
<b>Postconditions</b>	
<b>Exceptions</b>	<ol style="list-style-type: none"> <li>1. The Client cannot be identified; the Server assumes the Client is fully functional.</li> </ol>
<b>Comments</b>	<ol style="list-style-type: none"> <li>1. Some website features may not be “required” but only optional, in which case the client detection should only occur when trying to access the feature.</li> <li>2. Runtime browser testing could be performed with scripting to work out if the Client is <i>actually</i> incompatible.</li> </ol>

UC-57	<b>Dynamic Objects</b>
<b>Description</b>	Depending on the Client accessing the web application, a map component in a web application can be rendered using scripting technology, or only static HTML.
<b>Preconditions</b>	The map component cannot be rendered with scripting technology in their Client.
<b>Actors</b>	Visitor, Server
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. The Visitor visits the website with their incompatible Client.</li> <li>2. The Server recognises the Client is incompatible or misses some features.</li> <li>3. The Server instead renders the map component with a static HTML rendering.</li> </ol>
<b>Postconditions</b>	The Visitor can continue to use the site in the same manner as the scripted rendering.
<b>Exceptions</b>	<ol style="list-style-type: none"> <li>1. The Client cannot be identified; the Server assumes the Client is fully functional, and uses the scripted technology rendering.</li> <li>2. The Client is mis-identified; the Visitor can force the web application to switch to a different rendering.</li> </ol>
<b>Comments</b>	This differs from <i>Multiple Browser Support</i> in that it is concerned with the rendering of individual elements, not complete applications.

UC-58	<b>Store Data in Local Database</b>
<b>Description</b>	An application can use an offline technology, such as Google Gears [129], to store data locally in a local database which can be synchronised when network connectivity is restored.
<b>Preconditions</b>	<ol style="list-style-type: none"> <li>1. The User has Google Gears installed.</li> <li>2. Preconditions in <i>Communication with Plugins</i>.</li> <li>3. The User initially has network connectivity.</li> <li>4. The current web application supports offline operation.</li> </ol>
<b>Actors</b>	User, Client, Server
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. The User visits the web application.</li> <li>2. The Client creates a local client-side database to store changes.</li> <li>3. The User loses network connectivity; the application continues to function as expected.</li> <li>4. The User interacts with the offline application.</li> <li>5. The Client saves changes to the client-side database.</li> <li>6. The User restores network connectivity: <ol style="list-style-type: none"> <li>(a) The Client recognises the User has gone back online.</li> <li>(b) The Client contacts the Server and publishes its changes.</li> <li>(c) The Client deletes the information in its client-side database.</li> </ol> </li> <li>7. The User can continue to use the application as expected.</li> </ol>
<b>Postconditions</b>	The system is in a consistent state.
<b>Exceptions</b>	<ol style="list-style-type: none"> <li>1. The User does not have Google Gears installed; a message is displayed to the User informing them that work will otherwise be lost when network connectivity is lost.</li> <li>2. The Client recognises its database is out of sync with the Server database; the Client informs the User and gives them synchronisation options.</li> </ol>
<b>Comments</b>	<ol style="list-style-type: none"> <li>1. One cannot guarantee that client-side data will ever be submitted to the remote server; it may reside on the Client indefinitely.</li> <li>2. It is a reasonable assumption that data cannot be expected to be stored forever on the client.</li> <li>3. The major issue with storing data locally is synchronisation, especially between multiple independent clients.</li> <li>4. The major use case of offline technology appears to be in interacting with mostly-read-only data sources.</li> <li>5. Individual resources (images, videos) can be stored through <i>Store Resources Locally</i>.</li> </ol>

UC-59	<b>Store Resources Locally</b>
<b>Description</b>	An application can use an offline technology, such as Google Gears [129], to store web application resources (images, CSS, Javascript etc) on the local machine, to allow the application to go offline.
<b>Preconditions</b>	<ol style="list-style-type: none"> <li>1. The User has Google Gears installed.</li> <li>2. Preconditions in <i>Communication with Plugins</i>.</li> <li>3. The User initially has network connectivity.</li> <li>4. The current web application supports offline operation.</li> </ol>
<b>Actors</b>	User, Client, Server
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. The User visits the web application.</li> <li>2. The Client creates a list of resources necessary for offline operation, including a remote image.</li> <li>3. The Client downloads these resources, including the remote image, and stores them locally.</li> <li>4. The User loses network connectivity.</li> <li>5. <ol style="list-style-type: none"> <li>(a) The Client retrieves the image from the local machine.</li> <li>(b) The Client returns the image to the browser, which is displayed to the User identically to online operation.</li> </ol> </li> </ol>
<b>Postconditions</b>	
<b>Exceptions</b>	<ol style="list-style-type: none"> <li>1. Exceptions in <i>Store Data in Local Database</i>.</li> <li>2. An image is requested which is not in the local cache; either an empty image is displayed, or an error message is shown.</li> </ol>
<b>Comments</b>	<ol style="list-style-type: none"> <li>1. Compared to <i>Store Data in Local Database</i>, the resources loaded in this method cannot be modified.</li> <li>2. The offline technology must handle modified resources whenever the resource cache is synchronised.</li> </ol>

UC-60	<b>Multiple Client Threads</b>
<b>Description</b>	A rendered web application can execute in multiple threads, possibly asynchronously, on the Client.
<b>Preconditions</b>	The Client can run multiple threads.
<b>Actors</b>	User, Client
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. A Client starts an operation.</li> <li>2. The Client splits the operation execution into a second thread.</li> <li>3. The User continues to interact with the Client in one thread, while the other thread works in the background.</li> <li>4. At some point, the second thread will terminate.</li> </ol>
<b>Postconditions</b>	The second thread has terminated.
<b>Exceptions</b>	<ol style="list-style-type: none"> <li>1. The Client cannot run threads; either the threads can be run virtually, or an error message is displayed to the User.</li> <li>2. The second thread terminates unexpectedly; either the thread is restarted, or an error message is displayed to the User.</li> </ol>
<b>Comments</b>	<ol style="list-style-type: none"> <li>1. Currently no major browser supports multiple Javascript threads, but may be implemented in the future.</li> <li>2. A thread could be native or virtual (run with ticks in a single thread).</li> <li>3. The implementation of the <i>WorkerPool</i> in Google Gears [129] achieves some level of multiple Client threading.</li> </ol>

UC-61	<b>Multiple Server Threads</b>
<b>Description</b>	A web application can execute in multiple threads, possibly asynchronously, on the Server.
<b>Preconditions</b>	The Server can run multiple threads.
<b>Actors</b>	Server
<b>Normal Sequence</b>	<ol style="list-style-type: none"><li>1. The Server starts an operation.</li><li>2. The Server splits the operation execution into a second thread.</li><li>3. At some point, the second thread will terminate.</li></ol>
<b>Postconditions</b>	The second thread has terminated.
<b>Exceptions</b>	<ol style="list-style-type: none"><li>1. The Server cannot run threads; either the threads can be run virtually, or an error message is displayed to the Administrator.</li><li>2. The second thread terminates unexpectedly; either the thread is restarted, or an error message is displayed to the Administrator.</li></ol>
<b>Comments</b>	Server-side multiple threading is quite common, however concurrency and scalability issues often restrict its use.



UC-62	<b>Communication with Plugins</b>
<b>Description</b>	A client application can communicate with browser plugins; for example, interaction with Google Gears.
<b>Preconditions</b>	A Client has the Google Gears plugin installed; the Client can directly communicate with installed plugins.
<b>Actors</b>	User, Client, Server
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. The User visits a web application.</li> <li>2. The Client searches for an installed copy of Google Gears.</li> <li>3. The Client finds that Google Gears has been installed, and creates a new local instance of the Google Gears factory.</li> <li>4. The Google Gears factory can then be used to create additional objects, such as the <i>DataStore</i> and <i>LocalResourcePool</i>.</li> </ol>
<b>Postconditions</b>	
<b>Exceptions</b>	<ol style="list-style-type: none"> <li>1. Step 3: The user does not have Google Gears installed; either an error message can be displayed, or the exception can be ignored</li> </ol>
<b>Comments</b>	<ol style="list-style-type: none"> <li>1. Plugin technologies include ActiveX (for Internet Explorer) and NPAPI (for Firefox).</li> <li>2. Other use cases for communicating with plugins include identifying installed versions or executing functionality.</li> <li>3. <i>Flash Communication Support</i> is a particular instance of using plugins to interact with a rendered page.</li> </ol>

UC-63	<b>Scheduled Events</b>
<b>Description</b>	A Server can have a daily scheduled event (e.g. at 5am every day) to send new blog entries to every User on the system.
<b>Preconditions</b>	There is a list of Users on the system; the scheduled time has occurred (i.e. 5am).
<b>Actors</b>	Server
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. The Server iterates through recent blog entries and constructs a message of new entries.</li> <li>2. The Server iterates through all Users in the system.</li> <li>3. The Server sends an e-mail (<i>E-mailing Users</i>) to each User with this message.</li> <li>4. The Server proceeds onto the next user.</li> </ol>
<b>Postconditions</b>	All users have been processed.
<b>Exceptions</b>	<ol style="list-style-type: none"> <li>1. The scheduled event is skipped; the event is executed as soon as possible.</li> <li>2. The scheduled event is cancelled or unexpectedly terminated; an error message is displayed to the Administrator, and the process may be resumed in the future.</li> </ol>
<b>Comments</b>	See <i>Server Timer Support</i> for a process which occurs regularly, instead of at a specific scheduled time.

UC-64	<b>Custom API Publishing</b>
<b>Description</b>	A blogging website can publish a <i>custom</i> API (e.g. the LiveJournal API) to allow external software/websites to post. blog content
<b>Preconditions</b>	The Server has a published API endpoint, along with an expected API message format.
<b>Actors</b>	Server, User or Remote Server
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. The User or Remote Server constructs a <i>PostMessage</i> message.</li> <li>2. The User submits the <i>PostMessage</i> message to the Server's published API endpoint.</li> <li>3. The Server receives and parses this <i>PostMessage</i> message.</li> <li>4. The Server creates a new blog entry, based on the input from the posted message.</li> <li>5. The Server saves the blog entry, and returns an "OK" message.</li> </ol>
<b>Postconditions</b>	
<b>Exceptions</b>	<ol style="list-style-type: none"> <li>1. The message cannot be parsed; an error message is returned via the web service.</li> <li>2. The blog entry cannot be saved; an error message is returned via the web service.</li> </ol>
<b>Comments</b>	<ol style="list-style-type: none"> <li>1. With respect to the <i>Web Service</i> use case, this use case covers the publication of a <i>custom</i> API, not necessarily using a published protocol.</li> <li>2. This is often achieved through custom JSON [65], XML-based or REST-based [96] APIs.</li> </ol>

UC-65	<b>Runtime Interface Updates</b>
<b>Description</b>	The Client can update the user interface of a calendar site based on User input, without reloading or redirecting the Client.
<b>Preconditions</b>	The Client can run Javascript scripts; the User is viewing a monthly calendar page.
<b>Actors</b>	User, Client
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. The User selects "two weeks" as the calendar view.</li> <li>2. The Client replaces the monthly view of the calendar with a fortnightly view, without reloading or redirecting the Client.</li> </ol>
<b>Postconditions</b>	The User is viewing a fortnightly calendar page.
<b>Exceptions</b>	<ol style="list-style-type: none"> <li>1. The Client needs to load data asynchronously from the Server, but cannot contact it; the Client instead reloads the current page, or an error message is displayed to the User.</li> </ol>
<b>Comments</b>	This use case includes loading data asynchronously using AJAX technologies.

UC-66	<b>Out-of-Order Events</b>
<b>Description</b>	The Client can ignore events which come out of order, for example when requesting a list of search results that start with a particular prefix.
<b>Preconditions</b>	The Client can send/receive asynchronous messages.
<b>Actors</b>	User, Client, Server
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. The User enters in “a”.</li> <li>2. The Client sends an asynchronous request to the Server (<i>message 1</i>) for results starting with “a”.</li> <li>3. The Server receives <i>message 1</i>, and sends back a list of results (<i>response 1</i>).</li> <li>4. The User enters in “ab”.</li> <li>5. The Client sends an asynchronous request to the Server (<i>message 2</i>) for results starting with “ab”.</li> <li>6. The Server receives <i>message 2</i>, and sends back a list of results (<i>response 2</i>).</li> <li>7. The Client receives <i>response 2</i>, and displays a list of results starting with “ab”.</li> <li>8. The Client receives <i>response 1</i>, but recognises that message 2 was sent later; the Client ignores this response.</li> </ol>
<b>Postconditions</b>	A list of results starting with “ab” are displayed.
<b>Exceptions</b>	<ol style="list-style-type: none"> <li>1. <i>Response 2</i> does not arrive after a specified timeout period; the result from <i>response 1</i> is displayed, and <i>message 2</i> is resent.</li> </ol>
<b>Comments</b>	<ol style="list-style-type: none"> <li>1. In some cases, it may be desirable to display both responses, even if they are out of order.</li> <li>2. In extreme cases, where the order of messages is critically important, an additional layer of network control may be implemented (similar to the reliable transmission property of TCP).</li> </ol>

UC-67	<b>Backwards-Compatible Scripting</b>
<b>Description</b>	Non-essential functionality in scripted web applications do not affect the operation of web applications on incompatible clients; for example, a button should light up when it is clicked.
<b>Preconditions</b>	An incompatible client is accessing a web application with a button, that is using incompatible scripting.
<b>Actors</b>	User, Server
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. The User clicks on a button.</li> <li>2. The Client tries to “light up” the button.</li> <li>3. The Client-side animation fails, but the failure is caught by the Client.</li> <li>4. The execution of the button clicking continues as normal.</li> </ol>
<b>Postconditions</b>	The button is clicked.
<b>Exceptions</b>	
<b>Comments</b>	<ol style="list-style-type: none"> <li>1. If scripted functionality is actually required, then the Client should be redirected to a more suitable application rendering, as in <i>Incompatible Client Warning</i> or <i>Multiple Browser Support</i>.</li> <li>2. A less desirable alternative is for incompatible scripting to throw an error message, that prevents the button from functioning at all.</li> </ol>

UC-68	<b>Spellchecking</b>
<b>Description</b>	A form can be spellchecked, either at submit or at runtime.
<b>Preconditions</b>	A form can be edited by the User.
<b>Actors</b>	User, Server
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. The User enters in some incorrect text onto the form.</li> <li>2. The User submits the form to the server.</li> <li>3. The Server checks the submitted data for spellchecking, to ensure the text is entered in correctly.</li> <li>4. The submitted data is saved.</li> </ol>
<b>Postconditions</b>	The submitted data is fully spellchecked.
<b>Exceptions</b>	<ol style="list-style-type: none"> <li>1. Step 3, the submitted data is badly spelt; the Server presents the User with a list of possible corrections, and goes back to Step 1.</li> </ol>
<b>Comments</b>	<ol style="list-style-type: none"> <li>1. Some Clients may support spellchecking natively.</li> <li>2. If <i>Internationalisation Support</i> is supported, spellchecking must also take into account the different locales.</li> </ol>

UC-69	<b>Autocomplete</b>
<b>Description</b>	The user may enter data into a text field using autocomplete in order to simplify the data entry process.
<b>Preconditions</b>	An empty text field can be edited by the User.
<b>Actors</b>	User, Client, Server
<b>Normal Sequence</b>	<ol style="list-style-type: none"> <li>1. The User enters a few characters into the text field, and this represents the <i>query</i>.</li> <li>2. The Client contacts the Server with an asynchronous message to search for valid search results to the <i>query</i>.</li> <li>3. The Server identifies valid search results for the <i>query</i> and returns them to the Client.</li> <li>4. The Client displays a dropdown box of each of these search results.</li> <li>5. The User clicks one of the displayed search results.</li> <li>6. The Client hides the dropdown box, and replaces the text field with the value of the valid search result.</li> </ol>
<b>Postconditions</b>	The text field is no longer empty.
<b>Exceptions</b>	<ol style="list-style-type: none"> <li>1. Step 3, there are no valid search results; the Server returns an empty list, and the Client does not display any dropdown box.</li> <li>2. Step 4, the User has entered in additional characters into the text field, before the asynchronous search has completed; the Client ignores the results of this search, and proceeds from Step 2.</li> <li>3. Step 5, the User removes focus from the text field without selecting a search result; the Client hides the dropdown box.</li> </ol>
<b>Comments</b>	<ol style="list-style-type: none"> <li>1. This use case was discussed previously by Wright and Dietrich [368].</li> <li>2. Some of the exceptions in this use case are related to the <i>Out-of-Order Events</i> use case.</li> <li>3. The search interface may be provided as a public API, as in the <i>Custom API Publishing</i> use case.</li> <li>4. Navigation through the dropdown box of search results may be supported through <i>Keyboard Shortcuts</i>.</li> </ol>

## Appendix B

# Documentation By Example: Sync Wire

In this chapter, two important concepts behind the design and implementation of IAML will be illustrated through an example. In particular, this appendix will show the intended behaviour of model completion, where default conventions are used to automatically infer missing knowledge within a model instance; and [Wires](#), where a model developer can use instances of reusable patterns to implement common functionality within web applications. This example is derived from the Ticketiaml model instance presented earlier in Section 8.3.2.

As discussed earlier in Section 5.9.1, a [Sync Wire](#) may be used to keep the values of two elements synchronised, including user interface elements and instances of domain types. Within Ticketiaml, a [Sync Wire](#) is used in the implementation of the “Edit Event” page, where a manager may edit a selected *Event* instance as illustrated earlier in Figure 5.23; in particular, a [Sync Wire](#) is used to keep an [Input Form](#) and [Domain Iterator](#) synchronised.

Initially, the only contents within this form and iterator is a [Button](#) that is used to [save](#) the current instance within the [Domain Iterator](#). When model completion is executed, many elements will be inserted into the model instance as defined by the model completion rules in Sections I.94 and I.38:

1. **The builtin operations and properties of the iterator will be completed.** model completion

For the [Domain Iterator](#), operations such as [next](#) and [previous](#) will be completed with instances of [Builtin Operations](#); properties such as [has next](#) and [has previous](#) will be completed with instances of [Builtin Properties](#); and the current instance of the iterator will be completed with an instance of a [Domain Instance](#). The result of this step is illustrated in Figure B.1<sup>1</sup>.

2. **The feature instances of the [Domain Instance](#) will be completed.**

The newly-created [Domain Instance](#) will be completed with [Domain Feature Instances](#) for every [Domain Feature](#) defined by the [Domain Type](#) of the iterator, as discussed earlier in Section 5.6.3. Based on the schema of the *Event* type in Figure 5.19, this instance will be completed as illustrated in Figure B.2. Each [Domain Attribute Instance](#) must also be connected to an underlying [Domain Attribute](#), and this relationship is expressed as instances of [Extends Edges](#).

3. **The *Edit Event* [Input Form](#) will be completed with [Input Text Fields](#).**

As discussed earlier in Section 5.9.1, one of the key features of a [Sync Wire](#) is that it will automatically complete the contents of connected elements. This means that the *Edit Event*

---

<sup>1</sup>This figure also illustrates the use of the forward slash name prefix “/” to highlight elements generated through model completion, as discussed in Section 7.4.4.

**Input Form** will be completed with an **Input Text Field** for each **Domain Attribute Instance** within the connected **Domain Instance**<sup>2</sup>.


Another key feature of a **Sync Wire** is that it will also connect new sub-**Sync Wires** to these generated elements where appropriate. This means that each **Input Text Field** will be connected to its corresponding **Domain Attribute Instance** with a *new* **Sync Wire**. The *Edit Event Input Form* will therefore be completed as illustrated in Figure B.3<sup>3</sup>.

However, not all of the attributes of an *Event* should be editable by the manager; in particular, it should not be possible to edit the *id* of the *Event*, which is a primary and unique key for each *Event* instance. In the Ticketiaml model, the *Edit Event Input Form* defines “id” as an **overridden name**; as discussed in Section 7.5.2, this means that no **Input Text Field** for the *id* attribute instance will be generated.

4. **Each Input Text Field will be completed with the behaviour necessary to implement synchronisation with their corresponding Domain Attribute Instance.**

Finally, the underlying logic of a **Sync Wire** will be generated through model completion, as illustrated in Figure B.4<sup>4</sup>:

- (a) When the **Input Text Field** is accessed – either for the first time, or if the containing page is subsequently reloaded – the field should be initialised with the current value of the **Domain Attribute Instance**. This is achieved with an **ECA Rule** on the *onAccess* event of the text field that executes its **init** operation.
- (b) When the **Domain Attribute Instance** changes, the field should be updated with the value of the attribute instance. This is achieved with an **ECA Rule** on the *onChange* event of the attribute instance that executes the **update** operation on the text field.
- (c) Conversely, when the **Input Text Field** changes, the attribute instance should be updated with the value of the text field. This is achieved with an **ECA Rule** on the *onChange* event of the text field that executes the **update** operation on the attribute instance.

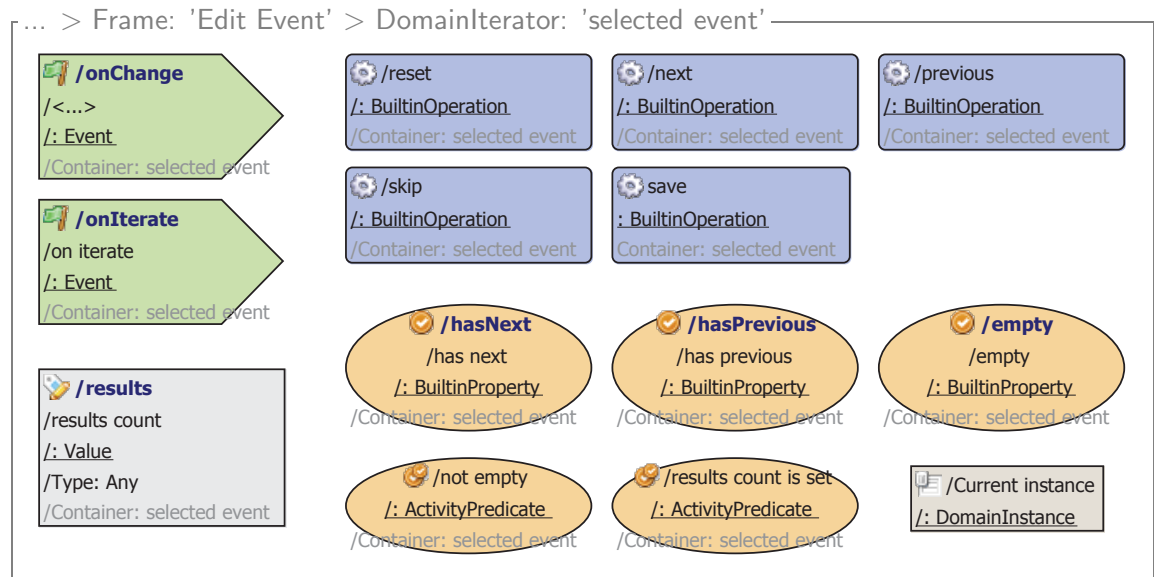
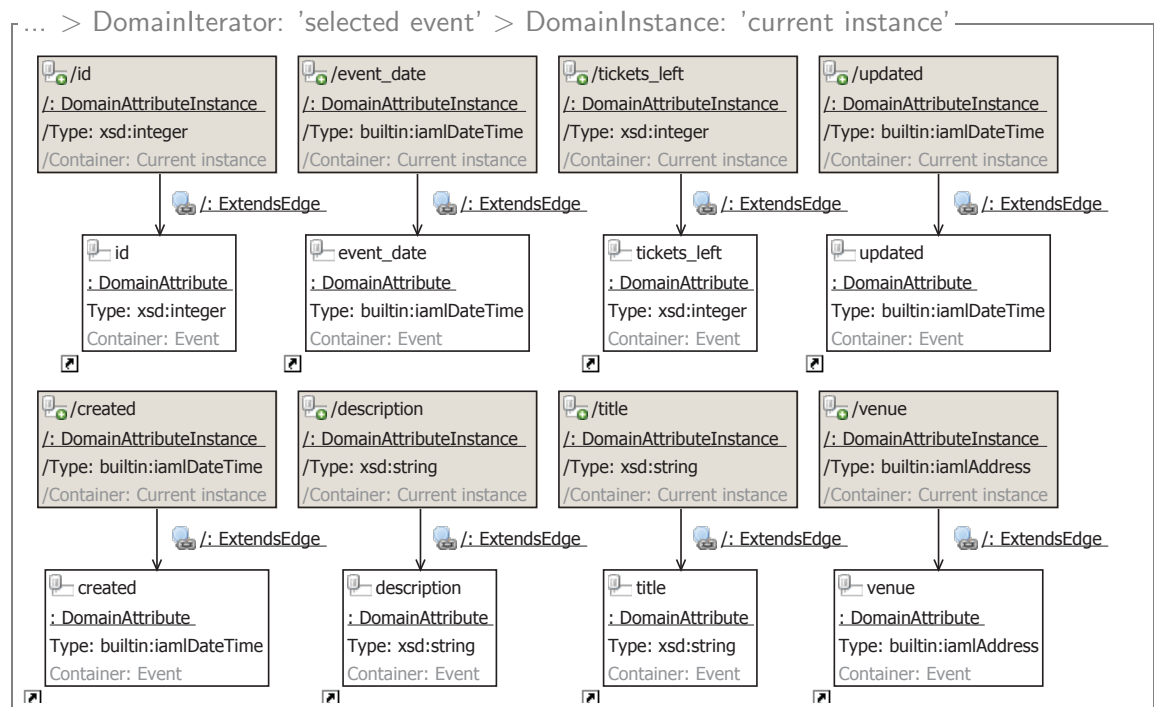
This model completion logic is performed for every **Input Text Field** within the *Edit Event Input Form* in a similar fashion. Since both the **Input Text Field** and **Domain Attribute Instance** have the same **name**, it is important to note that shortcut elements – those with the annotation  – are references to external elements, as discussed in Section 7.4.4.

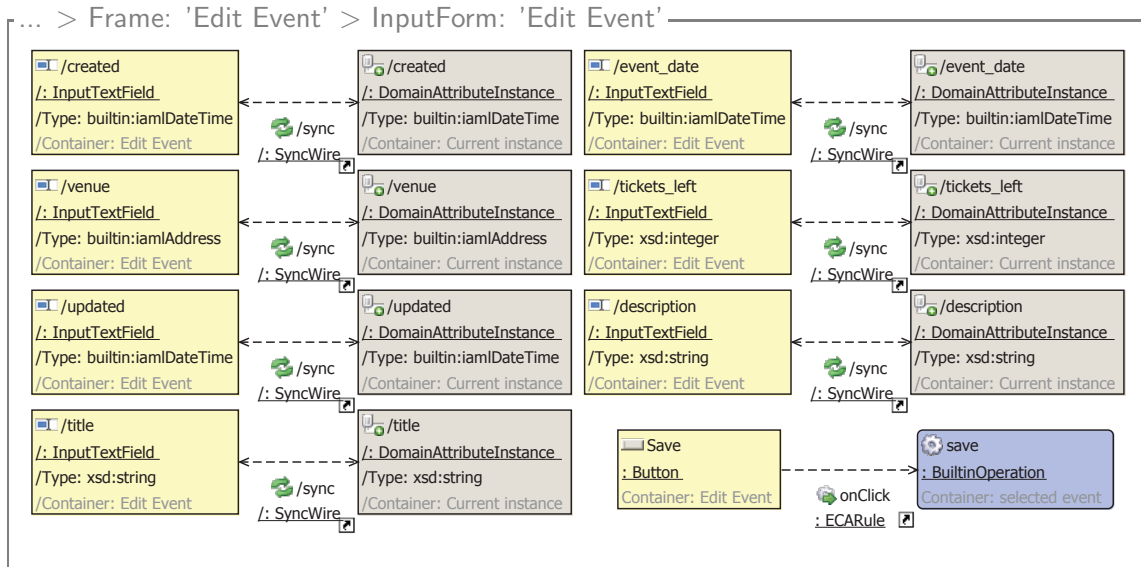
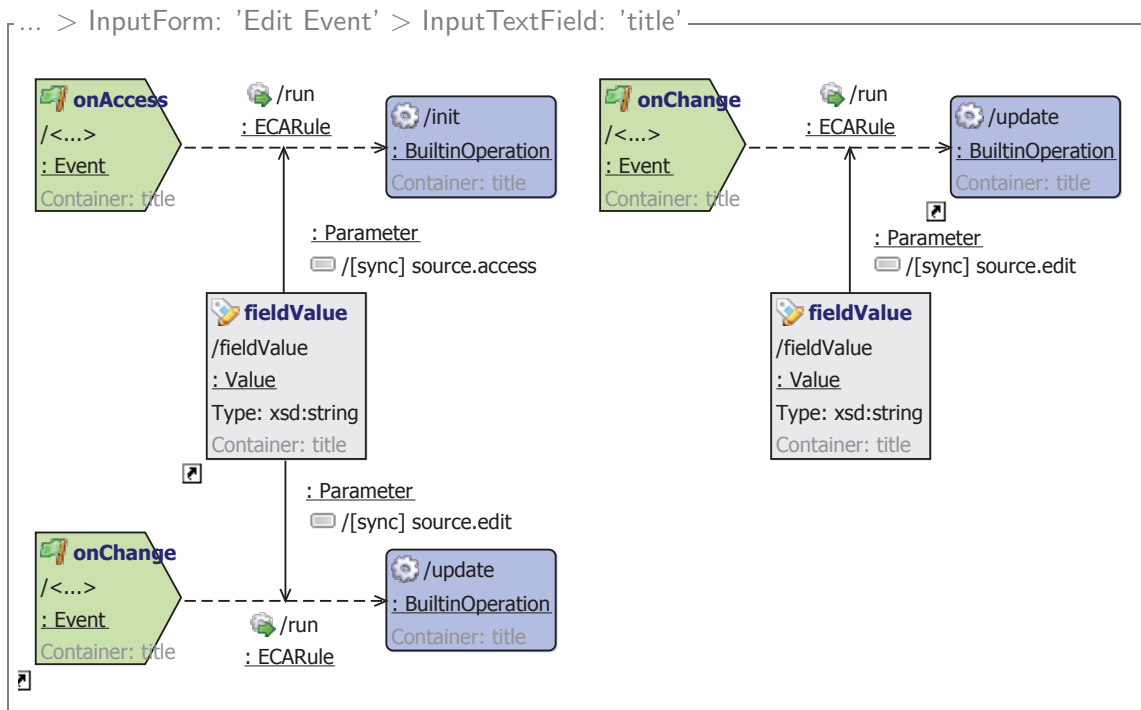
The “Edit Event” page of Ticketiaml forms a small part of the manual implementation of the web application, and relies heavily on model completion rules to implement common functionality. Once transformed through the code generation framework of IAML and loaded in a web browser, this completed functionality may be rendered as illustrated in Figure B.5.

<sup>2</sup>As discussed earlier in Section 5.6.4, since a **Domain Iterator** can only ever contain a single **Domain Instance**, a **Sync Wire** connected to a **Domain Instance** may connect directly to the iterator itself.

<sup>3</sup>This partial figure does not include elements generated by other model completion rules, including **Events** such as *onAccess* and **Builtin Operations** such as *show* and *hide*.

<sup>4</sup>Similarly to Figure B.3, this partial figure does not illustrate other elements normally generated such as the *onInput* and *onClick* events; **Builtin Operations** such as *show* and *hide*; or **Builtin Properties** such as *empty* and *not empty*.

Figure B.1: Ticketiaml: The completed contents of the *edit event* Domain IteratorFigure B.2: Ticketiaml: The completed contents of the *Event*-typed Domain Instance

Figure B.3: Ticketiaml: The completed contents of the *Edit Event* Input FormFigure B.4: Ticketiaml: The completed contents of the *title* Input Text Field, within the *Edit Event* Input Form, to implement synchronisation










---

[Home](#) [Browse Events](#) [Your Tickets](#) [Your Friends](#) [Recommended Events](#) [Logout](#)

Manager tools: [All Events](#) [Create New Event](#)

## Edit Event

Edit Event

 venue	<input type="text" value="123 Main St, Palmerston Nor"/>
 title	<input type="text" value="Event three"/>
 description	<input type="text" value="The third event with one ticke"/>
 created	<input type="text" value="Tue, 26 Jul 2011 19:35:19 +00"/>
 updated	<input type="text" value="Tue, 26 Jul 2011 19:35:19 +00"/>
 tickets_left	<input type="text" value="1"/>
 event_date	<input type="text" value="Fri, 26 Aug 2011 19:35:19 +00"/>

Save

Figure B.5: A screenshot of the *Edit Event* page implemented in Ticketiaml



## **Appendix C**

# **OpenBRR Evaluations of Model-driven Technologies**

In Chapter 6, fifteen projects were evaluated against the 28 open source quality metrics defined by OpenBRR [263]. The values of each metric for each technology is published here in Tables C.1, C.2 and C.3. For a given project, each individual metric may have a score between 1 (unacceptable) and 5 (excellent), based on the representative scores defined by OpenBRR.org [263].

Category	Alloy	ArgoUML	CrocoPat	Dresden OCL	Drools
<b>Usability</b>					
End-user UI	5	3	3	5	3
Installing prerequisites	5	5	5	4	4
Vanilla install	5	5	5	5	4
<b>Quality</b>					
Minor releases	1	5	1	3	3
Point/patch releases	1	5	1	1	3
Open bugs	5	4	5	5	3
Fixed bugs	5	3	1	4	2
Critical bugs	5	4	5	4	2
Age of critical bugs	5	4	5	1	1
<b>Security</b>					
Number of issues	5	5	5	5	1
Unpatched issues	5	5	5	5	1
Security web page	1	1	1	1	1
<b>Performance</b>					
Benchmarks	5	1	5	1	5
Tuning	1	1	5	1	5
<b>Scalability</b>					
Reference deployment	5	3	3	3	5
Scalable design	5	1	3	1	5
<b>Architecture</b>					
Third-party plug-ins	5	5	1	3	5
Public API	3	5	1	3	5
<b>Support</b>					
Mailing list volume	1	1	1	1	4
Professional support	1	1	1	1	5
<b>Documentation</b>					
Variety	4	3	5	4	5
User contributions	1	1	1	1	5
<b>Adoption</b>					
Books	2	1	1	1	3
Reference deployment	5	3	3	3	5
<b>Community</b>					
Mailing list volume	1	1	1	1	4
Unique contributors	1	2	1	2	3
<b>Professionalism</b>					
Project driver	2	2	2	2	4
Difficulty to enter	1	3	1	1	5
<b>Total</b>	<b>91</b>	<b>83</b>	<b>77</b>	<b>72</b>	<b>101</b>

Table C.1: Evaluations of model-driven technologies against OpenBRR ratings (1)

Category	EMFV OCL	EMF	GEF	GMF	Jena
<b>Usability</b>					
End-user UI	3	5	5	3	3
Installing prerequisites	5	5	5	5	5
Vanilla install	5	5	5	5	5
<b>Quality</b>					
Minor releases	3	5	1	3	5
Point/patch releases	5	5	5	1	1
Open bugs	5	3	3	3	5
Fixed bugs	1	4	3	3	2
Critical bugs	1	3	5	4	5
Age of critical bugs	5	1	5	1	5
<b>Security</b>					
Number of issues	5	5	5	5	5
Unpatched issues	5	5	5	5	5
Security web page	1	1	1	1	1
<b>Performance</b>					
Benchmarks	1	5	1	1	5
Tuning	1	5	1	3	5
<b>Scalability</b>					
Reference deployment	3	5	5	5	5
Scalable design	3	5	3	3	3
<b>Architecture</b>					
Third-party plug-ins	1	3	2	3	3
Public API	5	5	1	5	5
<b>Support</b>					
Mailing list volume	1	4	2	3	3
Professional support	5	5	5	5	1
<b>Documentation</b>					
Variety	5	5	4	3	4
User contributions	5	5	5	5	1
<b>Adoption</b>					
Books	1	3	3	2	1
Reference deployment	3	5	5	5	5
<b>Community</b>					
Mailing list volume	1	4	2	3	3
Unique contributors	1	4	2	3	1
<b>Professionalism</b>					
Project driver	5	5	5	5	4
Difficulty to enter	5	5	5	5	3
<b>Total</b>	<b>90</b>	<b>120</b>	<b>99</b>	<b>98</b>	<b>99</b>

Table C.2: Evaluations of model-driven technologies against OpenBRR ratings (2)

Category	JET	Marama	NuSMV	OAW	Xpand
<b>Usability</b>					
End-user UI	5	1	1	5	5
Installing prerequisites	5	3	5	5	5
Vanilla install	5	3	5	5	5
<b>Quality</b>					
Minor releases	5	1	3	1	5
Point/patch releases	3	3	3	1	1
Open bugs	5	5	1	5	5
Fixed bugs	4	3	1	5	2
Critical bugs	5	2	1	5	4
Age of critical bugs	5	1	1	5	1
<b>Security</b>					
Number of issues	5	5	1	5	5
Unpatched issues	5	5	1	5	5
Security web page	1	1	1	1	1
<b>Performance</b>					
Benchmarks	1	1	3	1	3
Tuning	1	1	3	1	5
<b>Scalability</b>					
Reference deployment	5	5	5	5	5
Scalable design	1	1	3	1	1
<b>Architecture</b>					
Third-party plug-ins	5	1	2	3	3
Public API	3	1	3	5	5
<b>Support</b>					
Mailing list volume	3	1	1	1	3
Professional support	5	1	1	5	5
<b>Documentation</b>					
Variety	3	3	4	4	5
User contributions	5	1	1	1	5
<b>Adoption</b>					
Books	1	1	1	3	1
Reference deployment	5	5	5	5	5
<b>Community</b>					
Mailing list volume	3	1	1	1	3
Unique contributors	1	1	1	1	1
<b>Professionalism</b>					
Project driver	5	2	2	4	5
Difficulty to enter	5	1	3	5	5
<b>Total</b>	<b>105</b>	<b>60</b>	<b>63</b>	<b>94</b>	<b>104</b>

Table C.3: Evaluations of model-driven technologies against OpenBRR ratings (3)

## Appendix D

# Model Checking using NuSMV: Implementing the *Infinitely Redirects* Constraint

As discussed earlier in Section 6.6.8, the implementation of the *infinitely redirects* model verification constraint in the NuSMV framework requires a considerable amount of definitions in order to define the behaviour of a web browser, and the functionality of the web application under consideration.

This appendix provides the necessary code to evaluate this constraint on a simple IAML model instance, which consists of two [Frames](#) – named *page1* and *page2* – which each define an *onAccess Event* that redirects to the other page through an [ECA Rule](#). As described in Section 7.7.4, the following NuSMV definitions are generated automatically through code generation templates using the openArchitectureWare framework [87].

```
-- main module
MODULE main

VAR
  -- all pages
  current_page : {null, page1, index, page2};
  -- allowing NuSMV to select pages at random
  browse_to_page : {null, page1, index, page2};
  -- flags
  operation_running : boolean;
  operation_finished : boolean;
  navigation_running : boolean;
  navigation_finished : boolean;

ASSIGN
  init (operation_running) := 0;
  init (operation_finished) := 0;
  init (navigation_finished) := 1;
  -- we start off without being on any page
  init (current_page) := null;
  -- click_button is chosen by nusmv
```

```

-- browse_to_page is chosen by nusmv

next (operation_running) := case
  1 : 0;
esac;

next (operation_finished) := case
  1 : 0;
esac;

-- browsing to a new page
next (current_page) := case
  -- cannot navigate away until finished navigating
  navigation_running = 1 : case
    -- does the current page have a redirection attached?
    current_page = page1 : -- an instant redirect
    page2;
    current_page = index : current_page;
    -- we must have stopped navigating
    current_page = page2 : -- an instant redirect
    page1;
    -- no redirections; stay where we are
    1 : current_page;
  esac;
  -- cannot navigate away while executing an operation
  operation_running = 1 : current_page;
  -- browse to another page chosen by NuSMV
  browse_to_page = page1 : page1;
  browse_to_page = index : index;
  browse_to_page = page2 : page2;
  -- finally, otherwise, stay where we are
  1 : current_page;
esac;

next (navigation_running) := case
  -- we are currently navigating
  navigation_running = 1 : case
    current_page = page1 : -- an instant redirect
    1;
    current_page = index : 0;
    -- we must have stopped navigating
    current_page = page2 : -- an instant redirect
    1;
    current_page = null : 0;
    -- should never occur
  esac;
  -- cannot navigate away while executing an operation
  operation_running = 1 : 0;
  -- browse to another page chosen by NuSMV
  browse_to_page = page1 : 1;

```



```

    browse_to_page = index : 1;
    browse_to_page = page2 : 1;
    -- otherwise, we are not navigating
    1 : 0;
esac;

next (navigation_finished) := case
  -- we are currently navigating
  navigation_running = 1 : case
    current_page = page1 : -- an instant redirect
    1;
    current_page = index : 1;
    -- we must have stopped navigating
    current_page = page2 : -- an instant redirect
    1;
    current_page = null : 0;
    -- should never occur
  esac;
  -- cannot navigate away while executing an operation
  operation_running = 1 : 1;
  -- browse to another page chosen by NuSMV
  browse_to_page = page1 : 0;
  browse_to_page = index : 0;
  browse_to_page = page2 : 0;
  -- otherwise, we are not navigating
  1 : 1;
esac;

-- checks for infinite loops in redirections

LTLSPEC
  G (((navigation_running = 1 -> !(F navigation_finished = 1)))
    U navigation_running = 0)

```



## Appendix E

# Description of the Attached Media

This thesis includes media containing the latest version of the proof-of-concept implementation of the IAML modelling environment, along with the source code of this environment and a copy of the metamodel documentation. To support this proof-of-concept implementation, a brief tutorial is also included in this appendix, which documents the process of defining an IAML model instance demonstrating the functionality of [Sync Wires](#).

### E.1 Media Contents

The contents of the attached media is fairly straightforward, but includes the following items:

#### **IAML-0.6-win32.zip**

This is a redistributable copy of the proof-of-concept IAML 0.6 modelling environment, based on a Windows version of Eclipse 3.5.2. The IAML environment has been installed from the IAML update site, and includes the experimental CrocoPat and NuSMV verification plugins. After extracting the environment to a local directory, Eclipse may be immediately executed; however, you may need to reinitialise the environment by executing the command `eclipse.exe -clean`.

#### **source**

This directory contains the full source code of the proof-of-concept implementation of IAML, as described in Chapter 7. This includes the source codes for the SimpleGMF framework described in Section 7.4.3, and the ModelDoc framework described by Wright [366]. This source code is also available online at the Subversion repository <http://iaml.googlecode.com/svn/trunk/>.

#### **iacleaner**

This directory contains the full source code of the *iacleaner* project used to implement the *Output Formatter* component, as described in Section 7.6.5. This source code is also available online at the Subversion repository <http://iacleaner.googlecode.com/svn/trunk/>.

## ticket2.0

This directory contains the source code for the three implementations of *Ticket 2.0*, as described in Section 8.3: *Ticketsf*, implemented using Symfony 1.0.12; *Ticketsf-mini*, implemented using Symfony 1.4.8; and *Ticketiaml*, implemented using IAML 0.6. This source code for each of these implementations are also available online at <http://openiaml.org/ticket20/>.

## model

This directory contains the full metamodel documentation of IAML, as generated by the ModelDoc framework [366]. This documentation is also provided in this thesis as Appendix I, and is accessible online at <http://openiaml.org/model/>.

## E.2 IAML Tutorial

This section will illustrate the basic concepts behind creating and working with IAML model instances within this Eclipse-based modelling environment. This documentation is derived from the end-user documentation of the project available online at <http://openiaml.org/docs/>. This tutorial assumes that you have installed the IAML modelling environment, and have installed the Apache HTTP Server and the PHP scripting language to process the generated web application; an installation guide to these environments is outside the scope of this appendix, but is available online at <http://openiaml.org/docs/install>.

### E.2.1 Creating a New Model Instance

Once a new project has been created within Eclipse, an IAML instance may be created by creating a new *IAML Diagram*. This may be achieved by right clicking the project and selecting “New > Other”, as illustrated in Figure E.1. In this dialog, browse down the list of available new wizards, and select the wizard “Example > Iaml Diagram” as illustrated in Figure E.2. Once selected, you will need to provide a diagram filename and a domain filename; in this tutorial, use the diagram filename `hello-world.iaml_diagram`, and the domain filename `hello-world.iaml`.

The resulting model instance will be presented through the IAML visual editor as illustrated in Figure E.3. By default, a new IAML model instance will contain a single **Frame** named “Home”, and a number of **EXSD Data Types** as a mapping to XSD primitive datatypes, such as `xsd:integer` and `xsd:string`.

### E.2.2 Editing the Model Instance

The “Home” page represents the default starting point of the web application, as the **ID** of this **Frame** is set to `index`. In this *Hello World* application, we will create a simple web site containing two text fields that are kept synchronous with each other. This is a fairly basic use case of Rich Internet Applications and may be described through a **Sync Wire** (Section 5.9.1).

Double click the “Home” page to open up a new editor window, as illustrated in Figure E.4. As described in Section 5.2.3, the IAML metamodel is designed in a hierarchical fashion, and the visual editor adheres to this design. We may edit the contents of the “Home” page – which by default, is empty – in this new editor window.

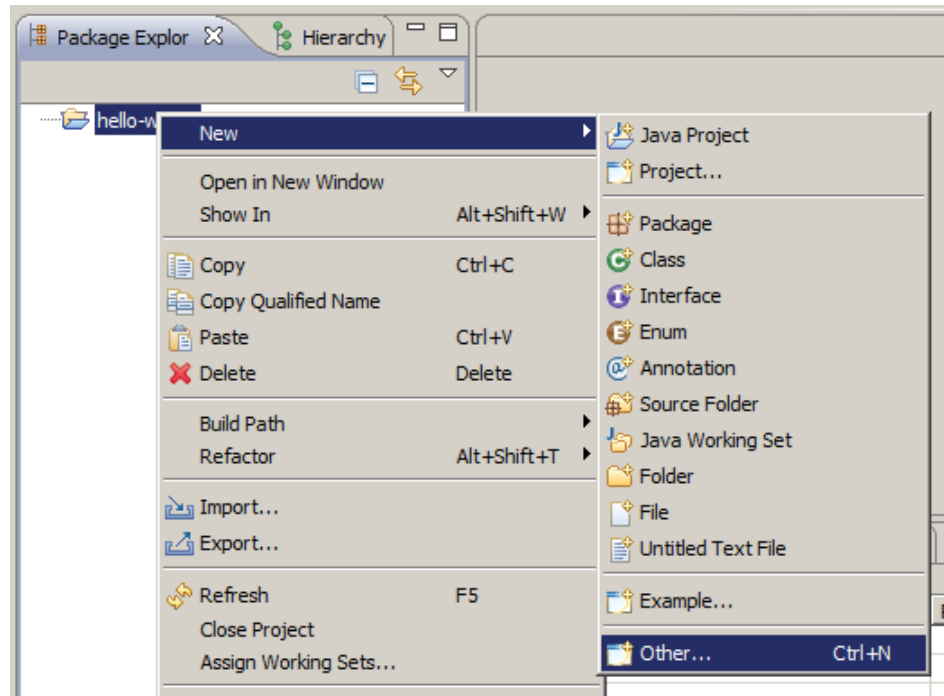


Figure E.1: Creating a new IAML model instance through the Eclipse wizard menu (1)

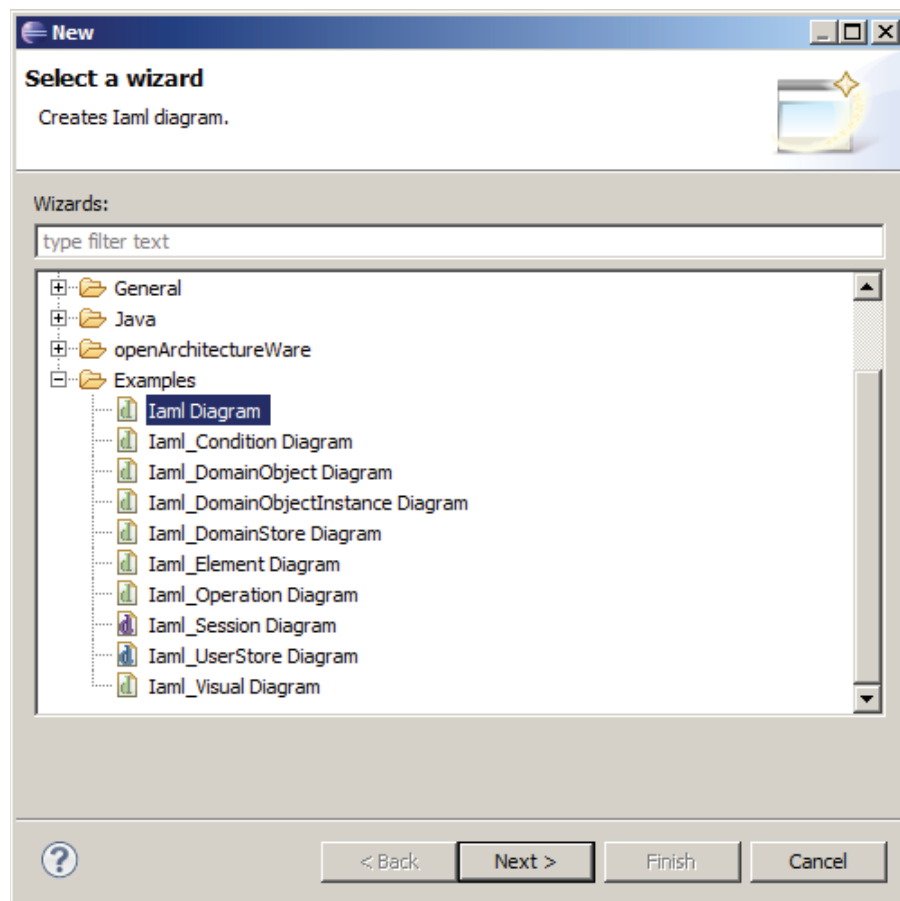


Figure E.2: Creating a new IAML model instance through the Eclipse wizard menu (2)

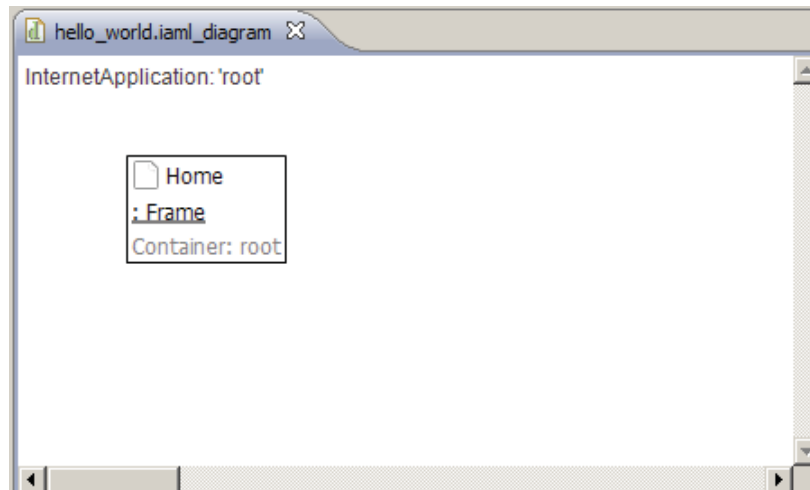


Figure E.3: The IAML visual editor for a new IAML model instance

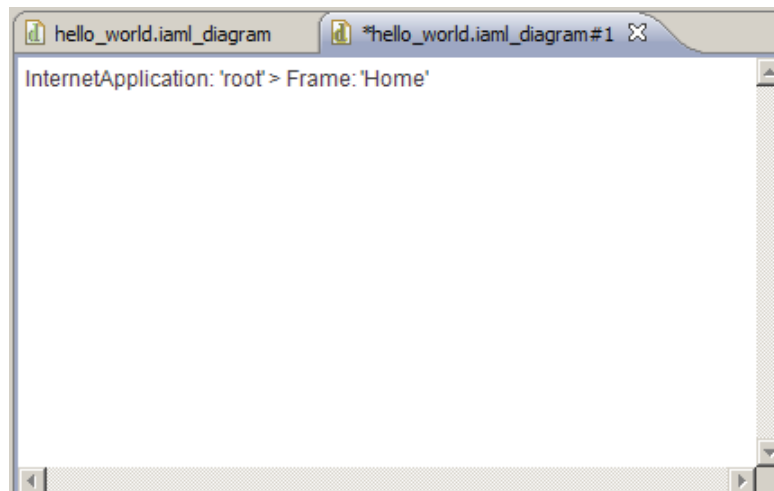


Figure E.4: The default contents of the “Home” Frame within a new IAML model instance

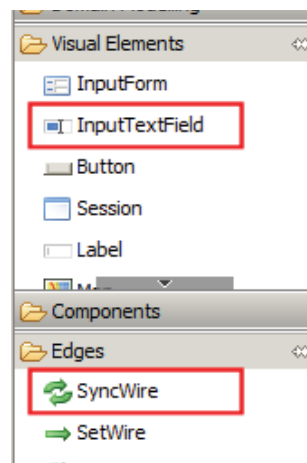


Figure E.5: The tool palette of the IAML visual editor

In this empty view, create two new text fields in this [Frame](#), naming them “Hello” and “World”. For each editor view, a list of elements that may be contained within the edited element are rendered in the tool palette of the editor. In IAML, text fields are available as [Input Text Fields](#), and are within the *Visual Elements* category of the palette as illustrated in Figure E.5. (A full description of the IAML visual editor interface has already been discussed in Section 7.4.)

By default, these two text fields would only accept input and persist it across instances of the application. In order to keep the contents of these two text fields synchronised with each other, it is necessary to connect them using a [Sync Wire](#), as described in Section 5.9.1. The new wire does not need to have a name, and elements without names will display the text “<...>”; in Figure E.6, this new wire has been named “sync”. This new model instance may now be saved by pressing Ctrl-S.

### E.2.3 Generating the Application

Now that a valid IAML model instance has been defined, we may automatically generate the server-side and client-side code necessary to implement the defined application. As described in the description of the *Code Generation* component in Section 7.6, the proof-of-concept implementation of IAML generates a combination of PHP, Javascript, CSS and HTML source code, which may then be served by a standard web server such as Apache.

In order to generate this source code, locate the `hello-world.iaml` file in the *Project Explorer* view of Eclipse. Right click this file, and select “IAML Tools > Generate Code”, as illustrated in Figure E.7. As described in Section 7.8 and illustrated in Figure E.8, this action completes the model instance through *model completion*, and then translates this completed model into source code through code generation templates. By default, this generated application is placed into a directory named `output` within the current Eclipse project.

### E.2.4 Configure the Web Server

This generated application may be accessed as a standard web application, served through any web server using PHP, by instructing the web server software to host the generated output directory. The technical details of the following step depend on the web server software that has been installed.

For example, if you are using the Apache HTTP Server to serve web applications locally, and your current Eclipse workspace is located at the path `C:\Workspace`, then you may set up an *Alias* to this generated code by adding the configuration details in Listing 24 to your `httpd.conf` configuration file. Once this configuration has been saved, restart the Apache server.

### E.2.5 Access the Generated Application

We can now access the generated application from any web browser. For example, the configuration details listed in Listing 24 will allow us to visit this application through the local address `http://localhost:8080/hello-world`, as illustrated in Figure E.9. Visiting this application will allow us to evaluate the functionality of the [Sync Wire](#); in particular, when the value of one of the two text fields changes, the other text field should automatically be updated to this new value.

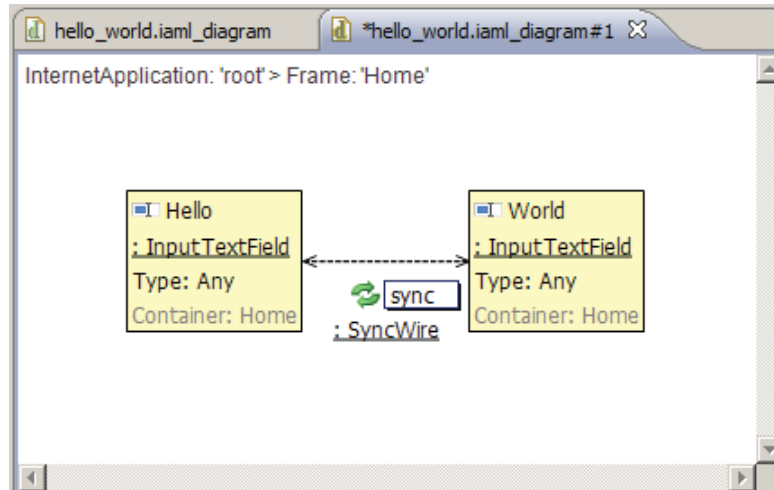


Figure E.6: The IAML visual editor representing a [Frame](#) containing two [Input Text Fields](#) connected by a [Sync Wire](#)

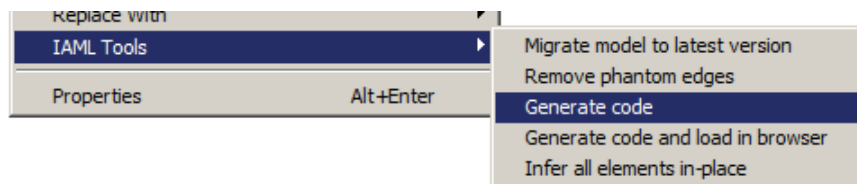


Figure E.7: Generating the source code for an IAML model instance (1)

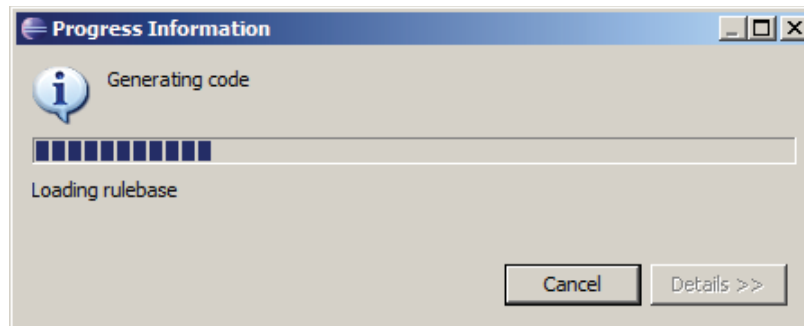


Figure E.8: Generating the source code for an IAML model instance (2)

```
Alias "/hello-world" "C:/Workspace/hello-world/output"
<Directory "C:/Workspace/hello-world/output">
  Options Indexes FollowSymLinks
  DirectoryIndex index.html index.php default.html default.php
  AllowOverride All
  Allow from All
</Directory>
```

Listing 24: Configuration details to host generated applications through the Apache HTTP Server



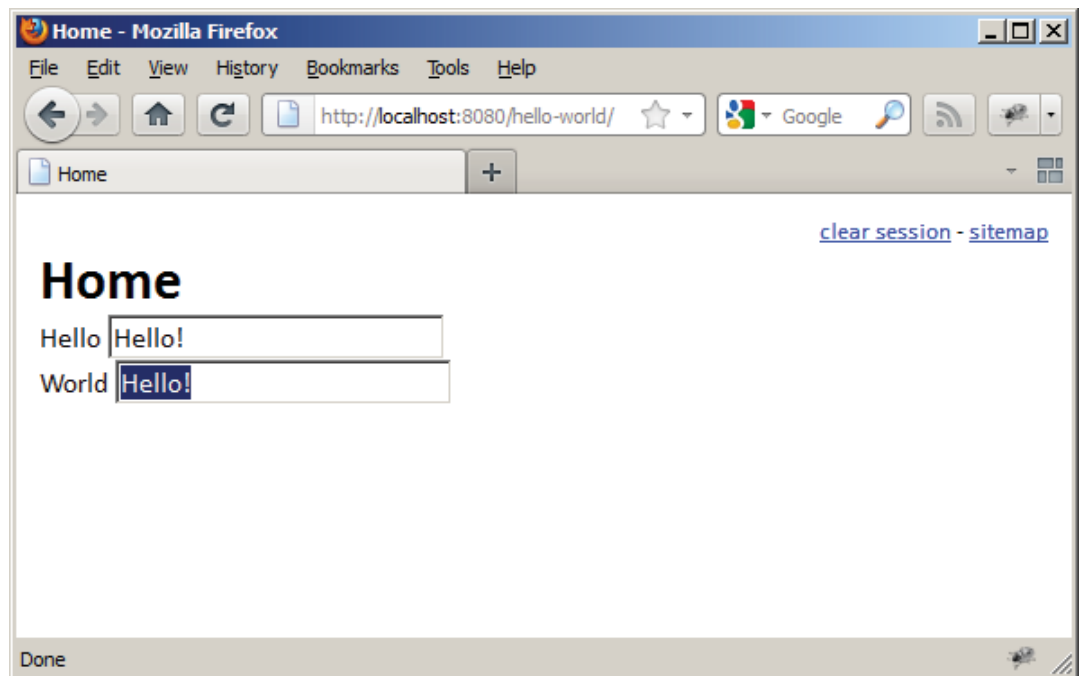


Figure E.9: Screenshot of the generated tutorial application accessed through Mozilla Firefox



## Appendix F

# XMI Representation of Ticketiaml

This appendix defines the partial implementation “Ticketiaml” of the *Ticket 2.0* benchmarking application within IAML using the XMI representation of the model instance, and this represents a complete IAML model instance necessary to implement the functionality of *Ticket 2.0*. The visual representation of this model instance, along with screenshots of the running generated application itself, were discussed earlier in Section 8.3.2. This file is also available on the attached media, as discussed later in Section E.1.

```
<?xml version="1.0" encoding="UTF-8"?>
<iaml:InternetApplication xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance"
xmlns:iaml="http://openiaml.org/model0.6" xmlns:iaml.domain="http://openiaml.org/model/do
main" xmlns:iaml.messaging="http://openiaml.org/model/messaging" xmlns:iaml.operations="h
ttp://openiaml.org/model/operations" xmlns:iaml.scopes="http://openiaml.org/model/scopes"
xmlns:iaml.users="http://openiaml.org/model/users" xmlns:iaml.visual="http://openiaml.or
g/model/visual" xmlns:iaml.wires="http://openiaml.org/model/wires" id="root" name="Ticket
2.0">
  <scopes xsi:type="iaml.visual:Frame" id="index" name="Home"/>
  <scopes xsi:type="iaml.scopes:Session" id="session_signup" name="Signup Session">
    <scopes xsi:type="iaml.visual:Frame" id="signup" name="Signup">
      <values id="value1" name="successful signup message" outParameterEdges="parameter1"
defaultValue="Your account has been created, you may now login."
      <parameterEdges id="parameter1" name="message" parameterValue="value1"
parameterTerm="eca4"/>
    </values>
    <messages xsi:type="iaml.messaging:Email" id="email_signup" name="Signup Email"
inWires="set1" subject="New account created" from="ticket20@openiaml.org" fromName="Ticke
t 2.0">
      <operations xsi:type="iaml.BuiltinOperation" rules="eca2" id="builtinoperation1"
name="send"/>
      <values id="signup_email_to" name="to" type="builtin_iamlEmail"
inWires="set3"/>
      <values id="signup_email_to_name" name="toName" inWires="set4"/>
      <onFailure id="signup_email_onFailure" name="onFailure" listeners="eca1">
        <ecaRules id="eca1" name="navigate" trigger="signup_email_onFailure" target="si
gnup_email_failed"/>
      </onFailure>
    </messages>
    <iterators id="iterator1" name="new user instance" outWires="sync1 set1" query="new
" outSelects="edge1">
```

```

        <wires xsi:type="iaml.wires:SyncWire" id="sync1" name="sync" from="iterator1" to="
"form1"/>
        <wires xsi:type="iaml.wires:SetWire" id="set1" name="set" from="iterator1" to="em
ail_signup"/>
        <selectEdges id="edge1" from="iterator1" to="named_users_db"/>
        <operations xsi:type="iaml:BuiltinOperation" rules="eca3" id="signup_user_save"
name="save" listeners="eca4 eca2">
            <ecaRules id="eca2" name="run" trigger="signup_user_save" target="builtinoperat
ion1" priority="100"/>
        </operations>
    </iterators>
    <children xsi:type="iaml.visual:InputForm" inWires="sync1" id="form1" name="Signup
Form">
        <operations xsi:type="iaml:BuiltinOperation" rules="eca5" id="signup_form_hide"
name="hide"/>
        <children xsi:type="iaml.visual:InputTextField" outWires="set2 set3" id="signup_e
mail" name="email" type="builtin_iamlEmail">
            <wires xsi:type="iaml.wires:SetWire" id="set2" name="set" from="signup_email"
to="label1"/>
            <wires xsi:type="iaml.wires:SetWire" id="set3" name="set" from="signup_email"
to="signup_email_to"/>
        </children>
        <children xsi:type="iaml.visual:InputTextField" outWires="set4" id="text1" name="
name" type="xsd_string">
            <wires xsi:type="iaml.wires:SetWire" id="set4" name="set" from="text1" to="sign
up_email_to_name"/>
        </children>
    </children>
    <children xsi:type="iaml.visual:Button" id="signup_create" name="Create Account"
listeners="eca3 eca5">
        <ecaRules id="eca3" name="onClick" trigger="signup_create" target="signup_user_sa
ve" priority="100"/>
        <ecaRules id="eca4" inParameterEdges="parameter1" name="run" trigger="signup_user
_save" target="user_login"/>
        <ecaRules id="eca5" name="click" trigger="signup_create" target="signup_form_hide
"/>
    </children>
</scopes>
<scopes xsi:type="iaml.visual:Frame" id="signup_email_failed" name="Email Sending Fai
led" rules="eca1">
    <children xsi:type="iaml.visual:Label" id="email_label1" name="label1" defaultValue
="An e-mail could not be sent to the e-mail address" renderOrder="1"/>
    <children xsi:type="iaml.visual:Label" id="email_label2" name="label3" defaultValue
="and you may wish to try signing up again." renderOrder="3"/>
    <children xsi:type="iaml.visual:Label" inWires="set2" id="label1" name="&lt;email a
ddress>" type="builtin_iamlEmail" renderOrder="2"/>
</scopes>
</scopes>
<scopes xsi:type="iaml.scopes:Session" id="session_user" name="User Session">
    <scopes xsi:type="iaml.visual:Frame" id="login" name="Login"/>
    <scopes xsi:type="iaml.visual:Frame" id="user_logout" name="logout"/>
    <scopes xsi:type="iaml.visual:Frame" id="user_invalid" name="Invalid Credentials"
rules="eca11"/>
    <scopes xsi:type="iaml.visual:Frame" id="user_tickets" name="Your Tickets">
        <iterators id="user_tickets_iterator" name="ticket iterator" outWires="set5"

```

```

inParameterEdges="parameter4" query="user_id = :id" limit="5" outSelects="edge2">
    <wires xsi:type="iaml.wires:SetWire" id="set5" name="set" from="user_tickets_iter
ator" to="user_tickets_list"/>
    <selectEdges id="edge2" from="user_tickets_iterator" to="tickets_db"/>
</iterators>
    <children xsi:type="iaml.visual:IteratorList" outWires="detail1" inWires="set5" id=
"user_tickets_list" name="Your Ticket List">
        <wires xsi:type="iaml.wires:DetailWire" id="detail1" name="detail" from="user_tic
kets_list" to="view_ticket"/>
        <overriddenNames>user_id</overriddenNames>
    </children>
</scopes>
<scopes xsi:type="iaml.visual:Frame" id="view_ticket" name="View Ticket" inWires="det
ail1">
    <children xsi:type="iaml.visual:InputForm" id="form2" name="View Ticket">
        <overriddenNames>user_id</overriddenNames>
        <children xsi:type="iaml.visual:Label" id="label2" name="event_id"
outParameterEdges="parameter2" type="xsd_integer">
            <parameterEdges id="parameter2" name="id" parameterValue="label2" parameterTerm
="eca6"/>
        </children>
        <children xsi:type="iaml.visual:Button" id="button1" name="View Event"
renderOrder="100" listeners="eca6">
            <ecaRules id="eca6" inParameterEdges="parameter2" name="onClick" trigger="butto
n1" target="view_event"/>
        </children>
    </children>
</scopes>
<iterators id="user_iterator" name="current instance" outSelects="edge3">
    <selectEdges id="edge3" from="user_iterator" to="named_users_db"/>
    <functions xsi:type="iaml.operations:ActivityPredicate" id="user_logged_in" name="n
ot empty"/>
    <currentInstance id="user_instance" name="instance">
        <featureInstances xsi:type="iaml.domain:DomainAttributeInstance"
outParameterEdges="parameter3 parameter4" id="ai1" type="xsd_integer" outExtendsEdges="ed
ge4" name="id">
            <parameterEdges id="parameter3" name="id" parameterValue="ai1" parameterTerm="m
anager_instance"/>
            <parameterEdges id="parameter4" name="id" parameterValue="ai1" parameterTerm="u
ser_tickets_iterator"/>
            <extendsEdges id="edge4" from="ai1" to="named_user_id"/>
        </featureInstances>
    </currentInstance>
</iterators>
<accessHandlers id="user_ach" name="Require Named User" outRequiresEdges="edge5">
    <requiresEdges id="edge5" from="user_ach" to="role_user"/>
</accessHandlers>
<loginHandlers id="user_login_handler" name="Session Login Handler" inParameterEdges=
"parameter12" type="USER"/>
</scopes>
<scopes xsi:type="iaml.scopes:Session" id="session1" name="Session Login Handler login"
>
    <scopes xsi:type="iaml.visual:Frame" id="user_login" name="login" rules="eca4">
        <parameters id="qp1" name="message" outParameterEdges="parameter5" defaultValue="">
            <parameterEdges id="parameter5" name="message" parameterValue="qp1" parameterTerm

```

```

="eca9"/>
    </parameters>
    <children xsi:type="iaml.visual:Label" id="login_message" name="message">
        <operations xsi:type="iaml.operations:ActivityOperation" rules="eca9" id="login_message_init" name="initialise or hide message label">
            <nodes xsi:type="iaml.operations:StartNode" id="startnode1" outExecutions="edge10"/>
            <nodes xsi:type="iaml.operations:DecisionNode" id="decisionnode1" name="empty?" outExecutions="edge11 edge12" inExecutions="edge10" inFlows="edge9"/>
            <nodes xsi:type="iaml.operations:FinishNode" id="finishnode1" inExecutions="edge14 edge15"/>
            <nodes xsi:type="iaml.operations:SetNode" id="setnode1" name="set value" outExecutions="edge13" inExecutions="edge11" outFlows="edge8" inFlows="edge7"/>
            <nodes xsi:type="iaml.operations:ExternalValue" id="externalvalue1" inFlows="edge8" value="login_message_value"/>
            <nodes xsi:type="iaml.operations:OperationCallNode" id="operationcallnode1" outExecutions="edge14" inExecutions="edge13" name="call show" listeners="eca7">
                <ecaRules id="eca7" name="run" trigger="operationcallnode1" target="login_message_show"/>
            </nodes>
            <nodes xsi:type="iaml.operations:OperationCallNode" id="operationcallnode2" outExecutions="edge15" inExecutions="edge12" name="call hide" listeners="eca8">
                <ecaRules id="eca8" name="run" trigger="operationcallnode2" target="login_message_hide"/>
            </nodes>
            <dataEdges id="edge7" from="activityparameter1" to="setnode1"/>
            <dataEdges id="edge8" from="setnode1" to="externalvalue1"/>
            <dataEdges id="edge9" from="activityparameter1" to="decisionnode1"/>
            <executionEdges id="edge10" from="startnode1" to="decisionnode1"/>
            <executionEdges id="edge11" name="n" from="decisionnode1" to="setnode1"/>
            <executionEdges id="edge12" name="y" from="decisionnode1" to="operationcallnode2"/>
            <executionEdges id="edge13" from="setnode1" to="operationcallnode1"/>
            <executionEdges id="edge14" from="operationcallnode1" to="finishnode1"/>
            <executionEdges id="edge15" from="operationcallnode2" to="finishnode1"/>
            <parameters id="activityparameter1" name="message" outFlows="edge7 edge9"/>
        </operations>
        <operations xsi:type="iaml:BuiltinOperation" rules="eca8" id="login_message_hide" name="hide"/>
        <operations xsi:type="iaml:BuiltinOperation" rules="eca7" id="login_message_show" name="show"/>
        <onAccess id="login_message_onAccess" name="onAccess" listeners="eca9">
            <ecaRules id="eca9" inParameterEdges="parameter5" name="run" trigger="login_message_onAccess" target="login_message_init"/>
        </onAccess>
        <fieldValue id="login_message_value" name="fieldValue"/>
    </children>
</scopes>
</scopes>
<scopes xsi:type="iaml.visual:Frame" id="all_events" name="Browse Events">
    <iterators id="all_events_iterator" name="list events iterator" outWires="set6" inParameterEdges="parameter6" query="(matches(title, :query) or matches(description, :query) or matches(venue, :query)) and event_date > now()" limit="5" outSelects="edge16">
        <wires xsi:type="iaml.wires:DetailWire" id="detail2" name="detail" from="all_events_list" to="view_event"/>
    </iterators>
</scopes>

```

```

        <wires xsi:type="iaml.wires:SetWire" id="set6" name="set" from="all_events_iterator
" to="all_events_list"/>
        <selectEdges id="edge16" from="all_events_iterator" to="events_db"/>
    </iterators>
    <children xsi:type="iaml.visual:IteratorList" outWires="detail2" inWires="set6" id="a
ll_events_list" name="events list" renderOrder="100">
        <overriddenNames>created</overriddenNames>
        <overriddenNames>updated</overriddenNames>
        <overriddenNames>description</overriddenNames>
        <children xsi:type="iaml.visual:Label" outWires="set7" id="all_events_venue" name="
venue" type="builtin_iamlAddress" renderOrder="30">
            <wires xsi:type="iaml.wires:SetWire" id="set7" name="set" from="all_events_venue"
to="all_events_map"/>
        </children>
        <children xsi:type="iaml.visual:Label" id="label3" name="title" renderOrder="20"/>
        <children xsi:type="iaml.visual:Button" id="button2" name="link" renderOrder="10"/>
        <children xsi:type="iaml.visual:Label" id="label4" name="event_date" renderOrder="4
0"/>
        <children xsi:type="iaml.visual:Label" id="label5" name="tickets_left" renderOrder=
"50"/>
    </children>
    <children xsi:type="iaml.visual:InputTextField" id="all_events_search" name="Event Se
arch" outParameterEdges="parameter6">
        <parameterEdges id="parameter6" name="query" parameterValue="all_events_search"
parameterTerm="all_events_iterator"/>
    </children>
    <children xsi:type="iaml.visual:Map" inWires="set7" id="all_events_map" name="Events
Map"/>
</scopes>
<scopes xsi:type="iaml.visual:Frame" id="view_event" name="View Event" inWires="detail2
" rules="eca10 eca16 eca6">
    <children xsi:type="iaml.visual:InputForm" id="view_event_form" name="View Event">
        <overriddenNames>created</overriddenNames>
        <overriddenNames>updated</overriddenNames>
        <overriddenNames>id</overriddenNames>
        <children xsi:type="iaml.visual:Label" id="view_event_title" name="title" type="xsd
_string"/>
        <children xsi:type="iaml.visual:Label" id="view_event_description" name="descriptio
n" type="xsd_string"/>
        <children xsi:type="iaml.visual:Label" outWires="set8" id="view_event_venue" name="
venue" type="xsd_string">
            <wires xsi:type="iaml.wires:SetWire" id="set8" name="set" from="view_event_venue"
to="view_event_map_point"/>
        </children>
        <children xsi:type="iaml.visual:Label" id="view_event_date" name="event_date" type=
"builtin_iamlDateTime"/>
        <children xsi:type="iaml.visual:Label" id="view_event_tickets" name="tickets_left"
type="xsd_integer"/>
    </children>
    <children xsi:type="iaml.visual:MapPoint" inWires="set8" id="view_event_map_point"
name="Event Map"/>
</scopes>
<scopes xsi:type="iaml.visual:Frame" id="events_rss" name="New Events" render="RSS20">
    <iterators id="events_rss_iterator" name="new events" outWires="set9" limit="5"
orderBy="attribute3" orderAscending="false" outSelects="edge17">

```

```

        <wires xsi:type="iaml.wires:SetWire" id="set9" name="set" from="events_rss_iterator
" to="events_rss_item"/>
        <selectEdges id="edge17" from="events_rss_iterator" to="events_db"/>
    </iterators>
    <children xsi:type="iaml.visual:InputForm" inWires="set9" id="events_rss_item" name="
Feed Item">
        <children xsi:type="iaml.visual:Button" id="events_rss_link" name="link" listeners=
"eca10">
            <ecaRules id="eca10" inParameterEdges="parameter7" name="onClick" trigger="events
_rss_link" target="view_event"/>
        </children>
        <children xsi:type="iaml.visual:Label" id="events_rss_id" name="id"
outParameterEdges="parameter7" type="xsd_integer">
            <parameterEdges id="parameter7" name="id" parameterValue="events_rss_id1"
parameterTerm="eca10"/>
        </children>
    </children>
</scopes>
<scopes xsi:type="iaml.scopes:Session" id="session_manager" name="Manager Session">
    <entryGate id="session_manager_gate" name="require manager instance" conditions="cond
ition1" listeners="eca11">
        <ecaRules id="eca11" name="fail" trigger="session_manager_gate" target="user_invali
d"/>
    </entryGate>
    <scopes xsi:type="iaml.visual:Frame" id="manager_logout" name="logout"/>
    <scopes xsi:type="iaml.visual:Frame" id="manager_login" generatedRule="" name="Login"
/>

    <scopes xsi:type="iaml.visual:Frame" id="event_new" name="Create New Event">
        <iterators id="event_new_iterator" name="new event" inWires="sync2" query="new"
outSelects="edge18">
            <selectEdges id="edge18" from="event_new_iterator" to="events_db"/>
            <operations xsi:type="iaml:BuiltinOperation" rules="eca13" id="event_new_save"
name="save" listeners="eca12">
                <ecaRules id="eca12" name="run" trigger="event_new_save" target="builtinoperati
on2"/>
            </operations>
            <operations xsi:type="iaml:BuiltinOperation" rules="eca12" id="builtinoperation2"
name="new"/>
        </iterators>
        <children xsi:type="iaml.visual:InputForm" outWires="sync2" id="event_new_form"
name="New Event">
            <wires xsi:type="iaml.wires:SyncWire" id="sync2" name="sync" from="event_new_form
" to="event_new_iterator"/>
            <children xsi:type="iaml.visual:Button" id="event_new_save_button" name="Save"
renderOrder="100" listeners="eca13">
                <ecaRules id="eca13" name="onClick" trigger="event_new_save_button" target="eve
nt_new_save"/>
            </children>
        </children>
    </scopes>
    <scopes xsi:type="iaml.visual:Frame" id="events_list" name="All Events">
        <iterators id="iterator2" name="all events iterator" outWires="set10" query="any"
limit="5" orderBy="attribute6" orderAscending="false" outSelects="edge19">
            <wires xsi:type="iaml.wires:SetWire" id="set10" name="set" from="iterator2" to="a
ll_events_list2"/>

```



```

        <selectEdges id="edge19" from="iterator2" to="events_db"/>
    </iterators>
    <children xsi:type="iaml.visual:IteratorList" inWires="set10" id="all_events_list2"
name="All Events">
        <overriddenNames>created</overriddenNames>
        <overriddenNames>updated</overriddenNames>
        <overriddenNames>description</overriddenNames>
        <children xsi:type="iaml.visual:Label" id="all_events_event_id" name="id"
outParameterEdges="parameter8">
            <parameterEdges id="parameter8" name="id" parameterValue="all_events_event_id"
parameterTerm="eca14"/>
        </children>
        <children xsi:type="iaml.visual:Button" id="all_events_edit" name="edit"
listeners="eca14">
            <ecaRules id="eca14" inParameterEdges="parameter8" name="onClick" trigger="all_
events_edit" target="event_edit"/>
        </children>
    </children>
</scopes>
<scopes xsi:type="iaml.visual:Frame" id="event_edit" name="Edit Event" rules="eca14">
    <operations xsi:type="iaml:BuiltinOperation" rules="eca15" id="builtinoperation3"
name="set current id"/>
    <onAccess id="event1" name="onAccess" listeners="eca15">
        <ecaRules id="eca15" inParameterEdges="parameter11" name="run" trigger="event1"
target="builtinoperation3"/>
    </onAccess>
    <values id="value2" name="current id" type="xsd_integer" outParameterEdges="paramet
er10 parameter9" defaultValue="0">
        <parameterEdges id="parameter9" name="id" parameterValue="value2" parameterTerm="
eca16"/>
    </values>
    <parameters id="qp2" name="id" outParameterEdges="parameter11">
        <parameterEdges id="parameter10" name="id" parameterValue="value2" parameterTerm=
"iterator3"/>
        <parameterEdges id="parameter11" name="id" parameterValue="qp2" parameterTerm="ec
a15"/>
    </parameters>
    <iterators id="iterator3" name="selected event" outWires="sync3" inParameterEdges="
parameter10" query="id = :id" outSelects="edge20">
        <wires xsi:type="iaml.wires:SyncWire" id="sync3" name="sync" from="iterator3" to=
"edit_event_form"/>
        <selectEdges id="edge20" from="iterator3" to="events_db"/>
        <operations xsi:type="iaml:BuiltinOperation" rules="eca17" id="builtinoperation4"
name="save" listeners="eca16">
            <ecaRules id="eca16" inParameterEdges="parameter9" name="navigate" trigger="bui
ltinoperation4" target="view_event"/>
        </operations>
    </iterators>
    <children xsi:type="iaml.visual:InputForm" inWires="sync3" id="edit_event_form"
name="Edit Event">
        <overriddenNames>id</overriddenNames>
        <children xsi:type="iaml.visual:Button" id="edit_event_save" name="Save"
renderOrder="100" listeners="eca17">
            <ecaRules id="eca17" name="onClick" trigger="edit_event_save" target="bui
ltinoperation4"/>

```

```

        </children>
    </children>
</scopes>
    <iterators id="manager_instance" name="current instance" inParameterEdges="parameter3
" query="named_user.id = :id" outSelects="edge21">
        <selectEdges id="edge21" from="manager_instance" to="source1"/>
        <functions xsi:type="iaml.operations:ActivityPredicate" id="manager_logged_in" name
="not empty" conditioned="condition1">
            <conditionEdges id="condition1" name="condition" function="manager_logged_in"
conditioned="session_manager_gate"/>
        </functions>
    </iterators>
</scopes>
    <types xsi:type="iaml.users:Role" name="named_user" id="role_user" outExtendsEdges="edg
e22" inExtendsEdges="edge24" inSchemas="schema1" inRequiresEdges="edge5">
        <eStructuralFeatures xsi:type="iaml.domain:DomainAttribute" name="name" eType="xsd_st
ring" id="named_user_name"/>
        <eStructuralFeatures xsi:type="iaml.domain:DomainAttribute" name="id" eType="xsd_inte
ger" id="named_user_id" inExtendsEdges="edge4" primaryKey="true"/>
        <parameterEdges id="parameter12" name="required" parameterValue="role1" parameterTerm
="user_login_handler"/>
        <extendsEdges id="edge22" from="role_user" to="role1"/>
    </types>
    <types xsi:type="iaml.users:Role" name="admin" id="role_admin" outExtendsEdges="edge23"
inSchemas="schema3">
        <extendsEdges id="edge23" from="role_admin" to="role_manager"/>
    </types>
    <types xsi:type="iaml.users:Role" name="manager" id="role_manager" outExtendsEdges="edg
e24" inExtendsEdges="edge23" inSchemas="schema2">
        <extendsEdges id="edge24" from="role_manager" to="role_user"/>
    </types>
    <types xsi:type="iaml.users:Role" name="User" id="role1" outParameterEdges="parameter12
" inExtendsEdges="edge22" inSchemas="schema4"/>
    <types name="Event" id="type_event" inSchemas="schema5">
        <eStructuralFeatures xsi:type="iaml.domain:DomainAttribute" name="venue" eType="built
in_iamlAddress" id="event_venue"/>
        <eStructuralFeatures xsi:type="iaml.domain:DomainAttribute" name="title" eType="xsd_s
tring" id="attribute1"/>
        <eStructuralFeatures xsi:type="iaml.domain:DomainAttribute" name="description" eType=
"xsd_string" id="attribute2"/>
        <eStructuralFeatures xsi:type="iaml.domain:DomainAttribute" name="created" eType="bui
ltn_iamlDateTime" id="attribute3"/>
        <eStructuralFeatures xsi:type="iaml.domain:DomainAttribute" name="updated" eType="bui
ltn_iamlDateTime" id="attribute4"/>
        <eStructuralFeatures xsi:type="iaml.domain:DomainAttribute" name="tickets_left" eType
="xsd_integer" id="attribute5"/>
        <eStructuralFeatures xsi:type="iaml.domain:DomainAttribute" name="event_date" eType="
buitln_iamlDateTime" id="attribute6"/>
        <eStructuralFeatures xsi:type="iaml.domain:DomainAttribute" name="id" eType="xsd_inte
ger" id="attribute7" primaryKey="true"/>
    </types>
    <types name="Ticket" id="type1" inSchemas="schema6">
        <eStructuralFeatures xsi:type="iaml.domain:DomainAttribute" name="user_id" eType="xsd
_integer" id="attribute8"/>
        <eStructuralFeatures xsi:type="iaml.domain:DomainAttribute" name="event_id" eType="xs

```

```

d_integer" id="attribute9"/>
</types>
<sources id="named_users_db" name="users.db (named user)" file="users.db" outSchemas="s
chema1" inSelects="edge1 edge3">
  <schemaEdges id="schema1" from="named_users_db" to="role_user"/>
  <schemaEdges id="schema2" from="source1" to="role_manager"/>
  <schemaEdges id="schema3" from="source2" to="role_admin"/>
</sources>
<sources id="source1" name="users.db (manager)" file="users.db" outSchemas="schema2"
inSelects="edge21"/>
<sources id="source2" name="users.db (admin)" file="users.db" outSchemas="schema3"/>
<sources id="users_db" name="users.db (user)" file="users.db" outSchemas="schema4">
  <schemaEdges id="schema4" from="users_db" to="role1"/>
</sources>
<sources id="events_db" name="events.db" file="events.db" outSchemas="schema5"
inSelects="edge16 edge17 edge18 edge19 edge20">
  <schemaEdges id="schema5" from="events_db" to="type_event"/>
</sources>
<sources id="tickets_db" name="tickets.db" file="tickets.db" outSchemas="schema6"
inSelects="edge2">
  <schemaEdges id="schema6" from="tickets_db" to="type1"/>
</sources>
<xsdDataTypes name="builtin:iamlInteger" id="builtin_iamlInteger">
  <definition href="platform:/plugin/org.openiaml.model/model/datatypes.xsd//iamlInteg
er;XSDSimpleTypeDefinition"/>
</xsdDataTypes>
<xsdDataTypes name="xsd:boolean" id="xsd_boolean">
  <definition href="platform:/plugin/org.eclipse.xsd/cache/www.w3.org/2001/XMLSchema.xs
d//boolean;XSDSimpleTypeDefinition=10"/>
</xsdDataTypes>
<xsdDataTypes name="xsd:integer" id="xsd_integer">
  <definition href="platform:/plugin/org.eclipse.xsd/cache/www.w3.org/2001/XMLSchema.xs
d//integer;XSDSimpleTypeDefinition=40"/>
</xsdDataTypes>
<xsdDataTypes name="xsd:string" id="xsd_string">
  <definition href="platform:/plugin/org.eclipse.xsd/cache/www.w3.org/2001/XMLSchema.xs
d//string;XSDSimpleTypeDefinition=9"/>
</xsdDataTypes>
<xsdDataTypes name="builtin:iamlAddress" id="builtin_iamlAddress">
  <definition href="platform:/plugin/org.openiaml.model/model/datatypes.xsd//iamlAddre
ss;XSDSimpleTypeDefinition=2"/>
</xsdDataTypes>
<xsdDataTypes name="builtin:iamlDateTime" id="builtin_iamlDateTime">
  <definition href="platform:/plugin/org.openiaml.model/model/datatypes.xsd//iamlDateT
ime;XSDSimpleTypeDefinition=4"/>
</xsdDataTypes>
<xsdDataTypes name="builtin:iamlEmail" id="builtin_iamlEmail">
  <definition href="platform:/plugin/org.openiaml.model/model/datatypes.xsd//iamlEmail
;XSDSimpleTypeDefinition=3"/>
</xsdDataTypes>
<xsdDataTypes name="builtin:iamlOpenIDURL" id="builtin_iamlOpenIDURL">
  <definition href="platform:/plugin/org.openiaml.model/model/datatypes.xsd//iamlOpenI
DURL;XSDSimpleTypeDefinition=6"/>
</xsdDataTypes>
<xsdDataTypes name="builtin:iamlString" id="builtin_iamlString">

```

```
<definition href="platform:/plugin/org.openiaml.model/model/datatypes.xsd#/iamlString;XSDSimpleTypeDefinition=1"/>
</xsdDataTypes>
<xsdDataTypes name="builtin:iamlURL" id="builtin_iamlURL">
  <definition href="platform:/plugin/org.openiaml.model/model/datatypes.xsd#/iamlURL;XSDSimpleTypeDefinition=5"/>
</xsdDataTypes>
</iaml:InternetApplication>
```

## Appendix G

# Language-specific Metrics of the Ticket 2.0 Implementations

As discussed earlier in Section 8.3, the *Ticket 2.0* benchmarking application was implemented using both the Symfony framework and the IAML language, allowing for a comparison of the two implementations in terms of a set of system metrics. In this appendix, the full set of metrics will be provided, both in terms of the programming languages used, and the different types of effort expended. The language-specific metrics were calculated using the *Cloc* code analysis tool<sup>1</sup>.

In Tables G.1 and G.2, system metrics of the two Symfony-based implementations of Ticket 2.0 named *Ticketsf* and *Ticketsf-mini* are presented. The implementation of these two applications required manual implementation within five programming languages, including the definition of server-side logic in PHP; client-side logic in Javascript; and configuration data in YAML. YAML configuration data was then used to generate a significant portion of the generated web application, particularly in terms of XML. These generated applications were then integrated with the Symfony runtime framework to complete the application.

In Table G.3, system metrics of the IAML-based implementation of Ticket 2.0 named *Ticketiaml* are presented. The implementation of *Ticketiaml* required the manual implementation of a single IAML model instance, a custom CSS stylesheet, and a number of templates used for e-mails and navigation. Through the model completion process, the IAML model instance was then transformed into the *intended model*, and this intended model was then transformed into a significant portion of the generated web application. This generated application could then be integrated with the IAML runtime framework (Section 7.6.3) to complete the application.

In these system metrics, unrelated technologies such as Dos Batch Script (.bat), make and Bourne Shell script (.sh) are not investigated. These are generally designed for use within conventional software development or systems administrator environments rather than within a RIA development environment, and are not considered to form part of a Ticket 2.0 implementation.

---

<sup>1</sup>The *Cloc* code analysis tool is available online at <http://cloc.sourceforge.net/>, and is provided under the GNU General Public License.

Number of Files				
Language	Manual	Generated	Runtime	Total
PHP	56	314	734	1,104
XML	0	4	4	8
Javascript	11	141	46	198
HTML	3	80	1	84
CSS	5	36	8	49
YAML	25	2	27	54
SQL	0	2	0	2
XSD	0	0	2	2
XSLT	0	0	1	1
DTD	0	0	1	1
Total	100	579	824	1,503

Lines of Code				
Language	Manual	Generated	Runtime	Total
PHP	992	58,031	71,461	130,484
XML	0	129,052	662	129,714
Javascript	1,409	20,701	9,650	31,760
HTML	3	30,696	4	30,703
CSS	125	1,418	1,141	2,684
YAML	679	41	385	1,105
SQL	0	395	0	395
XSD	0	0	300	300
XSLT	0	0	198	198
DTD	0	0	122	122
Total	3,208	240,334	83,923	327,465

Table G.1: File-based metrics of the development of Ticketsf within Symfony 1.0.12

Number of Files				
Language	Manual	Generated	Runtime	Total
PHP	26	38	2,023	2,087
XML	0	0	68	68
DTD	0	0	7	7
YAML	16	0	96	112
CSS	1	0	5	6
XSD	0	0	2	2
XSLT	0	0	1	1
Javascript	2	0	3	5
SQL	0	1	1	2
Total	45	39	2,206	2,290

Lines of Code				
Language	Manual	Generated	Runtime	Total
PHP	570	3,971	155,036	159,577
XML	0	0	6,384	6,384
DTD	0	0	3,535	3,535
YAML	194	0	1,637	1,831
CSS	192	0	699	891
XSD	0	0	346	346
XSLT	0	0	187	187
Javascript	54	0	102	156
SQL	0	58	27	85
Total	1,010	4,029	167,953	172,992

Table G.2: File-based metrics of the development of Ticketsf-mini within Symfony 1.4.8

<b>Number of Files</b>				
<b>Language</b>	<b>Manual</b>	<b>Generated</b>	<b>Runtime</b>	<b>Total</b>
PHP	2	568	67	637
IAML	1	1	0	2
XML	0	1	1	2
CSS	1	0	2	3
Javascript	0	1	9	10
HTML	0	1	8	9
SQL	3	0	0	3
Total	7	572	87	666

<b>Lines of Code</b>				
<b>Language</b>	<b>Manual</b>	<b>Generated</b>	<b>Runtime</b>	<b>Total</b>
PHP	53	75,824	6,877	82,754
IAML	321	4,212	0	4,533
XML	0	10	50	60
CSS	179	0	183	362
Javascript	0	1,606	6,199	7,805
HTML	0	47	197	244
SQL	53	0	0	53
Total	606	81,699	13,506	95,811

Table G.3: File-based metrics of the development of Ticketiaml within IAML 0.6



## Appendix H

# Cognitive Dimensions Evaluation of IAML

As discussed in Section 3.4.4, the *cognitive dimensions* framework proposed by Green and Petre [133] may be used to evaluate the usability of a visual modelling language. In this appendix, the thirteen dimensions of this framework will be briefly summarised, and evaluated against the visual representation of IAML as defined in Section 7.4.3.

The results of this subjective evaluation performed in Section 8.5.2 broadly suggest that the IAML visual representation satisfies each of these cognitive dimensions, but one cannot objectively quantify the measure in which a dimension is considered. Each of these dimensions should be considered when proposing a change to the existing IAML visual notation – such as introducing a new abstraction for the visual notation, modifying the structure of the IAML metamodel, or modifying one of the *Diagram Extensions* discussed in Section 7.4.4 – as the change can impact each of these cognitive dimensions in different ways [133, pg. 164–165].

### Abstraction Gradient

What are the minimum and maximum levels of abstraction? Can fragments be encapsulated?

It appears that this measurement simultaneously discusses the level of abstractions used in a language, and the depth in which users can – or must – develop their own abstractions. IAML is a language built exclusively on many different types of abstractions, but this diversity is necessary due to the wide range of functionality that a RIA possesses. Nevertheless, the abstractions used have been selected based on existing languages and visual metaphors, as discussed in Chapter 5.

As defined by Green and Petre [133], IAML may be considered somewhat *abstraction-tolerant*. The language may be used “as-is” without having to create new abstractions, however as the environment is developed within a model-driven environment, new abstractions may be independently defined through third-party extensions of the proof-of-concept implementation. The support for defining new abstractions is dependent on tool support, as discussed earlier in Section 9.3.8.

## Closeness of Mapping

What ‘programming games’ need to be learned?

As discussed in Section 5.15, each layer of the hierarchical visual notation of IAML is designed from an existing visual notation – such as UML class diagrams for domain modelling, or user interface design for interface modelling. This allows the visual syntax to map closely to the existing metaphors understood by the intended language users.

## Consistency

When some of the language has been learnt, how much of the rest can be inferred?

There are three core concepts of IAML visual syntax that need to be understood by a model developer: hierarchy; event-condition-action rules; and model completion. As discussed in Section 7.4.3, the IAML visual model adheres to a consistent design based on the information capacities of the syntax variables, and the standard Eclipse colour palette. The design decisions of the underlying IAML metamodel also respect consistency, as discussed in Section 5.2.

## Diffuseness/Terseness

How many symbols or graphic entities are required to express a meaning?

As discussed by Green and Petre [133], this measurement is difficult to quantify, as the terseness of a particular problem depends heavily on its closeness to the problem domain. As each metamodel element in IAML has a corresponding visual element, the language can be considered to be quite verbose; however, many of these elements are often only generated through the model completion process.

## Error-proneness

Does the design of the notation induce ‘careless mistakes’?

This dimension is also impacted by the *consistency* of the syntax and the language. To evaluate this dimension, four example categories of potential developer errors have been identified and evaluated, along with proposed techniques to reduce their occurrence.

1. *Invalid model instances*: A model developer may inadvertently create a model instance that violates the structural constraints of the metamodel; for example, not providing a **Complex Term** with the appropriate **Parameters** for the given **Function**. As discussed in Section 7.7.1, property verification is evaluated upon every model instance change to combat this, and violations displayed in realtime.
2. *Lack of context*: Due to the hierarchical nature of IAML model instances, a model developer may incorrectly refer to another model element due to a lack of context information of that element. For example, a model instance may validly contain two **Input Text Fields** with the *same* name contained within *different* elements. To combat this, additional visual notations – such as container name, metaclass name, and breadcrumbing – are provided by the *Diagram Extensions* component, discussed in Section 7.4.4.

3. *Misuse of model completion:* The model completion process occurs independently of the development environment, and model developers may not be fully aware of the semantics of the model completion rules used, causing errors in their designed model instance. Approaches to reduce these errors include adequate documentation (Section 7.5.2), and providing tools to integrate the two environments together (Section 7.8). Additionally, since model completion is additive and cannot remove existing elements, there is no risk of the model developer losing the results of manual development effort [369].
4. *Design errors:* Finally, a model developer may inadvertently create a model instance which satisfies the structural constraints of the IAML metamodel, but possesses a design-time flaw such as infinite redirection. As discussed in Section 7.7.4, recursive constraints and model checking verification may optionally be used to identify these cases.

### Hard Mental Operations

Are there places where the user needs to resort to fingers or pencilled annotation to keep track of what's happening?

A “hard mental operation” is defined as a construct that becomes incomprehensible as many are connected together, with no alternative way of designing the operation [133]. In IAML, these problematic constructs are avoided by using the Composite design pattern [118, pg. 163–173]. For example, a number of [Conditions](#) can be composed into a single [Complex Condition](#); a number of user interface elements ([Visible Things](#)) can be composed into a single wrapped [Frame](#); or a number of [Operations](#) can be composed into [Activity Operation](#). Mental complexity is also reduced by defining a [name](#) for most model elements.

### Hidden Dependencies

Is every dependency overtly indicated in both directions? Is the indication perceptual or only symbolic?

A hidden dependency is defined as “a relationship between two components such that one of them is dependent on the other, but that the dependency is not fully visible.” By default, hierarchical graphical editors developed using GMF suffer from this problem, as relationships with other elements are often not displayed. In response, the GetShortcuts controller was introduced (Section 7.4.4) to reduce this problem by fully populating all of the relationships of a given element for a given context.

### Premature Commitment

Do programmers have to make decisions before they have the information they need?

IAML reduces premature commitment by encoding common design patterns through model completion, and there are few cardinality limits. Model refactoring tools such as the *move into separate model* action can also reduce this concern.

## Progressive Evaluation

Can a partially-complete program be executed to obtain feedback on “How am I doing”?

Other than the model verification feedback provided through different verification languages, a model instance may be evaluated by generating the model instance into web application source code. Model completion also assists in progressive evaluation, as inference can automatically complete much of a base model. The smallest valid IAML model instance consists of only two elements – a single empty [Frame](#) contained by a root [Internet Application](#).

## Role-expressiveness

Can the reader see how each component of a program relates to the whole?

While IAML is designed according to the three-level viewpoint architecture of the MDA, the language does not directly integrate with a requirements-based metamodel. Nevertheless, GMF-based graphical editors support arbitrary notes and file attachments as secondary notation, allowing a developer to provide additional documentation.

## Secondary Notation

Can programmers use layout, colour, or other cues to convey extra meaning, above and beyond the ‘official’ semantics of the language?

One of the built-in features of a GMF-based graphical editor is that many forms of secondary notation can be applied to all model elements; a model developer may override any of the colours, lines, fonts or arrows used for any of the nodes or edges in a given representation. All secondary notation is serialised in the diagram metamodel (`iaml_diagram` files), which prevents the underlying model instance from being polluted with this representation information. The proof-of-concept implementation of IAML therefore strongly supports this dimension.

## Viscosity

How much effort is required to perform a single change?

In terms of the visual representation of IAML model instances, all nodes, edges and labels in a single view are elastic, which allows the arbitrary relayout and resizing of diagram elements. GMF supports the automatic relayout of model elements, which can significantly reduce the time necessary to make a visual change, and also supports reinitialising a diagram from a model instance from scratch. Model elements can also be cut, copied and pasted between views, providing editor functionality similar to that provided by a textual editor.

Certain model refactoring operations can still require a significant amount of effort – for example, changing the underlying type of an element, or replacing a [Single Condition](#) with a [Composite Condition](#) – and future work includes the development of additional model refactoring tools<sup>1</sup>.

---

<sup>1</sup>One existing model refactoring tool is the *move into separate model* action, discussed earlier in Section 7.4.5.

**Visibility and Juxtaposability**

Is every part of the code simultaneously visible (assuming a large enough display), or is it at least possible to juxtapose any two parts side-by-side at will? If the code is dispersed, is it at least possible to know in what order to read it?

Since IAML is designed as a hierarchical metamodel, it is possible to have two independent views of different parts of a model instance. The GMF framework supports displaying multiple views of a single model instance simultaneously, permitting good juxtaposability. Other features such as GMF's *Outline View* and the breadcrumbing diagram extension also improve the visibility and context of a model instance.



# Appendix I

## Model Element Reference

In this appendix, the complete definition of the IAML metamodel will be provided and discussed. This documentation includes the full definition of the intended behaviour of each metamodel element and attribute, and the proof-of-concept implementation of IAML – along with its extensive test suite, as discussed in Section 7.9 – supports this appendix as a reference implementation of the language.

This documentation is organised in terms of each type or abstract type defined in the IAML metamodel, and presented alphabetically. Each element type has a brief description (a *tagline*), and may optionally have additional documentation describing the intended behaviour of instances of that element type as an extension. For example, the documentation for [Domain Iterator](#) (pg. 395) describes the behaviour of the containing scope and failure handler mechanisms with greater detail than the discussion provided earlier in Section 5.6.3.

For each element type, the subtypes and supertypes of the element are described, along with a full definitions of the attributes, references, children (also known as containment references), functions, operations and events<sup>1</sup> that an instance of that type may define. Each element type also includes a discussion on the intent of the model completion rules that may apply to instances of this element, as discussed earlier in Section 7.5.2. These represent a verbalisation of each model completion rule itself through an `@inference` documentation tag, and represents a single Drools rule within the *Rule Set* component. A single model completion rule may involve more than a single element type, so a rule verbalisation may be reused across many element types.

This documentation source was generated automatically by the ModelDoc documentation framework, as discussed by Wright [366]. In particular, this documentation source represents the combination of parsing a number of different documentation sources, including the EMF metamodel instance and its annotations; GMF graphical editor definitions; Drools model completion rules; exported graphical instances of example models; source code comments throughout the proof-of-concept implementation and its test cases; and hand-written documentation provided through HTML files. This documentation is also available online in a hypertext format at <http://openiaml.org/model/>, and within the attached media described in Appendix E.

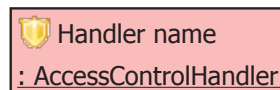
---

<sup>1</sup>The EMF metamodel does not natively support IAML concepts such as events; to bridge these IAML concepts over into the EMF domain, *functions*, *operations* and *events* are defined as EMF `EReferences` typed as a [Function](#), [Operation](#) or [Event](#).

## I.1 Access Control Handler

Within a [Scope](#), enforces that all user access must satisfy certain access control requirements such as [Roles](#) and [Permissions](#), as specified by outgoing [Requires Edges](#).

**Direct supertypes** [Action Edge Source](#), [Contains Functions](#), [Contains Operations](#), [Contains Values](#), [Contains Wires](#), [Generates Elements](#), [Named Element](#), [Parameter Edge Destination](#), [Requires Edges Source](#)



A [Scope](#) may contain at most one [Access Control Handler](#). All access to elements within this [Scope](#), or its children [Scopes](#), must first *satisfy* the [Access Control Handler](#). A [Access Control Handler](#) must have at least one outgoing [Requires Edge](#), and each [Requires Edge](#) must connect to either a [Role](#) or [Permission](#).

A [Requires Edge](#) may be connected to another [Requires Edge](#) with a [Constraint Edge](#) in order to support the composition of complex conditions. The composition of these outgoing [Requires Edges](#) with any connecting [Constraint Edges](#) represent the *access requirements* of the [Access Control Handler](#).

A [Access Control Handler](#) has an associated *user instance* [Domain Iterator](#), which will only be populated when the current user satisfies the *access requirements* of the handler.

A [Access Control Handler](#) may have an outgoing [ECA Rule](#) named “login”. If the current user does not satisfy the *access requirements* of this [Access Control Handler](#), the [ECA Rule](#) is executed.

### Model Completion Rules

- A [Domain Iterator](#) within a [Session](#) protected by both an [Access Control Handler](#) and [Login Handler](#) of type `DOMAIN_OBJECT` or `USER`, but the handlers both use different [Domain Types](#) (for example, with *inheritance*), and which does not specify a [Select Edge](#), will use the same [Domain Source](#) as the [Domain Type](#) used as a [Parameter](#) to the [Access Control Handler](#).
- A [Domain Iterator](#) within a [Session](#) protected by both an [Access Control Handler](#) and [Login Handler](#) of type `DOMAIN_OBJECT` or `USER`, but the handlers both use the same [Domain Types](#), and which does not specify a [Select Edge](#), will use the same [Domain Source](#) as the [Domain Type](#) used as a [Parameter](#) to the [Access Control Handler](#).
- A [Domain Iterator](#) within a [Session](#) protected by both an [Access Control Handler](#) and [Login Handler](#) of type `DOMAIN_OBJECT` or `USER`, but the [Access Control Handler](#) requires a [Permission](#) without a [Role](#), and which does not specify a [Select Edge](#), will use the same [Domain Source](#) as the [Domain Type](#) used as a [Parameter](#) to the [Access Control Handler](#).
- A [Login Handler](#) is generated for a [Session](#) that contains an [Access Control Handler](#).
- A [Login Handler](#) is generated for a [Frame](#) contained within a [Session](#) that contains an [Access Control Handler](#).
- If an [Access Control Handler](#) specifies a ‘login’ [ECA Rule](#) that goes to a [Frame](#), the same [Frame](#) will be used as the login [Frame](#) for the generated [Login Handler](#).
- If an [Access Control Handler](#) specifies a ‘logout’ [ECA Rule](#) that navigates to a [Frame](#), the same [Frame](#) will be used as the logout [Frame](#) for the generated [Login Handler](#).
- If an [Access Control Handler](#) specifies a ‘success’ [ECA Rule](#) that navigates to a [Frame](#), the same [Frame](#) will be used as the successful login [Frame](#) for the generated [Login Handler](#).



- A **Frame** within a **Session** protected by an **Access Control Handler** will have an **Operation** named ‘permissions check’.
- A **Frame** protected by an **Access Control Handler** will have an **Operation** named ‘page permissions check’.
- A **Frame** within a **Session** protected by an **Access Control Handler** will **execute** the ‘permissions check’ **Operation** when the frame is **accessed**.
- A **Frame** protected by an **Access Control Handler** will **execute** the ‘page permissions check’ **Operation** when the frame is **accessed**.
- The ‘permissions check’ **Operation** defined by an **Access Control Handler** protecting a **Frame** within a **Session** will **navigate** to the ‘login’ **Frame** within the same **Session** if the check fails.
- The ‘permissions check’ **Operation** defined by an **Access Control Handler** protecting a **Frame** within a **Session** will **navigate** to the ‘login’ **Frame** within a *separate* login **Session** if the check fails.
- An **Access Control Handler** protecting a **Session** with a defined **Role-based access requirement** will provide the *default Role* as a **Parameter** to the created **Login Handler**.
- An **Access Control Handler** has a **Builtin Operation** named ‘check permissions’.
- The **Domain Iterator** representing the current instance of an **Access Control Handler** check, protecting a **Session**, will be provided as a **Parameter** to the generated **Login Handler**.
- The **Domain Iterator** representing the current instance of an **Access Control Handler** check, protecting a **Frame** within a **Session**, will be provided as a **Parameter** to the generated **Login Handler**.

## I.2 Accessible

An element that can register when it is accessed.

**Interface.**

**Direct subtypes** Email, Scope, Visible Thing

**Indirect subtypes** Button, Frame, Input Form, Input Text Field, Internet Application, Iterator List, Label, Map, Map Point, Session

**Events**

- **onAccess** : [0..1] Event

This **Event** is executed whenever it is accessed, which may occur many times within a single request.

**Model Completion Rules**

- **Accessible** elements have an **Event** named **onAccess**.

## I.3 Action

A possible consequence of an **ECA Rule**, representing a “specification of parameterised behaviour”.

**Interface.**

**Direct subtypes** Frame, Gate, Operation

**Indirect subtypes** Activity Operation, Builtin Operation

**Frame**

If a [Frame](#) is executed as an [Action](#), then the browser is redirected to the selected [Frame](#). Any incoming [Parameters](#) to the connecting [ECA Rule](#) are passed as [Query Parameters](#) to the [Frame](#).

**Operation**

If an [Operation](#) is executed as an [Action](#), then the given [Operation](#) is executed. Any incoming [Parameters](#) to the connecting [ECA Rule](#) are passed as [Activity Parameters](#) to the [Operation](#).

If an [Operation](#) executed by a [ECA Rule](#) *fails*, all subsequent [ECA Rules](#) in the *current execution set* are cancelled, and the *failure handler* for the [Scope](#) containing the source of the [ECA Rule](#) is executed, as described by [Scope](#).

**References**

- [rules](#) : [0..\*] [ECA Rule](#) (opposite: [target](#))  
A list of incoming [ECA Rules](#).

## I.4 Action Edge Source

A possible trigger for an [ECA Rule](#).

**Interface.**

**Direct supertypes** [Contains Wires](#)

**Direct subtypes** [Access Control Handler](#), [Builtin Operation](#), [Button](#), [Email](#), [Event](#), [Gate](#), [Login Handler](#), [Operation Call Node](#), [Scope](#)

**Indirect subtypes** [Activity Operation](#), [Frame](#), [Internet Application](#), [Session](#)

This abstract interface is provided to simplify the implementation of [ECA Rules](#), as EMF does not support type unions in metamodel definitions. It is also provided as a form of syntactic sugar for using [ECA Rules](#).

For example, a [Button](#) is an [Action Edge Source](#) to allow “onClick” [ECA Rules](#) to be attached directly to the [Button](#), rather than the contained *onClick* [Event](#).

**References**

- [listeners](#) : [0..\*] [ECA Rule](#) (opposite: [trigger](#))  
A list of outgoing [ECA Rules](#).

## I.5 Activity Node

A primitive element of functionality as contained within an [Activity Operation](#).

**Abstract type.**

**Direct supertypes** [Generated Element](#)

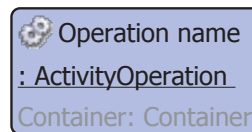
**Direct subtypes** Arithmetic, Cancel Node, Cast Node, Decision Node, External Value, Finish Node, Join Node, Operation Call Node, Set Node, Split Node, Start Node

The Activity Node model element is based on the UML *ActivityNode* abstract model element.

## I.6 Activity Operation

A user-defined Operation consisting of Activity Node connected by Execution Edges and Data Flow Edges.

**Direct supertypes** Builtin Operation, Contains Functions, Contains Operations, Contains Values, Contains Wires, Generates Elements



An Activity Operation may be defined through the composition of a number of Activity Nodes, connected by Execution Edges and Data Flow Edges. Execution Edges represent the execution flow of the particular operation, and Data Flow Edges represent the flow of data within the particular operation.

Each Decision Node and Operation Call Node defines a set of *failure edges* and *success edges*. For each node, all outgoing Execution Edges with a name beginning with the characters *n* or *f* represent the *failure edges*. All other Execution Edges from each node, including those with no name, represent the *success edges*.

A Activity Operation must contain exactly one Start Node, which must have at least one outgoing Execution Edge. When the Activity Operation is executed, the *execution flow* will begin from this Start Node, and will follow each outgoing Execution Edge sequentially.

### Containment References

- **data edges** : [0..\*] Data Flow Edge  
A list of Data Flow Edges necessary to define this Activity Operation.
- **execution edges** : [0..\*] Execution Edge  
A list of Execution Edges necessary to define this Activity Operation.
- **nodes** : [0..\*] Activity Node  
A list of Activity Nodes necessary to define this Activity Operation.
- **parameters** : [0..\*] Activity Parameter  
Allows access to Parameters provided to this Activity Operation as named Activity Parameters.
- **variables** : [0..\*] Temporary Variable  
A list of Temporary Variables necessary to define this Activity Operation.

### Model Completion Rules

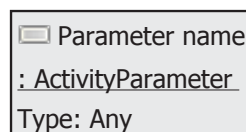
- A **Login Handler** of type **SECRET\_KEY** will create an **Activity Operation** ‘check key’ within the protected **Session**.
- An **Activity Operation** named ‘check key’, used to check the status of a **Login Handler** of type **SECRET\_KEY** against an incoming **Parameter** to the handler, will be completed with the necessary **Activity Nodes** to implement this functionality.
- An **Activity Operation** named ‘check instance’, used to check the status of a **Login Handler** of type **DOMAIN\_OBJECT** that a **Domain Iterator** is not **empty**, will be completed with the necessary **Activity Nodes** to implement this functionality.
- An **Activity Operation** named ‘check instance’, used to check the status of a **Login Handler** of type **USER** that a **Domain Iterator** is not **empty**, will be completed with the necessary **Activity Nodes** to implement this functionality.
- An **Activity Operation** named ‘do login’, used to actually implement the login operation of a **Login Handler** of type **SECRET\_KEY**, will be completed with the necessary **Activity Nodes** to implement this functionality.
- An **Activity Operation** named ‘do login’, used to actually implement the login operation of a **Login Handler** of type **DOMAIN\_KEY**, will be completed with the necessary **Activity Nodes** to implement this functionality.
- Every **Domain Attribute** provided as a **Parameter** to a **Login Handler** of type **DOMAIN\_OBJECT** will be available as a named **Activity Parameter** of the ‘do login’ **Activity Operation**.
- An **Activity Operation** named ‘do login’, used to actually implement the login operation of a **Login Handler** of type **USER**, will be completed with the necessary **Activity Nodes** to implement this functionality.
- Every **Domain Attribute** of the **Role** provided as a **Parameter** to a **Login Handler** of type **USER** will be available as a named **Activity Parameter** of the ‘do login’ **Activity Operation**.
- A **Login Handler** will define an **Operation** named ‘do logout’ within the protected **Scope**.
- An **Activity Operation** named ‘do logout’, used to actually implement the logout operation of a **Login Handler** of type **SECRET\_KEY**, will be completed with the necessary **Activity Nodes** to implement this functionality.
- An **Activity Operation** named ‘do logout’, used to actually implement the logout operation of a **Login Handler** of type **DOMAIN\_OBJECT**, will be completed with the necessary **Activity Nodes** to implement this functionality.
- An **Activity Operation** named ‘do logout’, used to actually implement the logout operation of a **Login Handler** of type **USER**, will be completed with the necessary **Activity Nodes** to implement this functionality.
- An **Activity Operation** named ‘do logout’ of a **Login Handler** of type **DOMAIN\_OBJECT** will reset all of the stored keys for each **Domain Attribute** used as a **Parameter** to the handler to an **empty Value**.
- An **Activity Operation** named ‘do logout’ of a **Login Handler** of type **USER** will reset all of the stored keys for each **Domain Attribute** of the **Role** used as a **Parameter** to the handler to an **empty Value**.
- A **Login Handler** of type **DOMAIN\_OBJECT** will create an **Activity Operation** ‘check instance’ within the protected **Session**.
- A **Login Handler** of type **USER** will create an **Activity Operation** ‘check instance’ within the protected **Session**.
- The **Activity Operation** ‘Set gate flag’ generated for a **Gate** protecting a **Frame**, with an outgoing **ECA Rule** named ‘first’ or ‘last’, will be completed with the contents necessary to set the **flag Value**.
- An empty **Activity Operation** named “update”, “refresh” or “init”, contained within a **Changeable**, will be completed with **Activity Nodes** to implement the particular operation.

- An empty **Activity Operation** named “set value XXX”, contained within a **Changeable**, used to set a **Value** other than **field value**, will be completed with **Activity Nodes** to implement the particular operation.
- The ‘last’ **Button** defined for navigating a **Domain Iterator**, as defined by a **Set Wire**, will contain an **Activity Operation** named ‘update target’.
- A **Frame** within a **Session** protected by an **Access Control Handler** will have an **Operation** named ‘permissions check’.
- A **Frame** protected by an **Access Control Handler** will have an **Operation** named ‘page permissions check’.
- A **Frame** within a **Session** protected by an **Access Control Handler** will **execute** the ‘permissions check’ **Operation** when the frame is **accessed**.
- A **Frame** protected by an **Access Control Handler** will **execute** the ‘page permissions check’ **Operation** when the frame is **accessed**.
- The ‘permissions check’ **Operation** defined by an **Access Control Handler** protecting a **Frame** within a **Session** will **navigate** to the ‘login’ **Frame** within the same **Session** if the check fails.
- The ‘permissions check’ **Operation** defined by an **Access Control Handler** protecting a **Frame** within a **Session** will **navigate** to the ‘login’ **Frame** within a *separate* login **Session** if the check fails.
- An **Activity Operation** named ‘permissions check’ within an **Access Control Handler** protecting a **Session**-contained **Frame**, used to check the status of the ‘check permissions’ **Operation** within the same handler, will be completed with the necessary **Activity Nodes** to implement this functionality.
- Two differently- **typed Input Text Fields** connected via a **Sync Wire** will define an **Operation** within the first field named ‘validate’.
- An **Activity Operation** named ‘validate’, used to check the validity of two type instances of two **Input Text Fields** connected via a **Sync Wire**, will be completed with the necessary **Activity Nodes** to implement this functionality.

## I.7 Activity Parameter

Allows access to **Parameters** provided to this **Activity Operation** or **Activity Predicate** as named **Activity Parameters**.

**Direct supertypes** **Data Flow Edges Source**, **Named Element**, **Parameter Edges Source**



An **Activity Parameter** defines a **name** and associated **type**. When an **Activity Operation** is being executed, the incoming **ECA Rule** can support incoming named **Parameters** in order to set these named operation parameters; parameters are passed by value, and not by reference. The values of these **Activity Parameters** do not persist across operation calls.

This model element may be removed from the IAML metamodel in the future<sup>2</sup>, as temporary variables may be replaceable with **Values**.

<sup>2</sup>Issue 279: *Replace TemporaryVariable element with a scoped Value.*

## References

- **type** : [0..1] EXSD Data Type  
The type of this Activity Parameter.

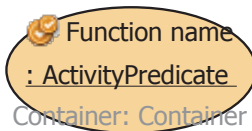
## Model Completion Rules

- Every Domain Attribute provided as a Parameter to a Login Handler of type DOMAIN\_OBJECT will be available as a named Activity Parameter of the ‘do login’ Activity Operation.
- Every Domain Attribute of the Role provided as a Parameter to a Login Handler of type USER will be available as a named Activity Parameter of the ‘do login’ Activity Operation.

## I.8 Activity Predicate

A user-defined Predicate consisting of Activity Node connected by Execution Edges and Data Flow Edges.

**Direct supertypes** Contains Functions, Contains Operations, Contains Values, Generates Elements, Predicate



An Activity Predicate may be defined through the composition of a number of Activity Nodes, connected by Execution Edges and Data Flow Edges. Execution Edges represent the *execution flow* of the particular condition, and Data Flow Edges represent the flow of data within the particular condition.

The definition of a Activity Predicate is identical to a Activity Operation, except on the behaviours of Finish Nodes and Cancel Nodes. If the execution flow of a Activity Predicate arrives at a Finish Node, the Predicate is taken to be *successful* and return true. Similarly, if the execution flow of a Activity Predicate arrives at a Cancel Node, the Predicate is taken to be *unsuccessful* and return false.

## Containment References

- **data edges** : [0..\*] Data Flow Edge  
A list of Data Flow Edges necessary to define this Activity Predicate.
- **execution edges** : [0..\*] Execution Edge  
A list of Execution Edges necessary to define this Activity Predicate.
- **nodes** : [0..\*] Activity Node  
A list of Activity Nodes necessary to define this Activity Predicate.
- **parameters** : [0..\*] Activity Parameter  
Allows access to Parameters provided to this Activity Predicate as named Activity Parameters.
- **variables** : [0..\*] Temporary Variable  
A list of Temporary Variables necessary to define this Activity Predicate.

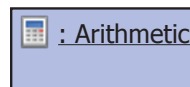
## Model Completion Rules

- An **Activity Predicate** named 'XXX is set', used to set a **named Value**, will be completed with the necessary **Activity Nodes** to implement this functionality.
- An **Activity Predicate** named 'can cast?', used to check casting for a **Changeable field value**, will be completed with the necessary **Activity Nodes** to implement this functionality.
- The 'check XXX' **Activity Predicate** for a **Gate**, with an outgoing **ECA Rule** named 'first' or 'last', will be completed with the contents necessary to evaluate the *flag Value*.
- A **Domain Iterator** will contain a generated **Activity Predicate** named 'not empty' - the inverse of the empty condition.
- The 'not empty' **Activity Predicate** for a **Domain Iterator**, will be completed with the contents necessary to check that the iterator is not empty.
- An **Activity Predicate** named 'update target', used to set the 'target' **Value** within the 'last' **Button** generated by a **Domain Iterator**, will be completed with the necessary **Activity Nodes** to update the **Value** with the current results **count** of that iterator.
- An **Activity Predicate** named 'permissions check' within an **Access Control Handler** protecting a **Frame**, used to check the status of the 'check permissions' **Operation** within the same handler, will be completed with the necessary **Activity Nodes** to implement this functionality.

## I.9 Arithmetic

An inline expression of arithmetic, operating upon all incoming **Data Flow Edge** values.

**Direct supertypes** **Activity Node**, **Data Flow Edge Destination**, **Data Flow Edges Source**



An **Arithmetic** must have at least one incoming **Data Flow Edge**, representing the source values. When a **Data Flow Edge** sourced from a **Arithmetic** is evaluated, the value provided represents the arithmetic combination of the source values according to the **operationType** of the **Arithmetic**.

Four **operationTypes** are defined for **Arithmetic**: AND, SUBTRACT, MULTIPLY and DIVIDE; each of these operations correspond to addition, subtraction, multiplication and division, respectively. For DIVIDE arithmetic, the divisor used in the arithmetic is the first incoming **Data Flow Edge** with the **name** "divisor". If the *divisor* used in a DIVIDE arithmetic is equal to zero, the value of the subsequent **Arithmetic** node will not be numeric ("not-a-number", or NaN).

This model element may be removed from the IAML metamodel in the future<sup>3</sup>, as these arithmetic operations can be replaced with **XQuery Functions**.

### Attributes

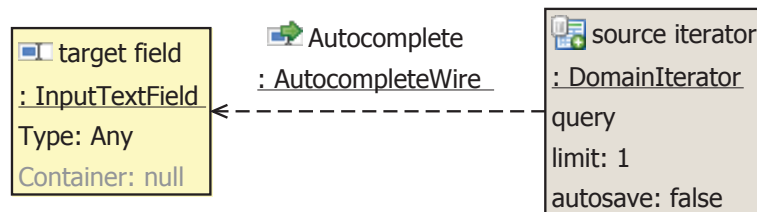
- **operationType**: **ArithmeticOperationTypes**  
The builtin arithmetic operation to perform.  
Possible values: ADD, SUBTRACT, MULTIPLY, DIVIDE

<sup>3</sup>Issue 220: Remove Arithmetic metamodel element.

## I.10 Autocomplete Wire

An **Autocomplete Wire** specifies that a target **Visible Thing** should support data entry through autocompletion from the source **Domain Iterator**, as specified by a matching **Domain Attribute**.

**Direct supertypes** **Wire**



An **Autocomplete Wire** is a unidirectional **Wire**, that connects a **Domain Iterator** to a **Visible Thing** (the *target element*). An **Autocomplete Wire** has an additional reference, **match**, which references a **Domain Attribute** from which to search by. The connected **Domain Type** for that iterator must have a **Domain Attribute** matching the name of the *target element*, as this will be the value used for a successful autocomplete interaction.

IAML does not specify whether the search should be performed on the client or the server. For small data sets, the database could be prefetched and cached by the client, in order to perform quick searches; for larger data sets, the database should remain on the server, and the search performed asynchronously.

### References

- **match** : [0..1] **Domain Attribute**

This **Autocomplete Wire** will match user input against the string representation of **instances** of this **Domain Attribute**.

### Model Completion Rules

- An **Autocomplete Wire**, connecting an **Domain Iterator** to a **Visible Thing**, will populate the containing **Frame** with an **Input Form** to contain autocompleted results.
- An **Autocomplete Wire**, connecting an **Domain Iterator** to a **Visible Thing**, will also create a **query Input Text Field** to accept user input, which will be used to search the **Domain Iterator**.
- The **query Input Text Field** created by an **Autocomplete Wire** will be connected as a **Parameter** to the connected **Domain Iterator**.
- An **Autocomplete Wire** connected to a **Domain Iterator** will automatically override the **query** of the **Iterator** based on what the wire **matches**.
- The **query Input Text Field** created by an **Autocomplete Wire** will **update** the field whenever it receives input.
- The **Input Form** results container generated by an **Autocomplete Wire** will be populated with a **results Iterator List** to display matched results in a list, using a **Set Wire**.
- Every **Label** within the **results Iterator List** of an **Autocomplete Wire** search will be provided an **ECA Rule** to **update** a **target element** – where the **name** of the **Domain Attribute Instance** matches the **name** of the target **Visible Thing** – when the **Label** is clicked.



- When a matching **Label** result is clicked from an **Autocomplete Wire** search, the *query Input Text Field* will be cleared to a blank **Value**.
- If the *query Input Text Field* for an **Autocomplete Wire** is empty, accessing the containing **Frame** will **hide** the *results Iterator List*.
- If the *query Input Text Field* for an **Autocomplete Wire** is not empty, accessing the containing **Frame** will **show** the *results Iterator List*.
- If the *query Input Text Field* for an **Autocomplete Wire** is changed and empty, the *results Iterator List* will be **hidden**.
- If the *query Input Text Field* for an **Autocomplete Wire** is changed and not empty, the *results Iterator List* will be **shown**.

## I.11 Boolean Property

A **Predicate** which uses the **container** of this predicate as an implicit **Parameter**.

**Abstract type.**

**Direct supertypes** **Predicate**

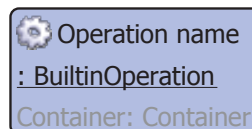
**Direct subtypes** **Builtin Property**

## I.12 Builtin Operation

A builtin reusable block of functionality with respect to its container.

**Direct supertypes** **Action Edge Source, Operation**

**Direct subtypes** **Activity Operation**



**hide, show**

The **hide** and **show** **Builtin Operations** are defined for all **Visible Things**. When executed, the **Operation** hides or shows its parent element respectively. The current visibility is stored within a **Value** contained within the field, and thus persists according to the containing **Scope**.

**init, update**

The **init** and **update** **Builtin Operations** are defined for all **Domain Attribute Instances** and **Visible Things**. The **Operation** accepts one **Activity Parameter**, *value*. When executed, the operation updates the *fieldValue* of the parent element to the provided *value*.

**send**

The **send Builtin Operation** is defined for all **Emails**. When executed, the **Operation** attempts to send the current **Email**. If the e-mail can successfully be delivered, the *onSuccess* event is triggered; if the e-mail cannot be successfully received, the *onFailure* event is triggered.

**next, previous**

The **next** and **previous Builtin Operations** are defined for all **Domain Iterators**. When executed, the **Operation** attempts to increment or decrement the current *instance pointer* of the **Domain Iterator**, and the current instance is reloaded from the **Domain Source**. If the resulting *instance pointer* is out-of-bounds for the **Domain Iterator**, an exception is thrown, and any defined *failure handlers* are triggered.

**skip, jump**

The **skip** and **jump Builtin Operations** are defined for all **Domain Iterators**. The **Operation** accepts one **Activity Parameter**, *value*. When executed, the **Operation** attempts to increment or set the current *instance pointer* of the **Domain Iterator** according to *value*, and the current instance is reloaded from the **Domain Source**. If the resulting *instance pointer* is out-of-bounds for the **Domain Iterator**, an exception is thrown, and any defined *failure handlers* are triggered.

**reset**

The **reset Builtin Operation** is defined for all **Domain Iterators**. When executed, the **Operation** attempts to reset the current *instance pointer* of the **Domain Iterator** to zero, and the current instance is reloaded from the **Domain Source**. If the resulting *instance pointer* is out-of-bounds for the **Domain Iterator**, an exception is thrown, and any defined *failure handlers* are triggered.

**save**

The **save Builtin Operation** is defined for all **Domain Iterators** and **Domain Attribute Instances**. When executed, the **Operation** attempts to save the current **Domain Iterator** or **Domain Attribute Instance** to the **Domain Source** from which the **Domain Iterator** is connected to. If the save is unsuccessful, an exception is thrown, and any defined *failure handlers* are triggered.

**new**

The **new Builtin Operation** is defined for all **Domain Iterators**. When executed, the **Operation** clears the value of each contained **Domain Attribute Instances** according to the semantics defined by **Domain Iterator**, and marks the current *instance pointer* as *new*.

**add permission, remove permission**

The **addPermission** and **removePermission Builtin Operations** are defined for all **Domain Iterators**. The **Operation** accepts one **Activity Parameter** of type **Permission**, *permission*. When executed, the **Operation** attempts to add or remove the given **Permission** to the current *user instance* as selected by the **Domain Iterator**. If the current **Domain Iterator** does not refer to a valid *user instance*, then an exception is thrown.

**add role, remove role**

The **addRole** and **removeRole Builtin Operations** are defined for all **Domain Iterators**. The **Operation** accepts one **Activity Parameter** of type **Role**, *role*. When executed, the **Operation** attempts to add or remove the given **Role** to the current *user instance* as selected by the **Domain Iterator**. If the current **Domain Iterator** does not refer to a valid *user instance*, then an exception is thrown.

When a **Role** is added to the current *user instance*, all **Permissions** provided by that **Role** through **Provides Edges** are also given to that *user instance*. If the *user instance* already possesses a given **Permission**, then there is no effect. Similarly, when a **Role** is removed from a *user instance*, all **Permissions** provided by that **Role** through **Provides Edges** are also removed from that *user instance*.

### check permissions

The **checkPermissions Builtin Operation** is defined for all **Access Control Handlers**. When executed, the **Operation** validates the current user according to the requirements of the **Access Control Handler**. If the user cannot be successfully validated, an exception is thrown.

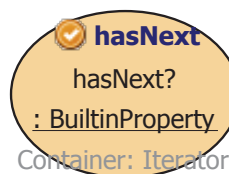
### Model Completion Rules

- **Domain Iterators** have a **Builtin Operation** named **previous**.
- **Domain Iterators** have a **Builtin Operation** named **next**.
- **Domain Iterators** have a **Builtin Operation** named **reset**.
- **Domain Iterators** have a **Builtin Operation** named **skip**.
- **Domain Iterators** have a **Builtin Operation** named **jump**.
- A **Visible Thing** has a **Builtin Operation** named **hide**.
- A **Visible Thing** has a **Builtin Operation** named **show**.
- An **Access Control Handler** has a **Builtin Operation** named 'check permissions'.
- A **Domain Iterator** contains an **Operation** named **save**.

## I.13 Builtin Property

A builtin **Boolean Property** with predefined behaviour, based on the **Builtin Property** type, and the type of its implicit container.

**Direct supertypes** **Boolean Property**



### empty, not empty

The **empty** and **notEmpty Builtin Property**s are defined for all **Changeable** elements; that is, elements which contain a **fieldValue**. A particular value is considered *empty* if and only if the instance cannot be cast to a **boolean** value, or if the casted **boolean** value is **false**.

The **empty Predicate** is a successful condition if and only if, with respect to the **Changeable** element that contains the **Predicate**, the **fieldValue** is either not set (according to the “fieldValue is set?” **Builtin Property** below), or can be considered *empty*. Conversely, the **notEmpty Predicate** is a successful condition if and only if the **empty Predicate** is not successful.

The **empty** element is also defined for all **Domain Iterator** elements, and this **Predicate** is considered successful if and only if there are no valid results for the current **Domain Iterator** query.

**fieldValue is set**

The "fieldValue is set?" [Builtin Property](#) is defined for all [Changeable](#) elements. This [Predicate](#) is a successful condition if and only if, with respect to the [Changeable](#) element that contains the [Predicate](#), the [fieldValue](#) is *not set* as described in [Value](#). This [Predicate](#) is therefore successful even if the current [fieldValue](#) can be considered *empty*.

**can cast**

The [canCast Builtin Property](#) is defined for all [Changeable](#) elements, and accepts one incoming value *value* as a [Activity Parameter](#). This [Predicate](#) is a successful condition if and only if the given *value* can be successfully cast to the type of the [fieldValue](#) of the [Changeable](#) that contains the [Predicate](#).

**can save**

The [canSave Builtin Property](#) is defined for all [Domain Instances](#) and [Domain Attribute Instances](#)<sup>4</sup>. This [Predicate](#) is a successful condition if and only if the current [Domain Instance](#) or [Domain Attribute Instance](#) can be saved successfully, according to the semantics of the “save” [Builtin Operation](#).

**has next, has previous**

The [hasNext](#) and [hasPrevious Builtin Property](#)s are defined for all [Domain Iterators](#). These [Predicates](#) are successful if and only if an attempt to increment or decrement the current *instance pointer* of the [Domain Iterator](#) – according to the semantics defined by the ‘next’ and ‘previous’ [Builtin Operations](#) – will not result in an out-of-bounds exception which would trigger any *failure handlers*.

**Attributes**

- [property](#) : BUILTIN\_PROPERTIES  
Selects the specific builtin functionality that this [Builtin Property](#) provides.  
Possible values: UNKNOWN, IS\_VISIBLE, CAN\_SAVE, CAN\_CAST, EMPTY, NOT\_EMPTY, IS\_SET, HAS\_NEXT, HAS\_PREVIOUS

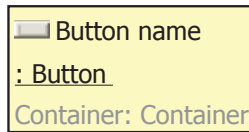
**Model Completion Rules**

- [Changeable](#) elements will contain a [Builtin Property](#) named ‘can cast?’.
- [Domain Iterators](#) have a [Builtin Property](#) named [has next](#).
- [Domain Iterators](#) have a [Builtin Property](#) named [has previous](#).
- [Domain Iterators](#) have a [Builtin Property](#) named [has next](#).
- [Visible Things](#) have an [Builtin Property](#) named [empty](#).
- [Visible Things](#) have an [Builtin Property](#) named [not empty](#).

**I.14 Button**

A clickable button.

**Direct supertypes** [Action Edge Source](#), [Visible Thing](#)



A [Button](#) represents a clickable button in the user interface. A [Button](#) may accept and lose focus. The [onInput Event](#) can never be triggered for a [Button](#). The [fieldValue](#) of the [Button](#) is used as the caption for the rendered button, and may be changed at runtime.

### Inherited Events

[onAccess](#) (from [Accessible](#)); [onChange](#) (from [Changeable](#)); [onClick](#) (from [Visible Thing](#)); [onInput](#) (from [Visible Thing](#))

### Model Completion Rules

- A [login Frame](#) for a [Login Handler](#) will contain an [Button](#) named 'login'.
- The 'login' [Button](#) in the [login Frame](#) for a [Login Handler](#) will execute the 'do login' [Operation](#) within the [Login Handler](#) when the button is [clicked](#).
- A [login Frame](#) for a [Login Handler](#) of type `SECRET_KEY` will contain an [Input Text Field](#) named 'login key'.

## I.15 Can Be Synced

An element that can be synchronised through a [Sync Wire](#) or [Set Wire](#).

### Interface.

**Direct supertypes** [Contains Functions](#), [Contains Operations](#), [Contains Wires](#), [Generated Element](#), [Generates Elements](#), [Named Element](#), [Wire Destination](#), [Wire Source](#)

**Direct subtypes** [Domain Feature Instance](#), [Domain Iterator](#), [Email](#), [Scope](#), [Visible Thing](#)

**Indirect subtypes** [Button](#), [Domain Attribute Instance](#), [Frame](#), [Input Form](#), [Input Text Field](#), [Internet Application](#), [Iterator List](#), [Label](#), [Map](#), [Map Point](#), [Session](#)

This abstract interface is provided to simplify the model completion implementation of [Set Wires](#) and [Sync Wires](#) within EMF, by moving common supertypes (such as [Contains Operations](#) and [Wire Source](#) into a single abstract supertype.

### Model Completion Rules

- For elements connected by a [Sync Wire](#), the [onAccess Event](#) executing the 'init' [Operation](#) will only [execute](#) if the source [field Value](#) can be cast [successfully](#).
- For elements connected by a [Sync Wire](#), the [onAccess Event](#) will only call the 'validate' [Operation](#) if the relevant [field value](#) is [set](#).

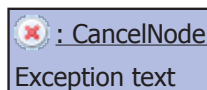
<sup>4</sup>Issue 230: Define 'can save?' for Domain Attribute Instances.

- A **Sync Wire** connecting two objects will create sub-**Sync Wires** between every contained object within these parent objects which have matching **names**.
- A **Sync Wire** connecting an **Domain Iterator** to a **Can Be Synced** object will create sub-**Sync Wires** between every contained object within the contained **Domain Instance** to every contained **Can Be Synced** object which have matching **names**.
- When a **Can Be Synced** element is connected to an **Can Be Synced** element by a **Sync Wire**, the **value** of the source element will be provided as a **Parameter** to the updating **ECA Rule**.
- When an **Input Text Field** is connected to a **Can Be Synced** element by an **instant Sync Wire**, the **value** of the source element will be provided as a **Parameter** to the updating **ECA Rule**.
- A **Set Wire** connecting two objects will create sub-**Set Wires** between every contained object within these parent objects which have matching **names**.
- A **Set Wire** connecting an **Domain Iterator** to a **Can Be Synced** object will create sub-**Set Wires** between every contained object within the contained **Domain Instance** to every contained **Can Be Synced** object which have matching **names**.
- A **Set Wire** connecting an **Can Be Synced** object to a **Domain Iterator** will create sub-**Set Wires** between every contained **Can Be Synced** object to every contained object within the contained **Domain Instance** which have matching **names**.
- When a **Can Be Synced** element is connected to an **Can Be Synced** element by a **Set Wire**, the **value** of the source element will be provided as a **Parameter** to the updating **ECA Rule**.
- When a **Changeable** element is connected to an **Can Be Synced** element by a **Set Wire**, the **value** of the source element will be provided as a **Parameter** to the updating **ECA Rule**.
- When an **Input Text Field** is connected to a **Can Be Synced** element by an **instant Set Wire**, the **value** of the source element will be provided as a **Parameter** to the updating **ECA Rule**.

## I.16 Cancel Node

Specifies that the current **Activity Operation** did not execute successfully, or the **Activity Predicate** must return **false**.

**Direct supertypes** **Activity Node**, **Execution Edge Destination**



An **Activity Operation** may contain any number of **Cancel Nodes**. When the *execution flow* arrives at a **Cancel Node**, the execution of the operation will finish, however the execution will be marked as a failure. The *failure handler* of the **ECA Rule** that executed the **Operation** will be called to handle this failure.

The **Cancel Node** element is based on the UML *FinalNode* (flow final) model element.

### Attributes

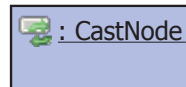
- **exceptionText** : string

Within an **Activity Operation**, this message will be provided to the developer as a failure reason if this exception is not caught by a *failure handler*.

## I.17 Cast Node

Allows one data [type](#) to be cast to another data [type](#). Has an outgoing “check” [Data Flow Edge](#) which can be used to check for invalid casts, otherwise a failing conversion is silent.

**Direct supertypes** [Activity Node](#), [Data Flow Edge Destination](#), [Data Flow Edges Source](#)



A [Activity Operation](#) may contain any number of [Cast Node](#) data flow nodes. A [Cast Node](#) must have exactly one incoming [Data Flow Edge](#), representing the *source* value and type. A [Cast Node](#) must also have exactly one outgoing [Data Flow Edge](#) to another *data flow node*, representing the *target* value and type.

A [Cast Node](#) may have additional outgoing [Data Flow Edges](#). If an outgoing [Data Flow Edge](#) is connected to a [Decision Node](#) named "can cast?", then the value of this [Data Flow Edge](#) represents whether the incoming *source* value to the [Cast Node](#) can be successfully cast to the *target* type. All other outgoing [Data Flow Edges](#) represent the forcibly cast value of the incoming *source* value to the *target* type; a forcibly cast value will not display an error message if the cast was not possible.

A “successful” cast is one where the source can be converted successfully *with no loss of information*. So, for example, the String “123” can be successfully cast to an Integer, but "123.4" cannot. However, this failure can be discarded if loss of information *is* acceptable; for example, "123.4“ forcibly cast to an Integer will return "123".

## I.18 Changeable

An element that has a [value](#), and can register when it can [change](#). Since it has a [value](#), it can be used as a [parameter](#).

**Interface.**

**Direct supertypes** [Contains Functions](#), [Generated Element](#), [Generates Elements](#), [Parameter Edges Source](#), [Wireable](#)

**Direct subtypes** [Domain Feature Instance](#), [Domain Iterator](#), [Visible Thing](#)

**Indirect subtypes** [Button](#), [Domain Attribute Instance](#), [Input Form](#), [Input Text Field](#), [Iterator List](#), [Label](#), [Map](#), [Map Point](#)

**Attributes**

- `defaultValue` : `string`

The default value of the [field value](#). This value should be the same as the [default value](#) of the contained [field value](#), and is provided to simplify end-user implementation.

**Containment References**

- **field value** : [0..1] **Value**

The current value of this **Changeable** element. This is always a valid **type** instance.

**References**

- **type** : [0..1] **EDataType** (from Ecore)

The type of this **Changeable** element and its contained **field value**.

**Events**

- **onChange** : [0..1] **Event**

This **Event** is triggered immediately after the **field value** of this element is changed.

**Model Completion Rules**

- A **Changeable** contains an untyped **Value** named **field value**.
- If set, the **default Value** is used as the default value of the created **field value Value**.
- **Changeable** elements will contain a **Builtin Property** named 'can cast?'.
- An **Activity Predicate** named 'can cast?', used to check casting for a **Changeable field value**, will be completed with the necessary **Activity Nodes** to implement this functionality.
- When a **Changeable** is connected to a **Value** by a **Set Wire**, the source element will **call** the 'set value XXX' **Operation** on the target when the source **changes**.
- When a **Changeable** element is connected to an **Can Be Synced** element by a **Set Wire**, the **value** of the source element will be provided as a **Parameter** to the updating **ECA Rule**.
- When a **Changeable** element is connected to a **Value** by a **Set Wire**, the **value** of the source element will be provided as a **Parameter** to the updating **ECA Rule**.
- **Changeable** elements have an **Event** named **onChange**.
- An empty **Activity Operation** named "update", "refresh" or "init", contained within a **Changeable**, will be completed with **Activity Nodes** to implement the particular operation.
- An empty **Activity Operation** named "set value XXX", contained within a **Changeable**, used to set a **Value** other than **field value**, will be completed with **Activity Nodes** to implement the particular operation.
- An **Activity Operation** named 'permissions check' within an **Access Control Handler** protecting a **Session-contained Frame**, used to check the status of the 'check permissions' **Operation** within the same handler, will be completed with the necessary **Activity Nodes** to implement this functionality.
- An **Activity Predicate** named 'permissions check' within an **Access Control Handler** protecting a **Frame**, used to check the status of the 'check permissions' **Operation** within the same handler, will be completed with the necessary **Activity Nodes** to implement this functionality.

**I.19 Complex Term**

Associates a **Function** with a set of **Parameter Values** as provided by its incoming **Parameters**.

**Abstract type.**

**Direct supertypes** **Parameter Edge Destination**, **Parameter Edges Source**, **Parameter Value**

**Direct subtypes** **Simple Condition**



**References**

- `conditioned` : Condition Edge Destination (opposite: `conditions`)

A `Complex Term` may be used in the definition of a `Condition`, if the referenced `Function` is a `Predicate`.

**Functions**

- `function` : Function (opposite: `conditioned`)

The function on which these `Parameter Values` will apply to when evaluated.

## I.20 Condition Edge Destination

An element which may be restricted by incoming `Conditions`.

**Interface.**

**Direct subtypes** Decision Node, ECA Rule, Gate, Wire

**Indirect subtypes** Autocomplete Wire, Detail Wire, Set Wire, Sync Wire

This abstract interface is provided to simplify the implementation of `Conditions`, as EMF does not support type unions in metamodel definitions.

**References**

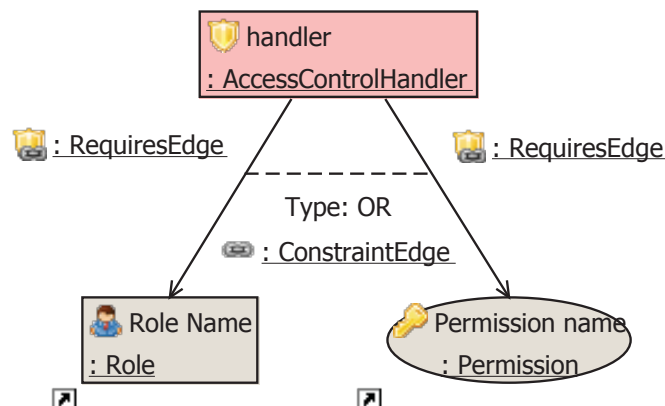
- `conditions` : [0..\*] Complex Term (opposite: `conditioned`)

A list of incoming `Conditions` that may restrict the behaviour of this element.

## I.21 Constraint Edge

Allows for the definition of complex requirements of `Requires Edges` through logical operators.

**Direct supertypes** Generated Element



**Attributes**

- **type** : [ConstraintTypes](#)  
The type of logical operator used in this [Constraint Edge](#).  
Possible values: AND, OR, XOR

**References**

- **from** : [Constraint Edges Source](#) (opposite: [out constraint edges](#))  
The first element in this [Constraint Edge](#) relationship.
- **to** : [Constraint Edge Destination](#) (opposite: [in constraint edges](#))  
The second element in this [Constraint Edge](#) relationship.

## I.22 Constraint Edge Destination

An element which may be used as the target of a [Constraint Edge](#).

**Interface.**

**Direct subtypes** [Requires Edge](#)

This abstract interface is provided to simplify the implementation of [Constraint Edges](#), as EMF does not support type unions in metamodel definitions.

**References**

- **in constraint edges** : [0..\*] [Constraint Edge](#) (opposite: [to](#))  
A list of outgoing [Constraint Edges](#).

## I.23 Constraint Edges Source

An element which may be used as the source of a [Constraint Edge](#).

**Interface.**

**Direct subtypes** [Requires Edge](#)

This abstract interface is provided to simplify the implementation of [Constraint Edges](#), as EMF does not support type unions in metamodel definitions.

**References**

- **out constraint edges** : [0..\*] [Constraint Edge](#) (opposite: [from](#))  
A list of incoming [Constraint Edges](#).

## I.24 Contains Functions

An element which can directly contain arbitrary **Functions**.

**Interface.**

**Direct subtypes** Access Control Handler, Activity Operation, Activity Predicate, Can Be Synced, Changeable, Domain Iterator, Email, Internet Application, Login Handler, Scope, Session, Visible Thing

**Indirect subtypes** Button, Domain Attribute Instance, Domain Feature Instance, Frame, Input Form, Input Text Field, Iterator List, Label, Map, Map Point

This abstract interface is provided to simplify the implementation of **Functions** within EMF, by moving common containment relationships into a single abstract supertype.

**Functions**

- **functions** : [0..\*] Function  
A list of contained **Functions**.

## I.25 Contains Operations

An element which can directly contain arbitrary **Operations**.

**Interface.**

**Direct subtypes** Access Control Handler, Activity Operation, Activity Predicate, Can Be Synced, Domain Feature Instance, Domain Instance, Domain Iterator, Email, Internet Application, Login Handler, Scope, Session, Visible Thing

**Indirect subtypes** Button, Domain Attribute Instance, Frame, Input Form, Input Text Field, Iterator List, Label, Map, Map Point

This abstract interface is provided to simplify the implementation of **Operations** within EMF, by moving common containment relationships into a single abstract supertype.

**Operations**

- **operations** : [0..\*] Operation  
A list of contained **Operations**.

## I.26 Contains Values

An element which can directly contain arbitrary **Values**, which can be used to programmatically set element attributes at runtime.

**Interface.**

**Direct subtypes** Access Control Handler, Activity Operation, Activity Predicate, Domain Iterator, Email, Internet Application, Login Handler, Scope, Visible Thing

**Indirect subtypes** Button, Frame, Input Form, Input Text Field, Iterator List, Label, Map, Map Point, Session

This abstract interface is provided to simplify the implementation of [Values](#) within EMF, by moving common containment relationships into a single abstract supertype.

#### Containment References

- [values](#) : [0..\*] [Value](#)  
A list of contained [Values](#).

## I.27 Contains Wires

A generic container of [Wires](#), [ECA Rules](#) and other edges.

#### Interface.

**Direct subtypes** Access Control Handler, Action Edge Source, Activity Operation, Can Be Synced, Domain Feature, Domain Instance, Domain Iterator, Domain Source, Domain Type, Email, Internet Application, Login Handler, Operation, Query Parameter, Requires Edge, Scope, Session, Visible Thing, Wire, Wire Source

**Indirect subtypes** Activity Predicate, Autocomplete Wire, Boolean Property, Builtin Operation, Builtin Property, Button, Changeable, Detail Wire, Domain Attribute, Domain Attribute Instance, Domain Feature Instance, Event, Frame, Function, Gate, Input Form, Input Text Field, Iterator List, Label, Map, Map Point, Operation Call Node, Predicate, Role, Set Wire, Sync Wire, Value, Wireable, XQuery Function, XQuery Predicate

This abstract interface is provided to simplify the implementation of the IAML proof-of-concept editor within EMF as GMF, by moving common containment relationships into a single abstract supertype. In particular:

1. Within EMF, all instances of model edges (such as [ECA Rules](#) and [Wires](#)) should be contained within another model element to prevent the edges from polluting the top-level `EResource` of the model instance (known as *phantom nodes* within EMF: see GMF bug 284135).
2. Within GMF, it is much simpler to support the creation of edge elements if the edge is contained within the element representing the edge source, rather than an arbitrary container elsewhere in the metamodel (GMF bug 249717).

#### Containment References

- [constraint edges](#) : [0..\*] [Constraint Edge](#)  
A list of contained [Constraint Edges](#).
- [eca rules](#) : [0..\*] [ECA Rule](#)  
A list of contained [ECA Rules](#).
- [extends edges](#) : [0..\*] [Extends Edge](#)  
A list of contained [Execution Edges](#).
- [parameter edges](#) : [0..\*] [Parameter](#)

A list of contained [Parameters](#).

- [provides edges : \[0..\\*\] Provides Edge](#)  
A list of contained [Provides Edges](#).
- [requires edges : \[0..\\*\] Requires Edge](#)  
A list of contained [Requires Edges](#).
- [schema edges : \[0..\\*\] Schema Edge](#)  
A list of contained [Schema Edges](#).
- [select edges : \[0..\\*\] Select Edge](#)  
A list of contained [Select Edges](#).
- [wires : \[0..\\*\] Wire](#)  
A list of contained [Wires](#).

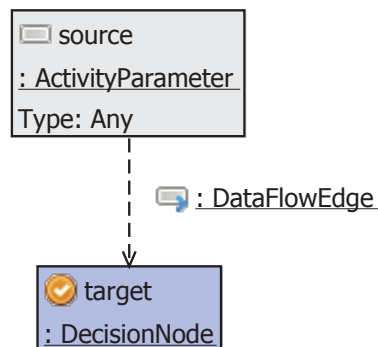
### Functions

- [condition edges : \[0..\\*\] Simple Condition](#)  
A list of contained [Simple Conditions](#).

## I.28 Data Flow Edge

Represents the flow of data from a data [source](#) to a data [destination](#).

**Direct supertypes** [Generated Element](#)



### References

- [from : Data Flow Edges Source](#) (opposite: out flows)  
The source of this [Data Flow Edge](#).
- [to : Data Flow Edge Destination](#) (opposite: in flows)  
The target of this [Data Flow Edge](#).

## I.29 Data Flow Edge Destination

A possible destination for a [Data Flow Edge](#).

#### Interface.

**Direct subtypes** [Arithmetic](#), [Cast Node](#), [Decision Node](#), [External Value](#), [Set Node](#), [Temporary Variable](#)

This abstract interface is provided to simplify the implementation of [Data Flow Edges](#), as EMF does not support type unions in metamodel definitions.

#### References

- [in flows](#) : [0..\*] [Data Flow Edge](#) (opposite: [to](#))  
A list of incoming [Data Flow Edges](#).

## I.30 Data Flow Edges Source

A possible source for a [Data Flow Edge](#).

#### Interface.

**Direct subtypes** [Activity Parameter](#), [Arithmetic](#), [Cast Node](#), [External Value](#), [Set Node](#), [Temporary Variable](#)

This abstract interface is provided to simplify the implementation of [Data Flow Edges](#), as EMF does not support type unions in metamodel definitions.

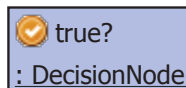
#### References

- [out flows](#) : [0..\*] [Data Flow Edge](#) (opposite: [from](#))  
A list of outgoing [Data Flow Edges](#).

## I.31 Decision Node

Based on its [name](#), considers the incoming [Conditions](#) or [Data Flow Edges](#), and follows passing or failing [Execution Edges](#) as appropriate.

**Direct supertypes** [Activity Node](#), [Condition Edge Destination](#), [Data Flow Edge Destination](#), [Execution Edge Destination](#), [Execution Edges Source](#), [Named Element](#)



A [Decision Node](#) can be used to control the flow of execution within a particular operation, and accepts any number of incoming [Data Flow Edges](#) and [Conditions](#), depending on the type of [Decision Node](#). A [Decision Node](#) must have at least one *failure edge* and one *success edge*.

An [Activity Operation](#) may contain any number of [Decision Nodes](#). When the *execution flow* arrives at a [Decision Node](#), all incoming edges are passed to the particular node, and the decision is evaluated as either true or false. If the [Decision Node](#) is considered to be false, the execution flow will follow all outgoing *failure edges* sequentially. If the [Decision Node](#) is considered to be true, the execution flow will follow all outgoing *success edges* sequentially.

The [Decision Node](#) element is based on the UML *DecisionNode* model element.

**true?**

A **Decision Node** named "true?" accepts any number of incoming **Data Flow Edges**. This node is true if and only if all incoming values can be cast to the `boolean` type, and are all considered true through logical conjunction.

**equal?**

A **Decision Node** named "equal?" accepts any number of incoming **Data Flow Edges**. This node is true if and only if all incoming values can be cast to the `string` type, and all string-cast representations can be considered equal in terms of the *fn:codepoint-equal* XQuery Function.

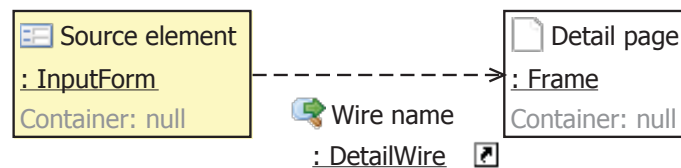
**can cast?**

A **Decision Node** named "can cast?" accepts one incoming **Data Flow Edge** (*source*), and has one outgoing **Data Flow Edge** (*target*). This node is true if and only if the current instance from the *source* edge can be successfully casted to the data type specified by the *target* edge, as discussed in **Cast Node**.

## I.32 Detail Wire

A **Detail Wire** specifies that an **Domain Instance** represented within a **Visible Thing** should be independently viewable on a separate **Frame**.

**Direct supertypes** **Wire**



A **Detail Wire** is a unidirectional **Wire**, that connects an **Input Form** or **Iterator List** (the *source view*) to a **Frame** (the *target view*). When the intended model is completed through *model completion*, a **Detail Wire** will complete functionality to allow a user to “zoom in” to a particular **Domain Type** instance.

The *source view* element must be connected via a **Set Wire** or a **Sync Wire** from a **Domain Iterator**. The **Domain Type** connected to this **Domain Iterator** must have a **primaryKey** defined, which will be passed as a **Query Parameter** to the *target view* **Frame**.

### Model Completion Rules

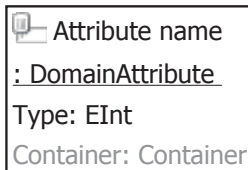
- When a **Detail Wire** is connected from a **Frame** that contains a **Domain Iterator** to a *target* **Frame**, a **Query Parameter** will be created in the *target* **Frame** to provide the primary key value.
- When a **Detail Wire** is connected from a **Frame** that contains a **Domain Iterator** to a *target* **Frame**, a new *selected* **Domain Iterator** will be used to select content based on primary key.
- If a **Visible Thing** to browse the *selected* **Domain Iterator** from a **Detail Wire** does not exist in the *target* **Frame**, a *view instance* **Input Form** will be created.
- A *view instance* **Input Form** generated by a **Detail Wire** will be connected with a **Set Wire** from the *selected* **Domain Iterator**.

- A *link Button* will be created in the container of the source *Domain Iterator* for a *Detail Wire*, in order to browse to the *target Frame*.
- The *link Button* generated for a *Detail Wire* will be connected with an *ECA Rule* to navigate to the *target Frame* when *clicked*.
- The navigation *rule* from the *link Button* generated for a *Detail Wire*, not contained by an *Iterator List*, will be supplied the current *value* within the source *Domain Instance*, representing the *primary key* of the source *Domain Type*.
- The navigation *rule* from the *link Button* generated for a *Detail Wire*, as contained by an *Iterator List*, will be supplied the current *value* within the source *Domain Instance*, representing the *primary key* of the source *Domain Type*.
- The *selected Domain Iterator* used to select one instance of a *Detail Wire* source is given the same *Domain Source* as the source *Domain Iterator*.

### I.33 Domain Attribute

A single domain-specific *typed* attribute of a particular *Domain Type*.

**Direct supertypes** *Domain Feature*



This model element is derived from the EMF *EAttribute* element.

#### Model Completion Rules

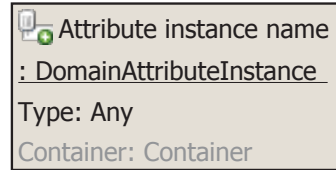
- A *login Frame* for a *Login Handler* of type *DOMAIN\_OBJECT* will contain an *Input Text Field* for every *Domain Attribute* defined as a *Parameter* to that *Login Handler*.
- A *login Frame* for a *Login Handler* of type *USER* will contain an *Input Text Field* for every non-primary *key Domain Attribute* within the *Role* defined as a *Parameter* to that *Login Handler*.
- A *Domain Type* which does not contain a primary *key Domain Attribute* will contain one named 'generated primary key'.
- A non- *primary key Domain Attribute* in a *Domain Type*, represented as a *supertype* of another *Domain Type*, will be copied into the subtype schema and marked as an *subattribute*.
- A *primary key Domain Attribute* in a *Domain Type*, represented as a *supertype* of another *Domain Type*, will be copied into a new 'ID' *Domain Attribute* and marked as a *foreign key*.
- An *Domain Instance* will be populated with *Domain Attribute Instances* for every *Domain Attribute* in the defining *Domain Type* of the instance's containing *Domain Iterator*.

### I.34 Domain Attribute Instance



A single instance of a [Domain Attribute](#) for a particular [Domain Type](#) instance.

**Direct supertypes** [Domain Feature Instance](#)



Each [Domain Attribute Instance](#) contains a [Predicate](#) `canSave`<sup>5</sup> as per the semantics defined by [Builtin Property](#); an [Operation](#) `save` as per the semantics defined by [Builtin Operation](#); and an [Event](#) `onChange` as per the semantics described by [Changeable](#).

### Inherited Events

`onChange` (from [Changeable](#))

### Model Completion Rules

- All [Domain Attribute Instances](#) will contain an [Operation](#) named “update”.
- An [Domain Instance](#) will be populated with [Domain Attribute Instances](#) for every [Domain Attribute](#) in the defining [Domain Type](#) of the instance’s containing [Domain Iterator](#).
- A [Domain Attribute Instance](#) created within a [Domain Instance](#) referencing a particular [Domain Attribute](#) will be marked as an extension of that attribute through an [Extends Edge](#).

## I.35 Domain Feature

A particular feature of a [Domain Type](#), which may be an [attribute](#) or a [reference](#).

**Abstract type.**

**Direct supertypes** [Contains Wires](#), [Extends Edge Destination](#), [Extends Edges Source](#), [Generated Element](#), [Generates Elements](#), [Parameter Edges Source](#)

**Direct subtypes** [Domain Attribute](#)

This model element is derived from the EMF `EStructuralFeature` element, and may be an instance of a [Domain Attribute](#) or as a [Domain Reference](#)<sup>6</sup>.

### Attributes

- `primaryKey` : `boolean`

Any number of [Domain Features](#) may be specified as the *primary key* for the containing [Domain Type](#).

<sup>5</sup>Issue 230: Define ‘can save?’ for [Domain Attribute Instances](#).

<sup>6</sup>Issue 228: Permit navigation through foreign keys of instances of [Domain Types](#).

## I.36 Domain Feature Instance

A single instance of a [Domain Feature](#), contained within a single [Domain Instance](#).

**Abstract type.**

**Direct supertypes** [Can Be Synced](#), [Changeable](#), [Contains Operations](#), [Extends Edge Destination](#), [Extends Edges Source](#)

**Direct subtypes** [Domain Attribute Instance](#)

### References

- **instance** : [0..1] [Domain Instance](#) (opposite: [feature instances](#))  
The containing [Domain Instance](#) of this feature instance.

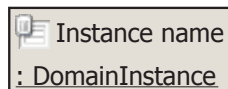
### Inherited Events

[onChange](#) (from [Changeable](#))

## I.37 Domain Instance

A single instance of a [Domain Type](#), as contained within a [Domain Iterator](#).

**Direct supertypes** [Contains Operations](#), [Contains Wires](#), [Generated Element](#), [Generates Elements](#), [Named Element](#), [Wireable](#)



### Containment References

- **feature instances** : [0..\*] [Domain Feature Instance](#) (opposite: [instance](#))  
All [Domain Features](#) specified by the instance [Domain Type](#) will be accessible at runtime via [Domain Feature Instances](#).

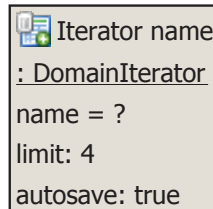
### Model Completion Rules

- An empty [Input Form](#) connected to a [Domain Iterator](#) with a [Sync Wire](#) will create an [Input Text Field](#) in that form for every [Domain Attribute Instance](#) that is not a [primary key](#) or [foreign key](#).
- An empty [Input Form](#) connected to a [Domain Iterator](#) with a [Set Wire](#) will create an [Input Text Field](#) in that form for every [Domain Attribute Instance](#) that is not a [primary key](#) or [generated](#).
- All [Domain Iterators](#) will be populated with a single instance of a [Domain Instance](#), representing the [current instance](#) of that iterator.
- An [Domain Instance](#) will be populated with [Domain Attribute Instances](#) for every [Domain Attribute](#) in the defining [Domain Type](#) of the instance's containing [Domain Iterator](#).

## I.38 Domain Iterator

Provides an iterable interface over instances of a particular [Domain Type](#), as selected from a particular [Domain Source](#).

**Direct supertypes** [Can Be Synced](#), [Changeable](#), [Contains Functions](#), [Contains Operations](#), [Contains Values](#), [Contains Wires](#), [Generates Elements](#), [Named Element](#), [Parameter Edge Destination](#), [Provides Edge Destination](#), [Provides Edges Source](#), [Wireable](#)



[Domain Iterators](#) allow for the access, and modification, of sources of data according to a particular schema. Access is defined by evaluating a [query](#) against named parameters for the query provided through [Parameters](#), with respect to a limit of results through the [limit](#) attribute.

A [Domain Iterator](#) is composed of a set of [Domain Attribute Instances](#), and this set is derived from the set of [Domain Attributes](#) defined by the [classifier Domain Type](#) of that [Domain Iterator](#).

### query

The [query](#) attribute of the [Domain Iterator](#) specifies the parameterised query that will be used to select [Domain Instances](#) from the connected [Domain Source](#). A [Domain Iterator](#) does not permit the execution of raw SQL commands, in order to improve the security of modelled web applications. Parameters to this [query](#) may be specified through [Parameters](#), and referenced within the [query](#) as *named parameters* (for example, a query of "id = 1" will use the incoming [Parameter](#) named "id"). A variety of database-independent query functions are provided:

Function	Returns
<code>matches(a, b)</code>	Performs a case-insensitive text search of a against the query b. Every word in b is matched, usually using a SQL keyword or function LIKE, against a.
<code>now()</code>	Returns the current date and time.

If the [query](#) is set to the value "new", then this [Domain Iterator](#) will instead create new instances of the given [Domain Type](#), which will be saved within the connected [Domain Source](#) if necessary.

### Containing Scope

Each [Domain Iterator](#) has an associated *containing scope* that represents the scope that a particular instance can be accessed:

- If the iterator is stored within a [Scope](#), then the iterator instance is unique to a given [Scope](#) instance. For example, a [Domain Iterator](#) contained within a [Session](#) can have one instance of that [Domain Iterator](#) per [Session](#) instance, and instances of that [Domain Iterator](#) cannot be accessed by other instances of the same [Session](#) scope.
- If the iterator is not stored within a [Scope](#), then the iterator instance is available globally according to the root [Internet Application](#). That is, only one instance of the [Domain Iterator](#) can ever exist.

## Failure Handlers

An outgoing [ECA Rule](#) from a particular [Domain Iterator](#) with the [name](#) "fail" is defined as a *failure handler* for the given iterator. If a [Domain Iterator](#) does not have such a handler, then the failure handler semantics of the containing [Scope](#) are used instead throughout the scope containment hierarchy.

## Attributes

**autosave** If the [autosave](#) attribute of this [Domain Iterator](#) is true, it will ensure all contained [Domain Attribute Instances](#) will automatically save the current [Domain Instance](#) (as pointed to by the cursor) whenever the attribute instance value is changed. The default value of this attribute is false.

If the [query](#) of this [Domain Iterator](#) is set to "new", and the [autosave](#) attribute is set to true, then a new [Domain Instance](#) will be created in the [Domain Source](#) immediately. If [autosave](#) is set to false, then a new [Domain Instance](#) will only be created in the [Domain Source](#) once the [save Operation](#) is called.

**new** The [new Operation](#) of a [Domain Iterator](#) forces a new result [Domain Instance](#) to be created, regardless of the current state of the contained [Domain Instance](#). If the [Domain Instance](#) has not been saved, then the changes will be lost.

**limit** By default, a [Domain Iterator](#) will select at most one matching result to the given [query](#). If the [limit](#) of a [Domain Iterator](#) is specified to a non-negative positive integer, then a [Domain Iterator](#) will select at most that many results. If the [limit](#) of a [Domain Iterator](#) is specified to zero, then the [Domain Iterator](#) will select all possible results.

**orderBy** The [orderBy](#) reference of a [Domain Iterator](#) describes the order in which multiple instances will be returned with respect to a single [Domain Attribute](#). If this reference is not set, then instances may be returned in any order. The direction of this order is specified by the [orderAscending](#) attribute of the [Domain Iterator](#).

## Operations

**reload** The [reload Operation](#) of a [Domain Iterator](#) reloads the current [Domain Instance](#) from the specified [Domain Source](#). When the [reload](#) operation is called upon a given [Domain Iterator](#) runtime instance, all current [Domain Attribute Instance](#) values are cleared, and reloaded from the specified [Domain Source](#). If the [Domain Source](#) can no longer provide the requested instance information, an error will occur.

When a [Domain Iterator](#) is accessed for the first time, or the current *instance pointer* is changed (for example, through calling the [next](#), [previous](#) or [reset Builtin Operations](#)), the [reload](#) operation is also executed.

**save** The [save Operation](#) of a [Domain Iterator](#) forces the current [Domain Instance](#) to be saved to the specified [Domain Source](#) of that iterator. Saving the current [Domain Instance](#) will not modify the current *instance pointer*. If the save is unsuccessful, an exception will be thrown and handled through the associated *failure handler* of the iterator.

**canSave** The [canSave Predicate](#) of a [Domain Iterator](#) is false if the current [Domain Source](#) for the iterator is read-only (e.g. an external RSS feed), or a contained [Domain Attribute Instance](#) currently has an invalid value instance.

**Role and Permission-based operations** A [Domain Iterator](#) which selects instances of a [Role](#) will have four [Operations](#) defined, in order to modify the current [Roles](#) and [Permissions](#) of the selected *user instance*: [addRole](#), [removeRole](#), [addPermission](#), and [removePermission](#). These are discussed in further detail as [Builtin Operations](#).

**Attributes**

- `autosave` : `boolean`  
If `true`, then any changes to the contained `Domain Instance` will also be automatically applied to its underlying `Domain Source`.
- `limit` : `integer`  
A limit to the number of `Domain Instances` retrieved and kept in runtime memory.
- `orderAscending` : `boolean`  
If `false`, the `order` will be sorted in descending order (Z-A).
- `query` : `string`  
The query used to select `Domain Instances`.

**Containment References**

- `current instance` : `[0..1] Domain Instance`  
The current `Domain Instance` as pointed to by the current instance `pointer`.
- `current pointer` : `[0..1] Value`  
The current value of the instance pointer. If there are no results or this is a new `query`, this will return -1.  
If there is one or many results, this will return a value between 0 and results-1.
- `results` : `[0..1] Value`  
The number of results in the current result set.

**References**

- `order by` : `[0..1] Domain Attribute`  
A `Domain Attribute` used to sort the results obtained from this iterators' `query`.
- `out selects` : `[0..*] Select Edge` (opposite: `from`)  
A list of outgoing `Select Edges`, specifying the `sources` of type instances.

**Events**

- `onIterate` : `[0..1] Event`  
This `Event` is triggered when the current instance `pointer` changes, or the current `instance` is reloaded.

**Inherited Events**

`onChange` (from `Changeable`)

**Functions**

- `can save` : `[0..1] Function`  
Returns `true` if the current `instance` can be saved to the specified `Domain Source` without throwing an error.
- `empty` : `[0..1] Function`  
Returns `true` if the current result set is empty; that is, there are no matching `Domain Instances` for the given `query`.

- **has next** : [0..1] Function  
Returns true if navigating to the **next** result will not be out-of-bounds for the given **query**.
- **has previous** : [0..1] Function  
Returns true if navigating to the **previous** result will not be out-of-bounds for the given **query**.

## Operations

- **jump** : [0..1] Operation  
Navigates to a specific instance result, where 0 is the first result.
- **next** : [0..1] Operation  
Navigate to the next result. Will throw an error if **has next** would return false.
- **previous** : [0..1] Operation  
Navigate to the previous result. Will throw an error if **has previous** would return false.
- **reset** : [0..1] Operation  
Resets the current instance **pointer** and reloads the current instance.
- **skip** : [0..1] Operation  
Navigate forwards over the **provided** number of results; negative parameters may be used to navigate backwards.

## Model Completion Rules

- **Domain Iterators** do *not* have a **Value** named **field value** created by default.
- A **Sync Wire** connecting a 'new' **Domain Iterator** to an **Input Form** should only execute the 'init' **Operation** on the created **Labels** when the label is accessed if the **Domain Instance** is not empty.
- When a **Domain Iterator** is connected to an **Iterator List** by a **Set Wire**, **Labels** will be generated for its non-generated, non-primary key **Domain Attribute Instances**.
- When a **Domain Iterator** is connected to an **Iterator List** by a **Set Wire**, a **hidden Label** will be generated for the primary key **Domain Attribute Instance**.
- A **Set Wire** connecting a **Domain Iterator** to an **Input Form** should only execute the 'init' **Operation** on the created **Labels** when the label is accessed if the **Domain Instance** is not empty.
- A **Set Wire** connecting a **Domain Iterator** to an **Input Form** should only execute the 'update' **Operation** on the created **Labels** when the label is **changed** if the **Domain Instance** is not empty.
- The **selected Domain Iterator** used to select one instance of a **Detail Wire** source is given the same **Domain Source** as the source **Domain Iterator**.
- The **query** of a **Domain Iterator** used to **set** a **Login Handler** of type DOMAIN\_OBJECT will be updated according to the **Domain Type** of the handler.
- The **query** of a **Domain Iterator** used to **set** a **Login Handler** of type USER will be set according to the **Domain Attributes** provided as **Parameters** to the handler.
- The **query** of a **Domain Iterator** used to **set** a **Login Handler** of type USER will be *updated* according to the **Domain Attributes** provided as **Parameters** to the handler.
- The **query** of a **Domain Iterator** used to **set** a **Login Handler** of type USER will be *updated* according to the **Role** of the handler.
- The **query** of a **Domain Iterator** used to **set** a **Login Handler** of type USER will be *updated* according to the **Role** of the handler.

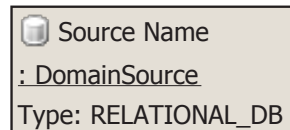
- A [Session](#) protected by a [Login Handler](#) of type `DOMAIN_OBJECT` will contain a [Domain Iterator](#) 'current instance' to represent the current domain object instance.
- A [Session](#) protected by a [Login Handler](#) of type `USER` will contain a [Domain Iterator](#) 'current instance' to represent the current domain object instance.
- [Domain Iterators](#) have a [Builtin Operation](#) named `previous`.
- [Domain Iterators](#) have a [Builtin Operation](#) named `next`.
- [Domain Iterators](#) have a [Builtin Operation](#) named `reset`.
- [Domain Iterators](#) have a [Builtin Operation](#) named `skip`.
- [Domain Iterators](#) have a [Builtin Operation](#) named `jump`.
- [Domain Iterators](#) have a [Builtin Property](#) named `has next`.
- [Domain Iterators](#) have a [Builtin Property](#) named `has previous`.
- [Domain Iterators](#) have a [Builtin Property](#) named `has next`.
- [Domain Iterators](#) have a [read only Value](#) named `results`.
- [Domain Iterators](#) have an [Event](#) named `onIterate`.
- A [Domain Iterator](#) will contain a generated [Activity Predicate](#) named 'not empty' - the inverse of the empty condition.
- The 'not empty' [Activity Predicate](#) for a [Domain Iterator](#), will be completed with the contents `necessary` to check that the iterator is not empty.
- An [Input Form](#) connected to a [Domain Iterator](#) using a [Set Wire](#) with a `limit` set (not 1), will contain [Buttons](#) named 'Next', 'Previous', 'First' and 'Last', in order to support navigation through the [Domain Iterator](#).
- An [Input Form](#) connected to a [Domain Iterator](#) using a [Set Wire](#) with a `limit` set (not 1), will contain a [Label](#) named 'Results'.
- The 'next' [Button](#) defined for navigating a [Domain Iterator](#), as defined by a [Set Wire](#), will `execute` the `next Operation` on that iterator when `clicked`.
- The 'next' [Button](#) defined for navigating a [Domain Iterator](#), as defined by a [Set Wire](#), will be conditionally `executed` only if the iterator's `has next Function` returns `true`.
- The 'previous' [Button](#) defined for navigating a [Domain Iterator](#), as defined by a [Set Wire](#), will `execute` the `previous Operation` on that iterator when `clicked`.
- The 'previous' [Button](#) defined for navigating a [Domain Iterator](#), as defined by a [Set Wire](#), will be conditionally `executed` only if the iterator's `has previous Function` returns `true`.
- The 'first' [Button](#) defined for navigating a [Domain Iterator](#), as defined by a [Set Wire](#), will `execute` the `reset Operation` on that iterator when `clicked`.
- The 'first' [Button](#) defined for navigating a [Domain Iterator](#), as defined by a [Set Wire](#), will be conditionally `executed` only if the iterator's `not empty Function` returns `true`.
- The 'last' [Button](#) defined for navigating a [Domain Iterator](#), as defined by a [Set Wire](#), will `execute` the `jump Operation` on that iterator when `clicked`.
- The 'last' [Button](#) defined for navigating a [Domain Iterator](#), as defined by a [Set Wire](#), will be conditionally `executed` only if the iterator's `not empty Function` returns `true`.
- The 'last' [Button](#) defined for navigating a [Domain Iterator](#), as defined by a [Set Wire](#), will contain a [Value](#) named 'target'.
- The 'target' [Value](#) defined in a 'last' [Button](#) defined for navigating a [Domain Iterator](#), as defined by a [Set Wire](#), will be used as a [Parameter](#) for executing the `jump Operation` on that iterator.
- The 'last' [Button](#) defined for navigating a [Domain Iterator](#), as defined by a [Set Wire](#), will contain an [Activity Operation](#) named 'update target'.
- The 'last' [Button](#) defined for navigating a [Domain Iterator](#), as defined by a [Set Wire](#), needs to `execute` the 'update target' [Operation](#) within that [Button](#) before calling `jump` on the iterator.

- The ‘results’ [Label](#) defined for navigating a [Domain Iterator](#), as defined by a [Set Wire](#), will be updated whenever the iterator [changes](#).
- The current results [count](#) of the [Domain Iterator](#) will be used as a [Parameter](#) to update the ‘results’ [Label](#) defined for navigating a [Domain Iterator](#) with a [Set Wire](#) when the iterator [changes](#).
- The ‘results’ [Label](#) defined for navigating a [Domain Iterator](#), as defined by a [Set Wire](#), will be updated whenever the containing [Input Form](#) is [accessed](#).
- The current results [count](#) of the [Domain Iterator](#) will be used as a [Parameter](#) to update the ‘results’ [Label](#) defined for navigating a [Domain Iterator](#) with a [Set Wire](#) when the containing [Input Form](#) is [accessed](#).
- All [Domain Iterators](#) will be populated with a single instance of a [Domain Instance](#), representing the [current instance](#) of that iterator.
- A [Domain Iterator](#) contains an [Operation](#) named [save](#).

## I.39 Domain Source

A specific source of instances of a particular [Domain Type](#).

**Direct supertypes** [Contains Wires](#), [Generates Elements](#), [Named Element](#)



### Attributes

- [cache](#) : integer  
If the [type](#) of this [Domain Source](#) is an RSS feed, values may be cached according to this timeout. If this value is 0, caching will not be permitted.
- [file](#) : FileReference  
If the [type](#) of this [Domain Source](#) is a file, this reference will specify its path.
- [type](#) : DomainStoreTypes  
The source type.  
Possible values: RELATIONAL\_DB, PROPERTIES\_FILE, RSS2\_0
- [url](#) : string  
If the [type](#) of this [Domain Source](#) is an RSS feed, this specifies the public URL of this external feed.

### References

- [in selects](#) : [0..\*] [Select Edge](#) (opposite: [to](#))  
A list of outgoing [Select Edges](#).
- [out schemas](#) : [0..\*] [Schema Edge](#) (opposite: [from](#))  
A list of incoming [Schema Edges](#) to specify the [Domain Types](#) supplied by this [Domain Source](#).



## Operations

- **refresh** : [0..1] Operation

If the **type** of this **Domain Source** is an RSS feed, forces the external feed to be reloaded regardless of the current **cache**.

## Model Completion Rules

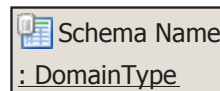
- A **Role** extending the *default Role* will be provided the same **Domain Source**.
- A **Domain Source** of **type** PROPERTIES\_FILE will generate a **Domain Type** named 'properties'.
- A **Domain Source** of **type** PROPERTIES\_FILE will populate the 'properties' **Domain Type** with string-typed **Domain Attributes** for each value in the source properties file.

# I.40 Domain Type

The composition of individual data **elements** into a domain-specific structure.

**Direct supertypes** Contains Wires, Extends Edge Destination, Extends Edges Source, Generated Element, Generates Elements, Parameter Edges Source, Requires Edges Source

**Direct subtypes** Role



A **Domain Type** can be composed of many **Domain Attributes**, each of which has a particular *data type*. Foreign keys are supported by declaring that a given **Domain Attributes** extends another through an **Extends Edge**. A **Domain Type** can *extend* another **Domain Type** through an **Extends Edge** to describe a simple form of inheritance. Multiple inheritance is supported, but circular inheritance is not allowed.

## Containment References

- **permissions** : [0..\*] Permission

If this **Domain Type** is a **Role**, a list of **Permissions** that this **Domain Type** may provide.

## References

- **in schemas** : [0..\*] Schema Edge (opposite: to)

A list of outgoing **Schema Edges** to specify the potential instance **sources** of this type.

## Model Completion Rules

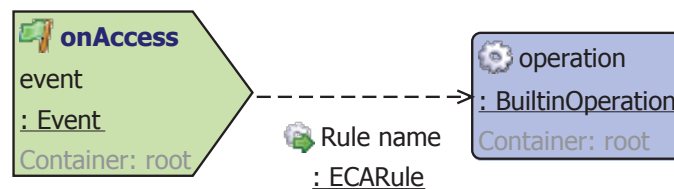
- A **Domain Source** of **type** PROPERTIES\_FILE will populate the 'properties' **Domain Type** with string-typed **Domain Attributes** for each value in the source properties file.
- A **Domain Type** which does not contain a primary **key Domain Attribute** will contain one named 'generated primary key'.

- A non- primary key Domain Attribute in a Domain Type, represented as a supertype of another Domain Type, will be copied into the subtype schema and marked as an subattribute.
- A primary key Domain Attribute in a Domain Type, represented as a supertype of another Domain Type, will be copied into a new 'ID' Domain Attribute and marked as a foreign key.

## I.41 ECA Rule

An ECA Rule defines how a triggering Event may execute a particular Action, as optionally restricted by incoming Conditions.

**Direct supertypes** Condition Edge Destination, Generated Element, Generates Elements, Named Element, Parameter Edge Destination



An ECA Rule specifies that a triggered Event will execute a specific Action if permitted to by any incoming Conditions. This ECA Rule may have incoming Parameters, which are passed by value at the time that the Event is triggered. This ECA Rule may also have incoming Conditions, which are evaluated at the time of the triggered Event; if these conditions are not true, then the Action will not be executed.

When an Event is triggered, all outgoing ECA Rules are sorted in descending order according to their integer property priority, and then evaluated in a sequential manner according to their incoming Predicates; this set is known as the *current execution set*. All executable ECA Rules are then followed sequentially in this order, and the type of action specified by the target Action are executed, until all ECA Rules are executed or an Action fails.

For an ECA Rule to be *executable*, all incoming Conditions must evaluate to true according to logical conjunction (AND). Conditions may be connected<sup>7</sup> with a Constraint Edge to change this behaviour to logical disjunction (OR) or exclusive disjunction (XOR). Any incoming Parameters to that ECA Rule are then calculated, and passed as named parameter values to the target Action.

### Attributes

- **priority** : integer

ECA Rules are executed in order of descending priority; that is, a higher priority ECA Rule will execute first.

### References

- **target** : Action (opposite: rules)  
The target, or consequence, of the ECA Rule.
- **trigger** : Action Edge Source (opposite: listeners)  
The source, or trigger, of the ECA Rule, normally an Event.

<sup>7</sup>Issue 134: Allow ConstraintWires to construct complex condition requirements for ConditionWires.


## Model Completion Rules

- If a **Login Handler** has an outgoing navigation **rule** named 'logout', then this navigation rule will be **executed** when the **logout Frame** is **accessed**.
- A **Frame** contained by a **Session** protected by a **Login Handler** of type **SECRET\_KEY** will **execute** the 'check key' **Operation** with a high **priority** when the frame is **accessed**.
- When the **logout Frame** for a **Login Handler** is **accessed**, the 'do logout' **Operation** will be **called**.
- If a **Login Handler** has a defined outgoing **ECA Rule** named 'logout', then this rule should be executed after a successful 'do logout' **Activity Operation**.
- If no 'success' **ECA Rule** is defined for a **Login Handler**, then the 'login successful' **Frame** will be defined as the 'success' **ECA Rule**.
- If no 'logout' **ECA Rule** is defined for a **Login Handler**, then the 'logout successful' **Frame** will be defined as the 'logout' **ECA Rule**.
- An **input entry Gate** will **update** the value of each **input Label** from the value of each similarly named **Visible Thing** in the **input Frame** when the protected **Scope** is **accessed**.
- All **conditions** against a **Sync Wire** will be copied to each **ECA Rule** subsequently created as connected to the **onChange Events**.
- All **conditions** against a **Sync Wire** will be copied to each **ECA Rule** subsequently created as connected to the **onAccess Events**.
- All **Parameters** for each **conditions** against a **Sync Wire** will be copied to each **ECA Rule** subsequently created as connected to the **onChange Events**.
- All **Parameters** for each **conditions** against a **Sync Wire** will be copied to each **ECA Rule** subsequently created as connected to the **onAccess Events**.
- All **conditions** against a **Set Wire** will be copied to each **ECA Rule** subsequently created as connected to the **onChange Events**.
- All **conditions** against a **Set Wire** will be copied to each **ECA Rule** subsequently created as connected to the **onChange Events**.
- All **Parameters** for each **conditions** against a **Set Wire** will be copied to each **Simple Condition** subsequently created for **ECA Rules** connected to the **onChange Events**.
- All **Parameters** for each **conditions** against a **Set Wire** will be copied to each **Simple Condition** subsequently created for **ECA Rules** connected to the **onAccess Events**.

## I.42 EXSD Data Type

Provides a bridge from XSD Simple Data Types to EData Types.

**Direct supertypes** Generated Element, Generates Elements

 xsd:boolean : EXSDDataType
---

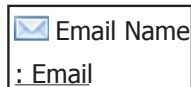
### References

- **definition** : [0..1] **XSDSimpleTypeDefinition** (from XSD)  
 The target XSD type definition.

## I.43 Email

A **Message** sent as an e-mail message to a particular recipient.

**Direct supertypes** Accessible, Action Edge Source, Can Be Synced, Contains Functions, Contains Operations, Contains Values, Contains Wires, Message, Wireable



An **Email** is designed to be used in the composition and delivery of mail messages. An **Email** may specify five directly contained **Values**, all of which must be **type** string. If the following **Email** attributes are defined as **Values**, they may be set at runtime to programmatically control the sender, recipient or subject of the **Email**:

- **from**, representing the e-mail address of the sender. This **Value** *must* be set for an **Email** to be sent.
- **fromName**, representing a human-readable name of the sender. This **Value** is optional.
- **to**, representing the e-mail address of the receiver. This **Value** *must* be set for an **Email** to be sent.
- **toName**, representing a human-readable name of the receiver. This **Value** is optional.
- **subject**, representing the subject of the mail message. This **Value** *must* be set for an **Email** to be sent.

An **Email** contains the **Operation send**, which will attempt to deliver the current **Email** instance as described by **Builtin Operation**. When an **Email** is composed, the *body* of the **Email** will be composed of all contained **Values** and their values at the time the **Email** was sent. The format of this body may use a *custom template* if this template is specified.

An **Email** contains the **Events** *onSent* and *onFailure*. These events are triggered once an **Email** has been successfully delivered and when an **Email** is unsuccessfully received, as discussed in **Event**.

An IAML model instance may contain any number of **Emails**. The *storage semantics* for a **Email** follow the storage semantics for its *containing scope*, similar to the *storage semantics* for the **Frame** element.

### Attributes

- **from** : string  
The sender e-mail address for this **Email**.
- **fromName** : string  
The sender name for this **Email**, which is optional.
- **subject** : string  
The subject for this **Email**.
- **to** : string  
The recipient e-mail address for this **Email**.
- **toName** : string  
The recipient name for this **Email**, which is optional.

**Containment References**

- **buttons** : [0..\*] **Button**

Within an **Email**, all **Buttons** will be represented as hyperlinks.

- **labels** : [0..\*] **Label**

Within an **Email**, all **Labels** will be represented as text labels.

**Events**

- **onFailure** : [0..1] **Event**

This **Event** is triggered if the sent e-mail has failed to arrive at its destination, which may occur at any point after it is sent.

- **onSent** : [0..1] **Event**

This **Event** is triggered when the **Email** is sent, however it is not guaranteed that it has been *successfully* sent.

**Inherited Events**

**onAccess** (from **Accessible**)

**Model Completion Rules**

- An **Input Form** connected to an **Email** with a **Set Wire** will create a **Label** in the target Email for every **Input Text Field** in the source form with the same **name**.
- An **Input Form** connected to an **Email** with a **Set Wire** will create a **Label** in the target Email for every **Label** in the source form with the same **name**.
- An **Input Form** connected to a **Map** with a **Set Wire** will create a **Map Point** in the target map for every **Input Text Field** in the source form with the same **name**.
- An **Email** contains a **Value** named 'to'.
- An **Email** contains a **Value** named 'toName'.
- An **Email** contains a **Value** named 'from'.
- An **Email** contains a **Value** named 'fromName'.
- An **Email** contains a **Value** named 'subject'.
- **Emails** have an **Event** named **onSent**.
- **Emails** have an **Event** named **onFailure**.

**I.44 Event**

The source of a triggerable event for a particular model element.

**Direct supertypes** **Action Edge Source**, **Named Element**

**onChange**

The *onChange* event is defined for all **Changeable** model elements. If the element is a **Visible Thing**, then *onChange* is triggered when a control loses the input focus *and* its contained **fieldValue** has been modified since it gained focus. For visual elements, *onChange* will *not* be triggered while the element still has the



### onInput

The *onInput* event is defined for all [Visible Thing](#) model elements. The event is triggered whenever its contained *fieldValue* is modified.

It is not guaranteed that *onInput* will ever fire; nor that it will fire upon every character keystroke. For performance reasons and to prevent excessive network requests, this event may be buffered during periods of intense input.

### onAccess

The *onAccess* event is defined for all [Accessible](#) model elements, which includes all [Visible Things](#), [Scopes](#) and [Emails](#). The *onAccess* event may be triggered multiple times within the same request.

- The *onAccess* [Event](#) is triggered on a [Visible Thing](#) when the element is *about to be* rendered to the client.
- The *onAccess* [Event](#) is triggered on a [Scope](#) when the scope is *about to be* accessed.
- The *onAccess* [Event](#) is triggered on an [Email](#) when the email is *about to be* rendered into a static form for delivery.

### onInit

The *onInit* event is defined for all [Scope](#) model elements, and it is fired whenever an instance of that [Scope](#) is initialised. A [Frame](#) is initialised whenever the [Frame](#) is about to be rendered to the client; and a [Session](#) is initialised when it is accessed for the first time by a *user*.

The *onInit* event is guaranteed to be triggered once before *onAccess* is triggered. It is not guaranteed that *onAccess* will be fired immediately after *onInit*, and it may never be fired (for example, within a [Session](#) that is referenced, but never accessed).

### onClick

The *onClick* event is defined for all [Visible Thing](#) model elements, and is triggered when the element is clicked by a pointer, or otherwise activated in a manner which simulates such a click. For composite visual elements, a *click* is *bubbled up* the containment hierarchy of [Visible Things](#).

### onSent

The *onSent* event is defined for all [Email](#) model elements, and is triggered when an [Email](#) has been successfully delivered to an outgoing email host. Once an *onSent* event has been triggered, the *onFailure* event may still be triggered.

### onFailure

The *onFailure* event is defined for all [Email](#) model elements, and is triggered when an [Email](#) could not be successfully delivered to an outgoing email host, according to the local e-mail delivery configuration. Once an *onFailure* event has been triggered, the *onSent* event cannot be triggered.

## onIterate

The *onIterate* event is defined for all [Domain Iterator](#) model elements, and is triggered immediately after the current *instance pointer* of a given [Domain Iterator](#) changes, or the current instance is reloaded through the *reload* operation.

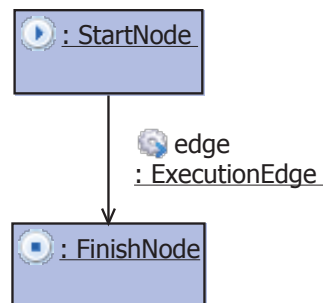
## Model Completion Rules

- [Emails](#) have an [Event](#) named *onSent*.
- [Emails](#) have an [Event](#) named *onFailure*.
- [Changeable](#) elements have an [Event](#) named *onChange*.
- [Accessible](#) elements have an [Event](#) named *onAccess*.
- [Scopes](#) have an [Event](#) named *onInit*.
- [Domain Iterators](#) have an [Event](#) named *onIterate*.
- [Visible Things](#) have an [Event](#) named *onInput*.
- [Visible Things](#) have an [Event](#) named *onClick*.

## I.45 Execution Edge

Represents the execution flow within an [Activity Operation](#) or [Activity Predicate](#).

**Direct supertypes** [Generated Element](#), [Named Element](#)



## References

- **from** : [Execution Edges Source](#) (opposite: out executions)  
The source of this [Execution Edge](#).
- **to** : [Execution Edge Destination](#) (opposite: in executions)  
The target of this [Execution Edge](#).

## I.46 Execution Edge Destination

A possible destination for an [Execution Edge](#).

**Interface.**

**Direct subtypes** Cancel Node, Decision Node, Finish Node, Join Node, Operation Call Node, Set Node, Split Node

This abstract interface is provided to simplify the implementation of Execution Edges, as EMF does not support type unions in metamodel definitions.

#### References

- **in executions** : [0..\*] Execution Edge (opposite: to)  
A list of incoming Execution Edges.

## I.47 Execution Edges Source

A possible source for an Execution Edge.

#### Interface.

**Direct subtypes** Decision Node, Join Node, Operation Call Node, Set Node, Split Node, Start Node

This abstract interface is provided to simplify the implementation of Execution Edges, as EMF does not support type unions in metamodel definitions.

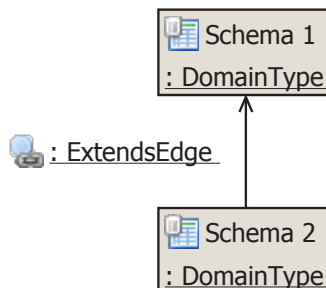
#### References

- **out executions** : [0..\*] Execution Edge (opposite: from)  
A list of outgoing Execution Edges.

## I.48 Extends Edge

A simple inheritance or subtyping relationship from the source element to the target element.

**Direct supertypes** Generated Element, Generates Elements



#### References

- **from** : Extends Edges Source (opposite: out extends edges)  
The source, or subtype, of this Requires Edge relationship.
- **to** : Extends Edge Destination (opposite: in extends edges)  
The target, or supertype, of this Requires Edge relationship.



## Model Completion Rules

- A [Domain Iterator](#) within a [Session](#) protected by both an [Access Control Handler](#) and [Login Handler](#) of type `DOMAIN_OBJECT` or `USER`, but the handlers both use different [Domain Types](#) (for example, with inheritance), and which does not specify a [Select Edge](#), will use the same [Domain Source](#) as the [Domain Type](#) used as a [Parameter](#) to the [Access Control Handler](#).
- A [Domain Iterator](#) within a [Session](#) protected by both an [Access Control Handler](#) and [Login Handler](#) of type `DOMAIN_OBJECT` or `USER`, but the handlers both use the same [Domain Types](#), and which does not specify a [Select Edge](#), will use the same [Domain Source](#) as the [Domain Type](#) used as a [Parameter](#) to the [Access Control Handler](#).
- A [Domain Iterator](#) within a [Session](#) protected by both an [Access Control Handler](#) and [Login Handler](#) of type `DOMAIN_OBJECT` or `USER`, but the [Access Control Handler](#) requires a [Permission](#) without a [Role](#), and which does not specify a [Select Edge](#), will use the same [Domain Source](#) as the [Domain Type](#) used as a [Parameter](#) to the [Access Control Handler](#).
- All [Roles](#) that do not extend another [Role](#) will extend the *default Role*.
- A [Role](#) extending the *default Role* will be provided the same [Domain Source](#).
- A non- [primary key Domain Attribute](#) in a [Domain Type](#), represented as a supertype of another [Domain Type](#), will be copied into the subtype schema and marked as an [subattribute](#).
- A [primary key Domain Attribute](#) in a [Domain Type](#), represented as a supertype of another [Domain Type](#), will be copied into a new 'ID' [Domain Attribute](#) and marked as a foreign key.
- A [Domain Attribute Instance](#) created within a [Domain Instance](#) referencing a particular [Domain Attribute](#) will be marked as an extension of that attribute through an [Extends Edge](#).

## I.49 Extends Edge Destination

An element which may define an incoming supertype relationship through [Extends Edge](#).

### Interface.

**Direct subtypes** [Domain Feature](#), [Domain Feature Instance](#), [Domain Type](#)

**Indirect subtypes** [Domain Attribute](#), [Domain Attribute Instance](#), [Role](#)

This abstract interface is provided to simplify the implementation of [Extends Edges](#), as EMF does not support type unions in metamodel definitions.

### References

- in [extends edges](#) : [0..\*] [Extends Edge](#) (opposite: to)  
A list of incoming [Extends Edges](#).

## I.50 Extends Edges Source

An element which may define an outgoing subtype relationship through [Extends Edge](#).

### Interface.

**Direct subtypes** Domain Feature, Domain Feature Instance, Domain Type

**Indirect subtypes** Domain Attribute, Domain Attribute Instance, Role

This abstract interface is provided to simplify the implementation of [Extends Edges](#), as EMF does not support type unions in metamodel definitions.

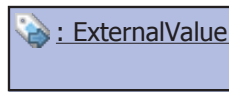
#### References

- `out extends edges : [0..*]` [Extends Edge](#) (opposite: from)  
A list of outgoing [Extends Edges](#).

## I.51 External Value

Acts as a bridge between [Activity Nodes](#) to external [Values](#).

**Direct supertypes** [Activity Node](#), [Data Flow Edge Destination](#), [Data Flow Edges Source](#), [Parameter Edges Source](#)



A bridge is necessary between [Activity Nodes](#) and [Values](#) as it should be possible to access values outside of an [Activity Operation](#) or [Activity Predicate](#), without having to define all [Values](#) as a [Data Flow Edges Source](#) or [Data Flow Edge Destination](#).

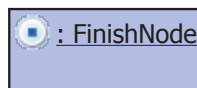
#### References

- `value : [0..1]` [Value](#)  
The referenced value of this [External Value](#).

## I.52 Finish Node

Specifies that the current [Activity Operation](#) executed successfully, or the [Activity Predicate](#) must return `true`.

**Direct supertypes** [Activity Node](#), [Execution Edge Destination](#)



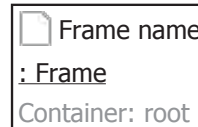
An [Activity Operation](#) may contain any number of [Finish Nodes](#). When the *execution flow* arrives at a [Finish Node](#), the execution of the operation will finish successfully.

The [Finish Node](#) model element is based on the UML *FinalNode* (activity final) model element.

## I.53 Frame

A block of content that may be accessed and rendered independently by a web browser, often as a single web page.

**Direct supertypes** [Action](#), [Parameter](#) [Edges](#) [Source](#), [Scope](#)



An IAML model instance may contain any number of [Frames](#). The *onInit* Event for an [Internet Application](#) is triggered every time the [Frame](#) is rendered to a client.

The *storage semantics* for a [Frame](#) follow the storage semantics for its *containing scope*. For example, if the [Frame](#) is directly contained by a [Session](#), then the storage semantics of the parent [Session](#) will be used; likewise, if the [Frame](#) is directly contained by an [Internet Application](#), then the storage semantics of the parent [Internet Application](#) will be used.

A [Frame](#) may contain any number of [Visible Things](#), and this set of elements defines the *user interface* for that [Frame](#). All [Visible Things](#) may be *rendered* to a user. When a [Visible Thing](#) is rendered to a user, all children [Visible Things](#) must also be rendered in order according to their *renderOrder*. A [Visible Thing](#) that is currently *hidden* must not be rendered until that [Visible Thing](#) is subsequently *shown* through the *show* Operation. The *name* property of the [Visible Thing](#) may be used as a label for the rendered element.

### Attributes

- **render** : [RenderTypes](#)

The format that the [Frame](#) should be rendered as; by default, to HTML/Javascript.

Possible values: HTML, RSS20

- **url** : string

The intended runtime URL of this [Frame](#), which is normally derived from the *id* of this element.

### Containment References

- **children** : [0..\*] [Visible Thing](#)

A [Frame](#) can directly contain other [Visible Things](#), which are then rendered to the client.

### Inherited Events

[onAccess](#) (from [Accessible](#)); [onInit](#) (from [Scope](#))

### Model Completion Rules

- A [Scope](#) protected by a [Login Handler](#) will create a *login* [Frame](#) named 'login' within the *login* [Session](#).
- The *login* [Frame](#) created for a [Login Handler](#) protecting a [Scope](#) will be provided as the login *rule* for the [Login Handler](#) if none is specified.

- A [Scope](#) protected by a [Login Handler](#) will create a *logout Frame* named 'logout' within the same [Scope](#).
- If no 'success' [ECA Rule](#) is defined for a [Login Handler](#), then a default 'login successful' [Frame](#) will be created.
- If no 'logout' [ECA Rule](#) is defined for a [Login Handler](#), then a default 'logout successful' [Frame](#) will be created.
- A [Login Handler](#) is generated for a [Frame](#) contained within a [Session](#) that contains an [Access Control Handler](#).
- A [Frame](#) within a [Session](#) protected by an [Access Control Handler](#) will have an [Operation](#) named 'permissions check'.
- A [Frame](#) protected by an [Access Control Handler](#) will have an [Operation](#) named 'page permissions check'.
- A [Frame](#) within a [Session](#) protected by an [Access Control Handler](#) will execute the 'permissions check' [Operation](#) when the frame is [accessed](#).
- A [Frame](#) protected by an [Access Control Handler](#) will execute the 'page permissions check' [Operation](#) when the frame is [accessed](#).
- The 'permissions check' [Operation](#) defined by an [Access Control Handler](#) protecting a [Frame](#) within a [Session](#) will [navigate](#) to the 'login' [Frame](#) within the same [Session](#) if the check fails.
- The 'permissions check' [Operation](#) defined by an [Access Control Handler](#) protecting a [Frame](#) within a [Session](#) will [navigate](#) to the 'login' [Frame](#) within a *separate* login [Session](#) if the check fails.

## I.54 Function

A [typed Function](#) defines a function signature as a sequence of [slots](#), each with a name and [type](#).

**Abstract type.**

**Direct supertypes** [Named Element](#), [Wire Source](#)

**Direct subtypes** [Predicate](#), [XQuery Function](#)

**Indirect subtypes** [Activity Predicate](#), [Boolean Property](#), [Builtin Property](#), [XQuery Predicate](#)

### Attributes

- [slotNames](#) : [0..\*] [string](#)  
A list of the named slots of this [Function](#).

### References

- [conditioned](#) : [0..\*] [Complex Term](#) (opposite: [function](#))  
A [Complex Term](#) may be used in the definition of a [Condition](#), if this [Function](#) is a [Predicate](#).
- [slot types](#) : [0..\*] [EClassifier](#) (from [Ecore](#))  
A list of the type of each slot of this [Function](#).
- [type](#) : [0..1] [EClassifier](#) (from [Ecore](#))  
The return type of this [Function](#).

## I.55 Gate

Restricts access into, or out of, a [Scope](#) until all incoming [Conditions](#) are satisfied.

**Direct supertypes** [Action](#), [Action Edge Source](#), [Condition Edge Destination](#), [Generated Element](#), [Generates Elements](#), [Named Element](#), [Requires Edges Source](#)



A [Scope](#) may contain a single entry [Gate](#) through [entryGate](#), and a single exit [Gate](#) through [exitGate](#).

### Entry Gate

An entry [Gate](#) must have at least one outgoing [ECA Rule](#) which has a destination of a [Frame](#); this first target [Frame](#) represents the *incoming gate*. If an entry [Gate](#) is executed as an [Action](#), and the entry [Gate](#) has been *satisfied*, the browser is redirected to the [Frame](#) which triggered the entry [Gate](#) (the *resume location*). If the satisfied entry [Gate](#) was not triggered by a [Frame](#), or the entry [Gate](#) is not satisfied, then the browser is redirected to the *incoming gate*.

When a [Frame](#) directly contained within this [Scope](#) is accessed or rendered, this entry [Gate](#) must first be satisfied; that is, all incoming [Conditions](#) must be considered *successful*. If the entry [Gate](#) is not satisfied, then the *failure rules* of that exit [Gate](#) are executed in descending [priority](#) order according to the execution semantics of [ECA Rule](#).

If an entry [Gate](#) is executed as an [Action](#), and the entry [Gate](#) has been *satisfied*, the browser is redirected to the [Frame](#) which triggered the entry [Gate](#) (the *resume location*). If the satisfied entry [Gate](#) was not triggered by a [Frame](#), or the entry [Gate](#) is not satisfied, then the browser is redirected to the *incoming gate*.

An entry [Gate](#) may have an outgoing [ECA Rule](#) to a [Frame](#) with a [name](#) “first”. This [Frame](#) must reside outside of the containing [Scope](#) of that [Gate](#). When this model is completed through *model completion*, this entry [Gate](#) will be extended to first check that the current user has successfully visited this [Frame](#).

### Exit Gate

An exit [Gate](#) must have at least one outgoing [ECA Rule](#) which has a destination of a [Frame](#); this first target [Frame](#) represents the *outgoing gate*. If an exit [Gate](#) is executed as an [Action](#), and the exit [Gate](#) has been *satisfied*, the browser is redirected to the [Frame](#) which triggered the exit [Gate](#) (the *resume location*). If the satisfied exit [Gate](#) was not triggered by a [Frame](#), or the exit [Gate](#) is not satisfied, then the browser is redirected to the *outgoing gate*.

When a [Frame](#) within the model, but not contained (directly or indirectly) within that [Scope](#) is accessed or rendered, this exit [Gate](#) must first be satisfied; that is, all incoming [Conditions](#) must be considered *successful*. If the exit [Gate](#) is not satisfied, then the *failure rules* of that exit [Gate](#) are executed in descending [priority](#) order according to the execution semantics of [ECA Rule](#).

If an exit [Gate](#) is executed as an [Action](#), and the exit [Gate](#) has been *satisfied*, the browser is redirected to the [Frame](#) which triggered the exit [Gate](#) (the *resume location*). If the satisfied exit [Gate](#) was not triggered by a [Frame](#), or the exit [Gate](#) is not satisfied, then the browser is redirected to the *outgoing gate*.

A exit [Gate](#) may have an outgoing [ECA Rule](#) to a [Frame](#) with a [name](#) “last”. This [Frame](#) must reside within the containing [Scope](#) of that exit [Gate](#). When this model is completed through *model completion*, the exit [Gate](#) will be extended to first check that the current user has successfully visited this [Frame](#).

## Model Completion Rules

- For exit [Gate](#)s protected by a ‘last’ [ECA Rule](#), the completed model only requires the target of the [ECA Rule](#) to be visited once.
- If a [Gate](#) is protected by a “first” [ECA Rule](#), then the destination [Frame](#) will have a new [Button](#) created called “Continue”
- An entry [Gate](#) can [require](#) a given [Label](#) to be set, creating a [Frame](#) that must be filled in before the containing [Scope](#) can be accessed.
- A [Gate](#) referencing a [Frame](#), with an outgoing [ECA Rule](#) named ‘first’, ‘last’ or ‘fail’, will create a [Button](#) named ‘continue’ in the referenced [Frame](#).
- The ‘continue’ [Button](#) generated for a [Gate](#), with an outgoing [ECA Rule](#) named ‘first’, ‘last’ or ‘fail’, will execute the [Gate](#) when [clicked](#).
- A [Scope](#) protected by a [Gate](#) named ‘XXX’, with an outgoing [ECA Rule](#) named ‘first’ or ‘last’, will contain a [flag Value](#) named ‘XXX flag’, in order to track the runtime status of the [Gate](#).
- A [flag Value](#) generated by a [Gate](#) with an outgoing [ECA Rule](#) named ‘first’ or ‘last’ is [accessed](#), will generate an [Activity Operation](#) in that [Scope](#) named ‘Set gate flag’.
- When a [Frame](#) protected by a [Gate](#) with an outgoing [ECA Rule](#) named ‘first’ or ‘last’ is [accessed](#), the ‘Set gate flag’ [Operation](#) will be [executed](#).
- The [Activity Operation](#) ‘Set gate flag’ generated for a [Gate](#) protecting a [Frame](#), with an outgoing [ECA Rule](#) named ‘first’ or ‘last’, will be completed with the contents [necessary](#) to set the [flag Value](#).
- A [Scope](#) protected by a [Gate](#) named ‘XXX’, with an outgoing [ECA Rule](#) named ‘first’ or ‘last’, will contain an [Activity Predicate](#) named ‘check XXX’ to check that the [Gate](#) has been satisfied.
- The ‘check XXX’ [Function](#) for a [Gate](#), with an outgoing [ECA Rule](#) named ‘first’ or ‘last’, will be provided as a runtime [Simple Condition](#) check to that [Gate](#).
- The ‘check XXX’ [Activity Predicate](#) for a [Gate](#), with an outgoing [ECA Rule](#) named ‘first’ or ‘last’, will be completed with the contents [necessary](#) to evaluate the [flag Value](#).
- A [Gate](#) protecting a [Scope](#) as an entry [gate](#) with an incoming [Requires Edge](#) from an *expected input value* (represented as a [Label](#)) – known as an *input entry Gate* – will create an *input Frame* to allow the user to enter in these expected values.
- An *input entry Gate* will create an outgoing navigation [rule](#) from the [Gate](#) named ‘fail’ to the *input Frame*.
- An *input Frame* generated by an *input entry Gate* will contain an [Input Text Field](#) for every [Label](#) required by that [Gate](#).
- An *input entry Gate* will check that every [Label](#) is set, through the ‘fieldValue is set’ [Function](#) of each [Label](#).
- An *input entry Gate* will [update](#) the value of each *input Label* from the value of each similarly named [Visible Thing](#) in the *input Frame* when the protected [Scope](#) is [accessed](#).
- An *input entry Gate* will only update the value of each *input Label* from [Visible Things](#) in the *input Frame* if the source elements are [set](#).
- An *input entry Gate* will [only](#) update the value of each *input Label* from [Visible Things](#) in the *input Frame* if the source elements can be [cast](#).

## I.56 Generated Element

An element which may be generated through *model completion*, and may be generated due to the presence of other [elements](#).

**Interface.**

**Direct subtypes** Activity Node, Can Be Synced, Changeable, Constraint Edge, Data Flow Edge, Domain Feature, Domain Instance, Domain Type, ECA Rule, EXSD Data Type, Execution Edge, Extends Edge, Gate, Named Element, Parameter, Provides Edge, Requires Edge, Schema Edge, Scope, Select Edge, Visible Thing, Wire

**Indirect subtypes** Access Control Handler, Activity Operation, Activity Parameter, Activity Predicate, Arithmetic, Autocomplete Wire, Boolean Property, Builtin Operation, Builtin Property, Button, Cancel Node, Cast Node, Complex Term, Decision Node, Detail Wire, Domain Attribute, Domain Attribute Instance, Domain Feature Instance, Domain Iterator, Domain Source, Email, Event, External Value, Finish Node, Frame, Function, Input Form, Input Text Field, Internet Application, Iterator List, Join Node, Label, Login Handler, Map, Map Point, Message, Operation, Operation Call Node, Parameter Value, Permission, Predicate, Query Parameter, Role, Session, Set Node, Set Wire, Simple Condition, Split Node, Start Node, Sync Wire, Temporary Variable, Value, XQuery Function, XQuery Predicate

**Attributes**

- `generatedRule` : string

To assist in debugging, this attribute may be populated with the name of the model completion rule that created this element. More extensive traceability support may be provided in the future.

- `id` : string

A unique identifier for this model element. Should be globally unique.

- `isGenerated` : boolean

Set to true if the current element has been generated.

**References**

- `generated by` : [0..\*] Generates Elements (opposite: generated elements)

A list of `elements` that generated this element; conversely, `overriding` the source will prevent this element from being created.

## I.57 Generates Elements

This element may be used within *model completion* to generate other `elements`. It can also be overridden.

**Interface.**

**Direct subtypes** Access Control Handler, Activity Operation, Activity Predicate, Can Be Synced, Changeable, Domain Feature, Domain Instance, Domain Iterator, Domain Source, Domain Type, ECA Rule, EXSD Data Type, Extends Edge, Gate, Internet Application, Login Handler, Message, Parameter Value, Schema Edge, Scope, Select Edge, Visible Thing, Wire

**Indirect subtypes** Autocomplete Wire, Button, Complex Term, Detail Wire, Domain Attribute, Domain Attribute Instance, Domain Feature Instance, Email, Frame, Input Form, Input Text Field, Iterator List, Label, Map, Map Point, Role, Session, Set Wire, Simple Condition, Sync Wire, Value

**Attributes**

- `overridden` : `boolean`  
If a `Generates Elements` is *overridden*, then this element will not generate any elements *directly*.
- `overriddenNames` : `[0..*] string`  
When applying model completion, no `Named Elements` with these `names` will be generated by this element.

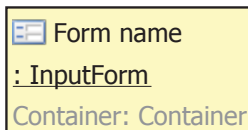
**References**

- `generated elements` : `[0..*] Generated Element` (opposite: `generated by`)  
A list of `Generated Elements` that this element has generated through *model completion*.

## I.58 Input Form

A group of related input elements, derived from the HTML FORM element.

**Direct supertypes** `Visible Thing`



An `Input Form` represents a group of related `Visible Things`. An `Input Form` cannot accept or lose focus. The `onInput Event` can never be triggered for an `Input Form`. The `name` of the `Input Text Field` may be used as a label or heading for the rendered form. The `fieldValue` of the `Input Form` has no renderable effect, and may be used to store temporary data.

**Inherited Events**

`onAccess` (from `Accessible`); `onChange` (from `Changeable`); `onClick` (from `Visible Thing`); `onInput` (from `Visible Thing`)

**Model Completion Rules**

- `Input Forms` do *not* have a `Value` named `field value` created by default.
- An `Input Form` connected to another `Input Form` with a `Sync Wire` will create an `Input Text Field` in the target form for every `Input Text Field` in the source form with the same `name`.
- An `Input Form` connected to another `Input Form` with a `Sync Wire` will create a `Label` in the target form for every `Label` in the source form with the same `name`.
- An `Input Form` connected to another `Input Form` with a `Set Wire` will create a `Label` in the target form for every `Input Text Field` in the source form with the same `name`.
- An `Input Form` connected to another `Input Form` with a `Set Wire` will create a `Label` in the target form for every `Label` in the source form with the same `name`.
- An `Input Form` connected to an `Email` with a `Set Wire` will create a `Label` in the target Email for every `Input Text Field` in the source form with the same `name`.

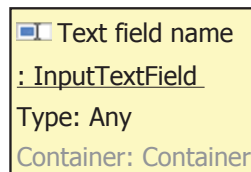


- An **Input Form** connected to an **Email** with a **Set Wire** will create a **Label** in the target Email for every **Label** in the source form with the same **name**.
- An **Input Form** connected to a **Map** with a **Set Wire** will create a **Map Point** in the target map for every **Input Text Field** in the source form with the same **name**.
- A **login Frame** for a **Login Handler** will contain an **Input Form** named 'login form'.
- An **Input Form** connected to a **Domain Iterator** using a **Set Wire** with a **limit** set (not 1), will contain **Buttons** named 'Next', 'Previous', 'First' and 'Last', in order to support navigation through the **Domain Iterator**.
- An **Input Form** connected to a **Domain Iterator** using a **Set Wire** with a **limit** set (not 1), will contain a **Label** named 'Results'.

## I.59 Input Text Field

A single text field, storing a single text value, which may be edited by the user.

**Direct supertypes** **Visible Thing**



An **Input Text Field** represents a user-editable text field in the user interface. An **Input Text Field** may accept and lose focus. The **onInput Event** may be triggered according to the semantics of **onInput**. The **name** of the **Input Text Field** may be used as a label for the rendered text field. The **fieldValue** of the **Button** is used as the current value of the rendered text field, and may be changed at runtime.

### Inherited Events

**onAccess** (from **Accessible**); **onChange** (from **Changeable**); **onClick** (from **Visible Thing**); **onInput** (from **Visible Thing**)

### Model Completion Rules

- A **Sync Wire** connecting two elements with a **Function** 'can sync?' will only permit synchronisation if the value can be cast **successfully**.
- When an **Input Text Field** is connected to a **Can Be Synced** element by an **instant Sync Wire**, the **value** of the source element will be provided as a **Parameter** to the updating **ECA Rule**.
- An **Input Form** connected to another **Input Form** with a **Sync Wire** will create an **Input Text Field** in the target form for every **Input Text Field** in the source form with the same **name**.
- When an **Input Text Field** is connected to a **Contains Operations** by an **instant Set Wire**, the source element will **call** the 'update' **Operation** on the target when the source input **changes**.
- When an **Input Text Field** is connected to a **Can Be Synced** element by an **instant Set Wire**, the **value** of the source element will be provided as a **Parameter** to the updating **ECA Rule**.
- A **login Frame** for a **Login Handler** of type **DOMAIN\_OBJECT** will contain an **Input Text Field** for every **Domain Attribute** defined as a **Parameter** to that **Login Handler**.

- A *login* **Frame** for a **Login Handler** of type **USER** will contain an **Input Text Field** for every non-primary key **Domain Attribute** within the **Role** defined as a **Parameter** to that **Login Handler**.
- **Input Text Fields** have an **Value** named **current input**.
- **Input Text Fields** will contain a **Label** named 'Warning'.
- Two differently-typed **Input Text Fields** connected via a **Sync Wire** will define an **Operation** within the first field named 'validate'.
- The 'validate' **Operation** defined to validate the type instances of two differently-typed **Input Text Fields** connected via a **Sync Wire** will be **executed** when one of the fields **change**.
- An **Activity Operation** named 'validate', used to check the validity of two type instances of two **Input Text Fields** connected via a **Sync Wire**, will be completed with the necessary **Activity Nodes** to implement this functionality.

## I.60 Internet Application

A **Scope** which represents the entire web application, and is the root element of all IAML model instances.

**Direct supertypes** **Contains Functions**, **Contains Operations**, **Contains Values**, **Contains Wires**, **Generates Elements**, **Named Element**, **Scope**

An IAML model instance must contain a single instance of an **Internet Application**. The *onInit* **Event** for an **Internet Application** is only executed once over the lifetime of the deployed application, and will not be triggered even if the application host is restarted.

The *storage semantics* for a **Internet Application** are as follows: For a **Value** directly contained within a **Internet Application**, any stored value is accessible globally; that is, across all clients, all browsers, all sessions and all requests over the lifetime of the deployed application. Similarly, a **Domain Iterator** is also accessible globally.

### Attributes

- **runtimeUrl** : string

The public root URL of the modelled web application.

### Containment References

- **sources** : [0..\*] **Domain Source**

A list of contained **Domain Sources**, representing the different sources of **Domain Types** in the web application.

- **types** : [0..\*] **Domain Type**

A list of contained **Domain Types**.

- **xsd data types** : [0..\*] **EXSD Data Type**

A list of contained **EXSD Data Types**, acting as bridges between EMF and XSD.

### Inherited Events

**onAccess** (from **Accessible**); **onInit** (from **Scope**)

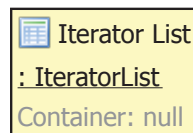
## Model Completion Rules

- An [Internet Application](#) will contain a [read only Value](#), named 'blank', which represents an empty string.

## I.61 Iterator List

When connected to a [Domain Iterator](#), an [Iterator List](#) can illustrate any number of [Domain Instances](#) as a list.

**Direct supertypes** [Visible Thing](#)



An [Iterator List](#) represents a list of elements retrieved from a particular [Domain Iterator](#). An [Iterator List](#) must be connected to a [Domain Iterator](#) via either a [Set Wire](#) or [Sync Wire](#); this iterator becomes the *connected iterator*. An [Iterator List](#) may accept and lose focus. The *onInput* Event can never be triggered for an [Iterator List](#).

When an [Iterator List](#) is rendered, the number of results from the *connected iterator* is obtained; all of the children directly contained in this [Iterator List](#) will be rendered as many times. These elements will be rendered in a table or grid style, with the [name](#) of each child element used as a heading.

### Inherited Events

[onAccess](#) (from [Accessible](#)); [onChange](#) (from [Changeable](#)); [onClick](#) (from [Visible Thing](#)); [onInput](#) (from [Visible Thing](#))

## Model Completion Rules

- The [Input Form](#) results container generated by an [Autocomplete Wire](#) will be populated with a *results Iterator List* to display matched results in a list, using a [Set Wire](#).
- Every [Label](#) within the *results Iterator List* of an [Autocomplete Wire](#) search will be provided an [ECA Rule](#) to [update](#) a [target](#) element – where the [name](#) of the [Domain Attribute Instance](#) matches the [name](#) of the target [Visible Thing](#) – when the [Label](#) is [clicked](#).

## I.62 Join Node

Joins multiple [split](#) execution threads back together. Halts until all threads are complete.

**Direct supertypes** [Activity Node](#), [Execution Edge Destination](#), [Execution Edges Source](#)



An **Activity Operation** may contain any number of **Join Nodes**. Each **Join Node** must have at least one incoming **Execution Edge**, and at least one outgoing **Execution Edge**.

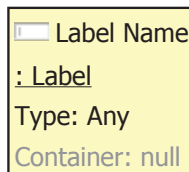
If an executing thread arrives at a **Join Node**, the thread will be removed from the associated *thread set* of that thread. If the *thread set* is not empty, execution of the thread will be suspended until such a point that the *thread set* is empty. If the *thread set* is empty, then the multithreaded execution originating from the original **Split Node** will be considered complete, and the *execution flow* will continue from the **Split Node** as normal. If an *unthreaded execution flow* arrives at a **Join Node**, an exception must occur.

The **Join Node** element is based off the UML *JoinNode* model element.

## I.63 Label

A static block of text that is not user-editable.

**Direct supertypes** Requires Edge Destination, Visible Thing



### Inherited Events

onAccess (from Accessible); onChange (from Changeable); onClick (from Visible Thing); onInput (from Visible Thing)

### Model Completion Rules

- An entry **Gate** can **require** a given **Label** to be set, creating a **Frame** that must be filled in before the containing **Scope** can be accessed.
- An **Input Form** connected to another **Input Form** with a **Sync Wire** will create a **Label** in the target form for every **Label** in the source form with the same **name**.
- An **Input Form** connected to another **Input Form** with a **Set Wire** will create a **Label** in the target form for every **Input Text Field** in the source form with the same **name**.
- An **Input Form** connected to another **Input Form** with a **Set Wire** will create a **Label** in the target form for every **Label** in the source form with the same **name**.
- When a **Domain Iterator** is connected to an **Input Form** by a **Set Wire**, **Labels** will be generated for its non-generated, non-primary key **Domain Attribute Instances**.
- **Input Text Fields** will contain a **Label** named 'Warning'.

## I.64 Login Handler

Within a [Scope](#), enforces that all user access must be authenticated through a particular login method.

**Direct supertypes** [Action Edge Source](#), [Contains Functions](#), [Contains Operations](#), [Contains Values](#), [Contains Wires](#), [Generates Elements](#), [Named Element](#), [Parameter Edge Destination](#), [Wireable](#)

■ Handler name  
: [LoginHandler](#)  
Type: <...>

A [Scope](#) may contain at most one [Login Handler](#). All access to elements within this [Scope](#), or its children [Scopes](#), must first *satisfy* the [Login Handler](#). The method in which a [Login Handler](#) may be *satisfied* depends on the *type* of the [Login Handler](#). A [Login Handler](#) must have at least one incoming [Parameter](#) in order to specify the necessary credentials.

A [Login Handler](#) must provide a *login* interface, in which the current user can provide credentials in order to satisfy the [Login Handler](#). A [Login Handler](#) must also provide a *logout* interface, in which the current user can remove credentials used to satisfy the [Login Handler](#). Any credentials supplied by the user to satisfy the [Login Handler](#) are stored according to the *storage semantics* of the protected [Scope](#).

### Secret Key

A *secret key* [Login Handler](#) must have exactly one incoming [Parameter](#) from a value instance element, specifying a particular password. When accessing a [Scope](#) protected by a *secret key* [Login Handler](#), this password must first be provided by the user.

### Domain Object

A *domain object* [Login Handler](#) must have at least one incoming [Parameter](#), each from a [Domain Attribute](#) instance. All incoming [Domain Attributes](#) must belong either directly or indirectly (through inheritance) to a single [Domain Type](#) (the *target schema*).

The *login* interface for this [Login Handler](#) must therefore provide one user interface element per incoming [Domain Attribute](#) in order to provide query values. When accessing a [Scope](#) protected by a *domain object* [Login Handler](#), these query values will be used in order to select a valid instance. A *domain object* [Login Handler](#) will only be satisfied if there exists at least one valid instance for these query values.

A *domain object* [Login Handler](#) may have any number of outgoing [Set Wires](#) to [Domain Iterators](#). When a [Login Handler](#) is satisfied, all of these [Domain Iterators](#) will be populated with the valid [Domain Type](#) instances selected. If this [Login Handler](#) is not satisfied, these [Domain Iterators](#) will be *empty*.

### Attributes

- **type** : [LoginHandlerTypes](#)  
The authentication method of this [Login Handler](#).  
Possible values: SECRET\_KEY, DOMAIN\_OBJECT, USER

### Model Completion Rules

- A **Scope** protected by a **Login Handler** will define a separate *login Session*, allowing session-unique authentication.
- A **Scope** protected by a **Login Handler** will create a *login Frame* named 'login' within the *login Session*.
- The *login Frame* created for a **Login Handler** protecting a **Scope** will be provided as the login rule for the **Login Handler** if none is specified.
- A **Scope** protected by a **Login Handler** will create a *logout Frame* named 'logout' within the same **Scope**.
- If a **Login Handler** has an outgoing navigation rule named 'logout', then this navigation rule will be executed when the *logout Frame* is accessed.
- A **Login Handler** of type **SECRET\_KEY** will define a **Value** named 'current login key' in the protected **Session** as the *stored key*, if no outgoing **Set Wire** named 'set' from the **Login Handler** is defined.
- A **Login Handler** of type **SECRET\_KEY** will connect to a **Value** named 'current login key' as the *stored key* using a **Set Wire** named 'set'.
- A **Login Handler** of type **SECRET\_KEY** will create an **Activity Operation** 'check key' within the protected **Session**.
- A **Frame** contained by a **Session** protected by a **Login Handler** of type **SECRET\_KEY** will execute the 'check key' **Operation** with a high priority when the frame is accessed.
- A **Frame** contained by a **Session** protected by a **Login Handler** of type **SECRET\_KEY** will provide the *stored key* as a **Parameter** to the executed 'check key' **Operation**.
- A **Frame** contained by a **Session** protected by a **Login Handler** of type **SECRET\_KEY** will navigate to the *login Frame* if the 'check key' **Operation** fails.
- An **Activity Operation** named 'check key', used to check the status of a **Login Handler** of type **SECRET\_KEY** against an incoming **Parameter** to the handler, will be completed with the necessary **Activity Nodes** to implement this functionality.
- An **Activity Operation** named 'check instance', used to check the status of a **Login Handler** of type **DOMAIN\_OBJECT** that a **Domain Iterator** is not empty, will be completed with the necessary **Activity Nodes** to implement this functionality.
- An **Activity Operation** named 'check instance', used to check the status of a **Login Handler** of type **USER** that a **Domain Iterator** is not empty, will be completed with the necessary **Activity Nodes** to implement this functionality.
- A *login Frame* for a **Login Handler** will contain an **Input Form** named 'login form'.
- A *login Frame* for a **Login Handler** will contain an **Button** named 'login'.
- The 'login' **Button** in the *login Frame* for a **Login Handler** will execute the 'do login' **Operation** within the **Login Handler** when the button is clicked.
- A *login Frame* for a **Login Handler** of type **SECRET\_KEY** will contain an **Input Text Field** named 'login key'.
- An **Activity Operation** named 'do login', used to actually implement the login operation of a **Login Handler** of type **SECRET\_KEY**, will be completed with the necessary **Activity Nodes** to implement this functionality.
- A *login Frame* for a **Login Handler** of type **DOMAIN\_OBJECT** will contain an **Input Text Field** for every **Domain Attribute** defined as a **Parameter** to that **Login Handler**.
- A *login Frame* for a **Login Handler** of type **USER** will contain an **Input Text Field** for every non-primary **key Domain Attribute** within the **Role** defined as a **Parameter** to that **Login Handler**.
- Every **Input Text Field** defined in the login form for the *login Frame* for a **Login Handler** of type **DOMAIN\_OBJECT** will be provided as a named **Parameter** to the 'do login' **Operation** of the handler.
- Every **Input Text Field** defined in the login form for the *login Frame* for a **Login Handler** of type **USER** will be provided as a named **Parameter** to the 'do login' **Operation** of the handler.

- A **Login Handler** will have an **Operation** named ‘do login’.
- An **Activity Operation** named ‘do login’, used to actually implement the login operation of a **Login Handler** of type **DOMAIN\_KEY**, will be completed with the necessary **Activity Nodes** to implement this functionality.
- Every **Domain Attribute** provided as a **Parameter** to a **Login Handler** of type **DOMAIN\_OBJECT** will be available as a named **Activity Parameter** of the ‘do login’ **Activity Operation**.
- An **Activity Operation** named ‘do login’, used to actually implement the login operation of a **Login Handler** of type **USER**, will be completed with the necessary **Activity Nodes** to implement this functionality.
- Every **Domain Attribute** of the **Role** provided as a **Parameter** to a **Login Handler** of type **USER** will be available as a named **Activity Parameter** of the ‘do login’ **Activity Operation**.
- If a **Login Handler** has a defined outgoing **ECA Rule** named ‘success’, then this rule should be executed after a successful ‘do login’ **Activity Operation**.
- If a **Login Handler** has a defined outgoing **ECA Rule** named ‘fail’, then this rule should be executed after a failed ‘do login’ **Activity Operation**.
- A **Login Handler** will define an **Operation** named ‘do logout’ within the protected **Scope**.
- When the *logout* **Frame** for a **Login Handler** is **accessed**, the ‘do logout’ **Operation** will be **called**.
- If a **Login Handler** has a defined outgoing **ECA Rule** named ‘logout’, then this rule should be executed after a successful ‘do logout’ **Activity Operation**.
- An **Activity Operation** named ‘do logout’, used to actually implement the logout operation of a **Login Handler** of type **SECRET\_KEY**, will be completed with the necessary **Activity Nodes** to implement this functionality.
- An **Activity Operation** named ‘do logout’, used to actually implement the logout operation of a **Login Handler** of type **DOMAIN\_OBJECT**, will be completed with the necessary **Activity Nodes** to implement this functionality.
- An **Activity Operation** named ‘do logout’, used to actually implement the logout operation of a **Login Handler** of type **USER**, will be completed with the necessary **Activity Nodes** to implement this functionality.
- An **Activity Operation** named ‘do logout’ of a **Login Handler** of type **DOMAIN\_OBJECT** will reset all of the stored keys for each **Domain Attribute** used as a **Parameter** to the handler to an empty **Value**.
- An **Activity Operation** named ‘do logout’ of a **Login Handler** of type **USER** will reset all of the stored keys for each **Domain Attribute** of the **Role** used as a **Parameter** to the handler to an empty **Value**.
- If no ‘success’ **ECA Rule** is defined for a **Login Handler**, then a default ‘login successful’ **Frame** will be created.
- If no ‘success’ **ECA Rule** is defined for a **Login Handler**, then the ‘login successful’ **Frame** will be defined as the ‘success’ **ECA Rule**.
- If no ‘logout’ **ECA Rule** is defined for a **Login Handler**, then a default ‘logout successful’ **Frame** will be created.
- If no ‘logout’ **ECA Rule** is defined for a **Login Handler**, then the ‘logout successful’ **Frame** will be defined as the ‘logout’ **ECA Rule**.
- The **query** of a **Domain Iterator** used to set a **Login Handler** of type **DOMAIN\_OBJECT** will be updated according to the **Domain Type** of the handler.
- The **query** of a **Domain Iterator** used to set a **Login Handler** of type **USER** will be set according to the **Domain Attributes** provided as **Parameters** to the handler.
- The **query** of a **Domain Iterator** used to set a **Login Handler** of type **USER** will be *updated* according to the **Domain Attributes** provided as **Parameters** to the handler.
- The **query** of a **Domain Iterator** used to set a **Login Handler** of type **USER** will be *updated* according to the **Role** of the handler.



- The [query](#) of a [Domain Iterator](#) used to [set](#) a [Login Handler](#) of [type](#) [USER](#) will be *updated* according to the [Role](#) of the handler.
- A [Login Handler](#) of [type](#) [DOMAIN\\_OBJECT](#) will define a [Value](#) named 'current XXX' in the protected [Session](#) for each [Domain Attribute](#) used as a [Parameter](#) to the handler.
- A [Login Handler](#) of [type](#) [USER](#) will define a [Value](#) named 'current XXX' in the protected [Session](#) for each [Domain Attribute](#) used as a [Parameter](#) to the handler.
- A [Login Handler](#) of [type](#) [DOMAIN\\_OBJECT](#) will connect each [Value](#) named 'current XXX' using [Set Wires](#) named 'set'.
- Each [Value](#) 'current XXX' [set](#) by a [Login Handler](#) of [type](#) [DOMAIN\\_OBJECT](#) will be provided as a [Parameter](#) to the handler.
- A [Login Handler](#) of [type](#) [USER](#) will connect each [Value](#) named 'current XXX' using [Set Wires](#) named 'set'.
- A [Login Handler](#) of [type](#) [DOMAIN\\_OBJECT](#) will create an [Activity Operation](#) 'check instance' within the protected [Session](#).
- A [Login Handler](#) of [type](#) [USER](#) will create an [Activity Operation](#) 'check instance' within the protected [Session](#).
- A [Session](#) protected by a [Login Handler](#) of [type](#) [DOMAIN\\_OBJECT](#) will contain a [Domain Iterator](#) 'current instance' to represent the current domain object instance.
- A [Domain Iterator](#) 'current instance' created for a [Login Handler](#) of [type](#) [DOMAIN\\_OBJECT](#) will be provided as the [set target](#) of the handler.
- A [Session](#) protected by a [Login Handler](#) of [type](#) [USER](#) will contain a [Domain Iterator](#) 'current instance' to represent the current domain object instance.
- A [Domain Iterator](#) 'current instance' created for a [Login Handler](#) of [type](#) [USER](#) will be provided as the [set target](#) of the handler.
- A [Domain Iterator](#) within a [Session](#) protected by a [Login Handler](#) of [type](#) [DOMAIN\\_OBJECT](#) without a specified [Select Edge](#) will use the same [Domain Source](#) as the [Domain Type](#) that defines the [Domain Attribute](#) used as a [Parameter](#) to that handler.
- A [Domain Iterator](#) within a [Session](#) protected by a [Login Handler](#) of [type](#) [USER](#) without a specified [Select Edge](#) will use the same [Domain Source](#) as the [Domain Type](#) that defines the [Domain Attribute](#) used as a [Parameter](#) to that handler.
- A [Domain Iterator](#) within a [Session](#) protected by a [Login Handler](#) of [type](#) [DOMAIN\\_OBJECT](#) or [USER](#) without a specified [Select Edge](#) will use the same [Domain Source](#) as the [Domain Type](#) used as a [Parameter](#) to that handler.
- A [Domain Iterator](#) within a [Session](#) protected by both an [Access Control Handler](#) and [Login Handler](#) of [type](#) [DOMAIN\\_OBJECT](#) or [USER](#), but the handlers both use different [Domain Types](#) (for example, with [inheritance](#)), and which does not specify a [Select Edge](#), will use the same [Domain Source](#) as the [Domain Type](#) used as a [Parameter](#) to the [Access Control Handler](#).
- A [Domain Iterator](#) within a [Session](#) protected by both an [Access Control Handler](#) and [Login Handler](#) of [type](#) [DOMAIN\\_OBJECT](#) or [USER](#), but the handlers both use the same [Domain Types](#), and which does not specify a [Select Edge](#), will use the same [Domain Source](#) as the [Domain Type](#) used as a [Parameter](#) to the [Access Control Handler](#).
- A [Domain Iterator](#) within a [Session](#) protected by both an [Access Control Handler](#) and [Login Handler](#) of [type](#) [DOMAIN\\_OBJECT](#) or [USER](#), but the [Access Control Handler](#) requires a [Permission](#) without a [Role](#), and which does not specify a [Select Edge](#), will use the same [Domain Source](#) as the [Domain Type](#) used as a [Parameter](#) to the [Access Control Handler](#).
- A [Login Handler](#) is generated for a [Session](#) that contains an [Access Control Handler](#).
- A [Login Handler](#) is generated for a [Frame](#) contained within a [Session](#) that contains an [Access Control Handler](#).

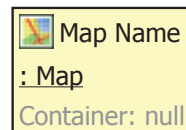


- An [Access Control Handler](#) protecting a [Session](#) with a defined [Role](#)-based access [requirement](#) will provide the *default Role* as a [Parameter](#) to the created [Login Handler](#).
- The [Domain Iterator](#) representing the current instance of an [Access Control Handler](#) check, protecting a [Session](#), will be provided as a [Parameter](#) to the generated [Login Handler](#).
- The [Domain Iterator](#) representing the current instance of an [Access Control Handler](#) check, protecting a [Frame](#) within a [Session](#), will be provided as a [Parameter](#) to the generated [Login Handler](#).

## I.65 Map

A geographical area represented as an interactive map. This [Map](#) may contain many separate [Map Points](#).

**Direct supertypes** [Visible Thing](#)



A [Map](#) represents a geographical map location that should be interactive – that is, the map can be navigated from within the browser. A [Map](#) may accept and lose focus. The [onInput Event](#) can never be triggered for a [Map](#). A [Map](#) may contain any number of [Map Points](#), all which will be rendered within the current [Map](#) as separate points.

The [fieldValue](#) of the [Map](#) is used as the location for the rendered map. If the [fieldValue](#) changes, the location of the map will be refreshed to the new location. If the [fieldValue](#) of the [Map](#) is empty, the map will revert to a default location. When the current location of the rendered map changes, the [fieldValue](#) of the [Map](#) should also change, and the [onChange Event](#) will be triggered<sup>8</sup>.

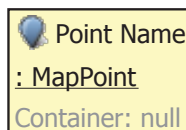
### Inherited Events

[onAccess](#) (from [Accessible](#)); [onChange](#) (from [Changeable](#)); [onClick](#) (from [Visible Thing](#)); [onInput](#) (from [Visible Thing](#))

## I.66 Map Point

A single geographical point, as opposed to a location represented by a [Map](#). A [Map Point](#) does not need to be contained within a [Map](#).

**Direct supertypes** [Visible Thing](#)



<sup>8</sup>Issue 225: *Map cannot track changes to its field value.*

A **Map Point** represents a single geographical point. A **Map Point** may accept and lose focus. The *onInput Event* can never be triggered for a **Map Point**. A **Map Point** does not need to be contained within a **Map** element. When a **Map Point** is rendered, all of the contained **children** must also be rendered.

The **fieldValue** of the **Map Point** is used as the location of the current point. If the **fieldValue** changes, the rendered map point will be refreshed to the new point. If the **fieldValue** of the **Map Point** is empty, the map point will not be displayed. If the **Map Point** is moved to a different location, the **fieldValue** of the **Map Point** should also change, and the *onChange Event* will be triggered<sup>9</sup>.

### Inherited Events

*onAccess* (from **Accessible**); *onChange* (from **Changeable**); *onClick* (from **Visible Thing**); *onInput* (from **Visible Thing**)

## I.67 Message

Encapsulated content which may be sent from a source to a recipient.

**Abstract type.**

**Direct supertypes** **Generates Elements**, **Named Element**

**Direct subtypes** **Email**

## I.68 Named Element

An element with a name.

**Interface.**

**Direct supertypes** **Generated Element**

**Direct subtypes** **Access Control Handler**, **Activity Parameter**, **Can Be Synced**, **Decision Node**, **Domain Instance**, **Domain Iterator**, **Domain Source**, **ECA Rule**, **Event**, **Execution Edge**, **Function**, **Gate**, **Internet Application**, **Login Handler**, **Message**, **Operation**, **Operation Call Node**, **Parameter**, **Parameter Value**, **Permission**, **Query Parameter**, **Scope**, **Session**, **Set Node**, **Temporary Variable**, **Visible Thing**, **Wire**

**Indirect subtypes** **Activity Operation**, **Activity Predicate**, **Autocomplete Wire**, **Boolean Property**, **Builtin Operation**, **Builtin Property**, **Button**, **Complex Term**, **Detail Wire**, **Domain Attribute Instance**, **Domain Feature Instance**, **Email**, **Frame**, **Input Form**, **Input Text Field**, **Iterator List**, **Label**, **Map**, **Map Point**, **Predicate**, **Set Wire**, **Simple Condition**, **Sync Wire**, **Value**, **XQuery Function**, **XQuery Predicate**

<sup>9</sup>Issue 225: *Map cannot track changes to its field value.*

**Attributes**

- `name`: string  
A non-unique string that names a given object.

**I.69 Operation**

A reusable block of functionality with respect to its parent `container`, which may be `predefined` or `user-defined`.

**Abstract type.**

**Direct supertypes** `Action`, `Contains Wires`, `Named Element`

**Direct subtypes** `Builtin Operation`

**Indirect subtypes** `Activity Operation`

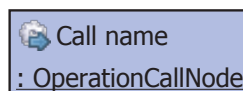
**Model Completion Rules**

- All non- `read only Values`, other than those `named` ‘fieldValue’, will have an `Builtin Operation` in its container named ‘set XXX’.
- All `Visible Things` (except `Iterator Lists` and `Input Forms`) will contain an `Operation` named “update”.
- All `Domain Attribute Instances` will contain an `Operation` named “update”.
- An `Operation` ‘set value XXX’ is created in the container of all `Values` that are the target of `Set Wires`.

**I.70 Operation Call Node**

A virtual `Event`, which permits an outgoing `ECA Rule` to be triggered.

**Direct supertypes** `Action Edge Source`, `Activity Node`, `Execution Edge Destination`, `Execution Edges Source`, `Named Element`



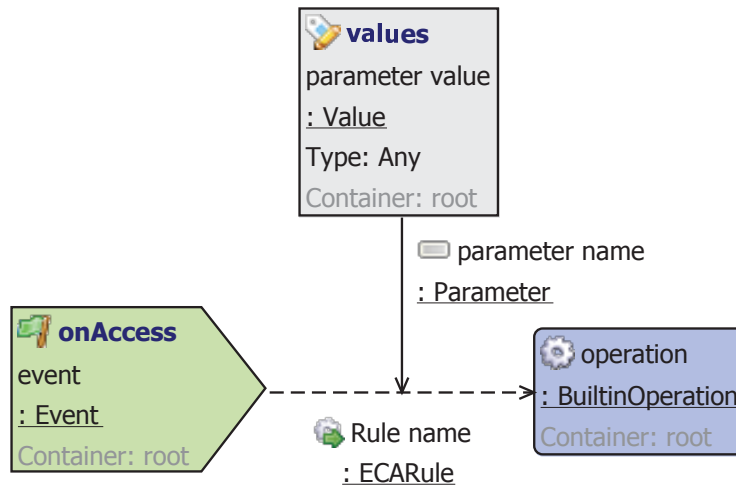
An `Activity Operation` may contain any number of `Operation Call Nodes`. Each `Operation Call Node` must have at least one outgoing `ECA Rule`. When the *execution flow* arrives at an `Operation Call Node`, each outgoing `ECA Rule` will be executed sequentially, according to the *priority* of each edge.

If the last `ECA Rule` executed from the `Operation Call Node` *fails*, the execution flow will follow all outgoing *failure edges* sequentially. Otherwise, the execution flow will follow all outgoing *success edges* sequentially.

**I.71 Parameter**

Associates a **Parameter Value** to a named slot of a **Function**, or as parameters ( **Query Parameters** or **Activity Parameters**) to an **Action**.

**Direct supertypes** Generated Element, Named Element



## References

- parameter term : Parameter Edge Destination (opposite: in parameter edges)  
The target of the **Parameter**.
- parameter value : Parameter Edges Source (opposite: out parameter edges)  
The source value of the **Parameter**.

## Model Completion Rules

- A **Sync Wire** connecting two elements with a **Function** 'can sync?' will only **permit** synchronisation if the value can be cast **successfully**.
- For elements connected by a **Sync Wire**, the 'onAccess' **Event** executing the 'init' **Operation** will only **execute** if the source field **Value** can be cast **successfully**.
- When a **Can Be Synced** element is connected to an **Can Be Synced** element by a **Sync Wire**, the **value** of the source element will be provided as a **Parameter** to the updating **ECA Rule**.
- When an **Input Text Field** is connected to a **Can Be Synced** element by an **instant Sync Wire**, the **value** of the source element will be provided as a **Parameter** to the updating **ECA Rule**.
- When a **Can Be Synced** element is connected to a **Can Be Synced** element by a **Sync Wire**, the **field value** of the source element will be used as a **Parameter** to initialise the target through the 'init' **Operation**.
- Every 'init' **Operation** executed when a **Session** is **initialised** due to a **Sync Wire** between a **Visible Thing** within that session, to another **element** outside of that session, will be provided the **field value** of the source element as a **Parameter**.
- Every 'init' **Operation** executed when a **Session** is **initialised** due to a **Sync Wire** between a **Frame** within that session, to another **element** outside of that session, will be provided the **field value** of the source element as a **Parameter**.

- When a **Can Be Synced** element is connected to an **Can Be Synced** element by a **Set Wire**, the **value** of the source element will be provided as a **Parameter** to the updating **ECA Rule**.
- When a **Changeable** element is connected to an **Can Be Synced** element by a **Set Wire**, the **value** of the source element will be provided as a **Parameter** to the updating **ECA Rule**.
- When a **Changeable** element is connected to a **Value** by a **Set Wire**, the **value** of the source element will be provided as a **Parameter** to the updating **ECA Rule**.
- When an **Input Text Field** is connected to a **Can Be Synced** element by an **instant Set Wire**, the **value** of the source element will be provided as a **Parameter** to the updating **ECA Rule**.
- When a **Can Be Synced** element is connected to an **Accessible** element by a **Set Wire**, the **field value** of the source element will be used as a **Parameter** to initialise the target through the ‘init’ **Operation**.
- Every ‘init’ **Operation** executed when a **Session** is initialised due to a **Set Wire** between a **Visible Thing** within that session, to another **element** outside of that session, will be provided the **field value** of the source element as a **Parameter**.
- Every ‘init’ **Operation** executed when a **Session** is initialised due to a **Set Wire** between a **Frame** within that session, to another **element** outside of that session, will be provided the **field value** of the source element as a **Parameter**.
- When a **Detail Wire** is connected from a **Frame** that contains a **Domain Iterator** to a **target Frame**, a new **selected Domain Iterator** will be used to select content based on primary key.
- The navigation **rule** from the **link Button** generated for a **Detail Wire**, not contained by an **Iterator List**, will be supplied the current **value** within the source **Domain Instance**, representing the **primary key** of the source **Domain Type**.
- The navigation **rule** from the **link Button** generated for a **Detail Wire**, as contained by an **Iterator List**, will be supplied the current **value** within the source **Domain Instance**, representing the **primary key** of the source **Domain Type**.
- The **query Input Text Field** created by an **Autocomplete Wire** will be connected as a **Parameter** to the connected **Domain Iterator**.
- The **query Input Text Field** created by an **Autocomplete Wire** will **update** the field whenever it receives **input**.
- Every **Label** within the **results Iterator List** of an **Autocomplete Wire** search will be provided an **ECA Rule** to **update** a **target** element – where the **name** of the **Domain Attribute Instance** matches the **name** of the target **Visible Thing** – when the **Label** is **clicked**.
- When a matching **Label** result is clicked from an **Autocomplete Wire** search, the **query Input Text Field** will be cleared to a blank **Value**.
- A **Frame** contained by a **Session** protected by a **Login Handler** of type **SECRET\_KEY** will provide the **stored key** as a **Parameter** to the executed ‘check key’ **Operation**.
- A **Frame** contained by a **Session** protected by a **Login Handler** of type **SECRET\_KEY** will **navigate** to the **login Frame** if the ‘check key’ **Operation** fails.
- A **login Frame** for a **Login Handler** of type **DOMAIN\_OBJECT** will contain an **Input Text Field** for every **Domain Attribute** defined as a **Parameter** to that **Login Handler**.
- A **login Frame** for a **Login Handler** of type **USER** will contain an **Input Text Field** for every non-primary **key Domain Attribute** within the **Role** defined as a **Parameter** to that **Login Handler**.
- Every **Input Text Field** defined in the login form for the **login Frame** for a **Login Handler** of type **DOMAIN\_OBJECT** will be provided as a named **Parameter** to the ‘do login’ **Operation** of the handler.
- Every **Input Text Field** defined in the login form for the **login Frame** for a **Login Handler** of type **USER** will be provided as a named **Parameter** to the ‘do login’ **Operation** of the handler.
- Each **Value** ‘current XXX’ set by a **Login Handler** of type **DOMAIN\_OBJECT** will be provided as a **Parameter** to the handler.

- The ‘target’ [Value](#) defined in a ‘last’ [Button](#) defined for navigating a [Domain Iterator](#), as defined by a [Set Wire](#), will be used as a [Parameter](#) for executing the [jump Operation](#) on that iterator.
- The current results [count](#) of the [Domain Iterator](#) will be used as a [Parameter](#) to update the ‘results’ [Label](#) defined for navigating a [Domain Iterator](#) with a [Set Wire](#) when the iterator [changes](#).
- The current results [count](#) of the [Domain Iterator](#) will be used as a [Parameter](#) to update the ‘results’ [Label](#) defined for navigating a [Domain Iterator](#) with a [Set Wire](#) when the containing [Input Form](#) is [accessed](#).
- All [Parameters](#) for each [conditions](#) against a [Sync Wire](#) will be copied to each [ECA Rule](#) subsequently created as connected to the [onChange Events](#).
- All [Parameters](#) for each [conditions](#) against a [Sync Wire](#) will be copied to each [ECA Rule](#) subsequently created as connected to the [onAccess Events](#).
- All [Parameters](#) for each [conditions](#) against a [Set Wire](#) will be copied to each [Simple Condition](#) subsequently created for [ECA Rules](#) connected to the [onChange Events](#).
- All [Parameters](#) for each [conditions](#) against a [Set Wire](#) will be copied to each [Simple Condition](#) subsequently created for [ECA Rules](#) connected to the [onAccess Events](#).
- An [Access Control Handler](#) protecting a [Session](#) with a defined [Role-based access requirement](#) will provide the *default Role* as a [Parameter](#) to the created [Login Handler](#).
- The [Domain Iterator](#) representing the current instance of an [Access Control Handler](#) check, protecting a [Session](#), will be provided as a [Parameter](#) to the generated [Login Handler](#).
- The [Domain Iterator](#) representing the current instance of an [Access Control Handler](#) check, protecting a [Frame](#) within a [Session](#), will be provided as a [Parameter](#) to the generated [Login Handler](#).

## I.72 Parameter Edge Destination

An element which may be used as the target of a [Parameter](#).

### Interface.

**Direct subtypes** [Access Control Handler](#), [Complex Term](#), [Domain Iterator](#), [ECA Rule](#), [Login Handler](#), [Wire](#)

**Indirect subtypes** [Autocomplete Wire](#), [Detail Wire](#), [Set Wire](#), [Simple Condition](#), [Sync Wire](#)

This abstract interface is provided to simplify the implementation of [Parameters](#), as EMF does not support type unions in metamodel definitions.

### References

- in [parameter edges](#) : [0..\*] [Parameter](#) (opposite: [parameter term](#))  
A list of incoming [Parameters](#).

## I.73 Parameter Edges Source

An element which may be used as the source of a [Parameter](#).

### Interface.

**Direct subtypes** Activity Parameter, Changeable, Complex Term, Domain Feature, Domain Type, External Value, Frame, Permission, Query Parameter, Temporary Variable, Value

**Indirect subtypes** Button, Domain Attribute, Domain Attribute Instance, Domain Feature Instance, Domain Iterator, Input Form, Input Text Field, Iterator List, Label, Map, Map Point, Role, Simple Condition, Visible Thing

This abstract interface is provided to simplify the implementation of [Parameters](#), as EMF does not support type unions in metamodel definitions.

#### References

- **out parameter edges** : [0..\*] [Parameter](#) (opposite: [parameter value](#))  
A list of outgoing [Parameters](#).

## I.74 Parameter Value

Abstractly defines an instance of a value with a particular [type](#).

**Abstract type.**

**Direct supertypes** [Generates Elements](#), [Named Element](#)

**Direct subtypes** [Complex Term](#), [Value](#)

**Indirect subtypes** [Simple Condition](#)

If the [type](#) of a typed value is given a null or empty URI, then the data type of the value is the default type.

#### References

- **type** : [0..1] [EClassifier](#) (from [Ecore](#))  
The runtime type of this [Parameter Value](#).

## I.75 Permission

A specific permission, which may or may not be associated with a [Role](#).

**Direct supertypes** [Named Element](#), [Parameter Edges Source](#), [Provides Edge Destination](#), [Requires Edge Destination](#)



A [Permission](#) may be *provided* by any number of [Roles](#), through the definition of [Provides Edge](#) instances.

## I.76 Predicate

A **Function** that is typed **boolean**.

**Abstract type.**

**Direct supertypes** [Function](#)

**Direct subtypes** [Activity Predicate](#), [Boolean Property](#), [XQuery Predicate](#)

**Indirect subtypes** [Builtin Property](#)

## I.77 Provides Edge

Indicates that the source [Role](#) or [Domain Iterator](#) provides the target [Permission](#).

**Direct supertypes** [Generated Element](#)



### References

- **from** : [Provides Edges Source](#) (opposite: [out provides edges](#))  
The source of this [Provides Edge](#) relationship.
- **to** : [Provides Edge Destination](#) (opposite: [in provides edges](#))  
The target of this [Provides Edge](#) relationship.

## I.78 Provides Edge Destination

An element which may be used as the destination of a [Provides Edge](#).

**Interface.**

**Direct subtypes** [Domain Iterator](#), [Permission](#)

This abstract interface is provided to simplify the implementation of [Provides Edges](#), as EMF does not support type unions in metamodel definitions. This model element may be removed from the IAML metamodel in the future<sup>10</sup>.

<sup>10</sup>Issue 278: *Remove ProvidesEdgesSource and ProvidesEdgeDestination abstract supertypes.*



**References**

- **in provides edges** : [0..\*] Provides Edge (opposite: to)  
A list of incoming Provides Edges.

**I.79 Provides Edges Source**

An element which may be used as the source of a Provides Edge.

**Interface.**

**Direct subtypes** Domain Iterator, Role

This abstract interface is provided to simplify the implementation of Provides Edges, as EMF does not support type unions in metamodel definitions. This model element may be removed from the IAML metamodel in the future<sup>11</sup>.

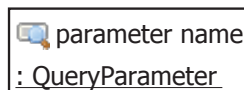
**References**

- **out provides edges** : [0..\*] Provides Edge (opposite: from)  
A list of outgoing Provides Edges.

**I.80 Query Parameter**

Runtime HTTP GET parameters of the currently requested URL.

**Direct supertypes** Contains Wires, Named Element, Parameter Edges Source



Query Parameters can only be used from within a containing Frame, although this Frame can be within a containment hierarchy of Scopes.

**Attributes**

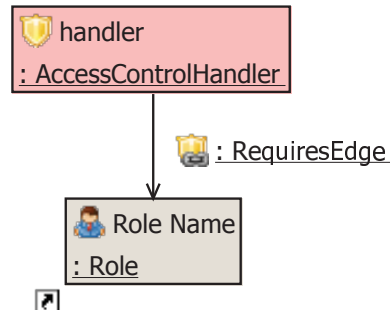
- **defaultValue** : string  
If set, provides a default value for this Query Parameter. This will prevent the application failing if the parameter has not been provided.

**I.81 Requires Edge**

<sup>11</sup>Issue 278: Remove ProvidesEdgesSource and ProvidesEdgeDestination abstract supertypes.

An access requirement for an [Access Control Handler](#), or a permission requirement for a [Role](#).

**Direct supertypes** [Constraint Edge Destination](#), [Constraint Edges Source](#), [Contains Wires](#), [Generated Element](#)



## References

- **from** : [Requires Edges Source](#) (opposite: [out requires edges](#))  
The source, or requiring party, of this [Requires Edge](#) relationship.
- **to** : [Requires Edge Destination](#) (opposite: [in requires edges](#))  
The target, or required element, of this [Requires Edge](#) relationship.

## Model Completion Rules

- An entry [Gate](#) can [require](#) a given [Label](#) to be set, creating a [Frame](#) that must be filled in before the containing [Scope](#) can be accessed.
- A [Gate](#) protecting a [Scope](#) as an entry [gate](#) with an incoming [Requires Edge](#) from an *expected input value* (represented as a [Label](#)) – known as an *input entry Gate* – will create an *input Frame* to allow the user to enter in these expected values.
- An *input entry Gate* will create an outgoing navigation [rule](#) from the [Gate](#) named ‘fail’ to the *input Frame*.
- An *input Frame* generated by an *input entry Gate* will contain an [Input Text Field](#) for every [Label](#) required by that [Gate](#).

## I.82 Requires Edge Destination

An element which may be used as the destination of a [Requires Edge](#).

**Interface.**

**Direct subtypes** [Label](#), [Permission](#), [Role](#)

**References**

- **in requires edges** : [0..\*] Requires Edge (opposite: to)  
A list of incoming Requires Edges.

**I.83 Requires Edges Source**

An element which may be used as the source of a Requires Edge.

**Interface.**

**Direct subtypes** Access Control Handler, Domain Type, Gate

**Indirect subtypes** Role

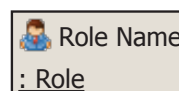
**References**

- **out requires edges** : [0..\*] Requires Edge (opposite: from)  
A list of outgoing Requires Edges.

**I.84 Role**

A specific role of a user within a web application using Role-Based Access Control, providing a particular set of Permissions.

**Direct supertypes** Domain Type, Provides Edges Source, Requires Edge Destination



Similarly to the inheritance hierarchy of Domain Type, a Role may inherit another Role in order to define a *role hierarchy*; this is provided by making a Role a subtype of Domain Type. That is, a child Role that extends a parent Role through an Extends Edge therefore will provide all Permissions that the parent Role provides.

A Role may also define role-specific attributes by defining Domain Attributes according to the semantics of Domain Type. These Domain Attributes subsequently become the *profile attributes* of the given Role.

An IAML model instance will have one Role with the **name** of “User”, representing the root Role type in an application, and this constraint may be satisfied through model completion. All Role instances in a model instance must inherit this Role. As the *default role*, this “User” role will contain a Domain Attribute with a **name** of “email” and a Domain Attribute with a **name** of “password”, representing the e-mail address and user password of a user’s profile respectively.

**Model Completion Rules**

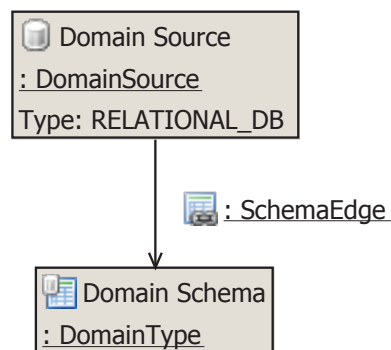
- A *login Frame* for a Login Handler of type USER will contain an Input Text Field for every non-primary key Domain Attribute within the Role defined as a Parameter to that Login Handler.
- A *default Role* named ‘User’ is defined for all Internet Applications.

- The *default Role* will contain an *Domain Attribute* named ‘email’.
- The *default Role* will contain an *Domain Attribute* named ‘password’.
- All *Roles* that do not *extend* another *Role* will *extend* the *default Role*.
- A *Role* *extending* the *default Role* will be provided the same *Domain Source*.

## I.85 Schema Edge

Defines the possible *sources* of data for a particular *Domain Type*.

**Direct supertypes** *Generated Element*, *Generates Elements*



### References

- *from* : [0..1] *Domain Source* (opposite: out schemas)  
The source *Domain Source*.
- *to* : [0..1] *Domain Type* (opposite: in schemas)  
The target *Domain Type*.

## I.86 Scope

A particular runtime scope of a web application, which permits lifecycle modelling. Scopes may contain other *Scopes* through a containment *hierarchy*.

**Abstract type.**

**Direct supertypes** *Accessible*, *Action Edge Source*, *Can Be Synced*, *Contains Functions*, *Contains Operations*, *Contains Values*, *Contains Wires*, *Generated Element*, *Generates Elements*, *Named Element*, *Wireable*

**Direct subtypes** *Frame*, *Internet Application*, *Session*

All scopes in IAML are instances of *Scope*, and a *Scope* may contain another *Scope* through a hierarchy of containment. A *Scope* may directly contain any number of *Scopes*. Every *Scope* will contain the pre-render lifecycle *Events* *onAccess* (through *Accessible*) and *onInit*.

When a [Scope](#) is rendered, the lifecycle events for that [Scope](#) must be executed in the following order. If a failure occurs during this process, the current lifecycle execution process is cancelled and the *failure handler* for the current [Scope](#) is executed. If an [ECA Rule](#) is executed which must redirect the user to another [Frame](#), the current lifecycle execution process is cancelled.

1. If this [Scope](#) is contained by a parent [Scope](#), then the pre-render lifecycle events for that parent [Scope](#) must be triggered.
2. If this [Scope](#) has not yet been *initialised* according to the storage semantics of that [Scope](#), then the *onInit Event* is triggered.
3. The *onAccess Event* is triggered.
4. All entry [Gates](#) contained within this [Scope](#) are sorted according to their [name](#). The *access semantics* of each entry [Gate](#) are then executed.
5. All exit [Gates](#) contained within this [Scope](#) are sorted according to their [name](#). The *access semantics* of each exit [Gate](#) are then executed.
6. The [Scope](#) is then rendered.

### Failure Handler

An outgoing [ECA Rule](#) from a particular [Scope](#) with the [name](#) "fail" is defined as a *failure handler* for the given scope. If an error occurs during the execution of the given [Scope](#) – for example, an [Operation](#) fails – then the failure handler for that [Scope](#) is executed according to the execution semantics of the [ECA Rule](#).

If the current [Scope](#) does not define a *failure handler*, then the *failure handler* of the containing [Scope](#) is used as the failure handler. If an exception is not caught through the containment hierarchy of [Scopes](#), then the exception is caught by the runtime environment and the web application will display a runtime error.

### Containment References

- [access handlers](#) : [0..\*] [Access Control Handler](#)  
A list of contained [Access Control Handlers](#).
- [entry gate](#) : [0..1] [Gate](#)  
A user accessing this [Scope](#) must first satisfy this [Gate](#) before its contents can be accessed.
- [exit gate](#) : [0..1] [Gate](#)  
Once this [Scope](#) has been accessed, a user must satisfy this [Gate](#) before the user may leave a [Frame](#) within this scope.
- [iterators](#) : [0..\*] [Domain Iterator](#)  
A list of contained [Domain Iterators](#).
- [login handlers](#) : [0..\*] [Login Handler](#)  
A list of contained [Login Handlers](#).
- [messages](#) : [0..\*] [Message](#)  
A list of contained [Messages](#).
- [parameters](#) : [0..\*] [Query Parameter](#)  
Runtime HTTP GET parameters of the current URL may be accessed as [Query Parameters](#).
- [scopes](#) : [0..\*] [Scope](#)  
A [Scope](#) can contain many children [Scopes](#) through a containment hierarchy.

Events

- `onInit` : [0..1] Event  
This Event is triggered once when this Scope is initialised.

Inherited Events

`onAccess` (from Accessible)

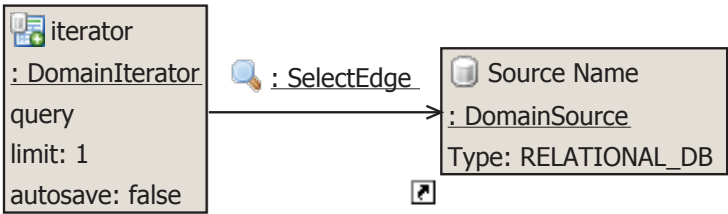
Model Completion Rules

- Scopes have an Event named `onInit`.
- A Scope protected by a Gate named 'XXX', with an outgoing ECA Rule named 'first' or 'last', will contain an Activity Predicate named 'check XXX' to check that the Gate has been satisfied.
- The 'check XXX' Function for a Gate, with an outgoing ECA Rule named 'first' or 'last', will be provided as a runtime Simple Condition check to that Gate.
- The 'check XXX' Activity Predicate for a Gate, with an outgoing ECA Rule named 'first' or 'last', will be completed with the contents necessary to evaluate the flag Value.

I.87 Select Edge

Defines the source of data types for a particular Domain Iterator.

Direct supertypes Generated Element, Generates Elements



References

- `from` : [0..1] Domain Iterator (opposite: out selects)  
The source Domain Iterator of this Select Edge.
- `to` : [0..1] Domain Source (opposite: in selects)  
The target Domain Source of this Select Edge.

Model Completion Rules

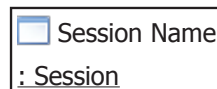
- A Domain Iterator within a Session protected by a Login Handler of type DOMAIN\_OBJECT without a specified Select Edge will use the same Domain Source as the Domain Type that defines the Domain Attribute used as a Parameter to that handler.

- A [Domain Iterator](#) within a [Session](#) protected by a [Login Handler](#) of type [USER](#) without a specified [Select Edge](#) will use the same [Domain Source](#) as the [Domain Type](#) that defines the [Domain Attribute](#) used as a [Parameter](#) to that handler.
- A [Domain Iterator](#) within a [Session](#) protected by a [Login Handler](#) of type [DOMAIN\\_OBJECT](#) or [USER](#) without a specified [Select Edge](#) will use the same [Domain Source](#) as the [Domain Type](#) used as a [Parameter](#) to that handler.
- A [Domain Iterator](#) within a [Session](#) protected by both an [Access Control Handler](#) and [Login Handler](#) of type [DOMAIN\\_OBJECT](#) or [USER](#), but the handlers both use different [Domain Types](#) (for example, with [inheritance](#)), and which does not specify a [Select Edge](#), will use the same [Domain Source](#) as the [Domain Type](#) used as a [Parameter](#) to the [Access Control Handler](#).
- A [Domain Iterator](#) within a [Session](#) protected by both an [Access Control Handler](#) and [Login Handler](#) of type [DOMAIN\\_OBJECT](#) or [USER](#), but the handlers both use the same [Domain Types](#), and which does not specify a [Select Edge](#), will use the same [Domain Source](#) as the [Domain Type](#) used as a [Parameter](#) to the [Access Control Handler](#).
- A [Domain Iterator](#) within a [Session](#) protected by both an [Access Control Handler](#) and [Login Handler](#) of type [DOMAIN\\_OBJECT](#) or [USER](#), but the [Access Control Handler](#) requires a [Permission](#) without a [Role](#), and which does not specify a [Select Edge](#), will use the same [Domain Source](#) as the [Domain Type](#) used as a [Parameter](#) to the [Access Control Handler](#).

## I.88 Session

A user session, which is only accessible by a single user.

**Direct supertypes** [Contains Functions](#), [Contains Operations](#), [Contains Wires](#), [Named Element](#), [Scope](#)



Every *visitor* to a web application is assigned a unique identifier in order to promote them to a *user*. If a [Session](#) is accessed by a user and no association exists between the users' unique identifier and the given [Session](#), then a new relationship is created and stored as part of the [Internet Application](#). The current user now *owns* that [Session](#). When this occurs, the *onInit* event for that [Session](#) is triggered.

If the current *user* restarts their web browser, the *user* will lose their unique identifier and will be demoted back into a *visitor*. Once a unique identifier has been lost, there is no way to regain access to a particular [Session](#), and a new [Session](#) relationship will have to be defined. Sessions may also *time out* after a specified period of activity, and this period of activity should be configurable by the model developer.

An IAML model instance may contain any number of [Sessions](#). The *storage semantics* for a [Session](#) are as follows: For a [Value](#) directly contained within a [Session](#), any stored value is only accessible to the user that *owns* the containing [Session](#). Once the [Session](#) ownership has been lost, any stored value is no longer accessible. Similarly, the values stored within a [Domain Iterator](#) (including the current *instance pointer*) are only accessible to the current *user*.

### Inherited Events

[onAccess](#) (from [Accessible](#)); [onInit](#) (from [Scope](#))

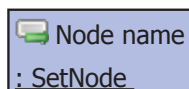
## Model Completion Rules

- A parent **Sync Wire** connecting a **Session**-contained **Visible Thing** to a **Contains Operations** element outside of that **Session**, will call the 'init' **Operation** generated by each sub-**Sync Wire** when the **Session** is initialised.
- A parent **Sync Wire** connecting a **Session**-contained **Frame** to a **Contains Operations** element outside of that **Session**, will call the 'init' **Operation** generated by each sub-**Sync Wire** when the **Session** is initialised.
- Every 'init' **Operation** executed when a **Session** is initialised due to a **Sync Wire** between a **Visible Thing** within that session, to another element outside of that session, will be provided the field value of the source element as a **Parameter**.
- Every 'init' **Operation** executed when a **Session** is initialised due to a **Sync Wire** between a **Frame** within that session, to another element outside of that session, will be provided the field value of the source element as a **Parameter**.
- A parent **Set Wire** connecting a **Session**-contained **Visible Thing** to a **Contains Operations** element outside of that **Session**, will call the 'init' **Operation** generated by each sub-**Set Wire** when the **Session** is initialised.
- A parent **Set Wire** connecting a **Session**-contained **Frame** to a **Contains Operations** element outside of that **Session**, will call the 'init' **Operation** generated by each sub-**Set Wire** when the **Session** is initialised.
- Every 'init' **Operation** executed when a **Session** is initialised due to a **Set Wire** between a **Visible Thing** within that session, to another element outside of that session, will be provided the field value of the source element as a **Parameter**.
- Every 'init' **Operation** executed when a **Session** is initialised due to a **Set Wire** between a **Frame** within that session, to another element outside of that session, will be provided the field value of the source element as a **Parameter**.
- A **Session** will contain at least a single **Frame**.
- A **Login Handler** is generated for a **Session** that contains an **Access Control Handler**.

## I.89 Set Node

Sets the value of all outgoing **Data Flow Edges** to the value of the first incoming **Data Flow Edge**.

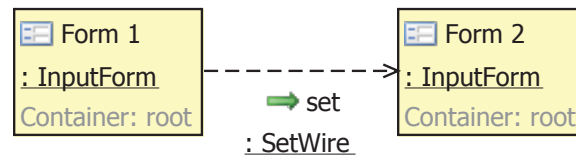
**Direct supertypes** Activity Node, Data Flow Edge Destination, Data Flow Edges Source, Execution Edge Destination, Execution Edges Source, Named Element



## I.90 Set Wire

When the value of a source element changes, the target element will be updated with this value (but not vice versa). Can be connected to any **Can Be Synced** model element.





### Direct supertypes Wire

A **Set Wire** will synchronise the **Can Be Synced** elements contained within each of the synchronised elements in a manner similar to a **Sync Wire**, except that this **Wire** is unidirectional; if the target value changes, the source value will not change. For a **Set Wire**, the *default element* created depends instead on the following semantics:

- If both the *source container* and the *target container* are **Input Forms**, then the type of the *default element* will be a **Label**.
- If the *source container* is a **Domain Iterator** and the *target container* is a **Input Form**, **Email** or **Iterator List**, then the type of the *default element* will be a **Label**.
- If the *source container* is a **Visible Thing** and the *target container* is a **Map**, then the type of the *default element* will be a **Map Point**.

If a **Domain Iterator** iterating over multiple results is connected to a **Input Form** with a **Sync Wire** or **Set Wire**, then the form will be populated in such a way to provide navigation over the data set. That is, **Buttons** will be created within the form to control the cursor over the iterator.

### Attributes

- **executeOnInput** : boolean

If true, then this **Set Wire** will *also* try to synchronise these two elements when the **onInput** event is triggered.

### Model Completion Rules

- A **Set Wire** connecting two objects will create sub-**Set Wires** between every contained object within these parent objects which have matching names.
- A **Set Wire** connecting an **Domain Iterator** to a **Can Be Synced** object will create sub-**Set Wires** between every contained object within the contained **Domain Instance** to every contained **Can Be Synced** object which have matching names.
- A **Set Wire** connecting an **Can Be Synced** object to a **Domain Iterator** will create sub-**Set Wires** between every contained **Can Be Synced** object to every contained object within the contained **Domain Instance** which have matching names.
- When a **Changeable** is connected to a **Contains Operations** by a **Set Wire**, the source element will call the 'update' **Operation** on the target when the source changes.
- When a **Changeable** is connected to a **Value** by a **Set Wire**, the source element will call the 'set value XXX' **Operation** on the target when the source changes.
- When an **Input Text Field** is connected to a **Contains Operations** by an instant **Set Wire**, the source element will call the 'update' **Operation** on the target when the source input changes.
- When a **Can Be Synced** element is connected to an **Can Be Synced** element by a **Set Wire**, the value of the source element will be provided as a **Parameter** to the updating **ECA Rule**.
- When a **Changeable** element is connected to an **Can Be Synced** element by a **Set Wire**, the value of the source element will be provided as a **Parameter** to the updating **ECA Rule**.

- When a **Changeable** element is connected to a **Value** by a **Set Wire**, the **value** of the source element will be provided as a **Parameter** to the updating **ECA Rule**.
- When an **Input Text Field** is connected to a **Can Be Synced** element by an **instant Set Wire**, the **value** of the source element will be provided as a **Parameter** to the updating **ECA Rule**.
- When a **Contains Operations** element is connected to an **Accessible** element by a **Set Wire**, the source element will **call** the **'init' Operation** on the target when the target is **accessed**.
- When a **Contains Operations** element is connected to an **Accessible** element by a **Set Wire**, the target element will **only** be initialised if the source value is set.
- When a **Can Be Synced** element is connected to an **Accessible** element by a **Set Wire**, the **field value** of the source element will be used as a **Parameter** to initialise the target through the **'init' Operation**.
- A sub-**Set Wire** generated for a parent **Set Wire** will also be restricted with all of the **Simple Conditions** of the parent wire.
- A sub-**Set Wire** generated for a parent **Set Wire** connected from a **Domain Iterator** will also be restricted with all of the **Simple Conditions** of the parent wire.
- A sub-**Set Wire** generated for a parent **Set Wire** connected to a **Domain Iterator** will also be restricted with all of the **Simple Conditions** of the parent wire.
- All restrictions derived by copying the **Simple Conditions** of parent **Set Wires** to their sub-**Set Wires** will also be provided with the same **Parameters**.
- All restrictions derived by copying the **Simple Conditions** of parent **Set Wires** from a **Domain Iterator** to their sub-**Set Wires** will also be provided with the same **Parameters**.
- All restrictions derived by copying the **Simple Conditions** of parent **Set Wires** to a **Domain Iterator** to their sub-**Set Wires** will also be provided with the same **Parameters**.
- A parent **Set Wire** connecting a **Session**-contained **Visible Thing** to a **Contains Operations** element outside of that **Session**, will **call** the **'init' Operation** generated by each sub-**Set Wire** when the **Session** is **initialised**.
- A parent **Set Wire** connecting a **Session**-contained **Frame** to a **Contains Operations** element outside of that **Session**, will **call** the **'init' Operation** generated by each sub-**Set Wire** when the **Session** is **initialised**.
- Every **'init' Operation** executed when a **Session** is **initialised** due to a **Set Wire** between a **Visible Thing** within that session, to another **element** outside of that session, will be provided the **field value** of the source element as a **Parameter**.
- Every **'init' Operation** executed when a **Session** is **initialised** due to a **Set Wire** between a **Frame** within that session, to another **element** outside of that session, will be provided the **field value** of the source element as a **Parameter**.
- An empty **Input Form** connected to a **Domain Iterator** with a **Set Wire** will create an **Input Text Field** in that form for every **Domain Attribute Instance** that is not a primary key or **generated**.
- An **Input Form** connected to another **Input Form** with a **Set Wire** will create a **Label** in the target form for every **Input Text Field** in the source form with the same **name**.
- An **Input Form** connected to another **Input Form** with a **Set Wire** will create a **Label** in the target form for every **Label** in the source form with the same **name**.
- An **Input Form** connected to an **Email** with a **Set Wire** will create a **Label** in the target Email for every **Input Text Field** in the source form with the same **name**.
- An **Input Form** connected to an **Email** with a **Set Wire** will create a **Label** in the target Email for every **Label** in the source form with the same **name**.
- An **Input Form** connected to a **Map** with a **Set Wire** will create a **Map Point** in the target map for every **Input Text Field** in the source form with the same **name**.
- When a **Domain Iterator** is connected to an **Input Form** by a **Set Wire**, **Labels** will be generated for its non-generated, non-primary key **Domain Attribute Instances**.

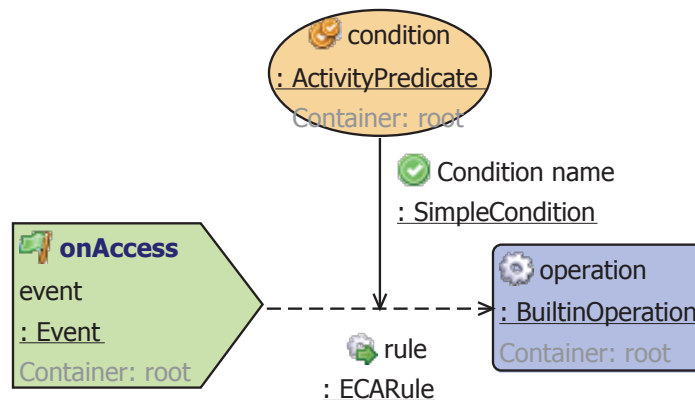
- When a **Domain Iterator** is connected to an **Iterator List** by a **Set Wire**, **Labels** will be generated for its non-generated, non-primary key **Domain Attribute Instances**.
- When a **Domain Iterator** is connected to an **Iterator List** by a **Set Wire**, a **hidden Label** will be generated for the primary key **Domain Attribute Instance**.
- A **Set Wire** connecting a **Domain Iterator** to an **Input Form** should only execute the 'init' **Operation** on the created **Labels** when the label is **accessed** if the **Domain Instance** is not empty.
- A **Set Wire** connecting a **Domain Iterator** to an **Input Form** should only execute the 'update' **Operation** on the created **Labels** when the label is **changed** if the **Domain Instance** is not empty.
- A **Login Handler** of type **DOMAIN\_OBJECT** will connect each **Value** named 'current XXX' using **Set Wires** named 'set'.
- A **Login Handler** of type **USER** will connect each **Value** named 'current XXX' using **Set Wires** named 'set'.
- A **Domain Iterator** 'current instance' created for a **Login Handler** of type **DOMAIN\_OBJECT** will be provided as the set **target** of the handler.
- A **Domain Iterator** 'current instance' created for a **Login Handler** of type **USER** will be provided as the set **target** of the handler.
- An **Operation** 'set value XXX' is created in the container of all **Values** that are the target of **Set Wires**.
- An **Input Form** connected to a **Domain Iterator** using a **Set Wire** with a **limit** set (not 1), will contain **Buttons** named 'Next', 'Previous', 'First' and 'Last', in order to support navigation through the **Domain Iterator**.
- An **Input Form** connected to a **Domain Iterator** using a **Set Wire** with a **limit** set (not 1), will contain a **Label** named 'Results'.
- The 'next' **Button** defined for navigating a **Domain Iterator**, as defined by a **Set Wire**, will **execute** the **next Operation** on that iterator when **clicked**.
- The 'next' **Button** defined for navigating a **Domain Iterator**, as defined by a **Set Wire**, will be conditionally **executed** only if the iterator's **has next Function** returns **true**.
- The 'previous' **Button** defined for navigating a **Domain Iterator**, as defined by a **Set Wire**, will **execute** the **previous Operation** on that iterator when **clicked**.
- The 'previous' **Button** defined for navigating a **Domain Iterator**, as defined by a **Set Wire**, will be conditionally **executed** only if the iterator's **has previous Function** returns **true**.
- The 'first' **Button** defined for navigating a **Domain Iterator**, as defined by a **Set Wire**, will **execute** the **reset Operation** on that iterator when **clicked**.
- The 'first' **Button** defined for navigating a **Domain Iterator**, as defined by a **Set Wire**, will be conditionally **executed** only if the iterator's **not empty Function** returns **true**.
- The 'last' **Button** defined for navigating a **Domain Iterator**, as defined by a **Set Wire**, will **execute** the **jump Operation** on that iterator when **clicked**.
- The 'last' **Button** defined for navigating a **Domain Iterator**, as defined by a **Set Wire**, will be conditionally **executed** only if the iterator's **not empty Function** returns **true**.
- The 'last' **Button** defined for navigating a **Domain Iterator**, as defined by a **Set Wire**, will contain a **Value** named 'target'.
- The 'target' **Value** defined in a 'last' **Button** defined for navigating a **Domain Iterator**, as defined by a **Set Wire**, will be used as a **Parameter** for executing the **jump Operation** on that iterator.
- The 'last' **Button** defined for navigating a **Domain Iterator**, as defined by a **Set Wire**, will contain an **Activity Operation** named 'update target'.
- The 'last' **Button** defined for navigating a **Domain Iterator**, as defined by a **Set Wire**, needs to **execute** the 'update target' **Operation** within that **Button** before calling **jump** on the iterator.

- The ‘results’ [Label](#) defined for navigating a [Domain Iterator](#), as defined by a [Set Wire](#), will be updated whenever the iterator [changes](#).
- The current results [count](#) of the [Domain Iterator](#) will be used as a [Parameter](#) to update the ‘results’ [Label](#) defined for navigating a [Domain Iterator](#) with a [Set Wire](#) when the iterator [changes](#).
- The ‘results’ [Label](#) defined for navigating a [Domain Iterator](#), as defined by a [Set Wire](#), will be updated whenever the containing [Input Form](#) is [accessed](#).
- The current results [count](#) of the [Domain Iterator](#) will be used as a [Parameter](#) to update the ‘results’ [Label](#) defined for navigating a [Domain Iterator](#) with a [Set Wire](#) when the containing [Input Form](#) is [accessed](#).
- All [conditions](#) against a [Set Wire](#) will be copied to each [ECA Rule](#) subsequently created as connected to the [onChange Events](#).
- All [conditions](#) against a [Set Wire](#) will be copied to each [ECA Rule](#) subsequently created as connected to the [onChange Events](#).
- All [Parameters](#) for each [conditions](#) against a [Set Wire](#) will be copied to each [Simple Condition](#) subsequently created for [ECA Rules](#) connected to the [onChange Events](#).
- All [Parameters](#) for each [conditions](#) against a [Set Wire](#) will be copied to each [Simple Condition](#) subsequently created for [ECA Rules](#) connected to the [onAccess Events](#).

## I.91 Simple Condition

May be used to conditionally prevent an [ECA Rule](#) from firing, if the evaluation of this [Complex Term](#) returns false.

**Direct supertypes** [Complex Term](#)



Currently only the [Simple Condition](#) subtype of the [Condition](#) abstract supertype is provided; the implementation of the [Complex Condition](#) subtype remains future work<sup>12</sup>.

### Model Completion Rules

- When a [Contains Operations](#) element is connected to an [Accessible](#) element by a [Sync Wire](#), the target element will [only](#) be initialised if the source value is set.

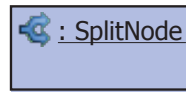
<sup>12</sup>Issue 226: Add *ComplexConditions* model elements.

- A sub- **Sync Wire** generated for a parent **Sync Wire** will also be restricted with all of the **Simple Conditions** of the parent wire.
- A sub- **Sync Wire** generated for a parent **Sync Wire** connected to a **Domain Iterator** will also be restricted with all of the **Simple Conditions** of the parent wire.
- All restrictions derived by copying the **Simple Conditions** of parent **Sync Wires** to their sub-**Sync Wires** will also be provided with the same **Parameters**.
- All restrictions derived by copying the **Simple Conditions** of parent **Sync Wires** involving a **Domain Iterator** to their sub-**Sync Wires** will also be provided with the same **Parameters**.
- When a **Contains Operations** element is connected to an **Accessible** element by a **Set Wire**, the target element will **only** be initialised if the source value is set.
- A sub- **Set Wire** generated for a parent **Set Wire** will also be restricted with all of the **Simple Conditions** of the parent wire.
- A sub- **Set Wire** generated for a parent **Set Wire** connected from a **Domain Iterator** will also be restricted with all of the **Simple Conditions** of the parent wire.
- A sub- **Set Wire** generated for a parent **Set Wire** connected to a **Domain Iterator** will also be restricted with all of the **Simple Conditions** of the parent wire.
- All restrictions derived by copying the **Simple Conditions** of parent **Set Wires** to their sub-**Set Wires** will also be provided with the same **Parameters**.
- All restrictions derived by copying the **Simple Conditions** of parent **Set Wires** from a **Domain Iterator** to their sub-**Set Wires** will also be provided with the same **Parameters**.
- All restrictions derived by copying the **Simple Conditions** of parent **Set Wires** to a **Domain Iterator** to their sub-**Set Wires** will also be provided with the same **Parameters**.
- An **input entry Gate** will **only** update the value of each **input Label** from **Visible Things** in the **input Frame** if the source elements can be **cast**.
- All **conditions** against a **Sync Wire** will be copied to each **ECA Rule** subsequently created as connected to the **onChange Events**.
- All **conditions** against a **Sync Wire** will be copied to each **ECA Rule** subsequently created as connected to the **onAccess Events**.
- All **Parameters** for each **conditions** against a **Sync Wire** will be copied to each **ECA Rule** subsequently created as connected to the **onChange Events**.
- All **Parameters** for each **conditions** against a **Sync Wire** will be copied to each **ECA Rule** subsequently created as connected to the **onAccess Events**.
- All **conditions** against a **Set Wire** will be copied to each **ECA Rule** subsequently created as connected to the **onChange Events**.
- All **conditions** against a **Set Wire** will be copied to each **ECA Rule** subsequently created as connected to the **onChange Events**.
- All **Parameters** for each **conditions** against a **Set Wire** will be copied to each **Simple Condition** subsequently created for **ECA Rules** connected to the **onChange Events**.
- All **Parameters** for each **conditions** against a **Set Wire** will be copied to each **Simple Condition** subsequently created for **ECA Rules** connected to the **onAccess Events**.

## I.92 Split Node

Execution flow can split off into multiple threads, reconnected with a **Join Node**.

**Direct supertypes** Activity Node, Execution Edge Destination, Execution Edges Source



An **Activity Operation** may contain any number of **Split Nodes**. Each **Split Node** must have at least one outgoing **Execution Edge**. Each outgoing **Execution Edge** is defined as a separate *thread* with respect to that particular **Split Node**, and every thread must eventually terminate at a **Join Node**.

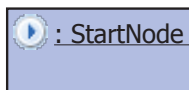
When the *execution flow* arrives at a **Split Node**, each outgoing thread will be executed according to some multithreaded policy, with each of these executed threads inserted into a *thread set* associated with the **Split Node**. Every thread has an associated *execution flow*. No guarantee is given as to the order in which each thread is executed, nor that any number of threads will execute simultaneously or that the implementation is actually within a multithreaded environment.

The **Split Node** element is based off the UML *ForkNode* model element.

## I.93 Start Node

An **Activity Operation** or **Activity Predicate** may contain a single **Start Node** as the starting point of its execution flow.

**Direct supertypes** **Activity Node**, **Execution Edges Source**



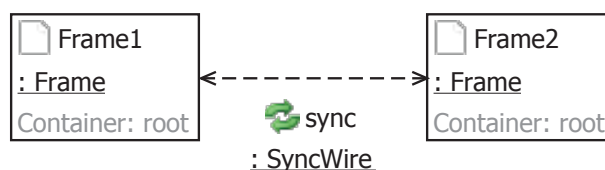
An **Activity Operation** must contain exactly one **Start Node**, which must have at least one outgoing **Execution Edge**. When the **Activity Operation** is executed, the *execution flow* will begin from this **Start Node**, and will follow each outgoing **Execution Edge** sequentially.

The **Start Node** element is based on the UML *InitialNode* model element.

## I.94 Sync Wire

When the value of a connected element changes, a **Sync Wire** will update the other connected element. Can be connected to any **Can Be Synced** model element.

**Direct supertypes** **Wire**



A **Sync Wire** is a bidirectional **Wire**, that may connect any two **Can Be Synced** model elements. When the intended model is completed through *model completion*, a **Sync Wire** will include elements necessary to keep the values of two associated **Can Be Synced** elements synchronised.



A **Sync Wire** will synchronise the **Can Be Synced** elements contained within each of the synchronised elements. That is, if a **Can Be Synced** element (the *triggering element*) contained within one **Sync Wire**-connected element (the *source container*) does not have an associated **Can Be Synced** element within the other **Sync Wire**-connected element (the *target container*), then the following process will occur:

1. For a **Sync Wire**, a *default element* will be created within the *target container*, as follows:
  - (a) If both the *source container* and the *target container* are **Visible Things**, then the type of the *default element* will be the same as the type of the *triggering element*.
  - (b) If the *source container* is a **Domain Iterator** and the *target container* is a **Input Form**, then the type of the *default element* will be a **Input Text Field**.
2. A new **Sync Wire** will be generated by the original **Sync Wire**. This new **Sync Wire** will connect the generated *default element* to the triggering element in the *source container*.
3. Any incoming **Conditions** and **Parameters** to the original **Sync Wire** will be duplicated with respect to the newly-created **Sync Wire**.

If a **Domain Iterator** iterating over multiple results is connected to a **Input Form** with a **Sync Wire** or **Set Wire**, then the form will be populated in such a way to provide navigation over the data set. That is, **Buttons** will be created within the form to control the cursor over the iterator.

### Attributes

- **executeOnInput** : boolean

If true, then this **Sync Wire** will *also* try to synchronise these two elements when the **onInput** event is triggered.

### Model Completion Rules

- A **Sync Wire** connecting two elements with a **Function** 'can sync?' will only permit synchronisation if the value can be cast **successfully**.
- For elements connected by a **Sync Wire**, the 'onAccess' **Event** executing the 'init' **Operation** will only **execute** if the source **field Value** can be cast **successfully**.
- For elements connected by a **Sync Wire**, the **onAccess Event** executing the 'init' **Operation** will only **execute** if the source **field Value** can be cast **successfully**.
- For elements connected by a **Sync Wire**, the **onAccess Event** will only call the 'validate' **Operation** if the relevant **field value** is set.
- A **Sync Wire** connecting two objects will create sub-**Sync Wires** between every contained object within these parent objects which have matching **names**.
- A **Sync Wire** connecting an **Domain Iterator** to a **Can Be Synced** object will create sub-**Sync Wires** between every contained object within the contained **Domain Instance** to every contained **Can Be Synced** object which have matching **names**.
- When a **Changeable** is connected to a **Contains Operations** by a **Sync Wire**, the source element will call the 'update' **Operation** on the target when the source **changes**.
- When an **Input Text Field** is connected to a **Contains Operations** by a **instant Sync Wire**, the source element will call the 'update' **Operation** on the target when the source input **changes**.
- When a **Can Be Synced** element is connected to an **Can Be Synced** element by a **Sync Wire**, the **value** of the source element will be provided as a **Parameter** to the updating **ECA Rule**.

- When an **Input Text Field** is connected to a **Can Be Synced** element by an **instant Sync Wire**, the **value** of the source element will be provided as a **Parameter** to the updating **ECA Rule**.
- When a **Contains Operations** element is connected to an **Accessible** element by a **Sync Wire**, the source element will **call** the 'init' **Operation** on the target when the target is **accessed**.
- When a **Contains Operations** element is connected to an **Accessible** element by a **Sync Wire**, the target element will **only** be initialised if the source value is set.
- When a **Can Be Synced** element is connected to a **Can Be Synced** element by a **Sync Wire**, the **field value** of the source element will be used as a **Parameter** to initialise the target through the 'init' **Operation**.
- A sub- **Sync Wire** generated for a parent **Sync Wire** will also be restricted with all of the **Simple Conditions** of the parent wire.
- A sub- **Sync Wire** generated for a parent **Sync Wire** connected to a **Domain Iterator** will also be restricted with all of the **Simple Conditions** of the parent wire.
- All restrictions derived by copying the **Simple Conditions** of parent **Sync Wires** to their sub-**Sync Wires** will also be provided with the same **Parameters**.
- All restrictions derived by copying the **Simple Conditions** of parent **Sync Wires** involving a **Domain Iterator** to their sub-**Sync Wires** will also be provided with the same **Parameters**.
- A parent **Sync Wire** connecting a **Session-contained Visible Thing** to a **Contains Operations** element outside of that **Session**, will **call** the 'init' **Operation** generated by each sub-**Sync Wire** when the **Session** is initialised.
- A parent **Sync Wire** connecting a **Session-contained Frame** to a **Contains Operations** element outside of that **Session**, will **call** the 'init' **Operation** generated by each sub-**Sync Wire** when the **Session** is initialised.
- Every 'init' **Operation** executed when a **Session** is initialised due to a **Sync Wire** between a **Visible Thing** within that session, to another **element** outside of that session, will be provided the **field value** of the source element as a **Parameter**.
- Every 'init' **Operation** executed when a **Session** is initialised due to a **Sync Wire** between a **Frame** within that session, to another **element** outside of that session, will be provided the **field value** of the source element as a **Parameter**.
- An empty **Input Form** connected to a **Domain Iterator** with a **Sync Wire** will create an **Input Text Field** in that form for every **Domain Attribute Instance** that is not a primary **key** or foreign **key**.
- An **Input Form** connected to another **Input Form** with a **Sync Wire** will create an **Input Text Field** in the target form for every **Input Text Field** in the source form with the same **name**.
- An **Input Form** connected to another **Input Form** with a **Sync Wire** will create a **Label** in the target form for every **Label** in the source form with the same **name**.
- A **Sync Wire** connecting a 'new' **Domain Iterator** to an **Input Form** should only execute the 'init' **Operation** on the created **Labels** when the label is **accessed** if the **Domain Instance** is not **empty**.
- All **conditions** against a **Sync Wire** will be copied to each **ECA Rule** subsequently created as connected to the **onChange Events**.
- All **conditions** against a **Sync Wire** will be copied to each **ECA Rule** subsequently created as connected to the **onAccess Events**.
- All **Parameters** for each **conditions** against a **Sync Wire** will be copied to each **ECA Rule** subsequently created as connected to the **onChange Events**.
- All **Parameters** for each **conditions** against a **Sync Wire** will be copied to each **ECA Rule** subsequently created as connected to the **onAccess Events**.
- Two differently- typed **Input Text Fields** connected via a **Sync Wire** will define an **Operation** within the first field named 'validate'.

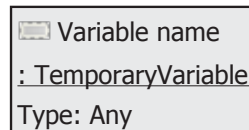


- The ‘validate’ [Operation](#) defined to validate the type instances of two differently-typed [Input Text Fields](#) connected via a [Sync Wire](#) will be [executed](#) when one of the fields [change](#).
- An [Activity Operation](#) named ‘validate’, used to check the validity of two type instances of two [Input Text Fields](#) connected via a [Sync Wire](#), will be completed with the necessary [Activity Nodes](#) to implement this functionality.

## I.95 Temporary Variable

A temporary [Value](#) instance, whose value will be lost once this [Activity Operation](#) or [Activity Predicate](#) is completed.

**Direct supertypes** [Data Flow Edge Destination](#), [Data Flow Edges Source](#), [Named Element](#), [Parameter Edges Source](#)



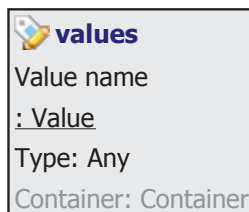
### References

- [type](#) : [0..1] [EXSD Data Type](#)  
The type of the [Temporary Variable](#).

## I.96 Value

Represents a single [typed](#) value, accessible and modifiable at runtime.

**Direct supertypes** [Parameter Edges Source](#), [Parameter Value](#), [Wireable](#)



[Values](#) have an associated [name](#) and [XSD type](#), and may also have a [defaultValue](#). The [defaultValue](#) of a [Value](#) is serialised as a [string](#), and this [string](#) instance is deserialised into a valid [type](#) instance defined by that [Value](#). Each [Value](#) has an associated *containing scope* that represents the storage method for the [Value](#):

- If the [Value](#) is stored within an [Activity Operation](#) or [Activity Predicate](#), then the containing scope of that value is the containing scope of the [Activity Operation](#)’s or [Activity Predicate](#)’s container.
- If the [Value](#) is stored within a [Scope](#), then the value instance is unique to a given [Scope](#). For example, a [Value](#) contained within a [Session](#) can only have one instance of that [Value](#) per [Session](#) instance, and instances of that [Value](#) cannot be accessed by other instances of the same [Session](#) scope.

- If the **Value** is not stored within a **Scope**, then the value instance is available globally according to the root **Internet Application**. That is, only one instance of the **Value** can ever exist.

If a particular **Value** has a defined **defaultValue**, then the **Value** will initially be considered as *unset*. However, if a **Value** is subsequently updated – for example, through the **update Builtin Operation** – the **Value** will now be considered *set*.

### Attributes

- **defaultValue** : string  
The initial value of this **Value**, as serialised to a string instance.
- **readOnly** : boolean  
If true, then this **Value** is considered read-only and cannot be modified at runtime.

### Model Completion Rules

- A **Changeable** contains an untyped **Value** named **field value**.
- If set, the **default Value** is used as the default value of the created **field value Value**.
- All **Values** not named 'fieldValue' will have an **Activity Predicate** in its container named 'XXX is set'.
- All **Values** named 'fieldValue' will have a **Builtin Property** in its container named 'fieldValue is set'.
- All non- **read only Values**, other than those named 'fieldValue', will have an **Builtin Operation** in its container named 'set XXX'.
- When a **Changeable** is connected to a **Value** by a **Set Wire**, the source element will **call** the 'set value XXX' **Operation** on the target when the source **changes**.
- When a **Changeable** element is connected to a **Value** by a **Set Wire**, the **value** of the source element will be provided as a **Parameter** to the updating **ECA Rule**.
- An **Email** contains a **Value** named 'to'.
- An **Email** contains a **Value** named 'toName'.
- An **Email** contains a **Value** named 'from'.
- An **Email** contains a **Value** named 'fromName'.
- An **Email** contains a **Value** named 'subject'.
- A **Login Handler** of type SECRET\_KEY will define a **Value** named 'current login key' in the protected **Session** as the *stored key*, if no outgoing **Set Wire** named 'set' from the **Login Handler** is defined.
- A **Login Handler** of type DOMAIN\_OBJECT will define a **Value** named 'current XXX' in the protected **Session** for each **Domain Attribute** used as a **Parameter** to the handler.
- A **Login Handler** of type USER will define a **Value** named 'current XXX' in the protected **Session** for each **Domain Attribute** used as a **Parameter** to the handler.
- **Domain Iterators** have a **read only Value** named **results**.
- **Input Text Fields** have an **Value** named **current input**.
- An **Operation** 'set value XXX' is created in the container of all **Values** that are the target of **Set Wires**.
- An empty **Activity Operation** named "set value XXX", contained within a **Changeable**, used to set a **Value** other than **field value**, will be completed with **Activity Nodes** to implement the particular operation.

## I.97 Visible Thing

An interface element that may be rendered to the user.

### Abstract type.

**Direct supertypes** Accessible, Can Be Synced, Changeable, Contains Functions, Contains Operations, Contains Values, Contains Wires, Generated Element, Generates Elements, Named Element, Wire Destination, Wire Source

**Direct subtypes** Button, Input Form, Input Text Field, Iterator List, Label, Map, Map Point

A [Visible Thing](#) is the abstract type for interface modelling, and may directly contain other [Visible Things](#) as its children. All [Visible Things](#) may be *rendered* to a user. When a [Visible Thing](#) is rendered to a user, all children [Visible Things](#) must also be rendered in order according to their `renderOrder`. A [Visible Thing](#) that is currently *hidden* must not be rendered until that [Visible Thing](#) is subsequently *shown*. The `name` of the [Visible Thing](#) may be used as a label for the rendered element.

All [Visible Things](#) define the `hide`, `show`, `init` and `update` Operations, which are implemented as [Builtin Operations](#). All [Visible Things](#) also define the `empty` and `notEmpty` Predicates, which are implemented as [Builtin Property](#)s.

### Attributes

- `renderOrder` : integer

[Visible Things](#) are rendered in an ascending `render order` order.

- `visible` : boolean

If false, this [Visible Thing](#) will not initially be rendered to the user, until it has been *shown*.

### Containment References

- `children` : [0..\*] [Visible Thing](#)

Interface elements may be composed of other interface elements.

- `current input` : [0..1] [Value](#)

The transient value of the [Visible Thing](#) ; for example, the current text input in a [Input Text Field](#) before it has been committed. This is *not* always a valid `type` instance.

### Events

- `onClick` : [0..1] [Event](#)

This [Event](#) is triggered when the user clicks a rendered user interface element.

- `onInput` : [0..1] [Event](#)

This [Event](#) is triggered when the user modifies the current value of an interface element, but before it has been committed (as per the `onChange` event).

### Inherited Events

`onAccess` (from [Accessible](#)); `onChange` (from [Changeable](#))

**Functions**

- **empty** : [0..1] Predicate  
Returns `false` if the current `field value` is both set and not empty.
- **not empty** : [0..1] Predicate  
Returns `true` if the current `field value` is both set and not empty.

**Model Completion Rules**

- **Visible Things** have an **Event** named `onInput`.
- **Visible Things** have an **Event** named `onClick`.
- **Visible Things** have an **Builtin Property** named `empty`.
- **Visible Things** have an **Builtin Property** named `not empty`.
- A **Visible Thing** has a **Builtin Operation** named `hide`.
- A **Visible Thing** has a **Builtin Operation** named `show`.
- All **Visible Things** (except **Iterator Lists** and **Input Forms**) will contain an **Operation** named “update”.
- All **Visible Things** (except **Iterator Lists** and **Input Forms**) will contain an **Operation** named “init”.

**I.98 Wire**

A **Wire** represents a common web application design pattern with respect to at least two **model elements**. The functionality of the **Wire** is provided by *model completion*, which generates additional functionality within the current model instance.

**Abstract type.**

**Direct supertypes** **Condition Edge Destination**, **Contains Wires**, **Generated Element**, **Generates Elements**, **Named Element**, **Parameter Edge Destination**

**Direct subtypes** **Autocomplete Wire**, **Detail Wire**, **Set Wire**, **Sync Wire**

A **Wire** is an abstract connection that represents additional functionality of a common web application design pattern, as defined by the subtype of the **Wire**. A **Wire** connects a **Wire Source** to a **Wire Destination**; this relationship may be unidirectional (such as a **Set Wire**) or bidirectional (such as a **Sync Wire**).

A **Wire** may optionally be constrained by a number of incoming **Predicates**, as specified through **Conditions**; these incoming **Predicates** will be subsequently applied to all **ECA Rules** generated through that **Wire**.

**References**

- **from** : **Wire Source** (opposite: out wires)  
The source of a **Wire**.
- **to** : **Wire Destination** (opposite: in wires)  
The target of a **Wire**.

**I.99 Wire Destination**

A possible destination for a [Wire](#).

#### Interface.

**Direct subtypes** [Can Be Synced](#), [Visible Thing](#), [Wireable](#)

**Indirect subtypes** [Button](#), [Changeable](#), [Domain Attribute Instance](#), [Domain Feature Instance](#), [Domain Instance](#), [Domain Iterator](#), [Email](#), [Frame](#), [Input Form](#), [Input Text Field](#), [Internet Application](#), [Iterator List](#), [Label](#), [Login Handler](#), [Map](#), [Map Point](#), [Scope](#), [Session](#), [Value](#)

This abstract interface is provided to simplify the implementation of [Wires](#), as EMF does not support type unions in metamodel definitions.

#### References

- [in wires : \[0..\\*\] Wire](#) (opposite: to)  
A list of incoming [Wires](#).

## I.100 Wire Source

A possible source for a [Wire](#).

#### Interface.

**Direct supertypes** [Contains Wires](#)

**Direct subtypes** [Can Be Synced](#), [Function](#), [Visible Thing](#), [Wireable](#)

**Indirect subtypes** [Activity Predicate](#), [Boolean Property](#), [Builtin Property](#), [Button](#), [Changeable](#), [Domain Attribute Instance](#), [Domain Feature Instance](#), [Domain Instance](#), [Domain Iterator](#), [Email](#), [Frame](#), [Input Form](#), [Input Text Field](#), [Internet Application](#), [Iterator List](#), [Label](#), [Login Handler](#), [Map](#), [Map Point](#), [Predicate](#), [Scope](#), [Session](#), [Value](#), [XQuery Function](#), [XQuery Predicate](#)

This abstract interface is provided to simplify the implementation of [Wires](#), as EMF does not support type unions in metamodel definitions.

#### References

- [out wires : \[0..\\*\] Wire](#) (opposite: from)  
A list of outgoing [Wires](#).

## I.101 Wireable

An interface to represent element types that can be both [sources](#) and [destinations](#) for [Wires](#).

#### Interface.

**Direct supertypes** [Wire Destination](#), [Wire Source](#)

**Direct subtypes** Changeable, Domain Instance, Domain Iterator, Email, Login Handler, Scope, Value

**Indirect subtypes** Button, Domain Attribute Instance, Domain Feature Instance, Frame, Input Form, Input Text Field, Internet Application, Iterator List, Label, Map, Map Point, Session, Visible Thing

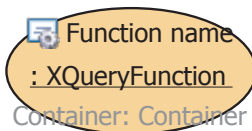
This abstract interface is provided to simplify the implementation of [Wires](#), as EMF does not support type unions in metamodel definitions.

## I.102 XQuery Function

Provides an interface to a predefined set of XQuery Functions as [Functions](#).

**Direct supertypes** [Function](#)

**Direct subtypes** [XQuery Predicate](#)



### Attributes

- **func** : XQUERY\_FUNCTIONS

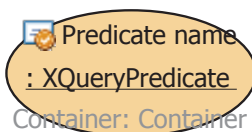
The XQuery function referenced by this [XQuery Function](#).

Possible values: OP\_NUMERIC\_ADD, OP\_NUMERIC\_SUBTRACT, OP\_NUMERIC\_MULTIPLY, OP\_NUMERIC\_DIVIDE, FN\_ABS, FN\_FLOOR, FN\_CONCAT, FN\_SUBSTRING, FN\_CONTAINS, FN\_CODEPOINT\_EQUAL, OP\_NUMERIC\_EQUAL, OP\_NUMERIC\_LESS\_THAN, OP\_NUMERIC\_GREATER\_THAN, FN\_NOT

## I.103 XQuery Predicate

An [XQuery Function](#) that is typed boolean.

**Direct supertypes** [Predicate](#), [XQuery Function](#)



# Bibliography

- [1] Acerbis, R., Bongio, A., Brambilla, M., Tisi, M., Ceri, S. and Tosetti, E. Developing eBusiness Solutions with a Model Driven Approach: The Case of Acer EMEA. In: *Proceedings of the 7th International Conference on Web Engineering (ICWE '07)*, pp. 539–544. 2007
- [2] Achour, M., Betz, F., Dovgal, A., Lopes, N., Magnusson, H., Richter, G., Seguy, D. and Vrana, J. *PHP Manual*, 2011  
<http://php.net/manual/en/>
- [3] Adobe Systems Incorporated. Adobe AIR, 2011  
<http://www.adobe.com/products/air/>
- [4] Adrion, W. R., Branstad, M. A. and Cherniavsky, J. C. Validation, Verification, and Testing of Computer Software. *ACM Comput. Surv.*, vol. 14, pp. 159–192, 1982
- [5] Alexander, R. The Real Costs of Aspect-Oriented Programming? *IEEE Software*, vol. 20(6), pp. 91–93, 2003
- [6] Allaire, J. Macromedia Flash MX – A Next-generation Rich Client. Tech. rep., Macromedia, Inc., 2002
- [7] Anastasopoulos, M. and Muthig, D. An Evaluation of Aspect-Oriented Programming as a Product Line Implementation Technology. In: *ICSR, Lecture Notes in Computer Science*, vol. 3107, pp. 141–156. Springer, 2004
- [8] Andy Prevost. PHPMailer, 2009  
<http://phpmailer.worxware.com/>
- [9] Aniszczyk, C. Using GEF with EMF. Tech. rep., IBM, 2005  
<http://www.eclipse.org/articles/Article-GEF-EMF/gef-emf.html>
- [10] Apache Software Foundation. Struts Framework, 2008  
<http://struts.apache.org/>
- [11] Atterer, R. Where Web Engineering Tool Support Ends: Building Usable Websites. In: *Proceedings of the 2005 ACM Symposium on Applied Computing (SAC '05)*, pp. 1684–1688. ACM Press, New York, NY, USA, 2005
- [12] Bacvanski, V. and Graff, P. Mastering Eclipse Modeling Framework. In: *EclipseCon*. Burlingame, California, 2005

- [13] Badros, G. J. JavaML: A Markup Language for Java Source Code. *Comput. Netw.*, vol. 33, pp. 159–177, 2000
- [14] Baier, C. and Katoen, J.-P. *Principles of Model Checking*. The MIT Press, 2008
- [15] Baker, P., Dai, Z. R., Grabowski, J., Øystein Haugen, Schieferdecker, I. and Williams, C. Test Execution with JUnit. In: *Model-Driven Testing*, pp. 149–156. Springer Berlin Heidelberg, 2008
- [16] Baldwin, C. Y. and Clark, K. B. *Design Rules, Volume 1: The Power of Modularity*, vol. 1. The MIT Press, 1 ed., 2000
- [17] Baresi, L., Colazzo, S., Mainetti, L. and Morasca, S. W2000: A Modelling Notation for Complex Web Applications. In: Mendes, E. and Mosley, N., eds., *Web Engineering*, pp. 335–364. Springer, 2006
- [18] Baroni, A. L. *Formal Definition of Object-Oriented Design Metrics*. Master's thesis, Vrije Universiteit Brussel, Brussels, Belgium, 2002
- [19] Baroni, A. L. and e Abreu, F. B. A Formal Library for Aiding Metrics Extraction. In: *Proceedings of the International Workshop on Object-Oriented Re-Engineering at ECOOP 2003*. Darmstadt, Germany, 2003
- [20] Barr, P., Biddle, R. and Noble, J. A Taxonomy of User Interface Metaphors. In: *Proceedings of the SIGCHI New Zealand Symposium on Computer-Human Interaction*. IEEE Press, 2002
- [21] Barry, C. and Lang, M. A Survey of Multimedia and Web Development Techniques and Methodology Usage. *IEEE MultiMedia*, vol. 8(2), pp. 52–60, 2001
- [22] Baumeister, H., Knapp, A., Koch, N. and Zhang, G. Modelling Adaptivity with Aspects. In: *Proceedings of the 5th International Conference on Web Engineering (ICWE '05)*, pp. 406–416. 2005
- [23] Beck, K. *Test-Driven Development by Example*. Addison-Wesley, 2003
- [24] Beck, K., Gamma, E. and Saff, D. *JUnit API*, 2010  
<http://junit.sourceforge.net/javadoc/>
- [25] Bellettini, C., Marchetto, A. and Trentini, A. WebUml: reverse engineering of web applications. In: *Proceedings of the 2004 ACM Symposium on Applied Computing (SAC '04)*, pp. 1662–1669. ACM, New York, NY, USA, 2004
- [26] Bellettini, C., Marchetto, A. and Trentini, A. TestUml: User-Metrics Driven Web Applications Testing. In: *Proceedings of the 2005 ACM Symposium on Applied Computing (SAC '05)*, pp. 1694–1698. ACM Press, New York, NY, USA, 2005
- [27] Ben-Kiki, O., Evans, C. and dot Net, I. YAML Ain't Markup Language (YAML™) Version 1.2. Tech. rep., Third Edition, 2009  
<http://yaml.org/spec/1.2/spec.html>
- [28] Bennett, S., McRobb, S. and Farmer, R. *Object-Oriented Systems Analysis and Design Using UML*. McGraw-Hill, London, 2nd ed., 2002



- [29] Bergmann, G., Ökrös, A., Ráth, I., Varró, D. and Varró, G. Incremental pattern matching in the viatra model transformation system. In: *Proceedings of the 3rd International Workshop on Graph and Model Transformations*, GRaMoT '08, pp. 25–32. ACM, New York, NY, USA, 2008
- [30] Berners-Lee, T. and Fischetti, M. *Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web by Its Inventor*. Harper San Francisco, 1999
- [31] Berners-Lee, T., Masinter, L. and McCahill, M. Uniform Resource Locators (URL). RFC 1738 (Proposed Standard), 1994
- [32] Beyer, D., Noack, A. and Lewerentz, C. Efficient Relational Calculation for Software Analysis. *IEEE Transactions on Software Engineering*, vol. 31(2), pp. 137–149, 2005
- [33] Bézivin, J. In Search of a Basic Principle for Model Driven Engineering. *CEPIS, UPGRADE, The European Journal for the Informatics Professional*, vol. 5(2), pp. 21–24, 2004
- [34] Bézivin, J., Dupé, G., Jouault, F., Pitette, G. and Rougui, J. E. First Experiments with the ATL Model Transformation Language: Transforming XSLT into XQuery. In: *2nd OOPSLA Workshop on Generative Techniques in the Context of Model Driven Architecture*. 2003
- [35] Blanc, X., Gervais, M.-P. and Sriplakich, P. Model Bus: Towards the Interoperability of Modelling Tools. pp. 17–32. 2005
- [36] Blustein, J., Ahmed, I. and Instone, K. An Evaluation of Look-ahead breadcrumbs for the WWW. In: *Proceedings of the 16th ACM Conference on Hypertext and Hypermedia (HT '05)*, pp. 202–204. ACM, 2005
- [37] Boehm, B. W. *Software Engineering Economics*. Prentice-Hall, 1981
- [38] Boley, H. Integrating Positional and Slotted Knowledge on the Semantic Web. *Journal of Emerging Technologies in Web Intelligence*, vol. 2(4), pp. 343–353, 2010
- [39] Boswell, D., King, B., Oeschger, I., Collins, P. and Murphy, E. *Creating Applications with Mozilla*. O'Reilly Series. O'Reilly, 2002
- [40] Bozzon, A., Comai, S., Fraternali, P. and Carughi, G. T. Conceptual Modeling and Code Generation for Rich Internet Applications. In: *Proceedings of the 6th International Conference on Web Engineering (ICWE '06)*, pp. 353–360. ACM Press, New York, NY, USA, 2006
- [41] Brambilla, M., Cabot, J. and Moreno, N. Tool Support for Model Checking of Web Application Designs. In: Baresi, L., Fraternali, P. and Houben, G.-J., eds., *ICWE, Lecture Notes in Computer Science*, vol. 4607, pp. 533–538. Springer, 2007
- [42] Brambilla, M., Preciado, J. C., Linaje, M. and Sanchez-Figueroa, F. Business Process-Based Conceptual Design of Rich Internet Applications. In: *Proceedings of the 8th International Conference on Web Engineering (WISE '08)*, pp. 155–161. IEEE Computer Society, Washington, DC, USA, 2008
- [43] Breslin, J., Passant, A. and Decker, S. *The Social Semantic Web*. Springer-Verlag, Heidelberg, 2009

- [44] Bruck, J. and Hussey, K. Customizing UML: Which Technique is Right for You? Tech. rep., 2007  
[http://www.eclipse.org/modeling/mdt/uml2/docs/articles/Customizing\\_UML2\\_Which\\_Technique\\_is\\_Right\\_For\\_You/article.html](http://www.eclipse.org/modeling/mdt/uml2/docs/articles/Customizing_UML2_Which_Technique_is_Right_For_You/article.html)
- [45] Buckley, A. JSR 14: Add Generic Types to the Java Programming Language. Tech. rep., Java Community Process, 2004  
<http://jcp.org/en/jsr/detail?id=14>
- [46] Cardelli, L. A Semantics of Multiple Inheritance. *Information and Computation*, vol. 76(2/3), pp. 138–164, 1988
- [47] Cardelli, L. Type Systems. In: Tucker, A. B., ed., *The Computer Science and Engineering Handbook*, pp. 2208–2236. CRC Press, 1997
- [48] Cardellini, V., Casalicchio, E., Colajanni, M. and Yu, P. S. The State of the Art in Locally Distributed Web-server Systems. *ACM Computing Surveys*, vol. 34, pp. 263–311, 2002
- [49] Carstensen, P. H. and Vogelsang, L. Design of Web-Based Information Systems - New Challenges for Systems Development? In: *Proceedings of the 9th European Conference on Information Systems (ECIS)*. 2001
- [50] Ceri, S., Fraternali, P. and Bongio, A. Web Modeling Language (WebML): A Modeling Language for Designing Web Sites. In: *Proceedings of the 9th International World Wide Web Conference on Computer Networks*, pp. 137–157. North-Holland Publishing Co., Amsterdam, The Netherlands, The Netherlands, 2000
- [51] Ceri, S., Fraternali, P., Bongio, A., Brambilla, M., Comai, S. and Matera, M. *Designing Data-Intensive Web Applications*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002
- [52] Ceri, S., Fraternali, P., Maurino, A. and Paraboschi, S. One-to-One Personalization of Data-Intensive Web Sites. In: *WebDB (Informal Proceedings)*, pp. 1–6. 1999
- [53] Chauvin, M., Tikhomirov, A. and Shatalin, A. GMF 2.2 New and Noteworthy: 2.2 M4, 2010  
[http://wiki.eclipse.org/GMF\\_2.2\\_New\\_and\\_Noteworthy#2.2\\_M4](http://wiki.eclipse.org/GMF_2.2_New_and_Noteworthy#2.2_M4)
- [54] Chen, P. P.-S. The Entity-Relationship Model – Toward a Unified View of Data. *ACM Trans. Database Syst.*, vol. 1, pp. 9–36, 1976
- [55] Chikofsky, E. J. and Cross II, J. H. Reverse Engineering and Design Recovery: A Taxonomy. *Software, IEEE*, vol. 7(1), pp. 13–17, 1990
- [56] Cimatti, A., Giunchiglia, E., Pistore, M., Roveri, M., Sebastiani, R. and Tacchella, A. Integrating BDD-based and SAT-based Symbolic Model Checking. In: *Proceedings of Frontiers of Combining Systems (FROCOS'02), LNAI*, vol. 2309. Springer, Santa Margherita, Italy, 2002
- [57] Clements, P. C. A Survey of Architecture Description Languages. In: *Proceedings of the 8th International Workshop on Software Specification and Design, IWSSD '96*, pp. 16–25. IEEE Computer Society, Washington, DC, USA, 1996

- [58] Collins-Sussman, B., Fitzpatrick, B. W. and Pilato, C. M. *Version Control with Subversion*. O'Reilly Media, Inc., 2004
- [59] Conallen, J. *Building Web applications with UML*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000
- [60] Cook, S. UML 2.4 is complete. WebLog, 2011  
<http://blogs.msdn.com/b/stevecook/archive/2011/01/26/uml-2-4-is-complete.aspx>
- [61] Cook, S., Jones, G., Kent, S. and Wills, A. *Domain-Specific Development with Visual Studio DSL Tools*. Addison-Wesley Professional, 2007
- [62] Cowderoy, A. Measures of Size and Complexity for Web-site Content. In: *Proceedings of the Combined 11th ESCOM Conference and the 3rd SCOPE Conference on Software Product Quality*, pp. 423–431. Munich, Germany, 2000
- [63] Crespo, Y., Marqués, J. M. and Rodríguez, J. J. On the Translation of Multiple Inheritance Hierarchies into Single Inheritance Hierarchies. In: Black, A., Ernst, E., Grogono, P. and Sakkinen, M., eds., *Proceedings of the Inheritance Workshop at ECOOP 2002*, pp. 30–37. Jyväskylä University Press, 2002
- [64] Creswell, J. W. *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches*. Sage Publications, 3rd ed., 2009
- [65] Crockford, D. The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627 (Informational), 2006
- [66] Czarnecki, K. and Helsen, S. Feature-based Survey of Model Transformation Approaches. *IBM Systems Journal*, vol. 45(3), pp. 621–645, 2006
- [67] da Silva, P. P. and Paton, N. W. A UML-Based Design Environment for Interactive Applications. In: *Proceedings of the 2nd International Workshop on User Interfaces to Data Intensive Systems (UIDIS '01)*, p. 60. IEEE Computer Society, Washington, DC, USA, 2001
- [68] da Silva, P. P. and Paton, N. W. User Interface Modeling in UMLi. *IEEE Software*, vol. 20, pp. 62–69, 2003
- [69] David, M. *Flash Mobile: Developing Android and iOS Applications*. Focal Press, 2011
- [70] de Boursetty, B. Using Gmail, Calendar and Docs without an Internet connection. Blog, 2011  
<http://gmailblog.blogspot.com/2011/08/using-gmail-calendar-and-docs-without.html>
- [71] Deitel, H. M. and Deitel, P. J. *Java<sup>TM</sup>: How to Program*. Prentice Hall Press, Upper Saddle River, NJ, USA, 7th ed., 2006
- [72] Demuth, B. The Dresden OCL Toolkit and its Role in Information Systems Development. In: *Proceedings of the 13th International Conference on Information Systems Development (ISD '04)*. Vilnius, Lithuania, 2004

- [73] Dennis, G. Re: Recursion in Alloy 3. Mailing List, 2004  
<http://permalink.gmane.org/gmane.comp.lang.alloy.general/30>
- [74] Deprez, J.-C. and Alexandre, S. Comparing Assessment Methodologies for Free/Open Source Software: OpenBRR and QSOS. *Product-Focused Software Process Improvement*, pp. 189–203, 2008
- [75] Deutsch, A., Sui, L. and Vianu, V. Specification and Verification of Data-driven Web Services. In: *Proceedings of the 23rd Symposium on Principles of Database Systems (PODS '04)*, pp. 71–82. ACM Press, New York, NY, USA, 2004
- [76] Di Martino, S., Ferrucci, F., Paolino, L., Sebillo, M., Tortora, G., Vitiello, G. and Avagliano, G. Towards the Automatic Generation of Web GIS. In: *Proceedings of the 15th Annual ACM International Symposium on Advances in Geographic Information Systems (GIS '07)*. ACM, New York, NY, USA, 2007
- [77] Dietrich, J., Hiller, J. and Schenke, B. Take - A Rule Compiler for Derivation Rules. In: Paschke, A. and Biletskiy, Y., eds., *RuleML, Lecture Notes in Computer Science*, vol. 4824, pp. 134–148. Springer, Orlando, Florida, 2007
- [78] Ditchendorf, T. Fluid, 2010  
<http://www.fluidapp.com/>
- [79] Dittrich, K. R. and Gatzju, S. Time Issues in Active Database Systems. In: *Proceedings of the International Workshop on an Infrastructure for Temporal Databases*. Arlington, Texas, 1993
- [80] Djurić, D., Gašević, D. and Devedžić, V. The Tao of Modeling Spaces. *Object Technology*, vol. 5(8), 2006
- [81] Dojo Foundation. The Dojo Toolkit, 2007  
<http://dojotoolkit.org/>
- [82] Eclipse Foundation. Eclipse Public License, Version 1.0, 2004  
<http://www.eclipse.org/legal/epl-v10.html>
- [83] Eclipse Foundation. Eclipse Foundation, Inc. Intellectual Property Policy. Tech. rep., Eclipse Foundation, Inc., 2008  
[http://www.eclipse.org/org/documents/Eclipse\\_IP\\_Policy.pdf](http://www.eclipse.org/org/documents/Eclipse_IP_Policy.pdf)
- [84] Eclipse Foundation. Eclipse Modeling Framework Project (EMF): Validation Framework, 2008  
<http://www.eclipse.org/modeling/emf/?project=validation>
- [85] Eclipse Foundation. Eclipse Projects, 2008  
<http://www.eclipse.org/projects/listofprojects.php>
- [86] ECMA International. ECMA-262: ECMAScript Language Definition, Edition 3. Tech. rep., ECMA International, 1999
- [87] Efftinge, S., Friese, P., Haase, A., Hübner, D., Kadura, C., Kolb, B., Köhnlein, J., Moroff, D., Thoms, K., Völter, M., Schönbach, P., Eysholdt, M. and Reinisch, S. *openArchitectureWare Documentation*, 2008

- [88] Eichinger, C. Web Application Architectures. In: Kappel, G., Pröll, B., Reich, S. and Retschitzegger, W., eds., *Web Engineering – The Discipline of Systematic Development of Web Applications*, chap. 4. John Wiley & Sons Ltd., England, 2006
- [89] Eshuis, R. and Wieringa, R. A Real-Time Execution Semantics for UML Activity Diagrams. In: *Proceedings of the 4th International Conference on Fundamental Approaches to Software Engineering (FASE '01)*, pp. 76–90. Springer-Verlag, London, UK, 2001
- [90] European Smalltalk User Group. About Seaside, 2008  
<http://seaside.st/about>
- [91] Fahl, W. *Marama Concepts*, 2007  
<https://wiki.auckland.ac.nz/display/csidst/Marama+Concepts>
- [92] Favre, J.-M. Foundations of Meta-Pyramids: Languages vs. Metamodels – Episode II: Story of Thotus the Baboon. *Language Engineering for Model-Driven Software Development*, vol. 4101, 2005
- [93] Fayad, M. and Schmidt, D. C. Object-oriented Application Frameworks. *Commun. ACM*, vol. 40(10), pp. 32–38, 1997
- [94] Fenton, N. E. and Pfleeger, S. L. *Software Metrics: A Rigorous and Practical Approach*. International Thompson computer, 2 ed., 1997
- [95] Ferraiolo, D. F., Sandhu, R., Gavrila, S., Kuhn, D. R. and Chandramouli, R. Proposed NIST Standard for Role-Based Access Control. *ACM Trans. Inf. Syst. Secur.*, vol. 4, pp. 224–274, 2001  
<http://doi.acm.org/10.1145/501978.501980>
- [96] Fielding, R. T. and Taylor, R. N. Principled Design of the Modern Web Architecture. *ACM Trans. Internet Technol.*, vol. 2, pp. 115–150, 2002
- [97] Filman, R. E., Elrad, T., Clarke, S. and Akşit, M., eds. *Aspect-Oriented Software Development*. Addison-Wesley, Boston, 2005
- [98] Fitting, M. *First-Order Logic and Automated Theorem Proving*. Springer-Verlag New York, Inc., New York, NY, USA, 1990
- [99] Flanagan, D. *JavaScript: The Definitive Guide*. Definitive Guide Series. O'Reilly, 2006
- [100] Fowler, M. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, MA, USA, 1999
- [101] Fowler, M. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002
- [102] Fowler, M. *Domain-Specific Languages*. Addison-Wesley Professional, 2010
- [103] France, R. B., Ray, I., Georg, G. and Ghosh, S. Aspect-oriented approach to early design modelling. *IEEE Software*, vol. 151(4), pp. 173–186, 2004

- [104] Fraternali, P. Tools and Approaches for Developing Data-intensive Web Applications: A Survey. *ACM Computing Surveys*, vol. 31(3), pp. 227–263, 1999
- [105] Fraternali, P., Comai, S., Bozzon, A. and Carughi, G. T. Engineering Rich Internet Applications with a Model-Driven Approach. *ACM Trans. Web*, vol. 4, pp. 7:1–7:47, 2010
- [106] Free Software Foundation. The GNU General Public License v3.0, 2007  
<http://www.gnu.org/licenses/gpl.html>
- [107] Free Software Foundation. GCC Runtime Library Exception, 2009  
<http://www.gnu.org/licenses/gcc-exception.html>
- [108] Free Software Foundation. Frequently Asked Questions about the GNU Licenses, 2011  
<http://www.gnu.org/licenses/gpl-faq.html>
- [109] Free Software Foundation. Various Licenses and Comments about Them, 2011  
<http://www.gnu.org/licenses/license-list.html>
- [110] Freed, N. and Borenstein, N. Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies. RFC 2045 (Draft Standard), 1996
- [111] Friedman-Hill, E. Re: JESS: Multi-inheritance. Mailing List, 2001  
<http://www.mail-archive.com/jess-users@sandia.gov/msg03143.html>
- [112] Friedman-Hill, E. J. *Jess, The Rule Engine for the Java Platform*, 2005  
<http://www.jessrules.com/jess/docs/61/>
- [113] Fu, Y., Dong, Z. and He, X. Formalizing and Validating UML Architecture Description of Web Systems. In: *Workshop Proceedings of the 6th International Conference on Web Engineering (ICWE '06)*. ACM, New York, NY, USA, 2006
- [114] Fuentes-Fernández, L. and Vallecillo-Moreno, A. An Introduction to UML Profiles. *The European Journal for the Informatics Professional*, vol. 5(2), 2004  
<http://www.upgrade-cepis.org/issues/2004/2/up5-2Editorial.pdf>
- [115] Gabbay, D. M., Hogger, C. J. and Robinson, J. A., eds. *Handbook of Logic in Artificial Intelligence and Logic Programming*, vol. 3. Oxford University Press, Inc., New York, NY, USA, 1994
- [116] Gaedke, M., Grossniklaus, M. and Díaz, O., eds. *Web Engineering, 9th International Conference, ICWE 2009, San Sebastián, Spain, June 24-26, 2009, Proceedings, Lecture Notes in Computer Science*, vol. 5648. Springer, 2009
- [117] Gamma, E. and Beck, K. *Contributing to Eclipse: Principles, Patterns, and Plugins*. Addison-Wesley, 2003
- [118] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995
- [119] Gane, N. and Beer, D. *New Media: The Key Concepts*. Berg Publishers, 2008



- [120] Garfinkel, S. and Spafford, G. *Web Security, Privacy and Commerce*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2nd ed., 2001
- [121] Garrett, J. J. Ajax: A New Approach to Web Applications. Tech. rep., 2005  
<http://www.adaptivepath.com/publications/essays/archives/000385.php/>
- [122] Gašević, D., Kaviani, N. and Hatala, M. On Metamodeling in Megamodels. In: Engels, G., Opdyke, B., Schmidt, D. C. and Weil, F., eds., *MoDELS, Lecture Notes in Computer Science*, vol. 4735, pp. 91–105. Springer, 2007
- [123] Giese, H. and Wagner, R. Incremental Model Synchronization with Triple Graph Grammars. In: Nierstrasz, O., Whittle, J., Harel, D. and Reggio, G., eds., *Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS), LNCS*, vol. 4199, pp. 543–557. Springer Verlag, 2006
- [124] Gitzel, R., Korthaus, A. and Schader, M. Using Established Web Engineering Knowledge in Model-Driven Approaches. *Sci. Comput. Program.*, vol. 66(2), pp. 105–124, 2007
- [125] Giurca, A. R2ML – The REWERSE I1 Rule Markup Language. Tech. rep., REWERSE Working Group I1, 2006  
<http://oxygen.informatik.tu-cottbus.de/reverse-i1/?q=R2ML>
- [126] Giurca, A. and Wagner, G. Rule Modeling and Interchange. In: *Proceedings of the 9th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2007)*, pp. 485–491. 2007
- [127] González, A. S., Ruiz, D. S. and Perez, G. M. EMF4CPP: a C++ Ecore Implementation. *DSDM 2010*, 2010
- [128] Google. Google Chrome, 2011  
<http://www.google.com/chrome>
- [129] Google Inc. Google Gears, 2007  
<http://gears.google.com>
- [130] Gosling, J., Joy, B., Steele, G. and Bracha, G. *The Java Language Specification*. 3rd ed., 2005
- [131] Gottlob, G., Koch, C. and Pichler, R. Efficient Algorithms for Processing XPath Queries. *ACM Trans. Database Syst.*, vol. 30, pp. 444–491, 2005
- [132] Grand, M. *Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with UML*, vol. 1. John Wiley & Sons, Inc., New York, NY, USA, 2002
- [133] Green, T. R. G. and Petre, M. Usability Analysis of Visual Programming Environments: A ‘Cognitive Dimensions’ Framework. *Journal of Visual Languages and Computing*, vol. 7(2), pp. 131–174, 1996
- [134] Groenewegen, D. M., Hemel, Z., Kats, L. C. and Visser, E. Webdsl: a domain-specific language for dynamic web applications. In: *Companion to the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '08)*, pp. 779–780. ACM, New York, NY, USA, 2008

- [135] Groenewegen, D. M. and Visser, E. Declarative Access Control for WebDSL: Combining Language Integration and Separation of Concerns. In: Schwabe, D. and Curbera, F., eds., *Proceedings of the 8th International Conference on Web Engineering (ICWE '08)*, pp. 175–188. IEEE CS Press, 2008
- [136] Gronback, R. C. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley Professional, 2009
- [137] Grundy, J., Hosking, J., Zhu, N. and Liu, N. Generating Domain-Specific Visual Language Editors from High-level Tool Specifications. In: *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE '06)*, pp. 25–36. IEEE Computer Society, Washington, DC, USA, 2006
- [138] Grundy, J. C., Hosking, J. G., Amor, R. W., Mugridge, W. B. and Li, Y. Domain-Specific Visual Languages for Specifying and Generating Data Mapping Systems. *Journal of Visual Languages & Computing*, vol. 15(3-4), pp. 243 – 263, 2004
- [139] Grundy, J. C., Hosking, J. G., Huh, J. and Li, K. N.-L. Marama: an Eclipse Meta-toolset for Generating Multi-view Environments. In: *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*, pp. 819–822. 2008
- [140] Gu, A., Henderson-Sellers, B. and Lowe, D. Web Modelling Languages: The Gap Between Requirements and Current Exemplars. In: *Proceedings of the 8th Australian World Wide Web Conference (AusWeb02)*. 2002
- [141] Gurevich, Y. Evolving Algebras 1993: Lipari Guide. In: Börger, E., ed., *Specification and Validation Methods*, pp. 9–36. Oxford University Press, Inc., New York, NY, USA, 1995
- [142] Haarslev, V. and Möller, R. Description of the RACER System and its Applications. In: Goble, C. A., McGuinness, D. L., Möller, R. and Patel-Schneider, P. F., eds., *Description Logics, CEUR Workshop Proceedings*, vol. 49. CEUR-WS.org, 2001
- [143] Haggett, P. and Chorley, R. J. Models, Paradigms and the New Geography. *Socioeconomic Models in Geography*, 1967
- [144] Hailpern, B. and Tarr, P. Model-driven development: The good, the bad, and the ugly. *IBM Systems Journal*, vol. 45, pp. 451–461, 2006
- [145] Hammer-Lahav, E. The OAuth 1.0 Protocol. RFC 5849 (Draft Standard), 2010
- [146] Hansen, T. and Vaudreuil, G. Message Disposition Notification. RFC 3798 (Draft Standard), 2004
- [147] Hanson, R. and Tacy, A. *GWT in Action: Easy Ajax with the Google Web Toolkit*. Wiley, Greenwich, CT, USA, 2007
- [148] Harel, D. and Rumpe, B. Modeling Languages: Syntax, Semantics and All That Stuff, Part I: The Basic Stuff. Tech. rep., 2000
- [149] Havelund, K. and Pressburger, T. Model Checking Java Programs using Java PathFinder. *Software Tools for Technology Transfer*, vol. 2(4), pp. 366–381, 2000



- [150] Heidenreich, F., Johannes, J., Karol, S., Seifert, M. and Wende, C. Derivation and Refinement of Textual Syntax for Models. In: *5th European Conference on Model-Driven Architecture Foundations and Applications (ECMDA-FA 2009)*. Enschede, The Netherlands, 2009
- [151] Henderson-Sellers, B. Bridging Metamodels and Ontologies in Software Engineering. *Journal of Systems and Software*, vol. 84(2), pp. 301–313, 2011
- [152] Henry, J., Lane, J. and Wright, J. JWebUnit, 2011  
<http://jwebunit.sourceforge.net/>
- [153] Hettel, T., Lawley, M. and Raymond, K. Model Synchronisation: Definitions for Round-Trip Engineering. In: *Proceedings of the First International Conference on Model Transformation (ICMT '08)*. Springer, 2008
- [154] Hillairet, G. emftriple: (Meta)Models on the Web of Data, 2011  
<http://code.google.com/p/emftriple/>
- [155] Hoare, C. A. R. An Axiomatic Basis for Computer Programming. *Commun. ACM*, vol. 12, pp. 576–580, 1969
- [156] Hoare, C. A. R. Hints on Programming Language Design. Tech. rep., Stanford, CA, USA, 1973
- [157] Huck, J., Morris, D., Ross, J., Knies, A., Mulder, H. and Zahir, R. Introducing the IA-64 Architecture. *Micro, IEEE*, vol. 20(5), pp. 12–23, 2000
- [158] Huh, J., Grundy, J., Hosking, J., Liu, K. and Amor, R. Integrated Data Mapping for a Software Meta-tool. In: *Proceedings of the Doctoral Symposium at the 20th Australian Software Engineering Conference (ASWEC 2009)*, pp. 111–120. 2009
- [159] Hunter, J. and Crawford, W. *Java Servlet Programming*. Java series. O'Reilly, 2nd ed., 2001
- [160] Huth, M. and Ryan, M. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2nd ed., 2004
- [161] IEEE. *IEEE Software Engineering Standards: Standard 610.12-1990*, 1990
- [162] Ihaka, R. and Gentleman, R. R: A Language for Data Analysis and Graphics. *Journal of Computational and Graphical Statistics*, vol. 5(3), pp. 299–314, 1996
- [163] Irawan, H., Stoll, D., Efftinge, S., Jockel, D. and Zarnekow, S. Xpand: New and Noteworthy, 2010  
[http://wiki.eclipse.org/Xpand/New\\_And\\_Noteworthy](http://wiki.eclipse.org/Xpand/New_And_Noteworthy)
- [164] ISO. *ISO 8879: Information processing — Text and office systems — Standard Generalized Markup Language (SGML)*. International Organization for Standardization, Geneva, Switzerland, 1986
- [165] ISO. *ISO/IEC 14977:1996: Information technology — Syntactic metalanguage — Extended BNF*. International Organization for Standardization, Geneva, Switzerland, 1996
- [166] ISO/IEC. *ISO/IEC 9126: Software Engineering – Product Quality, Part 1: Quality Model*. International Organization for Standardization, Geneva, Switzerland, 2001

- [167] ISO/IEC. *ISO/IEC 9126: Software Engineering – Product Quality, Part 3: Internal Metrics*. International Organization for Standardization, Geneva, Switzerland, 2002
- [168] ISO/IEC. *ISO/IEC 19503: Information technology – XML Metadata Interchange (XMI)*. International Organization for Standardization, Geneva, Switzerland, 2005
- [169] Jackson, D. *Software Abstractions: Logic, Language, and Analysis*. Cambridge, Mass., 2006
- [170] JetBrains. JetBrains Meta Programming System  
<http://www.jetbrains.com/mps/>
- [171] Johnson, R. J2EE Development Frameworks. *Computer*, vol. 38(1), pp. 107–110, 2005
- [172] Johnson, R., Hoeller, J., Arendsen, A., Risberg, T. and Sampaleanu, C. *Professional Java Development with the Spring Framework*. Wrox, Birmingham, UK, UK, 2005
- [173] Jouault, F. and Kurtev, I. Transforming Models with ATL. In: *Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005*. Jamaica, 2005
- [174] Jouault, F. and Kurtev, I. On the Architectural Alignment of ATL and QVT. In: *Proceedings of the 2006 ACM Symposium on Applied Computing (SAC '06)*, pp. 1188–1195. ACM Press, New York, NY, USA, 2006
- [175] Kappel, G., Pröll, B., Reich, S. and Retschitzegger, W. An Introduction to Web Engineering. In: Kappel, G., Pröll, B., Reich, S. and Retschitzegger, W., eds., *Web Engineering – The Discipline of Systematic Development of Web Applications*, chap. 1, pp. 1–22. John Wiley & Sons Ltd., England, 2006
- [176] Kappel, G., Retschitzegger, W. and Schwinger, W. Modeling Ubiquitous Web Applications: The WUML approach. In: *International Workshop on Data Semantics in Web Information Systems (DASWIS-2001)*. Yokohama, Japan, 2001
- [177] Karsai, G., Krahn, H., Pinkernell, C., Rumpe, B., Schneider, M. and Völkel, S. Design Guidelines for Domain Specific Languages. In: Rossi, M., Sprinkle, J., Gray, J. and Tolvanen, J.-P., eds., *Proceedings of the 9th OOPSLA Workshop on Domain-Specific Modeling (DSM '09)*, pp. 7–13
- [178] Kelly, S., Lyytinen, K. and Rossi, M. MetaEdit+: A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment. In: Constantopoulos, P., Mylopoulos, J. and Vassiliou, Y., eds., *CAiSE, Lecture Notes in Computer Science*, vol. 1080, pp. 1–21. Springer, 1996
- [179] Kelly, S. and Pohjonen, R. Worst Practices for Domain-Specific Modeling. *IEEE Software*, vol. 26, pp. 22–29, 2009
- [180] Kent, S. Model Driven Engineering. In: *Proceedings of the 3rd International Conference on Integrated Formal Methods (IFM '02)*, pp. 286–298. Springer-Verlag, London, UK, 2002
- [181] Kessler, P. Why can't I allocate 2GB of heap to the JVM on Windows, Part 2. Blog, 2004  
[http://blogs.oracle.com/moazam/entry/why\\_can\\_t\\_i\\_allocate](http://blogs.oracle.com/moazam/entry/why_can_t_i_allocate)
- [182] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M. and Irwin, J. Aspect-Oriented Programming. In: *ECOOP*, pp. 220–242. 1997

- [183] Kingsley-Hughes, A., Kingsley-Hughes, K. and Read, D. *VBScript: Programmer's Reference*. Wrox. Wrox/Wiley Pub., 2007
- [184] Klensin, J. Simple Mail Transfer Protocol. RFC 2821 (Proposed Standard), 2001
- [185] Kleppe, A. G., Warmer, J. and Bast, W. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003
- [186] Kline, K. *SQL in a Nutshell*. O'Reilly Media, Inc., 2001
- [187] Knapp, A., Koch, N., Moser, F. and Zhang, G. ArgoUWE: A CASE Tool for Web Applications. In: *First International Workshop on Engineering Methods to Support Information Systems Evolution (EMSISE 2003)*. 2003
- [188] Kobryn, C. UML 2001: A Standardization Odyssey. *Commun. ACM*, vol. 42, pp. 29–37, 1999
- [189] Koch, N. Transformation Techniques in the Model-Driven Development Process of UWE. In: *Proceedings of the 6th International Conference on Web Engineering (ICWE '06)*. ACM, New York, NY, USA, 2006
- [190] Koch, N. Classification of Model Transformation Techniques Used in UML-based Web Engineering. *Software, IET*, vol. 1(3), pp. 98–111, 2007
- [191] Koch, N., Baumeister, H., Hennicker, R. and Mandel, L. Extending UML to Model Navigation and Presentation in Web Applications. In: *Workshop on Modelling Web Applications in UML, UML 2000, York, UK*. 2000
- [192] Koch, N. and Kraus, A. The Expressive Power of UML-based Web Engineering. In: *IW-WOST'02*, pp. 105–119. 2002
- [193] Koch, N., Pigerl, M., Zhang, G. and Morozova, T. Patterns for the Model-Based Development of RIAs. In: Gaedke et al. [116], pp. 283–291
- [194] Koch, P.-P. Conditional comments, 2009  
<http://www.quirksmode.org/css/condcom.html>
- [195] Krasner, G. E. and Pope, S. T. A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System. *Object Oriented Programming*, vol. 1(3), pp. 26–49, 1988
- [196] Kroiß, C. and Koch, N. *The UWE Metamodel and Profile – User Guide and Reference*, 2008
- [197] Kroiss, C., Koch, N. and Knapp, A. UWE4JSF: A Model-Driven Generation Approach for Web Applications. In: Gaedke et al. [116], pp. 493–496
- [198] Kühne, T. Matters of (Meta-) Modeling. *Software and Systems Modeling*, vol. 5, pp. 369–385, 2006
- [199] Kuleshov, E. and Peter, K. *Eclipse UI Graphics: Style & Design*. The Eclipse Foundation, 2008  
[http://wiki.eclipse.org/UI\\_Graphics:\\_Design:\\_Style\\_&\\_Design](http://wiki.eclipse.org/UI_Graphics:_Design:_Style_&_Design)

- [200] Kuuskeri, J. and Mikkonen, T. REST Inspired Code Partitioning with a JavaScript Middleware. In: Daniel, F. and Facca, F., eds., *Current Trends in Web Engineering, Lecture Notes in Computer Science*, vol. 6385, pp. 244–255. Springer Berlin / Heidelberg, 2010
- [201] Lakoff, G. and Johnson, M. *Metaphors we live by*, vol. 111. Chicago London, 1980
- [202] Lang, M. and Fitzgerald, B. Hypermedia Systems Development Practices: A Survey. *IEEE Software*, vol. 22(2), pp. 68–75, 2005
- [203] Lano, K. and Bicarregui, J. Semantics and Transformations for UML Models. In: Bézivin, J. and Muller, P.-A., eds., *UML'98 - Beyond the Notation, LNCS*, vol. 1618, pp. 107–119. Springer, 1999
- [204] Lawton, G. Invasive Software: Who's Inside Your Computer? *Computer*, vol. 35, pp. 15–18, 2002
- [205] Leavitt, N. Will NoSQL Databases Live Up to Their Promise? *Computer*, vol. 43, pp. 12–14, 2010
- [206] Ledeczki, A., Maroti, M., Bakay, A., Karsai, G., Garrett, J., Thomason, C., Nordstrom, G., Sprinkle, J. and Volgyesi, P. The Generic Modeling Environment. In: *Workshop on Intelligent Signal Processing*. IEEE, 2001
- [207] Lerdorf, R., Tatroe, K. and MacIntyre, P. B. *Programming PHP*. O'Reilly, 2nd ed., 2006
- [208] Lerner, R. M. At the Forge: jQuery Plugins. *Linux J.*, vol. 2009, 2009
- [209] Li, K., Hosking, J. and Grundy, J. A Generalised Event Handling Framework. In: *Proceedings of the Knowledge Industry Survival Strategy Workshop at ASE 2009*. 2009
- [210] Li, L. *The VisualAge for Smalltalk Primer*. Advances in Object Technology. Cambridge University Press, 1998
- [211] Liang, S., Fodor, P., Wan, H. and Kifer, M. OpenRuleBench: An Analysis of the Performance of Rule Engines. In: Quemada, J., León, G., Maarek, Y. S. and Nejdl, W., eds., *WWW*, pp. 601–610. ACM, 2009
- [212] Lieberherr, K. J. and Holland, I. Assuring Good Style for Object-Oriented Programs. *Software, IEEE*, vol. 6(5), pp. 38–48, 2005
- [213] Liu, K. *Marama Meta-Tools User Manual*, 2008  
<http://www.cs.auckland.ac.nz/Nikau/marama/tutorial/MaramaMetaTools-UserManual.html>
- [214] Liu, N., Hosking, J. and Grundy, J. MaramaTatau: Extending a Domain Specific Visual Language Meta Tool with a Declarative Constraint Mechanism. In: *Proceedings of the 2007 IEEE Symposium on Visual Languages and Human Centric Computing (VL/HCC '07)*
- [215] Liu, N., Hosking, J. and Grundy, J. A Visual Language and Environment for Specifying User Interface Event Handling in Design Tools. In: *AUIC '07: Proceedings of the Eighth Australasian Conference on User Interface*, pp. 87–94. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 2007

- [216] Manes, A. T. *Web Services: A Manager's Guide*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003
- [217] Manolescu, I., Brambilla, M., Ceri, S., Comai, S. and Fraternali, P. Model-driven design and deployment of service-enabled Web applications. *ACM Trans. Inter. Tech.*, vol. 5(3), pp. 439–479, 2005
- [218] Manovich, L. *The Language of New Media*. The MIT Press, 2001
- [219] Marinilli, M. *Java deployment with JNLP and WebStart*. Kaleidoscope Series. Sams, 2002
- [220] Massoni, T., Gheyi, R. and Borba, P. A Model-Driven Approach to Formal Refactoring. In: *Companion to the 20th ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '05)*, pp. 124–125. ACM, New York, NY, USA, 2005
- [221] McBride, B. Jena: Implementing the RDF Model and Syntax Specification. In: *SemWeb*. 2001
- [222] McConnell, S. *Rapid Development: Taming Wild Software Schedules*. Microsoft Press, Redmond, WA, USA, 1st ed., 1996
- [223] Mendelson, E. *Introduction to Mathematical Logic*. Chapman & Hall/CRC, 1997
- [224] Mendes, E., Counsell, S. and Mosley, N. Towards a Taxonomy of Hypermedia and Web Application Size Metrics. In: *Proceedings of the 5th International Conference on Web Engineering (ICWE '05)*, pp. 110–123. 2005
- [225] Merialdo, P., Atzeni, P. and Mecca, G. Design and Development of Data-Intensive Web Sites: The Araneus Approach. *ACM Trans. Inter. Tech.*, vol. 3(1), pp. 49–92, 2003
- [226] Mernik, M., Heering, J. and Sloane, A. M. When and how to develop domain-specific languages. *ACM Computing Surveys*, vol. 37(4), pp. 316–344, 2005
- [227] Meservy, T. O. and Fenstermacher, K. D. Transforming Software Development: An MDA Road Map. *IEEE Computer*, vol. 38(9), pp. 52–58, 2005
- [228] Metz, C. AAA Protocols: Authentication, Authorization, and Accounting for the Internet. *Internet Computing, IEEE*, vol. 3(6), pp. 75–79, 1999
- [229] Meyer, B. *Object-Oriented Software Construction*. 2 ed.
- [230] Miao, H. and Zeng, H. Model Checking-based Verification of Web Application. In: *Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007)*, pp. 47–55. IEEE Computer Society, Washington, DC, USA, 2007
- [231] Michaelson, J. There's No Such Thing as a Free (Software) Lunch. *Queue*, vol. 2, pp. 40–47, 2004
- [232] Microsoft Developer Network. Extend your DSL by using MEF. Tech. rep., Microsoft, 2010 <http://msdn.microsoft.com/en-us/library/ff972471.aspx>
- [233] Mikkonen, T. and Taivalsaari, A. Web Applications – Spaghetti Code for the 21st Century. *Proceedings of the 6th International Conference on Software Engineering Research, Management and Applications*, pp. 319–328, 2008

- [234] Mockus, A., Fielding, R. T. and Herbsleb, J. D. Two Case Studies of Open Source Software Development: Apache and Mozilla. *ACM Trans. Softw. Eng. Methodol.*, vol. 11, pp. 309–346, 2002
- [235] Monperrus, M., Jézéquel, J.-M., Champeau, J. and Hoeltzener, B. Measuring models. In: Rech, J. and Bunse, C., eds., *Model-Driven Software Development: Integrating Quality Assurance*. IDEA Group, 2008
- [236] Moody, D. L. The “Physics” of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering. *IEEE Transactions on Software Engineering*, vol. 35, pp. 756–779, 2009
- [237] Moore, K. Simple Mail Transfer Protocol (SMTP) Service Extension for Delivery Status Notifications (DSNs). RFC 3461 (Draft Standard), 2003
- [238] Moreno, N., Fraternali, P. and Vallecillo, A. WebML Modelling in UML. *IET Software*, vol. 1(3), pp. 67–80, 2007
- [239] Morse, J. M. Approaches to Qualitative-Quantitative Methodological Triangulation. *Nursing Research*, vol. 40(2), pp. 120–123, 1991
- [240] Motik, B. On the Properties of Metamodeling in OWL. *J. Log. Comput.*, vol. 17(4), pp. 617–637, 2007
- [241] Mozilla Foundation. Mozilla Firefox, 2008  
<http://www.mozilla.org/en-US/firefox/3.0/releasenotes/>
- [242] Mozilla Foundation. Mozilla Prism, 2009  
<http://prism.mozillalabs.com/>
- [243] Mukerji, J. and Miller, J. Model-Driven Architecture Guide, v1.0.1. Tech. rep., Object Management Group, 2003  
<http://www.omg.org/cgi-bin/doc?omg/03-06-01>
- [244] Nagappan, N. and Ball, T. Evidence-Based Failure Prediction. In: Oram, A. and Wilson, G., eds., *Making Software*. 2011
- [245] Nicolae, O., Giurca, A. and Wagner, G. On Interchange between Drools and Jess. *Informatica (Slovenia)*, vol. 32(4), pp. 383–396, 2008
- [246] Nussbaumer, M. and Gaedke, M. Technologies for Web Applications. In: Kappel, G., Pröll, B., Reich, S. and Retschitzegger, W., eds., *Web Engineering – The Discipline of Systematic Development of Web Applications*, chap. 6. John Wiley & Sons Ltd., England, 2006
- [247] Object Management Group. What Is OMG-UML and Why Is It Important? Tech. rep., Object Management Group, 1997  
<http://cgi.omg.org/news/pr97/umlprimer.html>
- [248] Object Management Group. The Common Object Request Broker: Architecture and Specification, v2.3.1. Tech. rep., 1999  
<http://www.omg.org/cgi-bin/doc?formal/99-10-07>



- [249] Object Management Group. OMG Unified Modeling Language Specification, v1.4. Tech. rep., 2001  
<http://www.omg.org/spec/UML/1.4/>
- [250] Object Management Group. XML Metadata Interchange (XMI), v2.1. Tech. rep., 2005  
<http://www.omg.org/technology/documents/formal/xmi.htm>
- [251] Object Management Group. Meta Object Facility (MOF) Core Specification, v2.0. Tech. rep., 2006  
<http://www.omg.org/cgi-bin/doc?formal/2006-01-01>
- [252] Object Management Group. Object Constraint Language Specification, v2.0. Tech. rep., 2006  
<http://www.omg.org/cgi-bin/doc?formal/2006-05-01>
- [253] Object Management Group. OMG Unified Modeling Language (OMG UML), Infrastructure, v2.1.2. Tech. rep., 2007  
<http://www.omg.org/spec/UML/2.1.2/>
- [254] Object Management Group. OMG Unified Modeling Language (OMG UML), Superstructure, v2.1.2. Tech. rep., 2007  
<http://www.omg.org/spec/UML/2.1.2/>
- [255] Object Management Group. MOF Model to Text Transformation Language, v1.0. Tech. rep., 2008  
<http://www.omg.org/spec/QVT/1.1/>
- [256] Object Management Group. Semantics of Business Vocabulary and Business Rules (SBVR), v1.0. Tech. rep., 2008  
<http://www.omg.org/spec/SBVR/1.0/>
- [257] Object Management Group. Business Process Modeling and Notation (BPMN) Version 1.2. Tech. rep., 2009  
<http://www.omg.org/spec/BPMN/1.2/PDF>
- [258] Object Management Group. Meta Object Facility (MOF) Core Specification, v2.4. Tech. rep., 2010  
<http://www.omg.org/spec/MOF/2.4/Beta2/>
- [259] Object Management Group. Object Constraint Language, v2.2. Tech. rep., 2010  
<http://www.omg.org/cgi-bin/doc?formal/2010-02-01>
- [260] Object Management Group. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, v1.1. Tech. rep., 2011  
<http://www.omg.org/spec/QVT/1.1/>
- [261] Olsina, L., Lafuente, G. and Pastor, O. Towards a Reusable Repository for Web Metrics. *J. Web Eng.*, vol. 1(1), pp. 61–73, 2002
- [262] Open Source Initiative. The Open Source Definition  
<http://www.opensource.org/docs/osd>

- [263] OpenBRR.org. Business Readiness Rating for Open Source. Tech. rep., OpenBRR.org, 2005  
<http://www.openbrr.org/>
- [264] Oracle Corporation. *Java SE 6 Documentation*, 2011  
<http://docs.oracle.com/javase/6/docs/>
- [265] Origin, A. Method for Qualification and Selection of Open Source Software (QSOS), v1.6. Tech. rep., 2006  
<http://www.qsos.org>
- [266] Owens, M. *The Definitive Guide to SQLite*. Apress, Berkely, CA, USA, 2006
- [267] Paige, R. F., Ostroff, J. S. and Brooke, P. J. Principles for Modeling Language Design. *Information and Software Technology*, vol. 42(10), pp. 665–675, 2000
- [268] Papamarkos, G., Poulouvassilis, A. and Wood, P. T. Event-Condition-Action Rule Languages for the Semantic Web. In: *Proceedings of the 2003 International Workshop on Semantic Web and Databases (SWDB '03)*
- [269] Park, R. E. Software Size Measurement: A Framework for Counting Source Statements. Tech. Rep. CMU/SEI-92-TR- 20, ESC-TR-92-20, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213, 1992
- [270] Park, R. E., Goethert, W. B. and Florac, W. A. Goal Driven Software Measurement - A Guidebook. Tech. Rep. CMU/SEI-96-BH-002, Software Engineering Institute, Carnegie Mellon University, 1996
- [271] Pastor, O., Fons, J., Pelechano, V. and Abrahão, S. Conceptual Modelling of Web Applications: The OOWS Approach. In: Mendes, E. and Mosley, N., eds., *Web Engineering*, pp. 277–302. Springer, 2006
- [272] Plotkin, G. D. The origins of structural operational semantics. *J. Log. Algebr. Program.*, vol. 60-61, pp. 3–15, 2004
- [273] Pollice, G. Using the Rational Unified Process for Small Projects: Expanding upon eXtreme Programming. Tech. rep., Rational Software, 2001
- [274] Potencier, F. and Zaninotto, F. *The Definitive Guide to Symphony*. Apress, 2007
- [275] Preciado, J. C., Linaje, M., Sanchez, F. and Comai, S. Hypermedia Systems Development: Do We Really Need New Methods? In: *Proceedings of the Informing Science + IT Education Conference (IS2002)*. Cork, Ireland, 2002
- [276] Preciado, J. C., Linaje, M., Sanchez, F. and Comai, S. Necessity of Methodologies to Model Rich Internet Applications. In: *Proceedings of the 7th IEEE International Symposium on Web Site Evolution (WSE '05)*, pp. 7–13. IEEE Computer Society, Washington, DC, USA, 2005
- [277] Preece, J., Rogers, Y. and Sharp, H. *Interaction Design: Beyond Human-Computer Interaction*. Wiley, Indianapolis, IN, 2002
- [278] Pressman, R. S. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Higher Education, 5th ed., 2001



- [279] Proctor, M., Neale, M., Frandsen, M., Jr., S. G., Tirelli, E., Meyer, F. and Verlaenen, K. Drools Documentation. Tech. rep., JBoss.org, 2008  
<http://downloads.jboss.com/drools/docs/4.0.7.19894.GA/html/index.html>
- [280] Prototype Core Team. Prototype JavaScript Framework, 2007  
<http://www.prototypejs.org/>
- [281] Recordon, D. and Reed, D. OpenID 2.0: A Platform for User-centric Identity Management. In: *Proceedings of the 2nd ACM Workshop on Digital identity management (DIM '06)*, pp. 11–16. ACM, New York, NY, USA, 2006
- [282] Reis, C. and Gribble, S. D. Isolating Web Programs in Modern Browser Architectures. In: *Proceedings of the 4th ACM European Conference on Computer Systems, EuroSys '09*, pp. 219–232. ACM, New York, NY, USA, 2009
- [283] Resnick, P. Internet Message Format. RFC 5322 (Draft Standard), 2008
- [284] Reynolds, D. *Jena 2 Inference support*, 2010  
<http://jena.sourceforge.net/inference/>
- [285] Ricca, F. and Tonella, P. Analysis and Testing of Web Applications. p. 0025. IEEE Computer Society, Los Alamitos, CA, USA, 2001
- [286] Robbins, J. E. and Redmiles, D. F. Cognitive Support, UML Adherence, and XMI Interchange in Argo/UML. *Information and Software Technology*, vol. 42(2), pp. 79–89, 2000
- [287] Rose, L. M., Paige, R. F., Kolovos, D. S. and Polack, F. A. An Analysis of Approaches to Model Migration. In *Proceedings of the Joint MoDSE-MCCM Workshop at MODELS '09*, pp. 6–15, 2009
- [288] Rossi, G. and Schwabe, D. Model-Based Web Application Development. In: Mendes, E. and Mosley, N., eds., *Web Engineering*, pp. 303–333. Springer, 2006
- [289] Rossi, M. and Brinkkemper, S. Complexity Metrics for Systems Development Methods and Techniques. *Inf. Syst.*, vol. 21(2), pp. 209–227, 1996
- [290] RSS Advisory Board. RSS 2.0 Specification. Tech. rep., RSS Advisory Board, 2007  
<http://www.rssboard.org/rss-specification>
- [291] Rumbaugh, J. E. Notation Notes: Principles for Choosing Notation. *JOOP*, vol. 9(2), pp. 11–14, 1996
- [292] Samarati, P. and di Vimercati, S. D. C. Access Control: Policies, Models, and Mechanisms. *Foundations of Security Analysis and Design*, pp. 137–196, 2001
- [293] Sandhu, R. S., Coyne, E. J., Feinstein, H. L. and Youman, C. E. Role-Based Access Control Models. *Computer*, vol. 29(2), pp. 38–47, 1996
- [294] Sargent, R. G. Verification and Validation of Simulation models. In: *Proceedings of the 37th conference on Winter simulation, WSC '05*, pp. 130–143. Winter Simulation Conference, 2005

- [295] Schauerhuber, A. *aspectUWA - Applying Aspect-Orientation to the Model-Driven Development of Ubiquitous Web Applications*. Ph.D. thesis, Institut für Softwaretechnik und Interaktive Systeme, Vienna, Austria, 2007
- [296] Schauerhuber, A., Wimmer, M., Kapsammer, E., Schwinger, W. and Retschitzegger, W. Bridging WebML to Model-Driven Engineering: From DTDs to MOF. *IET Software*, vol. 1(3), pp. 81–97, 2007
- [297] Schewe, K.-D. The Challenges in Web Information Systems Development in 15 Pictures (Invited Talk). In: *ISTA*, pp. 204–215. 2005
- [298] Schleicher, A. and Westfechtel, B. Beyond Stereotyping: Metamodeling Approaches for the UML. In: Sprague, Jr., R. H., ed., *Proc. 34th Annual Hawaii International Conference on System Sciences (HICSS-34)*. IEEE Computer Society, 2001
- [299] Schmidt, D. A. *Denotational Semantics: A Methodology for Language Development*. William C. Brown Publishers, Dubuque, IA, USA, 1986
- [300] Schmidt, D. C. Guest Editor's Introduction: Model-Driven Engineering. *Computer*, vol. 39(2), pp. 25–31, 2006
- [301] Schwabe, D. A Conference Review System. In: *First International Workshop on Web-Oriented Software Technology*. 2001
- [302] Schwinger, W. and Koch, N. Modeling Web Applications. In: Kappel, G., Pröll, B., Reich, S. and Retschitzegger, W., eds., *Web Engineering – The Discipline of Systematic Development of Web Applications*, chap. 3. John Wiley & Sons Ltd., England, 2006
- [303] Sciascio, E. D., Donini, F. M., Mongiello, M. and Piscitelli, G. AnWeb: A System for Automatic Support to Web Application Verification. In: *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering (SEKE '02)*, pp. 609–616. ACM, New York, NY, USA, 2002
- [304] Seaborne, A. RDQL - A Query Language for RDF. Tech. rep., W3C Member Submission 9 January 2004, 2004  
<http://www.w3.org/Submission/RDQL/>
- [305] Selic, B. The Pragmatics of Model-Driven Development. *IEEE Software*, vol. 20, pp. 19–25, 2003
- [306] Selman, D. JSR 94: Java Rule Engine API. Tech. rep., Java Community Process, 2004  
<http://jcp.org/en/jsr/detail?id=94>
- [307] Selmi, S. S., Kraïem, N. and Ghézala, H. H. B. Toward a Comprehension View of Web Engineering. In: *Proceedings of the 5th International Conference on Web Engineering (ICWE '05)*, pp. 19–29. 2005
- [308] Sendall, S. and Kozaczynski, W. Model transformation: The heart and soul of model-driven software development. *Software, IEEE*, vol. 20(5), pp. 42–45, Sept.-Oct. 2003

- [309] Siau, K. and Cao, Q. Unified Modeling Language (UML) - A Complexity Analysis. vol. 12, pp. 26–34, 2001
- [310] Sirin, E., Parsia, B., Grau, B. C., Kalyanpur, A. and Katz, Y. Pellet: A Practical OWL-DL Reasoner. *J. Web Sem.*, vol. 5(2), pp. 51–53, 2007
- [311] Sottara, D., Mello, P. and Proctor, M. Adding Uncertainty to a Rete-OO Inference Engine. In: *Proceedings of the RuleML International Symposium (RuleML 2008)*, pp. 104–118. 2008
- [312] SpringSource. Spring.NET Application Framework, 2007  
<http://www.springframework.net/>
- [313] Sprinkle, J. *Metamodel Driven Model Migration*. Ph.D. thesis, Vanderbilt University, Nashville, TN 37203, 2003
- [314] Steinberg, D., Budinsky, F., Paternostro, M. and Merks, E. *EMF: Eclipse Modeling Framework*. Addison-Wesley Longman, Amsterdam, 2nd ed., 2009
- [315] Steindl, C., Ramler, R. and Altmann, J. Testing Web Applications. In: Kappel, G., Pröll, B., Reich, S. and Retschitzegger, W., eds., *Web Engineering – The Discipline of Systematic Development of Web Applications*, chap. 7. John Wiley & Sons Ltd., England, 2006
- [316] Strachey, C. Fundamental Concepts in Programming Languages. *Higher-Order and Symbolic Computation*, vol. 13(1/2), pp. 11–49, 2000
- [317] Tarski, A. *Introduction to Logic and to the Methodology of Deductive Sciences*. Oxford University Press, New York, 3 ed., 1965
- [318] The OSGi Alliance. *OSGi Service Platform Core Specification, Release 4.2*, 2009  
<http://www.osgi.org/Specifications>
- [319] The OWASP Foundation. OWASP Top 10: The Ten Most Critical Web Application Security Risks. Tech. rep., 2010  
[https://www.owasp.org/index.php/Top\\_10\\_2010](https://www.owasp.org/index.php/Top_10_2010)
- [320] The Web Standards Project. Acid Tests, 2011  
<http://www.acidtests.org/>
- [321] Thomas, D., Hansson, D., Breedts, L., Clark, M., Davidson, J. D., Gehrtland, J. and Schwarz, A. *Agile Web Development with Rails*. Pragmatic Bookshelf, 2 ed., 2006
- [322] Tolke, L. and Klink, M. *Cookbook for Developers of ArgoUML: An Introduction to Developing ArgoUML*, 2004
- [323] Topley, K. *J2ME in a Nutshell*. O'Reilly Media, 2002
- [324] Torres, V., Giner, P. and Pelechano, V. Developing BP-driven Web Applications through the use of MDE Techniques. *Software and Systems Modeling*, pp. 1–23, 2010
- [325] Tsarkov, D. and Horrocks, I. FaCT++ Description Logic Reasoner: System Description. In: Furbach, U. and Shankar, N., eds., *IJCAR, Lecture Notes in Computer Science*, vol. 4130, pp. 292–297. Springer, 2006

- [326] Vaishnavi, V. and Kuechler, B. Design Science Research in Information Systems. *Design Science Research in Information Systems and Technology*, 2004  
<http://www.desrist.org/desrist/>
- [327] van Gelder, A., Ross, K. A. and Schlipf, J. S. The Well-Founded Semantics for General Logic Programs. *Journal of the ACM*, vol. 38, pp. 620–650, 1991
- [328] van Welie, M., van der Veer, G. C. and Eliëns, A. Patterns as Tools for User Interface Design. In: *Tools for Working with Guidelines*, pp. 313–324. Springer-Verlag, London, Great Britain, 2001
- [329] Vlissides, J. M. and Linton, M. A. Unidraw: A Framework for Building Domain-specific Graphical Editors. *ACM Trans. Inf. Syst.*, vol. 8(3), pp. 237–268, 1990
- [330] Voss, A. argo2ecore, 2009  
<http://argo2ecore.sourceforge.net/>
- [331] Vépa, E., Bézin, J., Brunelière, H. and Jouault, F. Measuring Model Repositories. In: *Proceedings of the First Workshop on Model Size Metrics (MSM '06)*. 2006
- [332] W3C Group. Precision Graphics Markup Language (PGML). Tech. rep., W3C Note 10 April 1998, 1998  
<http://www.w3.org/TR/1998/NOTE-PGML-19980410.html>
- [333] W3C Group. HTML 4.01 Specification. Tech. rep., W3C Recommendation 24 December 1999, 1999  
<http://www.w3.org/TR/REC-html40/>
- [334] W3C Group. Document Object Model (DOM) Level 2 Events Specification. Tech. rep., W3C Recommendation 13 November 2000, 2000  
<http://www.w3.org/TR/DOM-Level-2-Events/>
- [335] W3C Group. Web Services Description Language (WSDL) 1.1. Tech. rep., W3C Note 15 March 2001, 2001  
<http://www.w3.org/TR/wsdl>
- [336] W3C Group. Document Object Model (DOM) Level 3 Core Specification. Tech. rep., W3C Recommendation 07 April 2004, 2004  
<http://www.w3.org/TR/DOM-Level-3-Core/>
- [337] W3C Group. OWL Web Ontology Language Semantics and Abstract Syntax . Tech. rep., W3C Recommendation 10 February 2004, 2004  
<http://www.w3.org/TR/owl-semantics/>
- [338] W3C Group. XML Schema Part 1: Structures Second Edition. Tech. rep., W3C Recommendation 28 October 2004, 2004  
<http://www.w3.org/TR/xmlschema-1/>
- [339] W3C Group. XML Schema Part 2: Datatypes Second Edition. Tech. rep., W3C Recommendation 28 October 2004, 2004  
<http://www.w3.org/TR/xmlschema-2/>

- [340] W3C Group. Simple Object Access Protocol (SOAP) Version 1.2. Tech. rep., W3C Recommendation 27 April 2007, 2007  
<http://www.w3.org/TR/soap12-part1/>
- [341] W3C Group. XSL Transformations (XSLT) Version 2.0. Tech. rep., W3C Recommendation 23 January 2007, 2007  
<http://www.w3.org/TR/xslt20/>
- [342] W3C Group. Extensible Markup Language (XML) 1.0 (Fifth Edition). Tech. rep., W3C Recommendation 26 November 2008, 2008  
<http://www.w3.org/TR/xml/>
- [343] W3C Group. HTML 5: A vocabulary and associated APIs for HTML and XHTML. Tech. rep., W3C Working Draft 26 February 2008, 2008  
<http://www.w3.org/html/wg/html5/>
- [344] W3C Group. Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification. Tech. rep., W3C Working Draft 07 December 2010, 2010  
<http://www.w3.org/TR/CSS21/>
- [345] W3C Group. XHTML™ 1.1: Module-based XHTML (Second Edition). Tech. rep., W3C Recommendation 23 November 2010, 2010  
<http://www.w3.org/TR/xhtml11/>
- [346] W3C Group. XHTML™ 2.0. Tech. rep., W3C Working Group Note 16 December 2010, 2010  
<http://www.w3.org/TR/xhtml2/>
- [347] W3C Group. XML Path Language (XPath) 2.0 (Second Edition). Tech. rep., W3C Recommendation 14 December 2010, 2010  
<http://www.w3.org/TR/xpath20/>
- [348] W3C Group. XMLHttpRequest. Tech. rep., W3C Candidate Recommendation 3 August 2010, 2010  
<http://www.w3.org/TR/XMLHttpRequest/>
- [349] W3C Group. XQuery 1.0: An XML Query Language (Second Edition). Tech. rep., W3C Recommendation 14 December 2010, 2010  
<http://www.w3.org/TR/xquery/>
- [350] W3C Group. XQuery 1.0 and XPath 2.0 Functions and Operators (Second Edition). Tech. rep., W3C Recommendation 14 December 2010, 2010  
<http://www.w3.org/TR/xpath-functions/>
- [351] W3C OWL Working Group. OWL 2 Web Ontology Language Document Overview. Tech. rep., W3C Recommendation 27 October 2009, 2009  
<http://www.w3.org/TR/owl2-overview/>
- [352] Wagner, G. The Agent-Object-Relationship Metamodel: Towards a Unified View of State and Behavior. *Information Systems*, vol. 28(5), pp. 475–504, 2003

- [353] Wagner, G., Antoniou, G., Tabet, S. and Boley, H. The Abstract Syntax of RuleML - Towards a General Web Rule Language Framework. In: *Proceedings of the 2004 International Conference on Web Intelligence (WI '04)*, pp. 628–631. IEEE Computer Society, Washington, DC, USA, 2004
- [354] Wagner, G., Antoniou, G., Taveter, K., Berndtsson, M. and Spreeuwenberg, S. A First-Version Visual Rule Language. deliverable I1-D1, Faculty of Technology Management – Eindhoven University of Technology, 2004
- [355] Wagner, G., Giurca, A. and Lukichev, S. Modeling Web Services with URML. In: *Proceedings of Workshop Semantics for Business Process Management*. Budva, Montenegro, 2006
- [356] Wagner, R. *Professional Flash Mobile Development: Creating Android and iPhone Applications*. Wrox, 2011
- [357] Wampler, D. Cat Fight in a Pet Store: J2EE vs. .NET. Tech. rep., ONJava.com, 2001  
<http://www.onjava.com/pub/a/onjava/2001/11/28/catfight.html>
- [358] Weaver, J. and Weaver, J. L. *JavaFX Script: Dynamic Java Scripting for Rich Internet/Client-side Applications*. Safari Books Online. Apress, 2007
- [359] WebRatio Group. WebRatio AJAX Extension, 2007  
<http://www.webratio.com/WebRatio-AJAX.do>
- [360] Wenz, C. *Essential Silverlight*. O'Reilly, 1st ed., 2008
- [361] White, S. A. Process Modeling Notations and Workflow Patterns. *BPTrends*, 2004
- [362] Wimmer, M., Kemper, A. and Seltzsam, S. Security for Web Applications. In: Kappel, G., Pröll, B., Reich, S. and Retschitzegger, W., eds., *Web Engineering – The Discipline of Systematic Development of Web Applications*, chap. 13. John Wiley & Sons Ltd., England, 2006
- [363] Winer, D. XML-RPC Specification. Tech. rep., UserLand Software, 1999  
<http://www.xmlrpc.com/spec>
- [364] Wirth, N. On the Design of Programming Languages. In: *IFIP Congress*, pp. 386–393. 1974
- [365] Wong, P. Y. H. and Gibbons, J. A Process Semantics for BPMN. In: *Proceedings of the 10th International Conference on Formal Engineering Methods (ICFEM '08)*, pp. 355–374. Springer, Kitakyushu-City, Japan, 2008
- [366] Wright, J. ModelDoc: A Model-Driven Framework for the Automated Generation of Modelling Language Documentation. Tech. rep., Massey University, New Zealand, 2012
- [367] Wright, J. and Dietrich, J. Requirements for Rich Internet Application Design Methodologies. In: *Proceedings of the 9th International Conference on Web Information Systems Engineering (WISE 2008)*. Auckland, New Zealand, 2008
- [368] Wright, J. and Dietrich, J. Survey of Existing Languages to Model Interactive Web Applications. In: *Proceedings of the 5th Asia-Pacific Conference on Conceptual Modelling (APCCM 2008)*. Wollongong, NSW, Australia, 2008



- [369] Wright, J. and Dietrich, J. Non-Monotonic Model Completion in Web Application Engineering. In: *Proceedings of the Doctoral Symposium at the 21st Australian Software Engineering Conference (ASWEC 2010)*. Auckland, New Zealand, 2010
- [370] Yap, P. S., Hosking, J. and Grundy, J. Automatic Diagram Layout Support for the Marama Meta-toolset. In: *Proceedings of the 2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, Pittsburgh, United States, 2011
- [371] Yu, J., Benatallah, B., Casati, F. and Daniel, F. Understanding Mashup Development. *IEEE Internet Computing*, vol. 12, pp. 44–52, 2008
- [372] Zandstra, M. Testing with PHPUnit. In: *PHP Objects, Patterns, and Practice*, pp. 379–405. Apress, 2010
- [373] Zelnick, N. Nifty Technology and Nonconformance: The Web in Crisis. *IEEE Computer*, vol. 31(10), pp. 115–116, 119, 1998
- [374] Zhang, G., Baumeister, H., Koch, N. and Knapp, A. Aspect-Oriented Modeling of Access Control in Web Applications. In: *6th International Workshop Aspect Oriented Modeling (AOM'05)*. Chicago, USA, 2005
- [375] Zhang, Y. and Xu, B. A Survey of Semantic Description Frameworks for Programming Languages. *SIGPLAN Notices*, vol. 39(3), pp. 14–30, 2004
- [376] Zhu, N., Grundy, J. and Hosking, J. Pounamu: A Meta-tool for Multi-View Visual Language Environment Construction. In: *Proceedings of the 2004 IEEE Symposium on Visual Languages and Human Centric Computing (VL/HCC '04)*, pp. 254–256. IEEE Computer Society, Washington, DC, USA, 2004
- [377] Zimmer, D. and Unland, R. On the Semantics of Complex Events in Active Database Management Systems. In: *Proceedings of the 15th International Conference on Data Engineering*, pp. 392–399. 1999





# Index

## Symbols

@generated tag, 163, 205  
@inference tag, 211, 367  
«generated» stereotype, 208  
«multiple» stereotype, 106

## A

abstract state machines, 49  
abstract syntax tree, 47, 103, 218  
abstraction, 31, 86  
access control, 15, 80–83, 136–140, 227, 263  
[Access Control Handler](#), 138  
    shape colour, 204  
access control list, 80  
[Accessible](#), 105  
ACL, *see* access control list  
[Action](#), 102  
    classifiers, 156  
    comparison to [Wires](#), 131  
ActiveX, 2, 289, 321  
[Activity Node](#)  
    shape colour, 204  
[Activity Operation](#), 131  
    example, 129  
    in Ticketiaml, 246  
    overriding, 131  
[Activity Parameter](#), 111  
[Activity Predicate](#), 130–131  
acyclical class inheritance constraint, 48, 49, 189  
aggregation, 57, 182  
AJAX, 2, 10, 143  
    component in prototype.js, 13  
    multithreading, 129  
Alloy, 187–188  
    OpenBRR review, 332  
anti-patterns  
    Primitive Obsession, 145

anySimpleType type, 73  
anyType type, 73  
AORML, 67  
Apache HTTP Server, 340  
    configuration, 343  
application platforms, 9  
Araneus, 24  
ArgoUML, 163–164, 170  
    OCL support, 190  
    OpenBRR review, 332  
ArgoUWE, 163  
arithmetic  
    in IAML, 375  
    in UML activity diagrams, 70, 127  
    reverse engineering, 47  
aspect-oriented modelling, 65, 84  
aspect-oriented programming, 65  
association classes, 99  
AST, *see* abstract syntax tree  
asynchronous Javascript and XML, *see* AJAX  
Atlas Transformation Language, 47  
atomic types, 72  
authentication, 80  
authorisation, 80, 285, 288  
autocomplete, 95, 134, 324  
    in UWE, 23  
[Autocomplete Wire](#), 95, 134

## B

base model, 52  
Basic RIAs, 86, 88, 229, 262, 268  
benchmarking application, 24  
[Boolean Property](#), 100  
boolean type, 73  
BPMN, 71  
breadcrumbing, 200, 206, 233, 362, 365  
broken links, 17, 54

- browser interaction, 227
- [Builtin Operation](#), 125, 325
  - overriding, 131
- [Builtin Property](#), 100, 130
- bundles, 79
- business logic, 12
- Business Process Modelling Notation, *see* BPMN
- [Button](#), 148
- C**
- C, 9, 222
- C++, 9
- C#, 164
- callbacks, 216
- cardinality, 75
- CASE tool, 20, 52
- casting, 106, 108, 383
  - instances of `iamlOpenIDURL`, 140
- CGI, 2
- change* event, 122
- change models, 47
- [Changeable](#), 105, 121, 268
- Checks, 175, 188–189, 264
  - implementation, 220–221
  - OpenBRR review, 334
- CIM, 41
- classic life cycle, 32
- classifiers, 208
  - metamodelling, 153
- click* event, 69, 123
- client-side technologies, 10
- Cloc, 357
- CMOF, 42
- code coverage, 224–225, 265
- code formatting, 217–218
- code generation, 45–46, 51
  - code coverage, 224
  - environments, 173–176, 264
  - implementation, 214–219, 222, 343
  - incremental generation, 270
  - metamodelling, *see* templating language
- Code Generation* component, 214
- code generation language, *see* templating language
- Code layer, 42
- coding conventions, 91
- cognitive dimensions, 60–61, 88, 259, 361–365
- cognitive distance, 65
- cognitive effectiveness, 60, 202, 258–259
- cohesion, 64
- coincidental refactoring, 271
- CommonJS, 10
- compilation, 45
- Completion* component, 211
- [Complex Condition](#), 98
- [Complex Term](#), 97, 103, 268
- complex types, 76
- complexity, 63–66
- composition, 73
- computation independent model, *see* CIM
- conceptual gap, 37
- concurrency, 12
- [Condition](#), 97
  - shape colour, 204
- condition modelling, *see* predicate modelling
  - decomposition, 103–105
- configuration file formats, 9
- connectives, 98
- consistency, 362
  - evaluation through model verification, 91
  - of modelling languages, 90
  - syntax, 91
- constraint expressiveness, 55–58, 182, 192, 193
- constraints, 48, 54
  - in UML, 21
- container names, 207, 362
- container reference, 100, 206
- containing scope, 110–111, 116, 215, 395
- containment feature, 200, 207–208
- containment references, 48, 100, 367
  - metrics, 28, 255
- content distribution network, 110
- content markup languages, 9
- context, 362
- conventions, 52
- cookies, 9
  - vulnerabilities, 12
- correctness, 55
- cost estimation, 26

coupling, 64  
CrocoPat, 56, 185–187, 191, 264  
    implementation, 221–222  
    OpenBRR review, 332  
CSS, 2, 9, 10, 90, 215, 232, 270, 357  
CTL specification language, 190  
current instance pointer, 119–120  
cursor, 78, 119  
custom template, 144

## D

DAC, *see* discretionary access control  
[Data Flow Node](#), 129  
data object table, 75, 114  
data reuse, 12  
data-intensive modelling languages, 72  
Database Broker design pattern, 77  
databases, 16, 228, 277, 280  
    query languages, 9  
date type, 73  
dateTime type, 73  
decomposition, 63–66  
*default checkbox* rule, 53, 177  
default role, 136, 435  
default type, 109, 263  
default value, 111, 450  
defect amplification, 17  
delivery status notifications, 124  
derivation cause, 182  
derived types, 73  
design decisions, 58  
design goals, 88–89  
design patterns, 12, *see* reusable patterns  
    Abstract Factory, 77  
    Command, 313  
    Composite, 146, 363  
    Database Broker, 77  
    Dynamic Linkage, 164  
    for consistency, 91  
    for data access, 77  
    Iterator, 77  
    Variant, 187  
design science research, 4–5  
desktop metaphor, 58

[Detail Wire](#), 133  
DHTML, 2  
*Diagram Actions* component, 209  
*Diagram Definitions* component, 201  
*Diagram Editors* component, 199  
*Diagram Extensions* component, 205  
diagram partitioning, 200  
discretionary access control, 82  
Document Object Model, *see* DOM  
documentation, 265  
documented conventions, 52  
DOM, 10, 98  
    *change* event, 122  
    *click* event, 69, 123  
    events, 69  
    of IAML model instances, 111  
[Domain Attribute](#), 113  
[Domain Attribute Instance](#), 95, 116  
    model completion, 325  
[Domain Feature](#), 113  
[Domain Feature Instance](#), 116  
[Domain Instance](#), 116  
    model completion, 325  
    modifying, 120  
    new, 120  
[Domain Iterator](#), 95, 116  
    autosave, 120  
    emulating looping, 126  
    failure, 119  
    interaction with a [Detail Wire](#), 133  
    interaction with an [Autocomplete Wire](#), 135  
    model completion, 325  
    modifying instances, 120  
    navigation, 119  
    new instances, 120  
    *onIterate* event, 124  
    query, 117  
    reloading, 116  
    RSS feeds, 117  
    shape colour, 204  
    user instances, 137  
domain modelling, 75–77, 112–122  
[Domain Source](#), 117  
    shape style, 203

- Domain Type, 113, 153
  - classifiers, 156
  - in Ticketiaml, 235
  - instantiation within MDA, 115
  - restriction through a [Login Handler](#), 137
  - shape colour, 204
  - shape style, 203
- domain-specific languages, 36–38, 50, 51
  - benefits, 37
  - comparison to metamodels, 38
  - drawbacks, 37
  - evolution, 32
  - for model completion, 177
  - in Drools, 180
  - in web development, 3
- domain-specific metamodel, 50
- domain-specific modelling language, 38
- domain-specific visual language, 58
- DOT graph description language, 223
- Dresden OCL
  - OpenBRR review, 332
- Dresden OCL toolkit, 164, 190, 221
- Drools, 180, 182–184, 264
  - documentation, 367
  - implementation, 214
  - incremental completion, 270
  - OpenBRR review, 332
- Drools* component, 214
- DSL, *see* domain-specific languages
- DTD, 41, 170
  - expressiveness, 172
  - meta-metamodel, 40
- dynamic templates, 169, 175, 197, 205
- dynamically checked language, 72
- E**
- EAnnotation, 163
- EAttribute, 113
- EBNF, *see* Extended Backus-Naur Form, 48
  - meta-metamodel, 40
- [ECA Rule](#), 101–102
  - comparison to [Wires](#), 131
  - failure, 119
  - model completion, 326
- ECA rules, 66, 101–103, 263, 362
- EClass, 28, 77, 101, 113, 255
- Eclipse
  - colour palette, 204, 362
  - framework, 160, 195
  - JDT, 217
  - M2T project, 174, 175
  - Modeling Framework, *see* EMF
  - Modeling Project, 163
- Eclipse Public License, 196, 218, 232
- eContainer, 100
- Ecore, 42, 77, 108, 114, 163, 228, 262
  - expressiveness, 172
  - in IAML, 112
  - meta-metamodel, 40
  - metamodel metrics, 255
  - serialisation to EMOF, 163
- EDataType, 114
- edges, 203, 204
  - shortcuts, 206
- efficiency of expression, 58
- EGenericType, 114
- EInt, 107
- Email, 300, 301
- [Email](#), 144
  - in Ticketiaml, 236
- EMF, 77, 162–163
  - as a metamodel for code generation, 173
  - code formatting, 217
  - datatypes, 107
  - documentation, 367
  - Dynamic EMF, 163
  - EMF.Edit, 199
  - events, 367
  - metamodel, *see* Ecore
  - model instance generation, 197
  - OpenBRR review, 333
  - type unions, 101
- EMF Validation framework, 163, 189–190, 264
  - implementation, 221
- EMFText, 269
- emftriple project, 178
- EMFV OCL
  - OpenBRR review, 333

- EMOF, 42, 77, 108, 163
  - English language
    - defining informal semantics, 49
    - defining syntax, 48
    - meta-metamodel, 40
  - enterprise application, 12
  - enterprise software, 12, 85–86
  - Entity-Relationship diagrams, *see* ER diagrams
  - EObject, 101
  - eOpposite, 28, 255, 256
  - EPackage, 28, 255
  - ER diagrams, 75–76, 114
    - within WebML, 75
  - EReference, 367
  - EString, 107
  - evaluation
    - feature comparison, 88
    - metamodelling metrics, 88
    - requirements, 88
    - Ticket 2.0, 88
    - visual notation, 88
  - Event, 122–124
    - classifiers, 156
    - shape colour, 204
    - shape style, 203
  - event-condition-action rules, *see* ECA rules
  - Event-Query-Filter-Action rules, 68
  - events, 14, 66–69, 122–124, 227
    - algebra, 68, 126
    - bubbling, 69, 123
    - cancelable, 69
    - client-side, 124
    - importance to RIAs, 14
    - in IAML, 101–103, 230
    - modelling language, 102, 143, 154, 156
    - rules, *see* ECA rules
    - server-side, 124
  - evolutionary model, 31, 32, 53, 87
  - exception handling, 142
  - execution environment, 159
  - Execution Flow Node, 128
  - EXSD Data Type, 108
  - Extended Backus-Naur Form, 40
  - Extends Edge, 113
  - extensibility, 270–271
    - code generation, 173
    - graphical editors, 168
    - metamodelling, 162
    - of UML, 21
  - Extensible Stylesheet Language Transformations,
    - see* XSLT
  - extension, 30, 73, 86
  - extensivity, 53
- F**
- Facebook, 14, 17–20, 276
  - factory function, 54
  - failure handlers, 119, 141, 396, 437
  - feature creep, 37
  - file, 110
  - files, 283
  - first-class citizens, 14
  - first-order logic, 49, 97, 108, 121
    - syntax, 97
  - Flash, 9, 232, 309, 310
  - folding, *see* aggregation
  - foreign keys
    - navigation, 122
  - forward engineering, 47
  - Frame, 150
  - frameworks, 13, 50, 51
    - comparison to modelling languages, 14
    - comparison to programming languages, 13
    - traceability, 48
  - free software, 196
  - Full RIAs, 86, 88, 268
  - Function, 97
    - classifiers, 156
    - shape colour, 204
    - shape style, 203
  - function calls, 100
  - functions, 97
- G**
- Gate, 142
    - shape colour, 204
  - GCC exception, 33

- GEF, 169
  - OpenBRR review, 333
- GEF\*, 170
- general-purpose languages, 37
- general-purpose metamodel, 50
- Generated Element**, 106
  - overriding, 211
- generated element notation, 325
- generated element notations, 208
- «generated» stereotype, 208
- Generates Elements**, 214
- Generator* component, 218
- Generic Modeling Environment, 161
- generic types, 77
- GetShortcuts class, 207, 363
- Gmail, 14, 17–20, 134, 276, 298
- GMF, 169–170, 199–208, 264
  - documentation, 367
  - extensibility, 205–206
  - initialise diagram file, 200, 364
  - Notation metamodel, 170, 255
  - OpenBRR review, 333
  - outline view, 365
  - secondary notation, 364
- GMF Notation
  - metamodel metrics, 255
- Google Accounts, 139, 311
- Google Calendar, 14, 17–20, 276
- Google Chrome, 2
- Google Docs, 14, 15, 17–20, 276, 304
- Google Gears, 10, 11, 129, 276, 308, 317, 318, 321
- Google Maps, 150, 218, 305
- Google Pages, 14, 17–20, 276, 304
- Google Reader, 14, 17–20, 276
- Google Talk, 298
- Google Web Toolkit, 13
- GPL, 33
- Graph Editing Framework, *see* GEF\*
- Graphical Editor* component, 198
- Graphical Editor Framework, *see* GEF
- graphical language guidelines, 61
- Graphical Modeling Framework, *see* GMF
- graphical modelling environments, 167–173, 264
  - graphical modelling language, *see* visual modelling
  - graphical object editors, 59
- H**
- Haskell, 9
- hierarchical modelling, 64, 91–92, 263, 362, 365
  - exporting diagrams, 209
  - operation modelling, 126
  - with visual metaphors, 92
- higher-order logic, 101
- Hoare logic, 49
- HTML, 2, 9, 19, 215, 230, 232, 270, 357
  - meta-metamodel, 40
- HTML 5, 2, 9, 10, 40, 308
- html2text, 217
- HTTP, 9
- hybrid software process model, 87, 229
- hypermedia, 5, 11
- I**
- i18n, 11, 86, 232, 268
- iacleaner, 218
  - source code, 339
- IAML, 85–157, 262–263, 357
  - development history, 195
  - graphical editor, 198–210, 340–343
  - instance unification, 156–157
  - issue tracker, 195
  - metamodel implementation, 197
  - metamodel metrics, 255
  - model completion rules, 211, 213
  - packages, 95
  - smallest valid model, 259, 364
  - source code, 339
  - source code repository, 195
  - Ticket 2.0 implementation, *see* Ticketiaml
  - tutorial, 340–343
  - visual model instance examples, 94, 103, 104, 113, 119, 121, 126, 129, 134, 135, 139, 142, 151, 200, 208, 327, 328
  - XMI serialisation, 197, 228, 347–356
- iaml files, 197, 198, 222, 340

- IAML Metamodel component, 197
- IAML Model component, 197
- IAML packages
  - Core, 95–106
  - overview, 95
- IAML test suite, 223–225
- iaml\_diagram files, 200, 209, 340, 364
- iamlEmail type, 93, 109
- iamlFile type, 110
- iamlIdentity type, 110, 139
- iamlImage type, 110
- iamlOpenIDURL type, 110, 139
- iamlPassword type, 110, 136
- iamlURL type, 109
- ID property, 198, 223
- idempotence, 53
- IDL, 69
- image, 110
- implicit container reference, 100, 102
- implicit slot, 100
- incremental generation, 45, 270
- infinitely redirects* constraint, 56, 181
- information capacity, 60
- injective functions, 47
- [Input Form](#), 148
  - model completion, 325
- [Input Text Field](#), 148
  - model completion, 325
- input validation, 111, 230, 290, 291
  - in UWE, 23
- insertion cache, 54, 181
  - implementation, 211
- instance pointer, 116
- instance unification, 153–157
- instantiation
  - in UML, 115
  - metamodelling, 153
- int type, 73
- integer overflow, 191
- integer type, 73
- intended model, 52
- interactivity, 11
- interface metaphors, 58
  - taxonomy, 58

- internationalisation, *see* i18n
- Internet, 1
- [Internet Application](#), 150
- invariants, 48
- ISO/IEC 9126, 12–13
- iterative development, 32, 90
- Iterator design pattern, 77
- [Iterator List](#), 148

## J

- J2ME, 9, 270
- Java, 9
  - applets, 9
  - as a metamodel for code generation, 173
  - meta-metamodel, 40
  - operations in IAML, 127
- Java Emitter Templates, *see* JET
- [Java Operation](#), 127
- Java Server Pages, *see* JSP
- JavaBeans, 164
- JavaFX
  - onReplace* event, 122
- Javascript, 2, 10, 215, 217, 232, 270, 357
  - injection vulnerability, 13
  - multithreading, 129
- JBoss Rules, *see* Drools
- Jena, 56, 178–179, 184–185
  - OpenBRR review, 333
- Jess, 179
- JET, 163, 169, 174–175
  - OpenBRR review, 334
- jQuery, 232
- jQuery framework, 13
- JSON, 145, 295, 322
- JSP, 2, 174
- JSR-94, 177, 219
- JUnit, 79, 224
- JUnit, 232, 268

## K

- Kaitiaki, 68

## L



- Label, 148
- language workbench, 161
- Last.fm, 14, 17–20, 276
- latency, 12
- Law of Demeter, 122
- layered architecture diagram, 96
- layout, 168, 258, 364
- lexical space, 108
- lifecycles, 15, 79–80, 227
  - events, 140, 143, 263
  - modelling, 140
- linear sequential model, 32, 87
- lines of code metric, 30, 255, 357
- LINQ, 9
- logic-based core, 97–101, 262
- Login Handler, 137
  - in Ticketiaml, 246
  - shape colour, 204
- long type, 73
- looping, 126
- LTL, 222
- LTL specification language, 190
- $L_{ver}$  expressiveness, 55–58, 182, 192, 193
- M**
- M0 layer, 39, 74
- M1 layer, 39, 45, 74
- M2 layer, 40, 45, 74
- M3 layer, 40, 42, 45, 49, 77
  - compliance, 162
  - of existing metamodels, 40
- MAC, *see* mandatory access control
- mail messages, 144, 404
- mandatory access control, 82
- manual effort, 254
- Map, 148
- Map Point, 148
- Marama project, 166–167, 171–172, 271
  - OpenBRR review, 334
- MaramaTorua project, 166
- mashups, 1, 2
- MDA, *see* Model Driven Architecture
- MDD, *see* model-driven development
- MDE, *see* Model-Driven Engineering
- message disposition notifications, 124
- messaging, 16, 143–145, 228
- Meta Object Facility, *see* MOF
- meta-metamodels, 40
  - of existing metamodels, 40
  - of existing technologies, 162
- Meta-Programming System, 161
- metaclass, 21
- metaclass name, 200, 207, 362
- MetaEdit+ , 161
- metamodel layer, 40
- metamodel refactoring, 92–93, 271–272
- metamodelling
  - comparison between classifiers and instances, 153–157
  - design principles, 89–93, 155
  - environments, 51, 161–167, 264
  - evolution in UML, 90
  - metrics, 28, 88, 255–257, 263, 267
  - semantics, 48–50
  - syntax, 48
- metamodelling architecture, 114–115, 153
- metamodels, 16, 36, 50, 51
  - as graphs, 28
  - comparison to domain-specific languages, 38
  - comparison to programming languages, 50
  - of models, 40
- metaphors, 58
- metrics, **26**
  - complexity, 28
  - graph-based, 28
  - language system, 29
  - metamodelling metrics, 28, 263
  - object-oriented software, 28
  - system metrics, 29, 262
  - web application metrics, 26
- Microsoft Passport, 110, 139, 311
- MIME, 144
- modality, 75
- Model Actions component, 222
- model checking, 57, 182, 192, 264
  - implementation, 222
- model completion, 51–54, 84, 93, 95, 105, 106, 112, 113, 119, 131, 137, 138, 263, 266,



- 362
- code coverage, 224
- consistency, 209, 263
- definition, 53
- documentation, 52, 211
- environments, 177–181, 264
- error-proneness, 363
- for model verification, 181
- implementation, 210–214, 222
- implementing a [Detail Wire](#), 133
- implementing a [Sync Wire](#), 325–326
- in Ticketiaml, 235, 246
- incremental completion, 270
- insertion cache, *see* insertion cache
- overriding, 211
- partial inference, 209
- performance, 224
- progressive evaluation, 259, 364
- removing generated elements, 106, 209
- rule modelling language, 156
- to implement reusable patterns, 84
- usability, 208
- visual notation, 208
- [Wires](#), 132
- Model Completion* component, 210
- model constraints, *see* model verification
- Model Driven Architecture, 39–43, 50, 51, 114
  - metamodelling architecture, 39, 262
  - type systems, 74–75
  - viewpoint architecture, 41, 262
- Model Edit* component, 199
- model instance verification, *see* model verification
- model instances, 36, 39–40, 50, 51
  - implementation, 197–198
  - in EMF, 197
  - semantics, 48–50
- model layer, 39
- model migration, 198
  - implementation, 223
- Model Migration* component, 198
- model refactoring, 363, 364
  - move into separate model, 209
- model transformations, 44–46, 50, 51, 84
  - architecture, 45
  - code generation, 45
  - consistency, 53
  - incremental transformations, 270
  - licensing, 33
  - model migration, *see* model migration
  - model-to-model, 46, 177
  - model-to-text, 46, 176
  - open source, 33
  - semantics, 50
- model validation, 55
- model verification, 33, 54–58, 228, 267
  - environments, 181–192, 264
  - evaluating consistency, 91
  - implementation, 219–222
  - language categories, *see* verification languages, 191–192
  - model checking, 363
  - progressive evaluation, 259, 364
  - properties, 362
  - recursive constraints, 363
  - support in metamodelling environments, 162
- Model Verification* component, 219
- model-driven code coverage, *see* code coverage
- model-driven development, 3, 38, 50, 51
- Model-Driven Engineering, 38, 50, 51
- model-driven graphical environments, 59
- model-driven testing, 33
- ModelBus, 165
- ModelDoc, 163, 209, 265, 340, 367
  - metamodel metrics, 255
  - model completion, 211
  - source code, 339
- modelling
  - aspect-oriented, *see* aspect-oriented modelling
  - big picture, 50
  - hierarchical, *see* hierarchical modelling
  - semantics, 48–50
  - software, 51
  - spaces, *see* modelling spaces
  - visual, *see* visual modelling
- modelling languages, *see* metamodels
  - abstraction, 31, 86
  - comparison to frameworks, 14

- complexity, 63–66
- consistency, 90, 91
- development with UML, 31
- domain-specific, *see* domain-specific languages
- evolution, 31, 32
- extension, 30, 86
- metrics, 28, 263
- problems in development, 31
- process model, 87
- restriction, 30, 86
- scalability, 90
- simplicity, 90
- standards-compliance, 91
- visual, *see* visual modelling
- modelling spaces, 43, 74, 107
- models, *see* model instances
- modifying domain data, 120
- modularity, 63–65
  - degrees, 63
- MOF, 42–43
  - Complete MOF, *see* CMOF
  - Ecore extension, 163
  - Essential MOF, *see* EMOF
  - meta-metamodel, 40
  - semantic domain model, 108
- MOFM2T, 46
- mousedown* event, 69
- MouseEvent, 69
- mouseup* event, 69
- Mozilla Firefox, 2
  - overridden values, 208
- multiple inheritance, 113, 115–116, 177, 262
  - emulation with single inheritance, 187
  - for role hierarchy, 136
- «multiple» stereotype, 106
- multithreading, 126, 129, 420, 446
- MVC, 169
- MySQL, 9

## N

- named association class, 97
- named parameters, 99
- navigability, 28, 255
- navigation, 191

- navigation space model, 92
- NCLOC metric, 30, 255
- negation, 98
  - in IAML, 125
- new domain data, 120, 396
- nodes, 203, 204
- non-functional requirements, 54
- non-monotonic reasoning, 53
- non-relational databases, 75
- NoSQL, 9, 75
- NuSMV, 190–192, 264
  - constraint example, 335–337
  - implementation, 222
  - OpenBRR review, 334

## O

- OAuth, 2, 139, 311
- OAW, *see* openArchitectureWare
- object-oriented databases, 75
- OCL, 49, 56, 72, 97, 111, 189–190, 203
  - Dresden OCL toolkit, *see* Dresden OCL toolkit
  - implementation, 221
  - metamodel metrics, 255
  - operations in IAML, 127
- OCL Operation**, 127
- offline applications, 11, 16, 86, 228, 268, 308
- onAccess* event, 123
  - implementation, 215
- onChange* event, 122
- onClick* event, 123, 146
- onDelete* event, 227
- onException* event, 142
- onFailure* event, 124, 227
- onFinish* event, 143
- onInit* event, 123
  - implementation, 215
- onInput* event, 122, 146
- onIterate* event, 124
- onSent* event, 123
- onTimeout* event, 227
- OOHDM, 24
- OOWS, 24
- open source, 33, 160
  - evaluation criteria, 160–161

- license, 195–196
- licenses, 33
- Open Web Application Security Project, *see* OWASP
- openArchitectureWare, 175, 202, 264, 269
  - Checks, *see* Checks
  - code formatting, 217
  - extensibility framework, *see* Xtend
  - implementation, 214, 220
  - OpenBRR review, 334
  - Xpand, *see* Xpand
  - Xtend, *see* Xtend
- openArchitectureWare component, 214
- OpenBRR, 160–161, 264, 331–335
- OpenID, 2, 16, 18, 139–140, 266, 311
  - in IAML, 110
  - in Symfony, 233
  - security, 139
- OpenID Gate, 140
- Operation
  - as a [Scope](#), 143
  - overriding, 131
  - shape colour, 204
  - shape style, 203
- operation modelling, 70–71, 125–131
  - with a textual language, *see* textual languages
- operational semantics, 49
- OQL, 9
- OSGi, 79
  - bundles, 160, 195
- Output Formatter component, 217
- overridden elements, 208
  - in Ticketiaml, 246
- «overridden» stereotype, 208
- OWASP, 13
- OWL, 2, 41, 56, 73, 178, 185
- OWL 2 Full, 56
- P**
- packages, 95
- page
  - scope, 80
- [Parameter](#), 97, 103, 268
  - named parameters, 99
- [Parameter Value](#), 97
- parameterised APIs, 89, 91, 117, 395
- parameterised types, 77
- parent names, *see* container names
- passwords, 110, 136, 137
- PDO, 13, 117
- Pellet, 179
- performance, 12
- Perl, 9
- [Permission](#), 136
  - shape colour, 204
  - shape style, 203
- Petri nets, 71
- PGML, 170
- phantom edges, 223
- PHP, 2, 9, 215, 217, 232, 270, 340, 357
  - injection vulnerability, 12
- PHPMailer, 217, 218
- PHPUnit, 232
- PIM, 41, 50
- Platform Configuration* component, 217
- platform independence, 16, 107, 228, 262, 270
- platform specific model, *see* PSM
- platform-independent model, *see* PIM
- plugins, 160
- POJOs, 174, 177, 179–181
- PostgreSQL, 9
- Pounamu project, 166
- [Predicate](#), 97
  - classifiers, 156
- predicate modelling, 71, 130
- Primitive Obsession, *see* anti-patterns
- primitive types, 72
- programming languages, 50, 51
  - comparison to frameworks, 13
  - comparison to metamodels, 50
  - design principles, 89
  - freeform, 217
- proof-of-concept implementation, 33, 87, 195–225
  - issues, 267
- Propel, 13, 117
- prototype.js, 217
- prototype.js framework, 13, 232
- prototyping, 31

- PSM, 41, 50
  - configuration in IAML, 217
  - implementation, 217
- pull-based, 143
- push-based, 2, 143
- Python, 9
- Q**
- QSOS, 160
- qualified procedure call, 100
- quality
  - open source, 160
- quantifiers, 98
- [Query Parameter](#), 111
- Query/Views/Transformations language, *see* QVT
- QVT, 46–47, 222
  - integration with Xpand, 176
- R**
- R2ML, 66, 67, 180
  - rule translation, 177
- rapid application development, 31
- Rational Rose, 163
- rationale, 58
- RBAC, 83, 136, 263
- RDF, 2, 178
- reality layer, 39
- recursion, *see* recursion
- refactoring, 271
  - metamodelling, *see* metamodel refactoring
- reference implementation, 6, 17, 87, 195–225, 367
- referential integrity, 145
- relational databases, 75, 113–114
- request, 143
  - scope, 80
- research design, 5
- research questions, 3–4
- REST, 2, 9, 322
- restriction, 30, 73, 86
- Rete algorithm, 177, 180
- reusable patterns
  - infrastructure, 105
  - metamodelling, 83–84
- reverse engineering, 47, 266
- reverse-engineering, 54
- REWERSE, 67
- Rich Internet Applications, 2, 9–11
  - features, 14–20, 261
  - scope, 80
  - security risks, *see* security risks
  - technologies, 10
- [Role](#), 136
  - in Ticketiaml, 235
- role hierarchy, 136, 235
- role-based access control, *see* RBAC
- root model element, 150, 206
- round-trip engineering, 45, 47, 176, 266
- RSS, 2, 16, 18, 117, 230, 276, 295
  - in Ticketiaml, 248
  - providing a feed in IAML, 150
- Ruby, 9
- Ruby on Rails framework, 13, 52
- rule engines, 177–181
  - benchmarks, 180
- Rule Set* component, 211, 224
- RuleML, 210
- runtime environment, 159, 160
- Runtime Library* component, 216, 357
- runtime URL property, 222
- S**
- saving domain data, 120, 396
- SBVR, 219
- scalability, 12, 90
- [Schema Edge](#), 116
- schemas, 75
- [Scope](#), 110, 123, 140–143
  - classifiers, 157
  - failure handlers, 141
- scope modelling, 143
- scopes, 80, 140–143, 263
- secondary notation, 364
- secret key, 137
- security risks, 12–13, 266–267
  - addressing through language design, 91
- [Select Edge](#), 116

- semantic domain model for constructs, 108
  - semantics
    - formal, 49
    - informal, 49
    - of metamodeling, 48–50
    - operational, 49
  - sentence, 98
  - server-side
    - languages, 9
  - service-enabled web applications, 1
  - [Session](#), 141
    - in Ticketiaml, 236–246
    - shape colour, 204
  - sessions, 9, 15, 80, 284, 286
    - hijacking, 12
    - in RBAC, 83
  - [Set Wire](#), 133
  - SGML
    - meta-metamodel, 40
  - shape design, 202–205, 265
  - shortcuts, 200, 206–207, 363
  - Silverlight, 9
  - [Simple Condition](#), 98, 103
  - SimpleGMF, 202, 265, 269
    - in metamodel refactoring, 271
    - source code, 339
  - simplicity, 90, 257
  - slots, 97
  - SMS, 144
  - SMTP, 9
  - SOAP, 9, 145, 295
  - software frameworks, 52
  - software process models, 31, 87
  - software quality model, 26
  - source code, 45
    - generation, *see* code generation
    - model, 35, 39
    - modelling spaces, 44
  - Sourceforge, 232
  - SQL, 9, 357
    - functions, 117
    - injection vulnerability, 12, 89
  - sqlite, 215, 270
  - standards-compliance, 16, 91, 228
  - statically checked language, 72, 106, 262
  - stereotypes, 21, 106, 208
  - storage semantics, 411, 418
  - Stratego/XT, 23
  - stratification, 54
  - string type, 73
  - strongly checked language, 72
  - structural operational semantics, 49
  - structure, *see* syntax
  - subtypes
    - subtype relation, 106
  - subtyping, 73, 76
  - Subversion, 195, 339
  - Symfony framework, 13, 52, 262, 266
    - Ticket 2.0 implementation, *see* Ticketsf, *see* Ticketsf-mini
  - [Sync Wire](#), 83, 93, 132
    - example, 325
    - implementation with model completion, 326
    - model completion, 325
  - syntax
    - first-order logic, 97
    - invariants, 48
    - of a metamodel, 48
    - visual, *see* visual notation, *see* visual syntax
  - system metaphors, 58
  - system reuse, 63
- ## T
- tagged values, 21
  - TDD, *see* test-driven development
  - template engines, 174
  - Templates* component, 215
  - templating language, 154
  - [Temporary Variable](#), 111
    - shape colour, 204
  - test cases
    - documentation, 367
    - in IAML, *see* IAML test suite
  - test-driven development, 33, 270
  - [Text Message](#), 144
  - textual languages, 255, 269
  - thesis statement, 3
  - throughput, 12

- Ticket 2.0, 6, 24–26, 34, 231, 261, 267
  - evaluation, 88, 262
  - IAML implementation, *see* Ticketiaml
  - implementations, 340, 357
  - optimisation, 268
  - Symfony implementation, *see* Ticketsf, *see* Ticketsf-mini
  - test cases, 268
- Ticketiaml, 26, 85, 233–252, 357
  - implementation, 233–252, 347–356
  - screenshots, 251, 329
  - source code, 340
  - use of model completion, 325–326
  - visual model instance, 233–248
  - visual model instance examples, 103, 113, 119, 121, 134, 139, 142, 151, 327, 328
- Ticketsf, 26, 232–233, 357
  - screenshots, 253
  - source code, 340
- Ticketsf-mini, 26, 252, 357
  - source code, 340
- time type, 73
- total cost of ownership, 1
- traceability, 48
- transient properties, 182
- transitive closure, 57, 182
- Turing completeness, 36, 126
- type reasoning, 73
- type systems, 72–75, 106–112, 262
  - implementation, 216
  - MDA architecture, 107–108
- type unions, 73, 101, 256
- typed language, 72
- types
  - atomic, 72
  - built-in, 107
  - complex, 76
  - derived, 73, 262
  - primitive, 72
  - subtyping, *see* subtyping
- U
  - UML, 3, 21–22, 58, 66, 76, 91, 115
    - component diagram, 195
    - deployment diagram, 151
    - developing a modelling language, 31
    - evolution, 90
    - evolutionary development, 32
    - extensibility, 21
    - hierarchical design, 65
    - hierarchical modelling, 92
    - history, 32, 39, 58
    - interaction sequence diagrams, 127
    - meta-metamodel, 40
    - metamodel metrics, 255
    - package diagram, 95
    - sequence diagrams, 77
    - shapes, 203
    - specification complexity, 67
    - state diagrams, 80
    - state machines, 80
    - stereotypes, *see* stereotypes
    - template parameters, 77
    - user interface modelling, 145
  - UML activity diagrams, 59, 66, 70, 87, 127–130, 152
    - control flow, 128
    - multithreading, 129
    - rationale, 127
  - UML class diagrams, 59, 76, 77, 91, 92, 114, 152, 362
    - «multiple» stereotype, 106
  - UML Profiles, 22, 67, 164
  - UML-based Rule Modelling Language, *see* URML
  - UML-based Web Engineering, *see* UWE
  - UMLi, 145
  - Unified Modeling Language, *see* UML
  - Unix permissions, 82
  - ur-type, 74
  - URL, 150
    - fragment, 19, 230
  - URML, 67
  - use cases, 17, 275–320
  - user instances, 137
  - user interface metaphor, 152
  - user interfaces, 12, 16, 145–150, 228
    - layout, 90
    - multiple, 86

- users, 15, 80–83, 136–140, 227, 263
  - authentication, *see* authentication
  - authorisation, *see* authorisation
  - identity, 110, 139
  - profile attributes, 95, 136
  - scope, 80
- UServ Business Rules, 180
- UWE, 3, 22–23, 45, 59, 80, 84
  - metamodel metrics, 255
  - navigation model, 152
  - user interface modelling, 145

## V

- V&V, 55
- validation, *see* model validation
  - comparison with verification, 55
- Value, 97, 110, 268
  - shape colour, 204
  - shape style, 203
- variables, 98, 121, 122
- variant type, 115
- VBScript, 10
- verification, 17, *see* model verification
  - comparison with validation, 55
- verification languages, 56–58, 264
  - functions-based, 57, 191, 220, 221
  - higher-order logic, 57
  - model checking, *see* model checking
  - relations-based, 57, 191, 221
- version, 160
- View Mapping Language, 60
- view mapping model, 269
- Visible Thing, 146–150, 153
  - classifiers, 156
  - shape style, 203
  - visibility, 95, 146
- visual layout, *see* layout
- visual metaphors, 58, 151–152, 267
  - in hierarchical modelling, 92
- visual modelling, 58–61
  - complexity, 63–66
  - environments, 167–173, 264
  - gallery of common shapes, 168
  - implementation, 198–210

- metaphors, *see* visual metaphors
- software, 59
- visual notation, 59–61
  - information capacity, *see* cognitive effectiveness
- Visual Studio, 164
- visual syntax, 202–205
- VisualAge for Smalltalk, 105, 152
- Visualization and Modeling SDK, 164
- VML, *see* View Mapping Language

## W

- W2000, 24
- W3C, 1
- waterfall model, 32
- weakly checked language, 72, 106, 109, 262
- web applications, 1–2
  - metrics, 26
  - non-functional verification of, 17, 54
  - rich, *see* Rich Internet Applications
  - security risks, *see* security risks
  - service-enabled, 1
  - technologies, 9
- web browser, 1
- web browsers, 15
- web development
  - with frameworks, 13
- Web Engineering, 2
- Web Information Systems, 24
- Web Ontology Language, *see* OWL
- web server, 1
- web services, 2, 9, 16, 144, 295
- WebDSL, 23
- WebML, 3, 20, 22, 59, 64
  - CASE tool, 20
  - metamodel metrics, 255
  - user interface modelling, 145
- WebRatio, 20
- Whitehorse framework, 164–166, 170–171
- Wire, 105, 131–135
  - classifiers, 156
  - comparison to Actions, 131
  - comparison to ECA Rules, 131



implementation through model completion, **Y**  
325–326  
inference rules, 132  
wires, 131–135, 263  
workflows, 71, 219  
World Wide Web, 1  
World Wide Web Consortium, *see* W3C  
WSDL, 9  
WUML, 22

## **X**

XHTML, 2, 9, 10  
XMI, 43, 163, 164, 197, 232, 347–356  
XML, 2, 9, 10, 19, 357  
meta-metamodel, 40  
schema validation, 76  
stylesheets, *see* XSLT  
XML Schema, 41, 48, 76, 114, 174, 228  
datatypes, *see* XSD datatypes  
expressiveness, 172  
metamodel metrics, 255  
typing, 73  
XML-RPC, 9, 145, 295  
XMLHttpRequest, 2, 10  
Xpand, 175–176, 264  
expressions language, 175  
implementation, 215  
incremental generation, 270  
XPath, 72, 174  
XQuery, 111, 174  
fn:contains function, 216  
fn:not function, 98, 125  
op:numeric-equal function, 125  
op:numeric-less-than function, 103  
**XQuery Function**  
implementation, 216  
XSD datatypes, 73, 107–108, 114, 262  
casting, 108  
within IAML, 108, 197  
**XSD Simple Types**, 101  
XSLT, 173–174, 222  
Xtend, 175, 188  
implementation, 215  
Xtext, 175, 269