

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.

**SAME**  
**Structured Analysis Modelling Environment**  
**The Design of an Executable Data Flow Diagram**  
**and Dictionary System**

A dissertation presented  
in partial fulfillment of the requirements  
for the degree of  
Doctor of Philosophy in Computer Science  
at Massey University

Thomas William George Docker

1989

## Abstract

The research reported in this thesis has been an investigation into the use of data flow diagrams as a prototyping tool for use during the analysis of a system. Data flow diagrams are one of the three main tools of structured systems analysis (the other two are a data dictionary, and some means for representing process' logic, such as minispecs).

The motivation for the research is a perceived need for better tools with which analysts and end-users can communicate during the requirements gathering process. Prototyping has been identified by many researchers and practitioners as such a tool. However, the output from the requirements analysis phase is the specification, which is a document that should provide the framework for all future developments of the proposed system (and should evolve with the system). Such a document should be provably correct. However this is seen as an ideal, and the most that can be hoped for is a document which contains within it a mixture of formality.

Executable data flow diagrams are considered to provide an environment which serves both as a means for communication between analysts and end-users (as they are considered relatively easy to understand by end-users), and as a method for providing a rigorous component of a specification. The rigour comes from the fact that, as demonstrated in this thesis, data flow diagrams can be given strict operational semantics based on low level ('fine-grain') data flow systems. This dual focus of executable data flow diagrams is considered significant.

Given the approach adopted in the research, executable data flow diagrams are able to provide an informal, flexible framework, with considerable abstraction capabilities, that can be used to develop executable models of a system. The number of concepts involved in providing this framework can be small. Apart from data flow diagrams themselves, the only other component proposed in the research is a system dictionary in which the definitions of data objects are stored. Procedural details are de-emphasised by treating the definition of data objects as statements in a single-assignment programming language during the execution of a model.

To support many of the ideas proposed in the research, a prototype implementation (of the prototype tool) has been carried out in Prolog on an Apple Macintosh. This system has been used to produce results that are included in this thesis, which demonstrate the general soundness of the research.

## Acknowledgements

I would like to thank Professor Graham Tate, my chief supervisor, who has provided useful guidance and support over the time taken to carry out and report on the research discussed here. I would also like to thank Professor Mark Apperley, who as my second supervisor, provided assistance at a critical time in the production of this thesis. As well as these, thanks go to Dr John Hudson and Chris Phillips for reading various portions of this tome. Last, but not least, the support provided by all the Dockers is much appreciated.

# Contents

List of figures and tables	xi
<b>Chapter 1: Introduction</b>	<b>3</b>
1.1 Motivation for the research	3
1.1.1 Methods, methodologies, tools, and techniques	6
1.1.2 Formal specifications, and formal methods	6
1.1.3 Informal, semi-formal, and formal	7
1.1.4 Semi-formal techniques in the specification of requirements	8
1.1.5 Software development environments	9
1.1.6 The software development process and the software life cycle	10
1.1.7 Models, executable models, and prototypes	12
<i>Prototypes, and prototyping</i>	13
1.2 Objectives of the research	14
1.3 The approach	14
1.4 Structure of the dissertation	16
<b>Chapter 2: Structured systems analysis</b>	<b>17</b>
2.1 Introduction	17
2.2 Component tools of SSA	18
2.3 Data flow diagrams	18
2.3.1 An application hierarchy of data flow diagrams	20
2.4 Data dictionary	23

2.4.1	Defining data objects.....	24
	<i>Data structures and abstractions</i> .....	24
2.5	Process transformations .....	26
2.5.1	Structured English.....	26
2.5.2	Decision tables.....	28
2.5.3	Decision trees.....	28
2.6	Combining the tools.....	30
2.7	Using SSA in specifying requirements .....	31
2.7.1	The positive features of data flow diagrams for use in specifying requirements .....	31
2.7.2	Common ways of misusing data flow diagrams.....	32
	<i>Avoiding procedural details in data flow diagrams</i> .....	35
	<i>Avoiding control and physical details in data flow diagrams</i> .....	36
2.8	A dictionary as a general resource .....	36
2.9	Executable data flow diagrams.....	38
2.10	Summary .....	39
 <b>Chapter 3: Data flow systems</b>		<b>40</b>
3.1	Introduction .....	40
3.1.1	An initial classification, and some definitions.....	41
3.2	Data-driven systems.....	42
3.2.1	Conditionals and loops .....	45
3.2.2	Karp and Miller – a reference data-driven model .....	51
3.2.3	Fine-grain data-driven architecture features .....	52
	<i>Direct communication</i> .....	53
	<i>Packet communication</i> .....	53
	<i>Static and dynamic architectures</i> .....	55
	<i>Enabling conditions and output conditions</i> .....	60
	<i>Summary of fine-grain data-driven systems</i> .....	61
3.3	Demand-driven systems .....	63
3.3.1	String reduction.....	64
3.3.2	Graph reduction .....	65
3.3.3	Demand-driven systems and functional languages.....	67
3.4	Data flow systems and data flow diagrams.....	68
3.4.1	Fine-grain data flow semantics applied to data flow diagrams.....	68
3.4.2	Input to output set transformations.....	72
3.4.3	Treating data flow diagrams and transformations independently.....	73
3.5	Summary .....	74

<b>Chapter 4: The data-driven model in SAME</b>	<b>79</b>
4.1 Introduction .....	79
4.2 The operational semantics of a simple data flow diagram model(DFDM1), and its comparison with the Karp and Miller data-driven model.....	81
4.2.1 External entities and data stores.....	84
<i>External entities</i> .....	84
<i>Data stores</i> .....	86
4.3 The operational semantics of DFDM2.....	87
4.3.1 Limited import and export sets.....	88
<i>Limited import sets</i> .....	88
<i>Conditional generation of data flows and limited export sets</i> .....	90
4.3.2 Composition and decomposition of group objects .....	93
4.4 Structural completeness of data flow diagrams .....	95
4.4.1 Structurally complete data flow diagrams .....	97
4.4.2 Structurally incomplete data flow diagrams .....	98
4.4.3 Invalid data flow diagrams .....	100
4.5 Levels of refinement.....	100
4.5.1 Hierarchy of data flow diagrams .....	101
4.5.2 Process sets .....	102
4.6 Applications in the top level model .....	103
4.7 Parallelism in the top level model.....	105
4.8 Deadlocks.....	106
4.9 Summary .....	107
<b>Chapter 5: The demand-driven model in SAME</b>	<b>108</b>
5.1 Introduction .....	108
5.2 The Ægis language .....	109
5.2.1 Options, conditionals and repeats .....	112
<i>Options</i> .....	112
<i>Conditionals</i> .....	112
<i>Repeats</i> .....	113
5.3 Demand-driven interpretation of Ægis definitions.....	116
5.3.1 Constructors .....	117
<i>Tuple constructors</i> .....	118
<i>Stream constructors</i> .....	119
<i>Basic type constructors</i> .....	120
<i>"Don't care" and empty values</i> .....	121
5.3.2 Operations.....	122

5.4	Naming and binding.....	123
5.4.1	Naming .....	123
	<i>Environment, program, and working variables</i> .....	123
	<i>Version control and naming</i> .....	124
	<i>Naming of objects within SAME</i> .....	124
5.4.2	Binding .....	126
5.5	Other characteristics of Ægis and the demand-driven executable environment .....	128
5.5.1	Referential transparency .....	128
5.5.2	Call-by-need and lazy evaluation .....	128
5.5.3	Typing and polymorphism .....	129
	<i>Strong, static, and dynamic typing</i> .....	130
	<i>Polymorphism</i> .....	131
5.6	Language design principles and Ægis.....	134
5.6.1	Procedural abstraction .....	135
5.6.2	Data type completeness .....	135
5.6.3	Declaration correspondence .....	135
5.7	Summary .....	136
 <b>Chapter 6: The complete architecture of SAME</b>		<b>137</b>
6.1	Introduction .....	137
6.2	A conceptual architecture for SAME .....	137
6.2.1	SYD.....	139
	<i>The structure of the dictionary, and the bindings between objects</i> .....	140
	<i>Data flow diagrams as views onto data objects in the dictionary</i> .....	141
6.2.2	SYP.....	146
	<i>Static definition facilities</i> .....	146
	<i>The external entity interface</i> .....	147
	<i>Data flow management (DFM)</i> .....	148
	<i>Multiprocessing and the scheduling of processors</i> .....	148
6.3	Specifications and executions .....	149
6.3.1	Specification of application environments, applications, data flow diagrams, and data objects.....	149
6.3.2	The execution of an application.....	149
	<i>Starvation</i> .....	150
	<i>Missing data objects</i> .....	150
	<i>Type conflicts</i> .....	151
	<i>Inconsistencies, and their interpretation</i> .....	152

<i>Semantic errors</i> .....	152
6.4 Data stores in SAME .....	152
6.4.1 Methods of access.....	154
6.4.2 Operations.....	155
6.4.3 Exceptions handling.....	156
6.4.4 Name mappings .....	156
6.4.5 Conceptual view of a data store.....	159
6.4.6 A data flow view of data stores.....	160
<i>Referential transparency</i> .....	161
6.5 Summary .....	162
<b>Chapter 7: An implementation</b> .....	<b>163</b>
7.1 Introduction .....	163
7.1.1 Main features of the implementation .....	164
7.1.2 major features of the full SAME system that have not been implemented.....	164
7.2 An introduction to the definition subsystem through a simple example – finding the real roots of a quadratic equation.....	165
7.2.1 Creating a new application, and drawing a data flow diagram.....	165
7.2.2 Defining data objects.....	167
7.2.3 Displaying data objects, their types, and their dependencies.....	168
7.3 Building and running an executable model .....	172
7.3.1 Defining an executable process set.....	172
7.3.2 Running the model .....	173
7.3.3 Controlling the execution process .....	176
7.3.4 Tracing the exercising of a model .....	177
7.3.5 Exporting to external entities.....	178
7.3.6 Execution time exceptions.....	179
7.3.7 Exercising processes.....	180
<i>The context of a process</i> .....	181
<i>The fundamental algorithm for creating object instances</i> .....	182
7.4 Applications with multiple levels of data flow diagrams.....	182
7.4.1 Refining (exploding) data flow diagrams.....	183
7.4.2 'Scope' of objects .....	184
7.4.3 Building an executable model.....	184
7.4.4 Hook composed data flow instances.....	185
7.5 More error examples .....	186
7.5.1 Missing data object definition.....	186
7.5.2 No importers for a data flow.....	187

7.6	Limited import sets, conditional exports, and loops.....	188
7.7	Prolog as the implementation language .....	191
7.8	Summary .....	192

## **Chapter 8: An example analysis** **193**

8.1	introduction.....	193
8.2	A SAME model of the order processing example.....	194
8.2.1	The application data flow diagram hierarchy.....	194
8.2.2	The data object definitions for the application.....	196
8.3	The first prototype.....	201
8.3.1	The data stores contents .....	202
8.3.2	Selected details from the development of the first prototype .....	202
8.4	The second prototype .....	211
8.5	Summary .....	216

## **Chapter 9: Alternative architectures** **219**

9.1	Introduction .....	219
9.2	Other executable coarse-grain data flow schemes.....	220
9.2.1	The LGDF approach of Babb.....	220
9.2.2	The Ada information management system prototyping environment of Burns and Kirkham.....	222
9.2.3	The DataLink environment of Strong.....	223
9.3	Structured Analysis Simulated Environment (SASE) .....	224
9.3.1	META.....	225
9.3.2	The SASE process sub-system .....	226
9.3.3	SASE as a means for building implementation models .....	227
9.4	Comparative summary .....	227
9.5	Networks of von Neumann systems.....	229
9.6	Summary .....	231

## **Chapter 10: Conclusions and further research** **232**

10.1	Summary and conclusions.....	232
10.1.1	Objectives of the research .....	233
10.1.2	That the executable model be rigorous enough to form part of the specification .....	233
10.1.3	That the tool should have a small number of (simple) concepts.....	234
10.1.4	That procedural details should be de-emphasised .....	234
10.1.5	That the tool should incorporate high levels of abstraction in a relatively simple manner.....	234

---

10.1.6	That the tool should make effective use of graphics.....	235
10.1.7	That the tool should provide 'soft' recovery from errors.....	235
10.1.8	That the tool should be able to execute 'incomplete' models.....	236
10.1.9	Primary objective.....	237
10.2	Further research.....	237
<b>Glossary</b>		<b>239</b>
<b>Bibliography</b>		<b>257</b>

# Figures and tables

## Figures

1.1	The waterfall model of the software life cycle, showing the overlapping of stages .....	12
2.1	Comparison of the Gane and Sarson, and De Marco data flow diagram notations.....	18
2.2	Context, or Level 0, data flow diagram for an order processing system .....	20
2.3	Level 1 refinement of process ORDER PROCESSING.....	21
2.4	Level 2 refinement of process PRODUCE INVOICE.....	22
2.5	The hierarchy of processes for the order processing application modelled in Figures 2.2 to 2.4.....	22
2.6	A possible data structure hierarchy of the INVOICE data flow shown in Figures 2.2 to 2.4.....	24
2.7	A structured English minispec for calculating the status of a customer .....	27
2.8	A decision table for calculating the status of a customer.....	28
2.9	A decision tree for calculating the status of a customer .....	29
2.10	An integrated view of three tools described in Section 2.5, showing how they combine to form a logical model of an application .....	30
2.11	Excerpt from a 'loose' data flow diagram in Wasserman <i>et al.</i> [WPS86] .....	33
2.12	Excerpt from a 'loose' data flow diagram in Booch [Bo86] .....	34

3.1	Data dependency graph for finding the (real) roots of a quadratic.....	43
3.2	Data flow graph for finding the (real) roots of a quadratic.....	44
3.3	A data-driven program for finding the (real) roots of a quadratic.....	45
3.4	A data flow graph for the conditional <code>if x &gt; y then a := v1 else a := v2</code> .....	46
3.5	A cyclic data flow graph for calculating the factorial of N.....	47
3.6	The general structure of a 'safe' while-loop in a data flow graph.....	48
3.7	The occurrence of deadlock in a data-driven program graph.....	49
3.8	The occurrence of a race condition.....	50
3.9	The functional structure of a processing element in a token storing data-driven system.....	54
3.10	The functional structure of a processing element in a token matching data-driven system.....	55
3.11	A conceptual snapshot of an Id data flow program showing the token <code>&lt;u.c.s.i, 4&gt;</code> on the arc connected to input port 2 of the instruction (activity) <code>s</code> .....	57
3.12	A data flow graph for the processing of the loop by the U-interpreter.....	59
3.13	A categorisation of data-driven machines. The machines discussed in this chapter are shown in the rectangles.....	61
3.14	A demand-driven program for finding the (real) roots of a quadratic.....	63
3.15	A string reduction execution sequence for the part of the program in Figure 3.14 which finds the first root.....	64
3.16	A graph reduction program corresponding to Figure 3.14.....	66
3.17	The program graph of Figure 3.16 with reverse pointers.....	67
3.18	Level 1 data flow diagram, and data dictionary definitions for finding the (real) roots of a quadratic equation.....	69
3.19	Level 1 data flow diagram for finding the (real) roots of a quadratic application.....	70
3.20	Accessing the data store CUSTOMERS using CUST_# as the key.....	71
3.21	Processing one COURSE_CODE against multiple STUDENT_# tokens.....	71
4.1	Level 0 data flow diagram for the order processing example.....	80
4.2	Level 1 data flow diagram.....	80
4.3	Level 2 data flow diagram for process PRODUCE INVOICE.....	80
4.4	Data flow diagram hierarchy for the order processing application, showing the leaf processes shaded.....	80
4.5	External entity e1, CUSTOMER, as the set {INVOICE, ORDER_DETAILS, UNFILLABLE_ORDER} of phantom nodes.....	86
4.6	An example which shows the decomposition and	

	composition of data flows in data flow diagrams.....	93
4.7	A structurally incomplete form of Figure 4.2.....	100
4.8	Possible different data flow process explosion trees created during the analysis of an application .....	101
4.9	Virtual leaf process data flow diagram, $\delta_{OP}$ , for the order processing application .....	104
4.10	A snapshot of order processing transaction histories .....	105
5.1	Dictionary definitions relating to INVOICE.....	110
5.2	Example invoice .....	114
5.3	Dependencies graph for INVOICE.....	117
5.5	The identity function <i>id</i> implemented in four languages that support parametric polymorphism.....	132
6.1	A conceptual architecture for SAME .....	138
6.2	Dictionary definitions relating to the objects in process 3, PRODUCE_INVOICE.....	139
6.3	An example invoice corresponding to the definitions in Figure 6.2.....	140
6.4	Data object dependencies in process 3, PRODUCE INVOICE.....	142
6.5	Data object dependencies in the refinement to process 3, PRODUCE INVOICE .....	143
6.6	Using an example to show the associations (bindings) between objects in SYD .....	145
6.7	Accessing the data store CUSTOMERS using CUST_# as the key .....	154
6.8	Adding a 'control' dimension to a data flow diagram in which the keys for accessing data store tuples (among other things) can be specified .....	155
6.9	Part of a data flow diagram implicitly showing multiple data flows referencing the same data store object (not necessarily the same instance).....	158
6.10	A conceptual view of a SAME data store.....	160
7.1	Naming an application .....	165
7.2	A Level 0 data flow diagram in the manner of Figure 3.18 .....	166
7.3	The structural details of the data flow diagram in Figure 7.2 .....	167
7.4	Defining the data object <i>coefficients</i> to be the tuple (a, b, c).....	167
7.5	A dialogue containing a menu for selecting the data objects to display .....	169
7.6	Display of all data objects currently in the dictionary .....	169
7.7	The internal representation of data object definitions for the roots example.....	169
7.8	Redundant <i>r<sub>hs</sub></i> facts which are used extensively in displaying data object dependencies.....	170
7.9	A listing of data objects showing their (inferred) types.....	170
7.10	A request to display the dependency graph, to the selected depth, of the data objects depended on by data flow	

	roots in process <code>findRootsOfQuadratic</code> .....	171
7.11	Data dependency graph for data flow roots in process <code>findRootsOfQuadratic</code> .....	172
7.12	Specifying the executable model process set.....	173
7.13	Request for user to supply external entity generated data flow instances.....	173
7.14	Sequence of requests for sub-object values for an instance of data flow <code>coefficients</code> .....	176
7.15	An example full trace.....	177
7.16	The executable model representation of external entity <code>analyst</code> .....	177
7.17	An example error display prompt generated by SAME during the creation of an instance of the data object <code>root1</code> . Particularly, a request to find the square root of -15 has been trapped. (The user has supplied a further invalid value. See Figure 7.18.).....	178
7.18	Following the user supplying an invalid value (as shown in Figure 7.17), SAME displays an error message. The user must supply a positive number before SAME will continue.....	179
7.19	Messages generated under full trace which relate to the two attempts to find the square root of a negative number.....	179
7.20	The data flow reduction graph for data flow roots evaluated in the context of process <code>findRootsOfQuadratic</code> .....	181
7.21	A particular refinement of the process <code>findRootsOfQuadratic</code> into the two processes <code>computeRoot1</code> and <code>computeRoot2</code> .....	183
7.22	A particular refinement of the process <code>findRootsOfQuadratic</code> into the two processes <code>computeRoot1</code> and <code>computeRoot2</code> .....	184
7.23	A request to form an application model from the leaf level processes that are descendants of the process <code>findRootsOfQuadratic</code> (namely the two processes <code>computeRoot1</code> and <code>computeRoot2</code> ).....	185
7.24	An instance of the data flow roots exported to the external entity <code>analyst</code> by the hook <code>roots</code> .....	185
7.25	Amendments to data object definitions for the roots application, with an omission in the definition of the object <code>sqr</code> .....	186
7.26	An error dialogue of the same general format as Figure 7.17, which indicates that no value could be found nor generated for data object <code>sqr</code> .....	187
7.27	Following the declaration of the data object <code>sqr</code> as <code>sqr&lt;=sqrt(bsq - fourAC)</code> , the object dependencies will be as shown.....	187
7.28	A different refinement of process <code>findRootsOfQuadratic</code> .....	188
7.29	An error dialogue stating that no importers exist for data flow <code>n1</code> .....	188
7.30	A data flow diagram which contains a loop.....	189

7.31	Data object definitions for the looping application; and an execution trace.....	190
7.32	Prompt to the user to define the action to take when a currency mismatch occurs, in the case where the automatic flushing of instances has been turned off.....	191
8.1	Level 0 data flow diagram for the revised order processing application .....	194
8.2	Level 1 refinement of <code>checkAndFillOrder</code> .....	195
8.3	Level 1 refinement of <code>produceInvoice</code> .....	196
8.4	Data object definitions .....	200
8.5	Data dependency graph for data object <code>invoice</code> .....	200
8.6	Data object definitions which differ from those given in Figure 8.4 .....	201
8.7	Data store tuples used in the first prototype .....	202
8.8	Data store access details for constructing instances of data flow <code>customer_details</code> .....	204
8.9	The objects to be mapped between the data flow <code>adjusted_credit</code> and the customer data store tuple component <code>cust_available_credit</code> .....	208
8.10	The generation of an invalid instance of <code>cust_available_credit</code> .....	210
8.11	The instance of <code>rejected_order</code> , which correctly identifies the customer's lack of available credit.....	211
8.12	Revised form of Figure 8.1, with the data store <code>parts</code> replaced by the external entity <code>parts</code> .....	212
8.13	An instance of data object <code>updated_part_details</code> which contains multiple <code>parts_remaining</code> instances .....	215
8.14	An instance of data object <code>invoice</code> which contains multiple line item instances .....	216
9.1	Executable META minispec for <code>process p3, PRODUCE INVOICE</code> .....	225
9.2	A conceptual structure for a coarse-grain processing element.....	230

**Tables**

I	Important properties of requirements and design specifications, as identified by Howden [Ho82a].....	4
II	The data dictionary language notation of De Marco .....	25
III	A comparison of some reported date-driven architectures.....	62
IV	Example tuple instances for specific definitions .....	118
V	Example tuple instances for group object definitions .....	121
VI	Example tuple instances using basic type constructors .....	121
VII	The possible bindings between dictionary objects.....	144
VIII	A comparison of some coarse-grain data flow schemes .....	228

# Part I

Part I contains the background material to the research reported in the dissertation.

In Chapter 1 the motivation for the research is described, along with a statement of the objectives. The principle objective has been to investigate the use of executable data flow diagrams as a prototyping tool for use during the analysis phase of the software life cycle. The approach adopted to achieve this objective is also given.

Chapters 2 and 3 contain discussions of the more important support material. In Chapter 2 structured systems analysis, which is the method that has data flow diagrams as a component tool, is discussed. Both advantages and disadvantages in the use of structured systems analysis, and data flow diagrams in particular, are enumerated.

In Chapter 3 low level ('fine-grain') data flow schemes are discussed, and characteristics which are particularly useful to a high level ('coarse-grain') data flow system are identified.

# Chapter 1

## Introduction

### 1.1 Motivation for the research

In the design of a software system, the output from a requirements capturing exercise is the **specification**, which is a document that contains an abstract computer-orientated representation of the set of **end-user** requirements.<sup>1</sup>

Producing a correct specification is seen to be the key to the successful, cost-effective development of software systems [Bo76]. There are, however, problems in knowing when a specification is correct, and even when it is complete; not least because of the problems of adequately specifying what is required in the first place. In the context of the specification of requirements, Howden has stated that ([Ho82a], p. 72):

*'The principle idea in the analysis of requirements specifications is to make sure that they have certain necessary properties.'*

Howden tabulates some of the more important of these properties, included here as Table I.

Some of the properties, notably completeness, must be viewed as ideals which cannot be achieved in many software development projects.

---

<sup>1</sup> Terms in bold type are included in the Glossary. In general, the term 'end-user(s)' will refer to the potential users of the system being analysed, who are considered not to be software developers. The terms 'user' and 'analyst' are used to refer to the person(s) carrying out the analysis. The term 'user' generally appears when the application of an analysis technique, or tool, is being discussed.

Property	Comments
Consistency	Specifications information must be internally consistent. If the information is duplicated in different documents, consistency between copies must be maintained.
Completeness	Specifications must be examined for missing or incomplete requirements and design information. All specification functions must be described, including important properties of data.
Necessity	Each part of the specified system should be necessary and not redundant.
Feasibility	The specified system should be feasible with existing hardware and technology.
Correctness	In some cases, it is possible to compare part of the specification with an external source for correctness.

Table I: Important properties of requirements and design specifications, as identified by Howden [Ho82a].

Parnas and Clements enumerated various problems in the area of software design [PC85]. Some of particular interest, are couched below in requirements specification terms:

- In most cases the end-users do not know exactly what they want and are unable to state what they do know.
- Even if the initial requirements were known, other requirements usually surface as progress is made in the development of the software.
- Even if all of the relevant facts had been elicited and included in the specification, experience shows that human beings are unable to fully comprehend the plethora of details that must be taken into account in order to progress into the design and building of a correct system.
- Even if all of the detail needed could be mastered, all but the most trivial projects are subject to change for external reasons. Some of those changes may invalidate previous requirements.
- Human errors can only be avoided if one can avoid the use of humans. No matter how rational the requirements specification process, no matter how well the relevant facts have been collected and organised, errors will be made.

These problems suggest that as requirements are likely to change during analysis, flexibility should exist in the methods and tools used to capture requirements. As well, consistency needs to be maintained. In fact, checking for consistency is seen to be the property in Table I which is the most achievable using computer tools. Given the right tools, computers are particularly good at this type of task.

The correct specification of requirements is seen as the key to the successful, cost-effective development of software systems [Bo76]. It is also generally agreed that to be able to validate requirements, they must be rigorously specified. As Davis succinctly puts it ([Da88], p. 1100):

*'Use a formal technique when you cannot afford to have the requirement misunderstood.'*

In an attempt to improve both the capturing of requirements, and the production of a specification document that can be effectively used throughout the software development process, considerable effort is being expended on developing formal specification methods (see, for example, [GT79a, BO85, Wa85, He86, Jo86, ZS86]). However, most, if not all, of the techniques proposed use formal methods and languages which require a reasonably sophisticated level of mathematical maturity to be fully understood. This tends to make them unsuitable as communications media between analysts and most end users; which is unfortunate, as a further major perceived parameter in the requirements capturing process is the active involvement of end users (see, for example, [Al84, BW79, CM83, De78, Ea82, IO84, MC83, Ri86, SP88]).

Speaking specifically about understanding software requirements specifications, Davis has observed that ([Da88], p. 1112):

*'understandability appears to be inversely proportional to the level of complexity and formality present.'*

There can be seen to be a tension between the need on the one hand for an unambiguous, succinct, specification of requirements as the output from the analysis process, and (at the least) the need to validate those requirements with end users.

Part of the purpose of the research reported herein has been an attempt to address some aspects of this tension by adding formality, in the shape of a strict syntax and operational semantics, to the **data flow diagrams of structured systems analysis (SSA)**, a semi-formal technique, to produce a **computer-assisted software engineering (CASE)** prototyping tool. Data flow diagrams are considered relatively easy to understand [De78, Ri86, YBC88], yet they have the potential to be viewed more formally as high level **data flow program graphs** [Ch79].

The subsequent sections of this introduction more fully develop some of the background to the research.

### 1.1.1 Methods, methodologies, tools, and techniques

Quite often confusion exists in the use of the words 'method' and 'methodology'. The sense in which they are used in this thesis is as follows [Fr80, MM85]:

*Definition:* A **method** consists of prescriptions for carrying out a certain type of work process; that is, it is a way of doing something. ♦

*Definition:* A **methodology** is a collection of *methods* and *tools*, along with the management and human-factors procedures necessary to their application. ♦

Also 'tool' is used with a particular meaning [Fr80, MM85]:

*Definition:* A **tool** is an aid, such as a program, a language, or documentation forms, that helps in the use of a *method*. ♦

Frequently, in this dissertation, the term 'technique' appears. It is used informally as an abstraction. For example, a set of objects may be described as 'techniques' when, say, some of them are 'methods' and the rest are (parts of) 'methodologies'.

### 1.1.2 Formal specifications, and formal methods

The application of formal methods is viewed by many as being necessary for the correct and unambiguous specification of objects (see, for example, [AP87, GM86, Jo86, LZ77]). Consequently considerable effort is being spent on research in this area. 'Formal methods' and 'formal specifications' are widely used terms that imply the use of strict syntax and semantics in the description of objects; whether the objects are statements, programs, requirements, or something else.

The following definitions make clear what is meant by 'formal specification' and the related term 'formal method':

*Definition:* A **formal specification** is a *specification* which has been defined completely in a language that is mathematically precise in both syntax and semantics. ♦

*Definition:* A **formal method** is a *method* with a rigorous mathematical basis. ♦

The extent to which formal methods can be successfully used is unknown. Although some formal methods have been used to specify significant applications [Su82, STE82], the correctness of the specifications has not been proved, and, in some cases, has been shown to be incorrect [Na82]. As discussed in the next section, it appears that the most that can be hoped for in practical situations is a specification in which amenable parts of the requirements have been formally specified [Na82]. Any specification which is not a formal specification will be described simply as a 'specification'. The integration of formal and informal specifications is considered necessary. As Gehani and McGettrick have put it ([GM86], p. vii):

*'Formal specifications do not render informal specifications obsolete or irrelevant; although they [formal specifications] can be checked to some degree for completeness, redundancy and ambiguity, and can be used in program verification, they are often hard to read and understand. Consequently, informal specifications are still necessary as an aid to the understanding of the system being designed; informal and formal specifications complement each other.'*

### 1.1.3 Informal, semi-formal, and formal

The problems with proving the correctness and general applicability of formal methods has led to the view that formal methods cannot be used without recourse to informal techniques for specifying requirements (nor even for specifying programs) [MM85, Na82, Fe88]. Naur has suggested that 'formality' should be viewed as an extension of 'informality' [Na82]. He states that

*'the meaning of any expression in formal mode depends entirely on a context which can only be described informally, the meaning of the formal mode having been introduced by means of informal statements.'*

Naur, himself, quotes from Zemanek discussing software development [Ze80]:  
*'No formalism makes any sense in itself; no formal structure has a meaning unless it is related to an informal environment [...] the beginning and the end of every task in the real world is informal.'*

The view of Naur is supported by Mathiassen and Munk-Madsen, who have taken Naur's arguments, which were directed at program development, and applied them to the more general area of systems development [MM85]. Both the views of Naur, and Mathiassen and Munk-Madsen, are supported here. As a consequence, the following are offered as definitions for 'informal' and 'formal' in the context of describing some object:

*Definition:* The **informal** description of an object is a description that is done without recourse to *formal methods*. ♦

*Definition:* The **formal** description of an object is a description that is done with recourse to *formal methods*. ♦

Note that a 'formal' description could include 'informal' descriptions within it, as it is 'with recourse to' rather than 'solely with'. The counter-argument does not apply: an 'informal' description contains no 'formal' descriptions within it.

It is possible to perceive of a spectrum of descriptions, going from informal at one end, to totally formal at the other end. This is in keeping with Naur's proposals [Na82].

The term 'semi-formal' is used loosely to describe any technique that is formal, but with distinctly informal components. An example would be the **structure charts** of **structured design** when interpreted using the algebraic approach(es) of Tse [St81, Ts85, Ts85a, Ts86, Ts87, YC79].

#### 1.1.4 Semi-formal techniques in the specification of requirements

Techniques of an informal nature for specifying requirements abound. The most widely used is narrative text, but this frequently results in large, ambiguous, and incomplete specifications that lead to communications problems between analysts and end users; particularly when attempts are made to validate requirements [De78, Da88]. Starting in the early 1970's, semi-formal **structured techniques** have been developed over the years in an attempt to improve both the approach to analysis, and to place the emphasis more on the graphical presentation of information as a better method of communications. Included in the structured approaches for the capturing and specification of requirements are, Structured Analysis and Design Technique (SADT) [Co85, Ro77, RS77, Di78, Th78], Information Systems work and Analysis of Changes (ISAC) [BH84, LGN81, Lu82], Software Requirements Engineering Methodology (SREM) [A177, A178, AD81, BBD77] which is more suited to embedded real-time systems, and the class of techniques called 'structured systems analysis' (SSA) [CB82, De78, GS79, We80].

All of these have quite powerful abstraction capabilities which allow, for example, objects in diagrams to be exploded into lower level diagrams in a **top-down** fashion.

SSA techniques are the most widely publicised and used techniques, and are based on data flow diagrams, which show the system in terms of data precedences: a **data-orientated** approach. The SSA techniques also happen to be the most informal of those mentioned. It is impossible to say whether their popularity is due to their relative simplicity, although some statistical evidence does exist to suggest that this may

be the case: in comparing the use of data flow diagrams and IDEFo (the graphically-based function modelling part of IDEF, a component of SADT), Yadav *et al.* concluded that data flow diagrams appear slightly easier to use [YBC88].

Though the graphical features of the SSA techniques are seen to aid communications between analysts and end users, they lack the necessary level of rigour to satisfactorily facilitate the validation of requirements [Fr80, Ri86]. The lack of rigour in these techniques stems from their generally free interpretation, which is due more to a lack of strict semantics than a lack of syntax. Unfortunately, this lack of rigour invites misuse [Do87]. It also leads to the possibility of incorrect, and ambiguous specifications. Consequently, as a specification technique, SSA suffers from many of the problems of narrative text. This is not surprising, because SSA still places a reasonably heavy reliance on the use of textual data, although its syntax is generally, but not completely, more formal than narrative text.

Some of the weaknesses of SSA are discussed in more detail in Chapter 2. At this time it should be noted that they exist, and that an attempt to add formality to SSA can be usefully applied to minimising the dependence on purely textual data. The means used to achieve this minimisation is sketched out in Section 1.3, while the details form the subject matter of Part II of this dissertation.

SSA has three major component tools which are of particular relevance in the dissertation. These are:

- *Data flow diagrams* – An **application** is modelled by a hierarchy of data flow diagrams which show how data flows through the application.
- *Data dictionary* – The description of data objects, and the transformations carried out on them (by **processes**), are maintained in a data dictionary.
- *Process specifications* – For each bottom level (**leaf**) process, its process specification (the **process logic**) describes how the data which flows into the process is transformed into the data which flows out of the process.

These and the other component tools will be discussed more fully in Chapter 2.

### 1.1.5 Software development environments

In looking to define any tool for the capturing of requirements, consideration should be given to the environment in which that tool will be focussed. The current approach in **software engineering** is to develop tools within a framework known as a **software development environment (SDE)**. SDEs are also known as **software engineering environments (SEEs)**, and **integrated project (or program) support environments (IPSEs)**.

The fundamental purpose of a SDE is to provide a computer-based set of methods and tools – a methodology – to support the **software (development) process**. The existence of a cohesive methodology is fundamental, as this encapsulates the process model used in software development. In Dowson's words ([Do86], p. 6),

*'We take the position that an unstructured "bag of tools" does not qualify as a software development environment.'*

Attempts have been made to define environments made up from existing methods and tools. Howden discusses the architecture for four possible SDEs, each based on the waterfall model of the software process [Ho82]. The differences between the environments is the number and sophistication of the methods and tools included. What is apparent is the large number of 'discontinuities' which exist between the different tools in each proposed environment. These discontinuities have to be bridged generally by manual means, which makes them error-prone and unsatisfactory for the development of other than small software projects.

The following definition emphasises the need for integration ([WD86], p. 5):

*Definition:* A **software development environment** is a coordinated collection of software tools organised to support some approach to software development or conform to some software process model. ♦

It is argued that the real value of a SDE comes from the integration between the various methods and tools that it uses. This integration is provided by a specialised **data base** environment. Conceptually, these specialised data bases have much in common with the more recent of the **data dictionary** systems, which also aim to provide an integrated view, and control, of (all) the objects in some context (whether, say, the context is an **enterprise**, or some division or department of that enterprise).

### 1.1.6 The software development process and the software life cycle

The underlying structure of a SDE is the particular software process development model adopted by the architects of the SDE. The purpose of this section is to determine what a process development model is, and whether a standard model and, hence, SDE exists into which the proposed tool could be usefully placed.

The software development process (also called the **software life cycle**) is frequently shown as consisting of a number of stages, such as *requirements*, *design*,

*implementation, testing, and operation and maintenance* [So85].<sup>2</sup> The activities carried out in each of these stages is described by Sommerville as ([So85], p. 3):

- *Requirements analysis and definition* – The system's services, constraints and goals are established by consultation with system end-users. Once they have been agreed, they must be defined in a manner which is understandable by both end-users and development staff.
- *System and software design* – Using the requirements definition as a base, the requirements are partitioned to either hardware or software systems. This process is termed systems design. Software design is the process of representing the functions of each software system in a manner which may be readily transformed to one or more computer programs.
- *Implementation and unit testing* – During this stage, the software design is realised as a set of programs or program units which are written in some executable programming language. Unit testing involves verifying that each unit meets its specification.
- *System testing* – The individual program units or programs are integrated and tested as a complete system to ensure that the software requirements have been met. After testing, the software system is delivered to the customer.
- *Operation and maintenance* – Normally (although not necessarily) this is the longest life cycle phase. The system is installed and put into practical use. The activity of maintenance involves correcting errors which were not discovered in earlier stages of the life cycle, improving the implementation of system units and enhancing the system's services as new requirements are perceived.

Figure 1.1 shows the **waterfall model** view of this process, including:

- The overlap between the stages – There are no 'clean' division points between the activities across stages.
- The feedback (and feed-forward) between the pre-operational development stages – The next stage in the process is dependent on work carried out in the previous stage(s) (feed-forward). Identifying errors, or accounting for changes, etc., require changes to previous stages (feedback).
- The feedback from the operational and maintenance stages – Once an application becomes live, errors may surface, or changes be required over time, which lead to a feedback to earlier stages.

---

<sup>2</sup> It is possible to define 'software development process' and 'software life cycle' to have significantly different meanings. Compare, for example, the definition for 'software (development) process' in the Glossary with the following definition for 'life cycle' ([MRY86], p. 83): '*The system life cycle is the period of time from the initial perception of need for a software version to when it is removed from service*'.

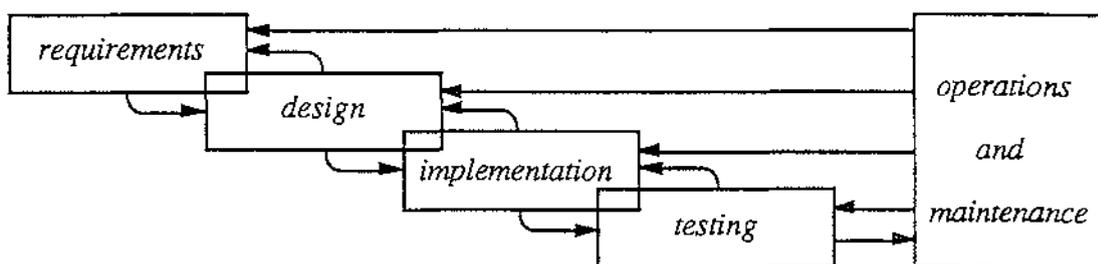


Figure 1.1: The waterfall model of the software life cycle, showing the overlapping of stages (based on Sommerville [So85], Figs 1.1 and 1.2).

The end points of the stages in the waterfall model are generally seen to coincide with major documentation and review points. They also tend to correspond with points at which major changes occur in the techniques and or environments used for the development, such as at the interface between (structured) design and implementation, where a switch is made from using two-dimensional **structure charts** to using a one-dimensional programming language [YC79].

The model in Figure 1.1 is extremely abstract, and a number of important features have been omitted, including:

- An indication of parallel activities within phases – Invariably, on other than the smallest projects, developers work in tandem. This is certainly true of the *implementation* phase, when a number of programmers will likely be concurrently developing modules.
- An indication of whether or not prototyping is supported, and if so, where.
- An explicit indication of where verification and validation take place.

Figure 1.1 highlights a current major problem in the description of the software process: the lack of a definitive **process metamodel** with which software process models can be described, and checked for correctness and completeness [PC85, WD86]. However, as this is a major research topic in itself, it will not be pursued further here. Instead, the waterfall model of Figure 1.1 is accepted as adequate for the purposes of the research reported herein.

### 1.1.7 Models, executable models, and prototypes

The use of models in analysis is now seen as fundamental. According to Quade ([Qu80], p. 31):

*'Analysis without some sort of model, explicit or otherwise, is impossible.'*

The following defines what a 'model' is understood to be:

*Definition:* A **model** of an object is a representation which specifies some but not all of the attributes of the object. ♦

In the development of computer software, models are seen to be most useful if they are executable [Ri86].

*Definition:* A **dynamic model** of an object is a model which can be made to carry out a set of operations, possibly in some specified sequence. ♦

An 'executable model' is merely a dynamic model, which in the context of software development specifies a software model that can be exercised on a computer.

### ***Prototypes, and prototyping***

As Carey and Mason have observed (CM83, p. 177), in computing:  
*'there appears to be little if any agreement on what a prototype is.'*

The following simple definition is considered adequate:

*Definition:* A **prototype** is a *model*. ♦

A prototype is either an abstraction of the object it is modelling, a 'mock-up', or it is a detailed representation of part of the object. SSA provides good facilities for modelling parts of systems, as described in Chapter 2.

By implication, the medium used to construct a prototype need not be the same as that used for the final object. A prototype of a menu system, for example, could be constructed using the transition diagram interpreter (TDI) part of RAPID/USE [WPS86], and then the real system could be constructed as part of a larger integrated project using a language such as PL/I.

*Definition:* **Prototyping** is a *method* for building and evaluating prototypes. ♦

The purpose of prototyping, as it is seen here, is the same as that stated by Carey and Mason ([CM83], p. 180):

*'Our focus in this paper is on improving the final information system product through use of prototypes to illuminate more clearly the [end-]user's real needs.'*

This view of prototyping, as a productive way for analysts and end-users to interact, is commonly held throughout the literature (see, for example, [AHN82, Al84, BW79, CM83, Ea82, IH87, JS85, KS85, MC83, NJ82, SP88]). No other purpose for prototyping is stressed here, although claims have been made for it as a replacement for the 'classical' software development process [NJ82]. See, for example, the discussion and references in Carey and Mason [CM83].

Different approaches to prototyping in computing have been enumerated [IH87, JS85]. Ince and Hekmatpour, provide the following taxonomy ([IH87], p. 9):

- *Throw-it-away' prototyping* – Which involves the production of an early version of a software system during requirements analysis. This is then used as a learning medium between the analyst and the end-user during the process of requirements elicitation and specification.
- *Incremental prototyping* – Where a system is developed one section at a time, but within a single overall software design.
- *Evolutionary prototyping* – Where a system is developed gradually to allow it to adapt to the inevitable changes that take place within an enterprise.

## 1.2 Objectives of the research

The principal objective of the research, has been to investigate the use of executable data flow diagrams as a prototyping tool during the analysis phase of the software life cycle.

Implicit in this objective are the following further objectives:

- That the executable model, which is a significant output of a prototyping exercise, be rigorous enough to form part of the specification, if required.
- That to serve as an adequate communications medium between analysts and end-users, the tool should:
  - have a small number of (simple) concepts;
  - de-emphasise procedural details;
  - incorporate high levels of abstraction in a relatively simple manner;
  - make effective use of graphics.
- To be an effective prototyping tool at the analysis stage, as well as the list of features just given, the tool should:
  - provide 'soft' recovery from errors;
  - be able to exercise 'incomplete' models.

## 1.3 The approach

In arriving at the objective(s) given in Section 1.2, the following five factors were identified as of particular importance to the successful capturing of requirements:

- *Active user involvement* – This is a long-held view in information systems development. De Brabander and Thiers cite a paper written in 1959 which proposes such an activity [DT84]. Active user involvement generally implies the need for informal and semi-formal methods and tools.
- *The use of graphical techniques in place of textual descriptions, wherever appropriate* – Graphic techniques abound in commerce: PERT charts, pie charts, histograms, and graphs, are notable examples. At the same time, purely textual descriptions have been much criticised [Da88, De78].
- *The use of executable models* – Particularly in the form of prototypes, as a means to illuminate clearly the needs of end-users [Al84, BW79, CM83, Ea82, MC83, Ri86, SP88]. A model should be viewed (at the least) as a form of documentation.
- *Powerful abstraction capabilities* – Analysis is a creative process which has to map complex real world problems into the specification of solutions [We81].
- *A specification should be unambiguous* – This implies the existence of strict semantics in the specification method(s), and ways of avoiding or checking for contradictions [AP87, Da88, GM86, Ho82a, Jo86, Ri86].

It was proposed that these factors can be addressed, to a significant degree, by adding formality to SSA.

Following an initial study into using the three SSA tools mentioned in Section 1.1.4, the approach adopted has been to specify the architecture for a tool based on two of those components – data flow diagrams, and the data dictionary – plus the development of a prototype (of the prototype system) to test out many of the ideas put forward.

The formality added to the data flow diagrams has three components:

- *A formal syntax for specifying data flow diagrams* – To ensure that only a consistent data flow hierarchy can be created, with valid data flow connections.
- *An operational semantics for data flow diagrams* – These define how a data flow diagram can be executed.
- *A consistent means of transforming data flows* – This is achieved by treating the definitions of data objects in the dictionary as programming language statements, when executing data flow diagram processes.

The tool is described as 'semi-formal'. Work to provide a completely formal 'back-end' is being undertaken separately from the research reported here.

Given the discussion in Section 1.1, the tool has not been fixed to any specific methodology or, by implication, to any specific SDE or software development process model. As a consequence of this, the tool can possibly have use beyond the requirements specification phase. However, this is not argued in the dissertation, but is suggested, in Chapter 10, as a possible topic for future research.

The tool is not considered a panacea for all the ills bedevelling the specifying of requirements. Again referring back to the discussions in previous sections, of necessity it is seen as one of a collection of informal to formal tools for use during analysis.

## 1.4 Structure of the dissertation

The thesis is structured in three parts. Part I contains this introductory chapter, and two further chapters which survey material relevant to the tool described in Part II.

Part II proposes a design for an executable data flow diagram tool in Chapters 4 to 6. Following this, in Chapter 7, a prototype implementation of the tool is discussed. Many of the ideas incorporated in the architecture of the executable data flow diagram environment have been incorporated into this prototype, which has been written in Prolog. It should be realised that no attempt was made to develop a complete commercial implementation. Having said this, the prototype source is over 400 Kbytes in size.

The final chapter in Part II contains a detailed example application developed on the system described in Chapter 7.

Part III contains two chapters. The first, Chapter 9, discusses other approaches to the execution of data flow diagrams. Included there is an outline description of a system that was also developed as part of this research, and is the precursor to the system described in Part II. Finally Chapter 10 discusses the findings of the research, and suggests further avenues of investigation.