M.O.O.T

META OBJECT ORIENTATED TOOL

A NOVEL META-CASE TOOL METHODOLOGY REPRESENTATION STRATEGY

A dissertation submitted in partial fulfilment of the
requirements for the degree of
Doctor of Philosophy in Computer Science

Massey University, New Zealand

David Charles Page

1998

Abstract

This thesis presents an investigation into current meta-CASE technology. The research focuses on CASE tool support for the concept of methodology, the representation of methodology syntax and semantics, and the support for re-use of methodology descriptions and software artefacts. A novel methodology representation strategy for meta-CASE tools is proposed and implemented with the development of a new meta-CASE tool (MOOT – Meta Object Orientated Tool).

The novel strategy propounded in this thesis uses an object-orientated meta-model and views methodology descriptions as potentially re-usable components. The coupling between methodology syntax and semantic descriptions is minimised so they can be re-used independently.

Two new modelling languages have been derived, to support the definition of syntax (NDL – Notation Definition Language) and semantics (SSL – Semantic Specification Language) of software engineering methodologies. Semantic descriptions are compiled to a platform independent representation (SSL-BC), which is executed on a purpose built virtual machine (SSL-VM). Late binding of syntax and semantic methodology descriptions is implemented with the development of Notation Semantic Mapping (NSM) tables. Two libraries of re-usable methodology description components, the Core Knowledge Base (CKB) and the Generic Object Orientated Knowledge Base (GOOKB), have been derived during this research.

Empirical results gained from applying the MOOT prototype demonstrated the flexibility, extensibility and potential of the novel methodology representation strategy. This approach permitted the implementation and modelling of UML and patterns, two recent advances of object technology that did not exist when the research commenced.

The novel strategy presented in this thesis is more than an untried theory. It has been implemented, applied and is being evaluated. Simply, it is real and it works.

DEDICATION

This thesis is lovingly dedicated to my parents

Michael Julius Page and Susan Evelyn Page

TABLE OF CONTENTS

APPENDICES

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

Colours

You reap what you sow. Put your face to the ground.
Here come the marching men. Your colours wrapped around.

The Sisters of Mercy

The research detailed in this thesis was supported by two PGSF funded research projects (MAU-503, MAU-807). Financial assistance was also received with a New Zealand postgraduate scholarship.

The following people deserve special mention.

My supervisors, who inspired me and taught me a great deal

Assoc. Prof. Daniela Mehandjiska-Stavreva, Prof. Mark Apperley

The Masters and Honours students who were also involved in this research

Paul Clark, Steven Adams, Hong Yu, Duane Griffin, Sarisha Dasari, Mi Duk Choi, Jonathan Ham

Two people that selflessly proof-read the thesis

Rachel Page, Wendy Browne

My family, without whom this would not have been possible

Michael Page, Susan Page, Audrey Isaac, Rachel Page, Jonathon Page, Ruth Page

My friends (you know who you are), who all helped without knowing it

Extra thanks to: Nick Earle, Paul Clark, Duane Griffin, Lisabeth Weston, Shamus Smith, Luke Usherwood, Marion Moore, Andrew Turvey, James Fulton, Steven Adams

GLOSSARY

The content of the glossary has been derived from a range of dictionaries (Collins, 1995; Nuttals, 1902; Readers Digest, 1988; Oxford, 1993; Mirriam-Webster, 1998), the Dictionary of Object Technology (Firesmith and Eykholt, 1995) and (D'Souza and Wills, 1998; Jacobson *et al.*, 1995; Pressman 1997; Schach, 1993, 1997; Somerville, 1996).

Abstraction. Any model that includes the most important, essential, or distinguishing aspects of something while suppressing or ignoring less important, immaterial, or diversionary details. The result of removing distinctions so as to emphasise commonalties.

Arity. The cardinality of something. For example the arity of a relation specifies the number of concepts that are involved in the relation.

Attribute. Any named property used as a data abstraction to describe its enclosing object, class or extent.

Behaviour. Anything that an organism does involving action and response to stimulation. The way in which someone behaves; also: an instance of such behaviour.

Bind. To place under certain constraints. To cohere or cause to cohere. To place under obligation; oblige.

Binding. Any selection of the appropriate method for an operation on receipt of a corresponding message.

Browser. Any view that allows you to access hierarchically organised and indexable information.

CASE Tool. A) Any computer based tool for software planning, development and evolution. This includes all examples of computer-based support for the managerial, administrative, or technical aspects of any part of a software development project. B) Products that assist the software engineer in developing and maintaining software.

CASE. An acronym that stands for Computer Assisted Software Engineering.

CKB. Core Knowledge Base. A library of methodology semantic components that implements a meta-model of methodology.

Class. Any uniquely identified abstraction (i.e. a model) of a *set* of logically related instances that share the same or similar characteristics. The combination of a type interface and associated type implementation.

Classification. The act of forming into a class or classes; a distribution into groups such as classes, orders, families, etc., according to some common relations or affinities.

Cohesion. The degree, to which something models a single abstraction, localising only features and responsibilities related to that abstraction.

Component. A) Any standard, reusable, previously implemented unit that is used to enhance the programming language constructs and to develop applications. B) An independently deliverable unit of software that encapsulates its design and implementation and offers interfaces to the out-side, by which it may be composed with other components to form a larger whole.

Coupling. The degree to which one thing depends on another. Low coupling is desirable because it produces better encapsulation, maintainability, and extendibility with fewer objects needlessly affected during iteration.

Encapsulation. To enclose in or as if in a capsule; the act of enclosing in a capsule. The physical localisation of features.

Engineering. The application of scientific principles to such practical ends as the design, construction, and operation of efficient, economical structures, equipment and systems. The application of science to the design, building, and use of machines, constructions etc.

GOOKB. Generic Object Orientated Knowledge Base. A library of methodology semantic components that implements a meta-model of concepts germane to all object-orientated methodologies.

Identity. Individuality.

Information Hiding. The deliberate and enforced hiding of information (e.g. design decisions, implementation details) from clients. The limiting of scope so that some information is invisible outside of the boundary of the scope.

Inheritance. The incremental construction of a new definition in terms of existing definitions without disturbing the original definitions and their clients.

Instance. Anything created from or corresponding to a definition.

Interface. The visible outside, user view of something.

Language. Any method of communicating ideas, as by a stream of signs, symbols, gestures or the like. The special vocabulary and usage of a scientific professional or other group. The speech or expression of ideas.

MDT. An acronym that stands for Methodology Description Table. The Methodology Description Table provides an index of the methodologies supported by MOOT.

Message Send. The sending of a message to an object.

Message. Any communication sent or received by an object.

Meta. A Greek prefix signifying beyond, after, with, among and frequently expressing change. Going beyond or transcending. Used with the name of a discipline to designate a

new but related discipline designed to deal critically with the original one. Of a higher or second-order kind.

Meta-CASE Tool. *A)* A meta-CASE tool is any tool that provides automated or semi-automated support for developing CASE tools. *B)* ... are CASE tools which are used to generate other CASE tools. *C)* A CASE tool that operates on CASE tools.

Meta-language. The natural language, formal language, or logical system used to discuss or analyse another system. A form of language used to discuss a language.

Method. *A)* Mode of procedure, logical arrangement, orderly arrangement, system of classification. A means or manner of procedure, especially a regular and systematic way of accomplishing anything. The procedures and techniques characteristic of a particular discipline or field of knowledge. A special form of procedure esp. in any branch of mental activity. *B)* A way of carrying out a complete phase such as such as design or integration. *C)* The hidden implementation of an associated operation.

Methodology CASE Tool. A CASE tool that supports one or more software development methodologies and attempts to span most of the software development life-cycle.

Methodology. The science of scientific method of classification. From the Greek method and logis (science). The system of principles, practices, and procedures applied to any specific branch of knowledge. The science of method; a body of methods used in a particular branch or activity.

Model. *A)* Archetype; a description or analogy used to help visualise something that cannot be directly observed; a system of postulates, data, and inferences presented as a mathematical description of an entity or state of affairs. A preliminary pattern or representation of an item not yet constructed. A tentative framework of ideas describing something intangible and used as a testing device. *B)* A model clarifies – for a person or group of people – some aspect or perspective on a thing or event.

MOOT. Meta Object Orientated Tool. A new meta-CASE tool developed as a result of this research.

NDL. Notation Definition Language. A new language used to define the syntax of a methodology in MOOT.

Notation. A system of characters, symbols, or abbreviated expressions used in an art or science or in mathematics or logic to express technical facts or quantities.

NSM. An acronym that stands for Notation-Semantic Mapping. NSM tables are used to implement late binding of NDL and SSL methodology descriptions.

Object. Any abstraction that models a single thing.

Operation. Any service that may be requested.

Polymorphism. The ability of a single name to refer to different things having different forms.

Process. A) A system of operations in the production of something. A series of actions, changes or functions that bring about an end or result. A course of action or proceeding, esp. a series of stages in manufacture or some other operation. B) ... the way we produce software. It starts with concept exploration and ends when the product is finally retired. C) ... the set of activities and associated results which produce a software product.

Relation. Connection by consanguinity or affinity; kinship; relationship; as, the relation of parents and children; an abstraction belonging to, or characteristic of, two entities or parts together.

Semantic. Of, or relating to, meaning in language.

Software Development Life-cycle (SDLC). A process by which software engineers build computer applications.

Software Engineering. A) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is the application of engineering to software. B) ... is concerned with the theories, methods and tools that are needed to develop software for computers. C) A discipline whose aim is the production of quality software that satisfies the user's needs, and is delivered on time and within budget.

Software Project. A software project consists of a set of models (built using a particular methodology) which collectively define the software being constructed.

SSL. Semantic Specification Language. A new object-orientated language used to define the semantics of a methodology in MOOT.

SSL-BC. SSL Byte Code. A platform independent, binary, representation of SSL, which is generated by the SSL compiler.

SSLC. SSL compiler.

SSL-VM. SSL Virtual Machine. A new virtual machine which supports efficient processing of SSL.

State. Any status, situation, condition, mode, or life-cycle phase of an object or class during which certain rules of overall behaviour (e.g. response to messages) apply.

Syntax. The way in which linguistic elements (as words) are put together to form constituents (as phrases or clauses).

Tool. A thing used in an occupation or pursuit. Any instrument of use or service.

Type. A lower taxonomic category selected as a standard of reference for a higher category. The declaration of the interface of any set of instances (e.g. objects) that conform to this common protocol.

PUBLICATIONS

The results of this research have been presented in the following refereed publications.

Phillips, C.H.E., Adams, S., **Page, D.** and Mehandjiska, D. (1998) The Design of the Client User Interface for a Meta Object-Oriented CASE tool, Proceedings of TOOLS Pacific'98, Monash Printing Services, Victoria, pp145-157

Page, D., Mehandjiska, D. and Phillips, C.H.E. (1998) Methodology Independent OO CASE: Supporting Methodology Engineering, Proceedings of Software Engineering: Education and Practise (SE:E&P'98), IEEE Computer Society Press, Dunedin, New Zealand, pp373-380

Phillips, C.H.E., Adams, S., **Page, D.** and Mehandjiska, D. (1998) Design of the User Interface for a Methodology Independent OO CASE Tool, Proceedings of OZCHI'98, IEEE Computer Society Press, Los Alamitos, California, pp106-114

Phillips, C.H.E., Mehandjiska, D. and **Page, D.** (1998) The Usability Component of a New Framework for the Evaluation of Object-Oriented CASE tools, Proceedings of Software Engineering: Education and Practise (SE:E&P'98), IEEE Computer Society Press, Dunedin, New Zealand, pp131-141

Page, D., Griffin, D., Usherwood, L. and Mehandjiska, D. (1997) Implementation of a Semantic Specification Language Interpreter for a Methodology Independent OO CASE Tool, Proceedings of IASTED International Cenference on Software Engineering (SE'97), ACTA Press, San Francisco, USA, pp239-242

Mehandjiska, D., **Page, D.**, Griffin, D. and Usherwood, L. (1997) Methodology Knowledge Representation and Interpretation for a Methodology Independent OO CASE Tool, Proceedings of IASTED International Cenference on Software Engineering (SE'97), ACTA Press, San Francisco, USA, pp243-247

Mehandjiska, D., **Page, D.** and Choi, M. D. (1996) Meta-Modelling and Methodology Support in Object-Oriented CASE Tools, Proceedings of 3rd International Conference on Object-Oriented Information Systems (OOIS'96), Eds. Patel, D., Sun, Y. and Patel, S., Springer-Verlag, London, pp370-386

Mehandjiska, D., **Page, D.** and Dasari, S. (1996) Generic Knowledge Base for a Methodology Independent Object-Oriented CASE Tool, Proceedings of the IASTED International Conference on Artificial Intelligence, Expert Systems and Neural Networks, Ed. Hamza, M., IASTED/Acta Press, Honolulu, Hawaii, pp23-26

Mehandjiska, D., Apperley, M. D., Phillips, C.H.E., Dasari, S. and **Page, D.** (1996) Advancing information technologies through CASE, Proceedings of the 19th Australasian Computer Science Conference (ACSC'96), Ed. Ramamohanarao, K., Melbourne, Australia, pp213-222.

Dasari, S., Mehandjiska, D. and **Page, D.** (1995) Construction of a Generic Knowledge Base for a Methodology Independent CASE Tool, Addendum to the Proceedings of The Second NZ International Two-Stream Conference on Artificial Neural Networks and Expert Systems (ANNES'95), Dunedin, pp466-473

Mehandjiska, D., Apperley, M.D., Phillips, C., **Page, D.** and Clark, P. (1995) A Methodology independent object oriented CASE tool, New Zealand Journal of Computing, Vol. 6, pp95-105

Mehandjiska, D., **Page, D.** and Ham, J. (1995) Template generator for methodology independent object oriented CASE tool, Proceedings of 2nd International Conference on Object-Oriented Information Systems (OOIS'95), Eds. Murphy, J. and Stone, B., Springer-Verlag, Dublin, Ireland, pp431-440

Page, D., Clark, P. and Mehandjiska, D. (1994) An Abstract Definition of Graphical Notations for Object Orientated Information Systems, Proceedings of 1st International Conference on Object-Oriented Information Systems (OOIS'94), Eds. Patel, D., Sun, Y. and Patel, S., Springer-Verlag, London, pp266-276

Mehandjiska, D., **Page, D.** and Clark, P. (1994) An Intelligent Object Oriented CASE Tool, Proceedings of 1st International Conference on Object-Oriented Information Systems (OOIS'94), Eds. Patel, D., Sun, Y. and Patel, S., Springer-Verlag, London, pp168-172

Section I

Literature Review

Chapter 1

Introduction

He that will not apply new remedies must expect new evils: for time is the greatest innovator, and if time of course alter things to the worse, and wisdom and counsel shall not alter them to the better, what shall be the end?

Francis Bacon

1.1 Introduction

Edward Yourdon once said, "CASE technology will help revolutionise the software industry". Unfortunately he was overly optimistic.

There can be no doubting the potential benefits of automation during the software development process, yet the adoption of CASE technology can only be described as lethargic. If the philosophy behind CASE technology is not at fault then what is the cause of its languid rate of adoption? The fault can only be ascribed to its execution.

> "Vendors have been selling products for years that are supposed to promote reusability, promote better design, and speed time to market. The problem is that few commercial products actually live up to even significant portions of their claims."
>
> *From the FreeCASE web site (FreeCASE 1998)*

Evaluation of CASE tools has revealed a number of shortcomings in many of the existing tools in use today (Brough, 1992; Brown, 1997; Crozier *et al.*, 1989; Gibson, 1988; Isazadeh and Lamb, 1997; Lang, 1991; Marttiin *et al.*, 1993; Mehandjiska *et al.*, 1994, 1995b, 1996a, 1997; Misra, 1990; Mosely, 1992; Nilson, 1990; Ovum, 1996; Papahristos and Gray, 1991; Phillips *et al.*, 1998a; Rossi *et al.*, 1992; Sorenson, 1988; Sumner, 1992; Vessey *et al.*, 1992).

This thesis details research which aims to develop novel methods and techniques to address the limitations of current CASE and meta-CASE technology with respect to methodology representation and customisation. The research is part of a three year project, funded by the Foundation of Research, Science and Technology (Adams, 1998; Clark, 1994; Choi, 1996; Dasari *et al.*, 1995; Gray, 1995; Griffin, 1997; Ham *et al.*, 1994; Mehandjiska *et al.*, 1994, 1995a, b, 1996a, c, 1997; Page *et al.*, 1994, 1997, 1998; Phillips *et al.*, 1998a-c; Yu, 1999). A new meta-CASE tool, which implements a new methodology representation strategy, has been developed during this research. Its prototype is presented in this thesis.

The remainder of this chapter defines the scope of the research detailed in this thesis. Initially some essential fundamental terms are introduced. A brief history of object-orientated software engineering methodologies is presented followed by a review of the current status of CASE technology. The term Methodology CASE tool is defined and the scope of the research is presented. The limitations of Methodology CASE tools are discussed from several perspectives and the objectives of the research are subsequently defined. Finally an overview of the research method and an outline of the remainder of the thesis is presented.

1.2 Fundamental Terms

> Don't sir, accustom yourself to use big words for little matters ... The practice of using words of disproportionate magnitude is, no doubt, too frequent.
>
> *Samuel Johnson*

The following terms are germane to this thesis and are pervasive throughout:

- Software Engineering Development Methodology
- Computer Aided Software Engineering (CASE)
- CASE tool
- Meta-modelling
- Meta-CASE tool

No attempt is made to adopt, or artificially create, definitions of these fundamental terms for the sole purpose of this study. Rather a pragmatic overview of these fundamental terms is presented by adopting a holistic approach. A range of definitions are introduced from English dictionaries (Collins, 1995; Nuttals, 1902; Oxford, 1990; Readers Digest,

1988; Mirriam-Webster, 1998) and used to facilitate the rationalisation of these terms. The reader is directed to the glossary, which includes accepted definitions from the literature (D'Souza and Wills, 1998; Firesmith and Eykholt, 1995; Jacobson *et al.*, 1995; Pressman, 1997; Schach, 1993, 1997; Somerville, 1996).

1.2.1 Software Engineering Development Methodology

> **Engineering. [rd]**[1] The application of scientific principles to such practical ends as the design, construction, and operation of efficient, economical structures, equipment and systems. **[oxf]** The application of science to the design, building, and use of machines, constructions etc.

Software engineering is the application of scientific principles to the design and construction of software systems. A software engineer applies these scientific principles by modelling within the context of a given problem domain. Boehm (1976) proposed a definition for software engineering: "the practical application of scientific knowledge in the design and construction of computer programs and the associated documentation required to develop, operate and maintain them."

> **Model. [web]** Archetype; a description or analogy used to help visualise something that cannot be directly observed; a system of postulates, data, and inferences presented as a mathematical description of an entity or state of affairs. **[rd]** A preliminary pattern or representation of an item not yet constructed. A tentative framework of ideas describing something intangible and used as a testing device.

A model is an abstraction of a problem domain that is built by concentrating on features a software engineer deems salient. Modelling is the process of deriving a model. Marttiin *et al.* (1993) state "A *model* is a simplified representation of a system." A model is built by applying well-tested scientific principles and is expressed in a language that encapsulates those principles.

> **Language. [rd]** Any method of communicating ideas, as by a stream of signs, symbols, gestures or the like. The special vocabulary and usage of a scientific professional or other group. **[nt]** The speech or expression of ideas.

---

[1] [rd] Readers Digest Dictionary. Readers Digest, 1993; [oxf] Consise Oxford Dictionary. Oxford, 1990; [web] Webster's Revised Unabridged Dictionary (Mirriam-Webster, 1998); [nt] Nuttals Standard Dictionary. Nuttals, 1902).

A language that is used to express models is called a modelling language. The syntax of a modelling language determines the 'phrases' that may be constructed with the language. The semantics of a modelling language determines how valid 'phrases' are interpreted and understood. The procedure followed to derive a syntactically correct model, which communicates the desired information, is the method.

> **Method. [nt]** Mode of procedure, logical arrangement, orderly arrangement, system of classification. **[rd]** A means or manner of procedure; especially, a regular and systematic way of accomplishing anything. The procedures and techniques characteristic of a particular discipline or field of knowledge. **[oxf]** A special form of procedure especially in any branch of mental activity.

Marttiin *et al.* (1993) state "A method is a set of steps and rules that define how a model is derived." Often the term language subsumes the term method, in software engineering[2]. Even in this situation the method exists and is implicitly 'do not break the rules of the language'. Figure 1-1 illustrates the relations between method, modelling language, model and system. A modelling language is used (by following an associated method) to define a model, which is an abstraction of a system.

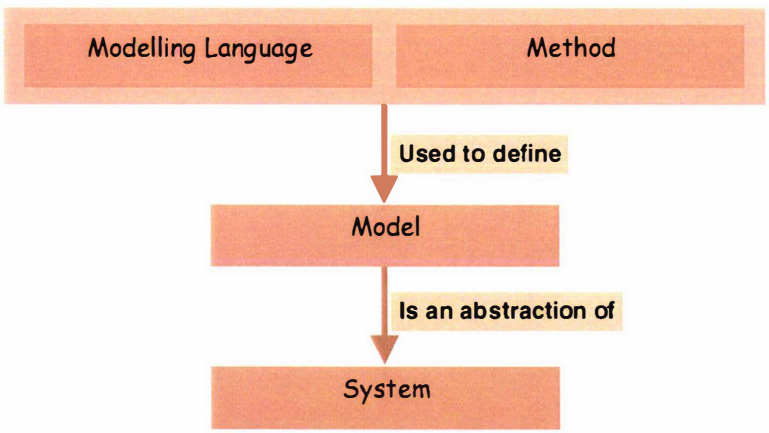


Figure 1-1 - Modelling

The meaning that can be conveyed by a model is subject to the facilities provided by the modelling language that is used to derive it. Smolander *et al.* (1991) state "A method embodies a set of concepts that determines what is perceived, a set of linguistic conventions and rules which govern how the perception is represented and

[2] The term *method* is also often used interchangeably with *methodology*. Sigfried (1996) notes "... there is no well established practice for the use of these two concepts."

communicated." A software engineer investigates the many dimensions of a problem domain by applying a range of different methods (and hence languages) to build a collection of models. The set of methods and modelling languages used to describe the dimensions of a software system is called a methodology[3].

> **Methodology. [nt]** The science of scientific method of classification. From the Greek method and logis (science). **[rd]** The system of principles, practices, and procedures applied to any specific branch of knowledge. **[web]** The science of method; a body of methods used in a particular branch or activity.

Marttiin *et al.* (1993) note "a methodology is an organised collection of methods." A software engineering methodology is a collection of methods that can be applied to build models of a software system such that the system is completely defined and can be built. Smolander *et al.* (1991) state "a methodology can be defined as an organised collection of methods and a set of rules which state whom, in what order, and in what way the methods are used." The procedure that is followed, to describe the dimensions of a software system, is called the process. Younessi and Henderson-Sellers (1998) note "a methodology is not just a set of notations and modelling rules ... a methodology must have a process dimension, thus implying a methodology includes or encompasses a process."[4]

> **Process. [rd]** A system of operations in the production of something. A series of actions, changes or functions that bring about an end or result. **[oxf]** A course of action or proceeding, esp. a series of stages in manufacture or some other operation.

A software engineering methodology promotes a set of software engineering principles that are deemed to be efficacious to the construction of software systems. Different methodologies may promote different sets of software engineering principles. It is natural in the development of any science that scientific principles change, evolve and are superseded. Naturally this is also true of software engineering methodologies.

---

[3] The concept of 'Methodology' is also discussed in more detail in chapter 6.

[4] Whether a methodology encapsulates a process, or is associated with one is an issue that is open to debate. Younessi and Henderson-Sellers (1998) offer some interesting arguments in this area.

1.2.2 Meta-Modelling

Meta is a prefix that is derived from the Greek language.

> **Meta. [nt]** A Greek prefix signifying beyond, after, with, among and frequently expressing change. **[rd]** Going beyond or transcending. **[web]** Used with the name of a discipline to designate a new but related discipline designed to deal critically with the original one. **[oxf]** Of a higher or second-order kind.

A meta-model is something 'beyond or transcending' a model. That which is beyond a model is the modelling language used to define it. Odell (1995) states "Basically a meta-model is a model that is used to talk about various kinds of models we wish to build." Tolvanen and Lyytinen (1993) note that "meta-modelling can be defined as a modelling process, which takes place one level of abstraction and logic higher than the standard modelling process." Figure 1-2 illustrates how applying the meta prefix indicates a shift in context and changes the focus of attention to something at a higher level of abstraction.

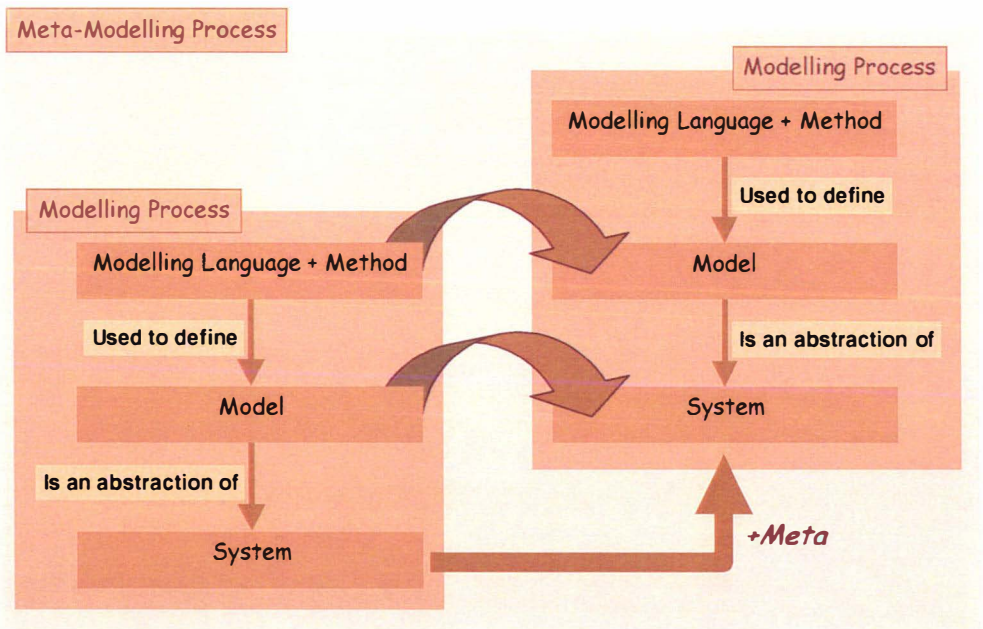


Figure 1-2 - Meta-modelling

Meta-modelling is concerned with *modelling* a *modelling process*. Tolvanen and Lyytinen (1993) note "the meta-model captures information about the concepts, representation forms, and use of a method." The model derived by the modelling process on the right-hand side of Figure 1-2 describes the application of the modelling language used in the

modelling process on the left-hand side. A modelling language is, therefore, a meta-model. Meta-models are built using a meta-modelling language.

> **Meta-language. [web]** The natural language, formal language, or logical system used to discuss or analyse another system. **[oxf]** A form of language used to discuss a language.

The meta-modelling process shown in Figure 1-2 can conceptually be performed infinitely. Tolvanen and Lyytinen (1993) note "Meta-modelling also uses its own tools which, in turn, can be described on one level higher in meta-metamodels (and so *ad infinitum*)." It is important to realise that the process of building a meta-model is modelling and that a meta-modelling language is a modelling language. The application of a number of meta prefixes indicates the relation of languages and derived models, to a point of reference.

1.2.3 Computer Aided Software Engineering (CASE)

The acronym CASE also represents terms such as Computer Assisted Software Engineering and Computer Automated Software Engineering.

> **Aid. [oxf]** A person or thing that helps; promote or encourage **[web]** Help; succour; assistance; relief.
>
> **Assist. [oxf]** Help; an act of helping. **[web]** To lend aid; to help
>
> **Automation. [oxf]** The use of automatic equipment to save mental and manual labour. The automatic control of the manufacture of a product through its successive stages. **[web]** The technique of making an apparatus, a process, or a system operate automatically.

In the context of Computer Aided Software Engineering it is clear that the computer is used to help, promote and encourage the practice of software engineering. CASE is a very general, all embracing, term.

1.2.4 CASE Tool

> **Tool. [oxf]** A thing used in an occupation or pursuit. **[web]** Any instrument of use or service.

A CASE tool is *any* computer based system that may be used during the software development process. A more detailed discussion and classification of CASE tools is given in section 1.4.

1.2.5 Meta-CASE and Meta-CASE Tool

There are many possible interpretations of the term Meta-CASE. Webster's (Miriam-Webster, 1998) dictionary states that meta can be "used with the name of a discipline to designate a new but related discipline designed to deal critically with the original one." Meta-CASE, therefore, can be interpreted as the discipline of critically dealing with computer aided software engineering. The meaning preferred in this study, however, is a higher or second-order kind of computer aided software engineering. A meta-CASE tool is therefore a higher or second-order kind of CASE tool. In the context of this thesis a general definition of meta-CASE tool is 'a computer based system used to assist the development of CASE tools.' The term meta-CASE tool encompasses all tools that are designed for the sole purpose of developing CASE tools. A more detailed discussion of meta-CASE tools is given in chapter 2.

1.3 Object-Orientated Software Development Methodologies

Three generations of Object-orientated methodologies have been identified. A multiplicity of methodologies was developed in the late 80s and early 90s. Some of the most prominent first generation object-orientated methodologies are given in Table 1-1.

Methodology	**Year**	**Methodologist**
Object Oriented Analysis	1988	Shlaer and Mellor, 1988
Object Oriented Analysis and Design	1991	Shlaer and Mellor, 1991
Responsibility Driven Design (RDD)	1990	Wirfs-Brock *et al.*, 1990
Object Oriented Analysis, Design	1991	Coad and Yourdon, 1990, 1991a, b
Object Oriented Design (OOD)	1991	Booch, 1991
Object Modelling Technique (OMT)	1991	Rumbaugh *et al.*, 1991
Object Oriented Analysis and Design	1993	Martin and Odell, 1993
Object Oriented Software Engineering (OOSE)	1993	Jacobson *et al.*, 1993

Table 1-1 - First generation object-orientated methodologies

These first generation methodologies generally covered the 'design' phase of software development and were atypically developed independently from each other. They extended ideas from object-orientated programming and also earlier non object-orientated methodologies (such as information engineering and structured analysis and design). Many methodologies that were introduced near the end of the first generation also began to consider analysis.

The first generation methodologies were applied and evaluated. The limitations that were identified prompted the emergence of second generation methodologies. Many first generation methodologies were extended to span more of the software development life-cycle (e.g. Booch OOD (Booch, 1991) → Booch OOA&D (Booch, 1994)). New methodologies were developed which simply 'borrowed the best from the rest' (Muller, 1997). For example Ian Graham's SOMA (Graham, 1994) extended Coad and Yourdon by incorporating business rules. The Fusion method (Coleman *et al.*, 1993) extended ●MT by incorporating responsibility driven design (RDD) and in-house techniques specific to Hewlett-Packard. The Booch method (Booch, 1994) also incorporated ideas from OMT and RDD. Over fifty different first and second generation object-orientated methodologies existed by 1995[5] (Muller, 1997). Some of the most well known second generation methodologies are given in Table 1-2.

Methodology	**Year**	**Methodologist**
●bject Oriented Analysis and Design (OOA&D)	1994	Booch, 1994
Semantic Object Oriented Modelling Approach (SOMA)	1994	Graham, 1994
Methodology for Object Oriented Software Engineering Systems (MOSES)	1994	Henderson-Sellers and Edwards, 1994
Advanced Object Modelling	1995	Martin and Odell, 1995
Fusion	1993	Coleman *et al.*, 1993
●bject Modelling Technique (OMT) (v2)	1994	Rumbaugh, 1995a, b
Business Object Notation (B●N)	1994	Walden and Nerson, 1995

Table 1-2 - Second generation object-orientated methodologies

[5] This period of time has been referred to as the 'methodology wars' (Henderson-Sellers, 1996).

Many comparisons of object-orientated methodologies have been published (Arnold *et al.*, 1991; Brinkkemper *et al.*, 1998; Cribbs *et al.*, 1992; de Champeaux and Faure, 1992; Fichman and Kemerer, 1992; Fung *et al.*, 1997; Hong *et al.*, 1993; Hutt, 1994; Loy, 1990; Monarchi and Puhr, 1992; Object Agency, 1998; Sharble and Cohen, 1993; Taylor, 1998; van den Goor *et al.*, 1992; Yourdon and Argila, 1996). These studies focused on the differences between methodologies rather than on identifying common aspects (Henderson-Sellers, 1996).

The results of these studies indicated that whilst many of the methodologies propounded different sets of terms and notations, there was a common awareness of the goals and process of object-orientated modelling. Work began on identifying and quantifying the common aspects of object-orientated methodologies in 1995 (Booch and Rumbaugh, 1995; Rational, 1997a, b; Henderson-Sellers and Bulthuis 1996a, b, 1997; Henderson-Sellers and Firesmith, 1997a). It was at this time that the Object Management Group (OMG) "re-established an OOAD working group/task force to ... standardise ... OO methodologies" (Henderson-Sellers, 1996).

The appearance of two third generation software development approaches was one of the results of these developments:

- Unified Modelling Language[6] (UML) (Booch and Rumbaugh, 1995; Booch *et al.*, 1999; Douglas, 1998; Fowler and Scott, 1997; Jacobson *et al.*, 1996, 1999; Muller, 1997; Rational, 1997a, b, 1998; Rumbaugh *et al.*, 1999; OMG, 1997c-j; Quatrani, 1997; UML-RTF, 1998; Warmer and Kleppe, 1999).

- OPEN[7] (COTAR, 1998; Firesmith *et al.*, 1997; Firesmith and Henderson-Sellers, 1998a, b; Graham and Henderson-Sellers, 1997; Graham *et al.*, 1997; Henderson-Sellers, 1996, 1997, 1998; Henderson-Sellers and Bulthuis, 1996a, b, 1997; Henderson-Sellers and Graham, 1996; Henderson-Sellers and Firesmith, 1997a, b; Henderson-Sellers *et al.*, 1996, 1997a-d; OPEN, 1996, 1998).

[6] UML does not propound a particular process so many consider that it is not a methodology, but a collection of interrelated modelling languages. UML is discussed briefly in chapter 2.

[7] OPEN is discussed briefly in chapter 2.

The appearance of patterns, frameworks and component engineering in the last five years (Ayoma, 1998; Bergner *et al.*, 1998; Booch, 1996; Brown, 1997; Brown and Jaeger, 1998; D'Souza and Wills, 1998; Firesmith, 1993; Fowler, 1997; Gamma *et al.*, 1995; Goldberg and Rubin, 1995; Meyer, 1995, 1997; Pree, 1994; Schmidt and Assmann, 1998; Seacord *et al.*, 1998; Short, 1997; Sigfried, 1996; Taylor, 1998; Webster, 1995; Weiderman *et al.*, 1997; Wills and D'Souza, 1997) is significant and signals a new phase in the development of software engineering. New methodologies have been developed to address the emerging technology of Components Based Development (CBD) (Ayoma, 1998; Bergner *et al.*, 1998; Brown and Jaeger, 1998; Schmidt and Assmann, 1998; Short, 1997). An example is Catalysis (D'Souza and Wills, 1998; ICON, 1998):

> "**Catalysis.**
> A next-generation UML-based method for the systematic development of object and component based systems, using precise modelling techniques and frameworks, to reflect and support an adaptive enterprise."
>
> *From the ICON computing website (ICON, 1998)*

1.4 CASE Technology

A ***C***omputer ***A***ided ***S***oftware ***E***ngineering (CASE) tool is any computer based tool for software planning, development and evolution. This definition includes all examples of computer-based support for the managerial, administrative, or technical aspects of any part of a software development project.

The principle objective of CASE technology is to reinforce and support an engineering approach to software development and evolution by providing computer based assistance, which translates to low-defect solutions and enhanced productivity (Brough, 1992; Haine, 1992; Nilsson, 1990; Quantrani, 1997; Marttiin, 1994; Senn, 1990; Sumner, 1992; Verhoef *et al.*, 1991). Sumner (1992) summarised the benefits of CASE as the introduction of engineering-like discipline into the system development process and the creation of a common repository of design documentation. Nilsson (1990) notes that "the main benefit of CASE is that people who perform requirements gathering and specification need not use 'pen and paper' techniques for drawing diagrams and that the diagrams can be integrated with a data dictionary". Senn (1990) states that "CASE tools

are important because they speed development, automate tedious tasks, and enforce standards and procedures."

CASE tools have been categorised in many different ways. For example they have been classified in terms of functionality, their relation to the software development life-cycle and the level of inter-tool integration that they support (Beynon-Davies, 1989; Nilsson, 1990; Pressman, 1997; Wallnau, 1992; Wallnau and Feiler, 1991; Whitten *et al.*, 1994; Zarella, 1990).

This thesis is concerned with CASE tools that implement software development methodologies and support activities across the entire software development life-cycle. Whitten *et al.* (1994) adopts the term 'cross life-cycle CASE' to classify tools that support activities across the entire software development life-cycle. The name adopted in the thesis for a CASE tool of this type is a Methodology CASE tool.

> **Methodology CASE Tool**. A CASE tool that supports one or more software development methodologies and attempts to span most of the software development life-cycle.

Use of the term CASE tool in the remainder of the thesis specifically relates to Methodology CASE tools and not to CASE tools in general (such as compilers and debuggers). This thesis is concerned with tools that support object-orientated software engineering, so its primary focus is on object-orientated Methodology CASE tools.

Use of the term meta-CASE tool in the remainder of the thesis specifically relates to meta Methodology CASE tools. A meta Methodology CASE tool is a meta-CASE tool that is used to develop Methodology CASE tools.

A Brief History of CASE

Table 1-3 describes the history of CASE tools. It is taken from the CASE Tool home page at the University of Sunderland (Ferguson, 1998).

Early CASE tools addressed mostly form and representation issues of software development methodologies and focused on capturing a set of diagrams for the software engineer (Brough, 1992; Verhoef *et al.*, 1991). As these tools evolved they supported completeness, correctness and consistency checking (Sorenson, 1988). These tools mainly

supported structured software engineering techniques (Haine, 1992; Hoffman and Strooper, 1995), focusing on specific phases of the software development life-cycle (Haine, 1992; Nilsson, 1990).

Early 80s	Computer aided documentation
	Computer aided diagramming
	Analysis and design tools
Mid 80s	Automatic design analysis and checking
	Automated system information repository
Late 80s	Automatic code generation from design specification
	Linking design automation
Early 90s	Intelligent methodology driver
	Habitable user interface reusability as a development methodology

Table 1-3 - History of CASE tools (Ferguson, 1998)

Large-scale software development demanded enhanced support across the entire software development process from methodologies (Younessi and Henderson-Sellers, 1998) and CASE tool developers (Brown, 1997; Haine, 1992; Nilsson, 1990). Assistance was required for the requirements definition, design and implementation phases of the software development life-cycle, testing, documentation and version control (Sorenson, 1988; Sorenson *et al.*, 1988). The term front-end (or upper-CASE) tool was introduced to classify tools that supported phases of the software development life-cycle up to, and including, design (Nilsson, 1990; Beynon-Davies, 1989). The term back-end (or lower-CASE) tool was introduced to classify tools that supported phases beyond design (Nilsson, 1990; Beynon-Davies, 1989).

At this time object-orientated methodologies were attracting more attention from the software development industry (Behforooz and Hudson, 1996). They were being revised to encompass analysis, domain and business modelling in addition to design (Younessi and Henderson-Sellers, 1998). CASE tools had to address these developments by spanning more of the software development life-cycle (Brown, 1997; Mehandjiska *et al.*, 1994, 1995, 1996b; Page *et al.*, 1998).

1.5 Methodology CASE Tools

The evolution of Methodology CASE tools was investigated during the inception of the research. Existing CASE tools were evaluated focusing on methodology support, life-cycle support, functionality and usability. A new evaluation framework, which encapsulated these evaluation criteria, was derived[8] as an extension of the Software Engineering Institute's framework for evaluating CASE tools (Mosely, 1992). The central organising principle of the new framework is the classification hierarchy of CASE tool categories shown in Figure 1-3.

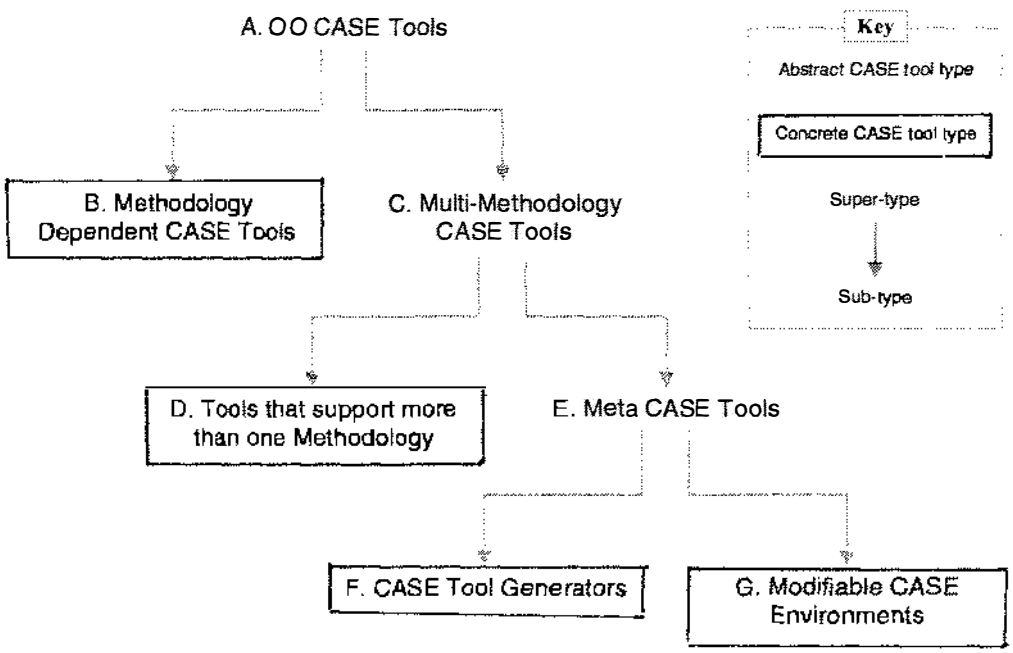


Figure 1-3 : Classification hierarchy of CASE tool categories

Evaluation criteria are associated with nodes in the classification hierarchy at an appropriate level of abstraction and are further classified in terms of usability, methodology support, life-cycle support and functionality. Each evaluation criterion is therefore classified in two ways: a) based on the CASE tool category it is relevant to and b) based on the type of CASE tool property it evaluates. The structure of the classification hierarchy permits evaluation criteria to be specialised and refined in a systematic way. A classification based evaluation framework provides the necessary flexibility needed to cope with changing CASE and software engineering technology and

[8] A brief synopsis of the evaluation framework is presented in appendix I.

can be easily extended in the future. Some of the results obtained by applying the new evaluation framework are presented in (Choi, 1996; Gray, 1995; Phillips *et al.*, 1998a).

Object-orientated methodology CASE tools (node *A* of the hierarchy in Figure 1-3) can be categorised as either 'Methodology dependent CASE tools' (node *B* of the hierarchy in Figure 1-3) or 'Multi-methodology CASE tools' (node *C* of the hierarchy in Figure 1-3).

1.5.1 Methodology Dependent CASE Tools

These tools support a single object-orientated software engineering methodology. They are often older tools that typically support a single phase of the software life-cycle[9]. Examples of tools in this category include ObjecTool (supports Coad and Yourdon), ShowCASE (supports Booch'91), Objectory (supports Jacobson), OEW (supports Martin and Odell) (Innovative Software, 1998) and early versions of Rational Rose (supports Booch'91 and Booch'94) (Rational, 1998).

The most fundamental limitation of tools in this category is that a company is constrained to adopt a single software engineering methodology. This prevents software development companies from choosing the most suitable methodology for the problem at hand. In addition, companies may choose to mix and match concepts from more than one methodology. It is not possible for a CASE tool vendor to predict these kinds of decisions and demands. The limited flexibility of methodology dependent CASE tools is therefore a barrier to the adoption of CASE tools by industry.

1.5.2 Multi-Methodology CASE Tools

Tools of this category (node *C* of the hierarchy in Figure 1-3) attempt to address the deficiencies of methodology dependent CASE tools by supporting several different methodologies. Some tools simply attempt to implement more than one methodology whilst others provide some type of customisation facilities to support multiple methodologies. The techniques used differentiate multi-methodology CASE tools into two sub categories 'Tools that support more than one methodology' (node D of the hierarchy in Figure 1-3) and 'Meta-CASE tools' (node *E* of the hierarchy in Figure 1-3).

---

[9] Methodology dependent CASE tools may also support more than a single phase of the software development life cycle, if the methodology they implement provides such support.

1.5.3 Tools that Support More than One Methodology

Some Object-Oriented CASE tools such as COOL:Teamwork (Sterling, 1998), COOL:Jex (Sterling, 1998), CASET, MacA&D and WinA&D (ExcelSoftware, 1998), ObjecTime and System Architect (Popkin, 1998) claim to support more than one methodology. For example Teamwork supports structured analysis and design as well as several object-orientated analysis and object-orientated design methodologies.

In general, tools in this category do not support the methodologies completely and support is restricted to subsets of each methodology. Usually only visualisation of the users' project using a range of different graphical notations is provided. The user cannot customise these tools; only the tool proprietor may extend or modify them.

1.5.4 Meta-CASE Tools

A meta-CASE tool provides automated or semi-automated support for developing CASE tools (Alderson, 1991; ASD, 1995a, b, 1998; Coxhead and Fisher, 1994a, b; Coxhead *et al.*, 1994; Demetrovics *et al.*, 1982; Findeisen, 1993, 1994a-d; Gadwal *et al.*, 1994a, b; JrCASE, 1998; Lincoln, 1994, 1998; Lo, 1995; Lyytinen *et al.*, 1994; Maokai and Scott, 1998; MetaCASE consulting, 1996a, b, 1998; Marttiin 1994; Marttiin *et al.*, 1993; mip GmbH, 1998a-d; Scott, 1998; Smolander *et al.*, 1991; Sorenson *et al.*, 1988; Tolvanen and Lyytinen, 1993; Zhuang, 1994; Zhuang *et al.*, 1995). Meta-CASE tools are based on an underlying meta-model, which is used to describe the languages, concepts and relations propounded by a methodology. The majority of meta-CASE tools use a data model as their meta-model (e.g. variants of the Entity Relationship Diagram).

Meta-CASE tools can be further classified as 'CASE tools generators' (node *F* of the hierarchy in Figure 1-3) and 'Modifiable CASE Environments' (node *G* of the hierarchy in Figure 1-3). More detailed analysis of meta-CASE tools is presented in section 2.3.

1.5.5 CASE Tool Generators

A CASE tool generator is a meta-CASE tool that supports the construction of standalone CASE tools. Meta-CASE tools of this type often provide a set of libraries and a programmer API to support the construction of standalone tools. Some CASE tool generators allow individual tools to share a common repository. The tools developed by a

CASE tool generator often exhibit a similar look and feel in their user interface and have a similar structure in their repositories[10]. The main advantage of a CASE tool generator is the greatly reduced development time for an individual tool. CASE tools generated by these systems often suffer from poor user interfaces as CASE tool generators typically focus on specifying modelling languages at the expense of due consideration to Human Computer Interaction (HCI) principles.

Paradigm+ (Platinum, 1998), Software through Pictures (STP, 1998), Toolbuilder (Lincoln, 1998) and MetaView (Gadwal *et al.*, 1994a, b; Findeisen, 1993, 1994a-d; Lo, 1995; Sorenson *et al.*, 1988; Zhuang, 1994; Zhuang *et al.*, 1995) are CASE tool Generators.

1.5.6 Modifiable CASE Environments

Tools in this category attempt to combine the benefits of a meta-CASE tool and a multi-methodology CASE tool. These tools allow their methodologies to be modified and may be extended to support new methodologies. Meta-CASE tools of this type usually provide a set of methodology description languages that are used to define methodologies.

A modifiable CASE environment has two types of user. Methodology engineers use a modifiable CASE environment to manipulate descriptions of software engineering *methodologies*. Software engineers use a modifiable CASE environment to manipulate descriptions of software engineering *projects*.

The main problem with modifiable CASE environments is poor support for the concept of methodology. Often the user of such an environment is presented with an extremely large collection of methodologies (Lyytinen *et al.*, 1994; MetaCASE Consulting, 1996a, b, 1998; Smolander *et al.*, 1991; Tolvanen and Lyytinen, 1993). In addition the relation between different methodologies and methods is often not clear.

Modifiable CASE environments do present significant possibilities for the support of re-use amongst software development projects as these tools have detailed information regarding the methodologies they implement. However these tools do not consider re-use

[10] This is often a consequence of choosing some type of data model as the meta-model.

explicitly. Any claim for the support of re-use is only ever matched by simple import/export facilities or by accidental re-use[11] (ASD, 1998; Lyytinen *et al.*, 1994; MarkV, 1998; MetaCASE Consulting, 1996a, b, 1998; Smolander *et al.*, 1991; Tolvanen and Lyytinen, 1993). Generally these tools only support accidental re-use of methodology descriptions.

Modifiable CASE environments typically focus on specifying modelling languages at the expense of due consideration to Human Computer Interaction (HCI) principles and suffer from poor user interfaces.

Graphical Designer (ASD, 1998), ObjectMaker (MarkV, 1998), MetaEdit+ (Lyytinen *et al.*, 1994; MerridanMarketing, 1998; MetaCASE Consulting, 1996a, b, 1998; Smolander *et al.*, 1991; Tolvanen and Lyytinen, 1993) are examples of Modifiable CASE environments.

1.6 Limitations of Methodology CASE Tools

CASE tools have promised high gains in terms of enhanced productivity, lower defect solutions and faster time to market. Yet many organisations have not adopted CASE technology (Beynon-Davies, 1989; Bay, 1998; Huff *et al.*, 1992; Malmborg, 1992; Oakes *et al.*, 1992; Sørensen, 1993; Vessey *et al.*, 1992; Wallnau, 1992; Wallnau and Feiler, 1991; Zarella, 1990; Zarella *et al.*, 1991). Many of the reasons for the poor adoption of CASE tools are epitomised by the FreeCASE project (FreeCASE, 1998). FreeCASE is a methodology dependent CASE tool (Figure 1-3 - Classification hierarchy of CASE tool categories) that is being developed by volunteers from the free software community.

> "FreeCASE will be a first of a kind product. It will be a team orientated tool for object-oriented analysis and design. It will ... support UML 1.1 ... It will forward-generate and reverse engineer source code in multiple languages. It will support a networked repository, allowing for development over the Internet. It will also provide versioning and code management capabilities. Additionally, it will support a client running on multiple platforms."
>
> *From the FreeCASE website (FreeCASE, 1998)*

[11] "The developers discover that certain parts of a product may be re-used in another product, after it has been completed" Aranow, 1996 .

Whilst the project is in its infancy and, as yet, does not appear to provide anything novel, it is interesting because of its motives:

> "Vendors have been selling products for years that are supposed to promote reusability, promote better design, and speed time to market. The problem is that few commercial products actually live up to even significant portions of their claims. Worse, the price of entry is anywhere from $800 to $5000 PER USER! I find this to be unacceptable."
>
> *From the FreeCASE website (FreeCASE, 1998)*

The very existence of such a project is indicative of the potential of CASE technology and also of the failure to deliver on that potential.

A large body of work related to the adoption of CASE technology exists (Beynon-Davies, 1989; Day, 1998; Huff *et al.*, 1992; Malmborg, 1992; Mathiassen and Sørensen, 1995; Oakes *et al.*, 1992; Schottland, 1996; Sørensen, 1993; Vessey *et al.*, 1992; Wallnau, 1992; Wallnau and Feiler, 1991; Zarella, 1990; Zarella *et al.*, 1991). Oakes *et al.* (1992) report that the major problems associated with the adoption of CASE tools are[12]:

- The wide variation in quality and value within a single type of tool.
- The relatively short time that many types of CASE tool have been in use in organisations.
- The wide difference in the adoption practices of various organisations.
- The general lack of detailed metric data for previous and current projects.
- The wide range of project domains.
- The confounding impact of changes to methods and processes that are often associated with the adoption of CASE tools.
- The potential bias of organisations reporting CASE gains or losses.

[12] This list is from a technical report published by the Software Engineering Institute (SEI 1998) dealing with the adoption of CASE tools. Their definition of CASE is the "range of interrelated tools that support the software engineering process."

Artsy (1995) notes in a position paper for the OOPSLA workshop 'Meta-modelling in OO':

> "Very few tools implementing an OOA&D method do have an explicit meta-model, and even fewer publish it. Without such a model, the tool's user cannot know precisely how accurately does the tool view or implement certain concepts ... Furthermore, even when the tool has an explicit meta-model, but the tool is not model-driven, it is inflexible to change whenever the OOM evolves (and since OOMs do evolve relatively often, tools are becoming obsolete too soon)."

The limitations of CASE tools can be considered from two perspectives. The first is from the point of view of companies adopting CASE technology (organisational perspective). The second is from the point of view of CASE tools themselves (CASE Tool perspective)[13].

1.6.1 Limitations from the Organisational Perspective

These limitations are related to the effect the adoption of CASE technology can have within an organisation.

- **High cost of adoption**

  The adoption of CASE technology can be a major investment for a company. The price of CASE tools can vary greatly depending on the functionality and features provided by the CASE tools. In addition the training costs associated with adopting a CASE tool can be prohibitive.

- **High learning curve**

  The learning curve associated with CASE tools can be high. CASE tools are not simple products to master, especially given their emphasis on collaborative work and that their affect is across the software development life-cycle.

[13] Additional limitations of meta CASE tools are presented in chapter 2.

- **Long payback period**

  The payback period for adopting CASE tools can be long (i.e., years of time). This is because the advantages of adopting CASE technology may not become clear until the first products are completed with the assistance of CASE tools. Payback is in terms of faster time to market, better quality products and lower maintenance costs.

- **Lack of customisation**

  Many companies utilise in-house methodologies or processes. Their means of work may also be a modification or extension of a popular, accepted methodology. Such practices are not supported well by current CASE technology, as most CASE tools are rigid and do not allow customisation.

- **Lack of standards**

  A plethora of CASE tools exist, which vary significantly in terms of quality, usability and functionality. This is related to the large number of object-orientated methodologies, the lack of industry standards and immaturity of the CASE industry.

- **Culture shock**

  CASE tools propound a collaborative approach to software engineering and emphasise the importance of the pre-implementation phases of the software development process. This can be a culture shock for many organisations.

  In addition, some people feel that CASE tools will 'de-skill' and 'constrain' them rather than enhance their productivity.

- **Lack of flexibility**

  Companies have significant investments in legacy systems and existing software projects documented using different methodologies. Existing CASE tools are inflexible and do not allow companies to preserve their investments in existing technology, systems and methodologies.

In addition, many CASE tools do not integrate well into the existing operation of an organisation. This means that changes are required to accommodate a new tool. People in general are resistant to change.

1.6.2 Limitations from the CASE Tool Perspective

These limitations are related to the characteristics and functionality of current CASE tools.

- **Methodology specific**

  The majority of CASE tools are methodology specific which makes it difficult to justify the significant investment required, in terms of time, training and resources, to adopt CASE technology. Use of such tools also places constraints on an organisation to use the methodologies they support in order to justify their initial investment.

- **Limited support for the software development life-cycle**

  The support of the entire software development life-cycle is limited. Whilst many tools provide some support for reverse engineering and re-engineering, few support requirements gathering for example. This is also because of the limited support of the entire spectrum of software engineering activities by existing methodologies.

- **Poor support for all aspects of a methodology**

  The support of a methodology that is provided by a CASE tool is often limited to a collection of diagram editors, which correspond to the various modelling languages that the methodology provides. The concepts of process and method are often ignored.

- **Poor usability**

  The usability of CASE tools, from a HCI perspective, is often poor. CASE tools are generally rigid and force users to conform to a set means of working. In addition the possibilities that are available with current Human-Computer-Interaction (HCI) techniques are generally not considered.

Most CASE tools are simple implementations of existing 'pen and paper' techniques with the addition of correctness and consistency checking. They do not support techniques such as logical distortion and novel interaction styles.

- **Poor support for re-use**

  CASE tools do not provide support for re-use between user projects. This will become a more important limitation in the future as the trend toward adoption of object-orientated technology continues. Whilst it is true that object-orientated technology does not guarantee re-use it is accepted that one of the principle objectives of object-orientated technology is to enable re-use. This should therefore also be a key objective of a CASE tool that supports object-orientated methodologies.

- **Poor support for migration of software engineering projects**

  CASE tools do not allow software artefacts to be re-used, if they are built with different methodologies. Consequently a company cannot effectively make use of previous modelling results.

  Some CASE tools, however, do attempt to implement data interchange formats such as the Case Data Interchange Format (CDIF) (EIA CDIF, 1994a-h, 1996; Flatscher, 1996). CDIF is discussed in chapter 2.

- **Lack of intelligence**

  The level of assistance provided by CASE tools to software engineers is limited to the capture and consistency checking of a set of diagrams. No consideration is given to things such as intelligent feedback on work as it is completed, auto-correction and quality analysis.

1.7 Objectives of the Research

This research is part of the PGSF funded research project titled "Advancing information technologies through CASE", which aims to develop novel methods and techniques for addressing the limitations of current CASE and meta-CASE technology.

The objectives of the research detailed in this thesis are:

- Develop a novel meta-CASE tool methodology representation strategy that:
  - Uses an object-orientated meta-model.
  - Allows methodology descriptions to be re-used.
  - Minimises the coupling between methodology syntax and semantic descriptions such that methodology syntax and semantic descriptions can be re-used independently.
  - Permits software engineering projects to be re-used, even if they are built with different methodologies.
- Design and implement a prototype meta-CASE tool that realises the new methodology representation strategy via the development of:
  - Languages that support the description of syntax and semantics of a methodology.
  - The efficient execution strategy of syntax and semantic descriptions.

The new CASE tool that has been developed during the research to satisfy these objectives is called MOOT (***M***eta ***O***bject ***O***rientated ***T***ool).

1.8 Method

The following steps summarise the approach adopted to satisfy the objectives described in section 1.7:

A. Compare, contrast and evaluate existing CASE tools and meta-CASE tools to identify limitations of current CASE technology. A detailed comparison of meta-CASE tools is presented in chapter 2.

B. Define the rationale and goals of the MOOT project based on the identified limitations of current CASE technology. Investigate a possible meta-systems approach based on an object-orientated meta-model.

C. Devise a representation scheme for methodology descriptions in MOOT.

D. Develop a meta-model of the concept of methodology with the representation scheme defined in C.

E. Derive a means of processing the methodology descriptions defined in step C.

F. Design the architecture of a meta-CASE tool based on the work in steps B - E.

G. Realise a prototype of the system defined in step F that is suitable for assessing the efficacy of the representation scheme for methodology descriptions (defined in step C).

H. Validate the meta-systems approach by modelling object-orientated methodologies and implementing support for design patterns.

1.9 Outline of the Thesis

The overall outline of the thesis is illustrated in Figure 1-4.

The thesis is structured into nine chapters, which are grouped into three sections:

- Literature review (chapter one and two)
- Research description (chapter three to chapter seven)
- Results, discussion and review (chapter eight and nine)

Chapter two presents a review of meta-modelling and meta-CASE tools. Chapters three to seven cover the research undertaken. The overall architecture and design philosophy of a new meta-CASE tool is discussed in chapter three. Chapters four, five, six and seven discuss the languages and mechanisms used to represent and process methodology descriptions. Chapter six also outlines the facilities for re-use of methodology descriptions and user projects that these techniques provide. Chapter eight presents results of using the prototype meta-CASE tool. Chapter nine is a review chapter in which the contribution of this research is examined and further work is identified.

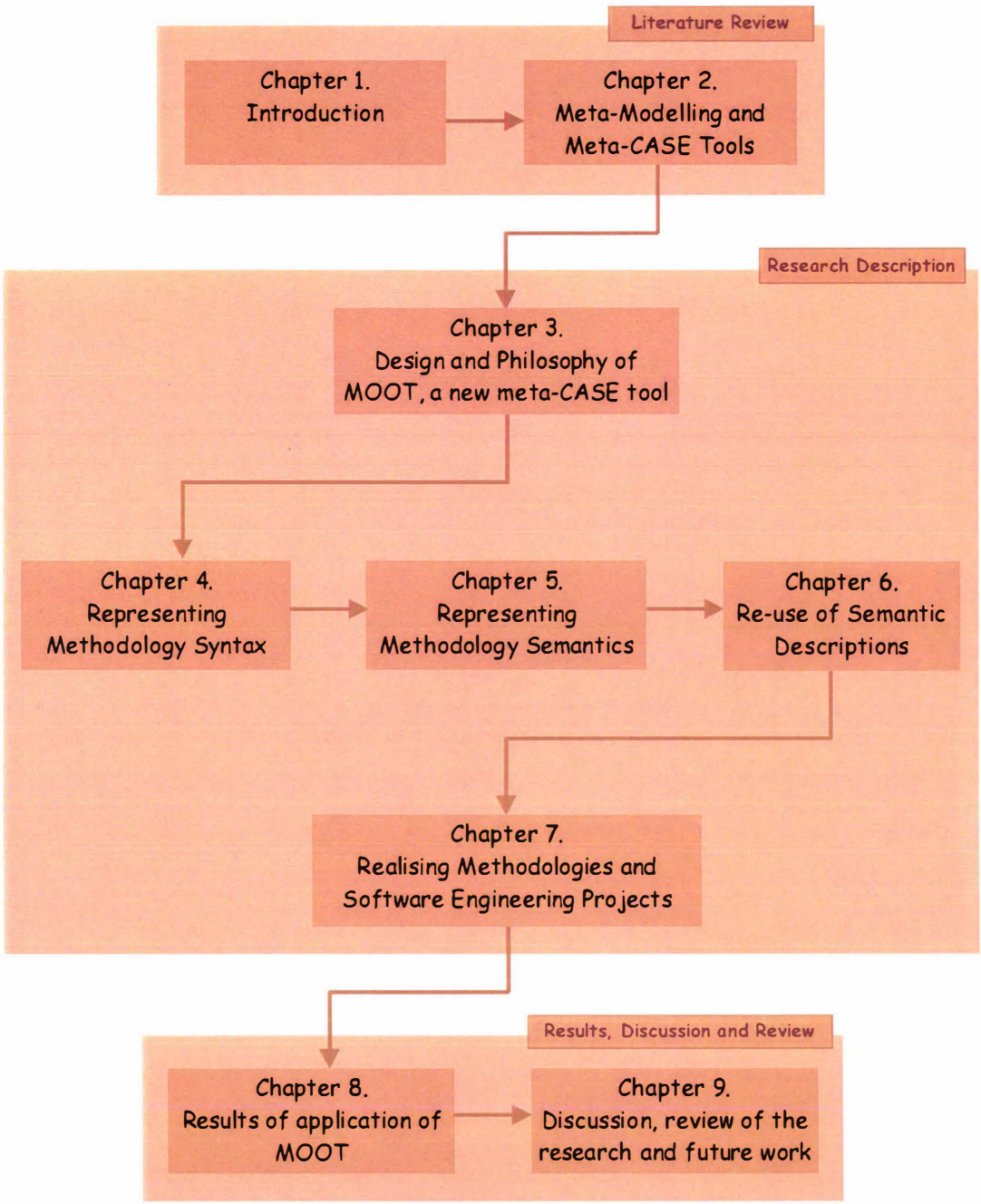


Figure 1-4 - Thesis outline

Chapter 2

Meta-Modelling and Meta-CASE Tools

There are two ways of constructing a software design. One way is to make it so simple there are no obvious deficiencies. And the other is to make it so complicated that there are no obvious deficiencies.

C A R Hoare

2.1 Introduction

This chapter presents a review of meta-modelling and meta-CASE tools. Initially the meta-modelling process is discussed in relation to a four layer meta-modelling architecture (EIA CDIF, 1994a; Ernst, 1996; OMG, 1997a, i). A review of some important applications of meta-modelling in software engineering is presented. The relation of meta-modelling to meta-CASE tools is discussed and a representative sample of meta-CASE tools is critically reviewed. A summary of the limitations of existing meta-CASE tools is derived based on the review.

2.2 Meta-Modelling

Problems cannot be solved at the same level of awareness that created them.

Albert Einstein

Meta-modelling is an activity that is germane to many problem domains (Metamodel.com, 1998). Examples include modelling business rules (Blanchard, 1995), the development of databases (Demphlous and Lebastard, 1995; Sahraoui *et al.*, 1995) and the translation of architecture description languages (Barbacci and Weinstock, 1998).

The generally accepted framework for meta-modeling is based on a four layer architecture (OMG, 1997i). The layers, from the most abstract (left) to the least abstract (right), are:

meta-metamodel → meta-model → model → user objects

Table 2-1 presents a description of each layer. It is taken from the UML semantics guide (v1.1) (OMG, 1997i). Similar tables may be found in (EIA CDIF, 1994a; Ernst, 1996; OMG, 1997a).

Layer	Description	Example
Meta-metamodel	The infrastructure for a meta-modelling architecture. Defines the language for specifying meta-models.	*MetaClass, MetaAttribute, MetaOperation*
Meta-model	An instance of a meta-metamodel. Defines the language for specifying a model.	*Class, Attribute, Operation, Component*
Model	An instance of a meta-model. Defines a language to describe an information domain.	*StockShare, askPrice, sellLimitOrder, StockOrderQuoteServer*
User objects (user data)	An instance of a model. Defines a specific information domain.	*<Acme_Software_Share_98789>, 654.6, sell_limit_order, <Stock_Quote_Svr_32123>*

Table 2-1 - Four layer meta-modelling architecture

Figure 2-1 shows that two meta-modelling steps (described in section 1.2.2) are required to implement the four layer architecture of Table 2-1.

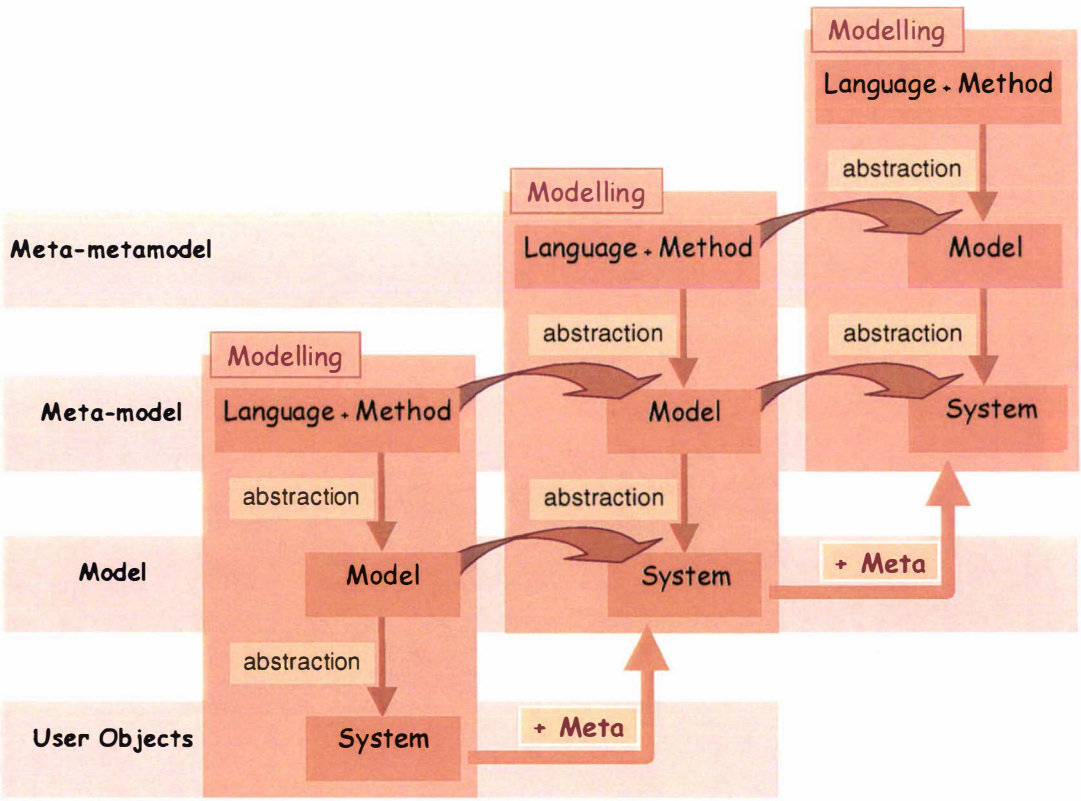


Figure 2-1 - Four layer meta-modelling process

Figure 2-1 also demonstrates that three languages are required to implement the four layer architecture. Practically, it is possible that the same language may be used more than once. Tolvanen and Lyytinen (1993) note "meta-modelling also uses its own methods and tools which, in turn, can be described on one level higher in meta-metamodels (and so *ad infinitum*)."

The nature of the particular problem a meta-modelling approach is applied to dictates the choice of languages and methods used and the resulting meta-metamodel, meta-models and models. Examples[14] of the application of meta-modelling in the software engineering field include:

- The OMG Meta Object Facility (OMG, 1997a, b)
- The Unified Modelling Language (Booch and Rumbaugh, 1995; Rational, 1997a, b; OMG, 1997c-j; UML-RTF, 1998)
- The Common Object Meta-Modelling Architecture (Henderson-Sellers and Bulthuis, 1996a, b, 1997; Henderson-Sellers and Firesmith, 1997a)
- OPEN Modelling Language (Firesmith *et al.*, 1997, 1998b; Henderson-Sellers and Graham, 1996; Henderson-Sellers *et al.*, 1997a, b)
- The OOram meta model developed by Taskon A/S, Reich Technologies and Humans and Technology (Reenskaug *et al.*, 1996; Taskon A/S, 1997)
- The Case Date Interchange Format family of standards (EIA CDIF, 1994a-h, 1996; Flatscher, 1996)
- The ISO/CDIF meta-model (ISO, 1998b)
- The MetaData Interchange Format Standard (MDC, 1997, 1998)

2.2.1 The OMG Meta Object Facility

The Object Management Group's (OMG) ***M***eta ***O***bject ***F***acility (MOF) defines an object-orientated meta-metamodel (the MOF model), which is used to define meta-

[14] MetaCASE tools are also an example of the application of meta-modelling. They are discussed in section 2.3.

models in various domains (OMG, 1997a, b). Example domains include object-orientated analysis and design, the application development life-cycle, data warehouse management and business object management.

> "The main purpose of the OMG MOF is to provide a set of CORBA interfaces that can be used to define and manipulate a set of interoperable meta-models. The MOF is a key building block in the construction of CORBA based distributed development environments."
>
> *From the Meta Object Facility (MOF) Specification (OMG, 1997a)*

The MOF Model is defined in terms of itself[15]. The MOF Model is also the meta-metamodel of the UML submission for the Object Analysis and Design Facility (OA&DF) to the OMG (OMG, 1997i). The OMG MOF specification clearly states the importance of the MOF development:

> "This attempt at OMG to integrate the Meta Object Facility and the Object Analysis and Design Facility (OA&DF) is expected to be a critical step in developing meta-data standards that will begin addressing the application development life-cycle. This standard is even more important now considering the profound impact that Distributed Objects and the Internet are having on development methodologies that favour object-oriented and component-based development environments. The use of repositories and meta-data management in these environments is a well recognised industry trend."
>
> *From the Meta Object Facility (MOF) Specification (OMG, 1997a)*

The OMG MOF is also being aligned with the meta-metamodels submitted for the OMG OA&D facility and the EIA CDIF standard.

[15] The language used to define the MOF model is the language defined by the MOF model. The UML semantics guide calls this approach meta-circular (OMG, 1997i).

2.2.2 Unified Modelling Language

UML is a visual object-orientated modelling language targeted toward describing object-orientated systems. Its initial development started with the unification of three existing object-orientated software engineering methodologies: OMT (Rumbaugh, 1991, 1995a, b), Booch (Booch, 1991, 1994) and OOSE (Jacobson *et al.*, 1993).

> "The Unified Modelling Language (UML) is a general purpose modelling language that is designed to specify, visualise, construct and document the artefacts of a software system. The UML is simple and powerful. The language is based on a small number of core concepts that most object oriented developers can easily learn and apply. The core concepts can be combined across a wide range of domains."
>
> *From the UML Semantics, v1.1 (OMG, 1997i)*

The UML consists of two parts:

- *UML Semantics.* A meta-model that defines the abstract syntax and semantics of UML object modelling concepts.
- *UML Notation.* A graphical notation for the visual representation of the UML semantics.

The UML meta-model is expressed in a subset of UML. The implicit meta-metamodel[16] is the same as the OMG MOF (Meta Object Facility) model (OMG, 1997a, b).

The UML started as the Unified Method in 1995. The draft specification of the Unified Method (version 0.8) contains an informal meta-model that encompasses the concepts and associations used in object-orientated analysis and design (Booch and Rumbaugh, 1995). It also contains a collection of papers on specific aspects of that meta-model. The Unified Method was then renamed the Unified Modelling Language to reflect that the process was to be defined at a later stage.

[16] The UML semantics guide notes "If there is not an explicit meta-metamodel, there is an implicit meta-metamodel associated with every meta-model" (OMG, 1997i).

In January of 1997 UML version 1.0 was submitted to the OMG for the Object Analysis and Design Facility (OA&DF) (Rational, 1997a, b). The Object Management Group adopted the UML version 1.1 as a standard for the OMG OA&DF in September of 1997 (OMG, 1997c-j).

UML is currently under revision (UML-RTF, 1998) with a projected completion date, for version 1.3, of January of 1999. Revision 1.4 is expected to be complete by April of 1999.

2.2.3 COMMA

COMMA stands for **C**ommon **O**bject **M**ethodology **M**etamodel **A**rchitecture. Henderson-Sellers and Bulthuis initiated the project in 1995 (Henderson-Sellers and Bulthuis, 1996a, b, 1997; Henderson-Sellers and Firesmith, 1997a).

> "The major goal of the COMMA project is to highlight the commonalties of object-orientated methods by describing their underlying meta-models ... in order to focus on the areas of agreement."
>
> *From the COMMA project: First steps (Henderson-Sellers et al., 1996b)*

The COMMA project consisted of three phases:

- Identification of the methodologies to be modelled. Derivation of an appropriate meta-level notation and modelling syntax[17] for this purpose.
- Derivation of meta-models for a number of methodologies.
- Construction of a core meta-model.

Fourteen different object-orientated methodologies were modelled during the COMMA project. The meta-language used during the project was purpose designed[18].

> "When we began the COMMA project in January 1995, we unfortunately found existing object-orientated meta-modelling techniques to be inadequate for COMMA, the notations and semantics usually being extensions of structured notations and not possessing object-orientated features,

[17] This corresponds to a meta-metamodel, in terms of a four layer meta modelling architecture.

[18] In reality, the meta language is a simple object-orientated language, which has explicit support for roles and a notation that is an amalgam of several others

particularly inheritance; for example, OPRR and GOPRR, the notation used by Bulthuis and the use of ER by Eckert and Golder."

From An Overview of the COMMA Project (Henderson-Sellers and Bulthuis, 1996a)

The COMMA project greatly influenced the development of the OPEN methodology and the OPEN modelling language.

2.2.4 Open Modelling Language

The ***O***PEN ***M***odelling ***L***anguage (OML) is one aspect of the larger OPEN project (Firesmith *et al.*, 1997; Firesmith and Henderson-Sellers 1998a, b; Henderson-Sellers and Graham, 1996; Henderson-Sellers *et al.*, 1996, 1997a, b). OPEN has been derived from object-orientated software engineering approaches of SOMA (Graham, 1994), MOSES (Henderson-Sellers and Edwards, 1994), and Firesmith with contributions from a group of 32 researchers and methodologists, collectively known as the OPEN consortium.

"OPEN consists of a full life-cycle process-centred OO methodology with emphasis on *inter alia*, reuse, quality, organisational issues including people and project management ... It has a meta-model and notation which are collectively called the OPEN Modelling Language – OML has exactly the same scope as the UML ... "

From: Evaluating Third generation OO Software Development Approaches (Henderson-Sellers and Firesmith, 1997b)

The OPEN meta-model is characterised by and emphasises responsibilities, unidirectional associations and the inclusion of roles (based on the work on OOram (Reenskaug *et al.*, 1996; Taskon A/S, 1997)). The OPEN meta-model is based on the core COMMA meta-model.

2.2.5 OOram

The Taskon A/S / Reich Technologies / Humans and Technology OOram meta-model was created primarily for its submission to the Object Management Group in response to the request for proposals for the Object Analysis and Design Facility (Taskon A/S, 1997).

"The main contributions of this proposal are its overall architecture and its system abstraction. The architecture is object-orientated and unifies a number of powerful abstractions. The system abstraction combines the power of use cases, responsibility driven design, and role modelling; it can be thought of as an extension of the UML and OML object models."

From the OOram Metamodel, v 1.0 (Taskon A/S, 1997)

The main goal of the OOram meta-model was to contribute concepts that the authors considered were missing from mainstream object-orientated methodologies. Its primary focus is role modelling, class modelling, and system relations. The OOram meta-model is an object-orientated meta-model, which is described in terms of itself.

2.2.6 CASE Data Interchange Format

CDIF (CDIF, 1998) is a standards body sponsored by the EIA (EIA, 1998) (Electronic Industries Association) and the ISO (ISO, 1998a) (International Standards Organisation), whose mission is to enable data interchange between modelling tools.

"CDIF has been developed to define the structure and content of a transfer that may be used to exchange data between two CASE tools. The fundamental objectives of the CDIF Family of Standards are: to provide a precise, unambiguous definition of information to be transferred; to define a transfer that may be read and understood directly (i.e., without interpretation by a computer); to provide the importer with sufficient information to enable the importer to reproduce the transferred data consistent with the original sense."

From the CDIF CASE Data Interchange Format – Overview, Extract of Interim Standard (ELA CDIF, 1994a)

The EIA initiated the development of CDIF in October of 1987. The goal of this work has been to permit the results of modelling work, performed with various techniques, to be transferred between CASE Tools. CDIF defines a series of meta-models (which are called subject areas) for modelling techniques, using an Entity Relationship model (the meta-metamodel) (EIA CDIF, 1994a-h, 1996; Flatscher, 1996).

In 1991 the EIA CDIF interim standard extended the CDIF meta-metamodel to allow entity types to be interpreted as 'classes' and 'sub-classed' in refined meta-models (EIA CDIF, 1994a). The 1994 standard supported sub-classing of relationship types as well. Figure 2-2 shows the current CDIF meta-metamodel taken from the 'Extract of Interim Standard CDIF Framework for Modelling and Extensibility' (EIA CDIF, 1994b). Non directed lines indicate a sub-classing relationship among entity types, where the more abstract concept is placed physically above the less abstract concept. A directed line indicates a directed association between two entity types.

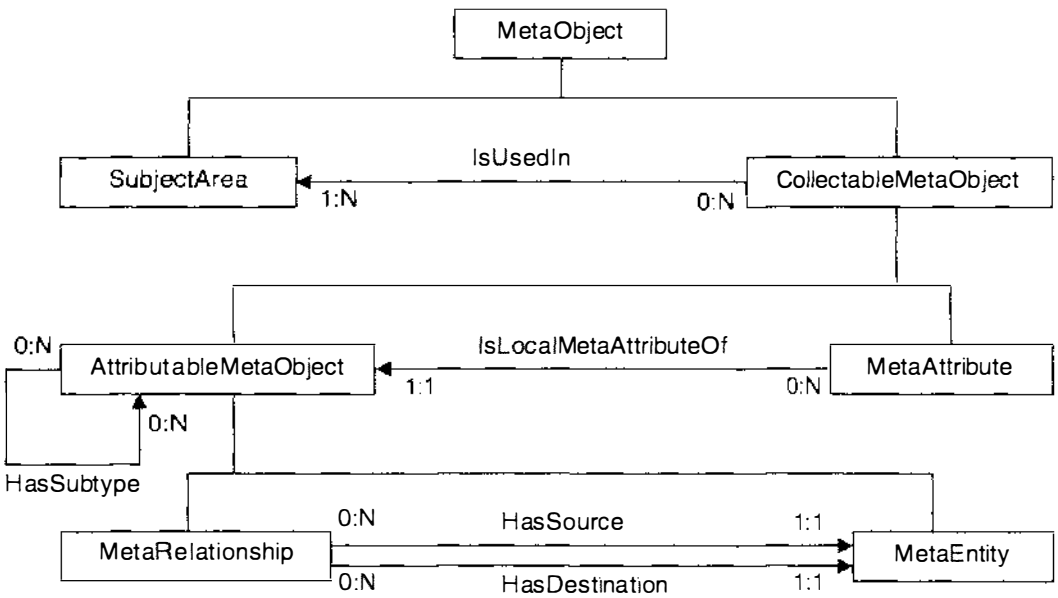


Figure 2-2 - CDIF Meta-metamodel (EIA CDIF, 1994b)

CDIF has defined subject areas for modelling techniques such as data modelling, data flow modelling, state event modelling and object-orientated analysis and design. Work is under way to integrate CDIF with the UML (Ernst, 1996).

2.2.7 ISO/CDIF Meta-Model

ISO/IEC JTC1/SC7/WG11 (ISO, 1998b) is the international body responsible for standardising information such as meta-models for software engineering activities. It is informally known as ISO/CDIF. Much of the work performed by ISO/CDIF corresponds to EIA/CDIF projects. ISO/CDIF also co-ordinates with other organisations such as the Object Management Group.

2.2.8 MetaData Interchange Facility

The MetaData Interchange Facility (MDIF) is developed by the Metadata Coalition (MDC) (MDC, 1997, 1998). The goal of the MDC is to create a vendor-independent, industry-defined and maintained standard access mechanism and standard application programming interface (API) for meta-data.

> "To enable full-scale enterprise data management, different IT tools must be able to freely and easily access, update, and share meta-data. The only viable mechanism to enable disparate tools from different vendors to exchange meta-data is a common meta-data interchange specification with guidelines to which the different vendors' tools can comply. ... The MetaData Interchange Specification initiative brings industry vendors and users together to address a variety of problems and issues regarding the exchange, sharing, and management of meta-data."
>
> *From the Metadata Interchange Specification version 1.1 (MDC, 1997)*

The MetaData Interchange Specification uses an ER meta-metamodel to describe the entities and relationships that are used to represent meta-data in the MDIF.

2.3 Meta-CASE Tools

Two types of meta-CASE tool were identified in section 1.5.4. These were CASE Tool Generator and Modifiable CASE Environment.

CASE Tool Generator

A CASE tool generator is a meta-tool, which supports the construction of standalone CASE tools. Figure 2-3 shows two common CASE tool generator configuration. A tool description is composed of a methodology specification and a tool configuration definition.

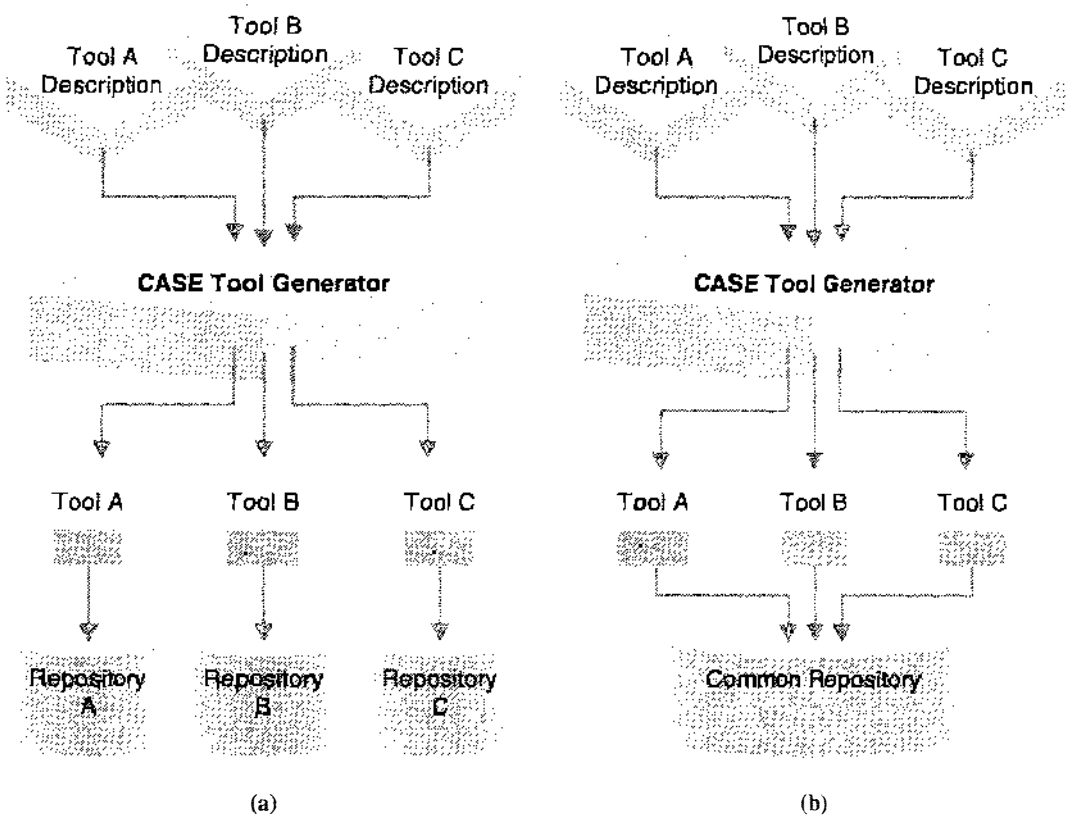


Figure 2-3 - CASE tool generators

A CASE tool generator is parameterised by a set of CASE tool descriptions. Each tool description in Figure 2-3 (a) is translated into a separate standalone CASE tool. The generated tools are completely separate and each has an individual repository for software engineering projects. Figure 2-3 (b) shows a slightly different approach, where each of the generated tools share a common repository.

In both cases, the methodology specification and the software projects do not coexist in the same repository. Moreover each generated CASE tool supports a single methodology.

Modifiable CASE Environment

A modifiable CASE environment allows methodology descriptions to be modified and may be extended to support new methodologies. Figure 2-4 shows the common configuration of a modifiable CASE environment.

The key difference between a modifiable CASE environment and a CASE tool generator is the integration of methodology descriptions and software projects into one repository.

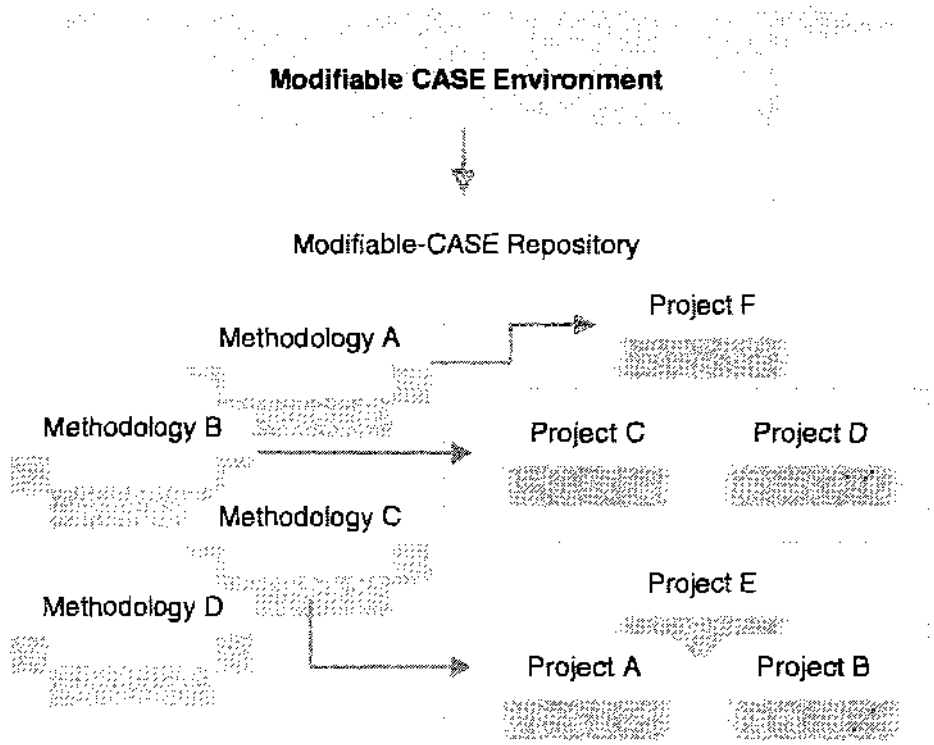


Figure 2-4 - Architecture of a modifiable CASE environment

Both types of meta-CASE tool provide a meta-model, which is used to define methodologies. This meta-model is either: explicit and supported by a set of definition languages (ASD, 1995a, b, 1998; Coxhead and Fisher *et al.*, 1994a, b; Froehlich, 1994; Coxhead *et al.*, 1994; Findeisen, 1993, 1994a-d; Froehlich, 1994; Gadwal *et al.*, 1994a, b; Lincoln, 1994, 1998; L●, 1995; Lyytinen *et al.*, 1994; Smolander *et al.*, 1991; Tolvanen and Lyytinen, 1993; MetaCASE consulting, 1996a, b, 1998; Sorenson *et al.*, 1988; Verhoef *et al.*, 1991; Zhuang, 1994; Zhuang *et al.*, 1995) or implicit and supported by one or more libraries and an application programming interface (rCASE, 1998; Maokai and Scott, 1998; mip GmbH, 1998a-d; Scott, 1998). Figure 2-5 illustrates the relation between the four layer meta-modelling architecture discussed in section 2.2 and a meta-CASE tool meta-model.

A software engineer builds descriptions of software that is to be constructed. Each description corresponds to a software engineering project in Figure 2-5. Each project consists of a set of models, which collectively define the software. The software corresponds to the 'user objects' level of the meta-modelling architecture and the software project corresponds to the 'model' level.

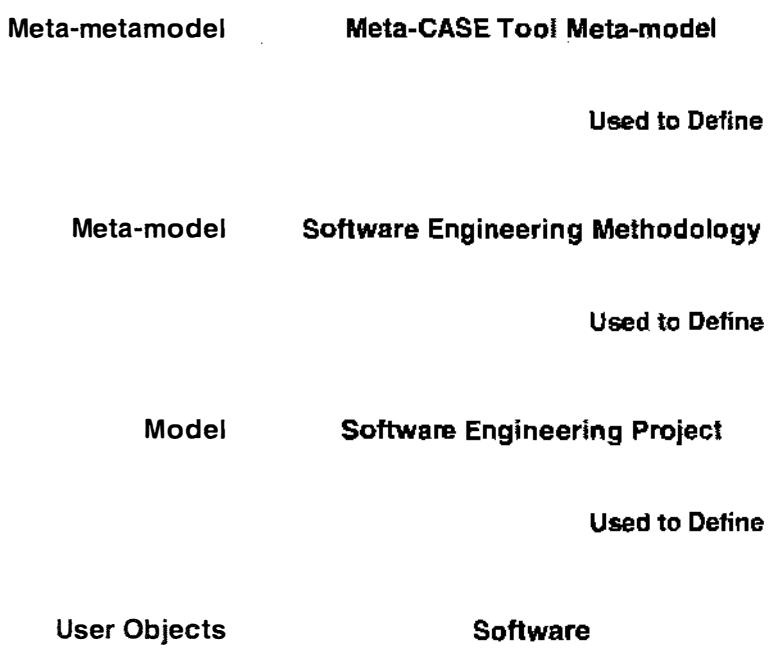


Figure 2-5 - Meta-CASE tools and the four layer meta-modelling architecture

Each model in a software engineering project is defined using a modelling language provided by a software engineering methodology. The modelling languages are meta-models of the models in the software engineering project and thus correspond to the meta-model level of the meta-modelling architecture.

Each methodology in a meta-CASE tool is defined in terms of the meta-CASE tool meta-model. This meta-model provides the language the methodology engineers use to define methodologies. The meta-CASE tool meta-model therefore corresponds to the meta-metamodel level of the meta-modelling architecture.

2.3.1 Framework for Discussion of Meta-CASE Tools

The discussion of meta-CASE tools is based on a framework, which has been designed to evaluate the properties of meta-CASE tools related to methodology representation.

The properties considered include:

1. *Underlying meta-model (representation of semantics)*

   What is the modelling language supported? How is the modelling language implemented? Are there any limitations of the meta-model?

2. *Representation of syntax*

   How tightly coupled are the semantic and syntax descriptions? What are the limitations of the mechanism for describing syntax.

3. *Support for the concept of methodology*

   Does the tool support the concept of methodology at all? Does the tool support the concepts of method and process? Does it only support the definition of modelling languages?

4. *Support for re-use*

   Does the tool embrace re-use of methodology descriptions? Is re-use of software projects supported?

5. *Problems and limitations*

Table 2-2 contains a non-exhaustive, representative sample of meta-CASE tools. Tools with a ✓ next to them will be discussed with respect to the framework. These tools have been selected because they are widely used and referenced in the literature.[19] They utilise a range of meta-models (for example EARA/GE, GOPRR and Class based) and are representative of the types of meta-CASE tool.

Research Tool	**Commercial Tool**
MetaView (MetaView, 1998) ✓	Alfabet (Alfabet, 1998) ✓
Meta-Edit and MetaEdit+ (MetaCASE consulting, 1998) ✓	ToolBuilder (Alderson, 1991; Lincoln, 1998) ✓
CASEMaker (JrCASE, 1998; Maokai and Scott, 1998; Scott, 1998)	Graphical Designer (ASD, 1998) ✓
KOGGE (Sahraoui *et al.*, 1995; Ebert *et al.*, 1996)	ObjectMaker (MarkV, 1994, 1998)
Ramatik	Paradigm Plus (Platinum, 1998)
MetaPlex	Software though Pictures (STP, 1998)
Socrates (Verhoef *et al.*, 1991)	

Table 2-2 Meta-CASE tools

[19] Many meta-CASE tools are commercial products and detailed technical information is difficult to obtain.

2.3.2 MetaView

MetaView is a CASE tool generator (Figure 1-3 - Classification hierarchy of CASE tool categories). It was developed at the University of Saskatchewan and University of Alberta, Canada (Gadwal *et al.*, 1994a, b; Findeisen, 1993, 1994a-d; Froehlich, 1994; Lo, 1995; Sorenson *et al.*, 1988; Zhuang, 1994; Zhuang *et al.*, 1995).

2.3.2.1 Underlying meta-model

MetaView introduces an extension of the entity relationship (ER) data model called EARA/GE (***E***ntity ***A***ggregate ***R***ole ***A***ttribute with ***G***raphical ***E***xtension). The novel parts of this meta-model are the support for aggregates, specialisation and the graphical extensions (Gadwal *et al.*, 1994a, b; Findeisen, 1993, 1994a-d; Lo, 1995; Sorenson *et al.*, 1988; Zhuang, 1994; Zhuang *et al.*, 1995).

An aggregate is a heterogeneous collection of entities and relationships. The entities and relationships belonging to an aggregate are called its components. EARA supports an aggregation relationship, which is a special association between an entity and an aggregate. This relationship is also called an entity explosion and is used to represent hierarchical decomposition.

Each entity, relationship and aggregate has a type. These types can be built into specialisation hierarchies where subtypes inherit the relationships and attributes of their super-types. The properties of entities, relationships and aggregates are represented by attributes.

Methodologies in MetaView are defined in a specially designed language called Environment Definition Language (EDL) (Gadwal *et al.*, 1994a). EDL provides features that correspond to the concepts supported by EARA/GE.

Constraints are defined after an EARA data model, which describes a methodology, is constructed. The constraints either guard the consistency of the specification (consistency constraints) or ensure that the software specification is complete (completeness constraints). Constraints are written in a separate language called Environment Constraints Language (ECL) (Findeisen, 1994d).

2.3.2.2 *Representation of syntax*

Methodology graphical notations are modelled with an extension to the EARA model called the Graphical Extension (GE) (Findeisen, 1993).

The graphical extension is designed to support two-dimensional, non-animated diagramming techniques. It introduces the following graphical types: icon, edge and diagram as subtypes of entity, relationship and aggregation respectively.

The graphical extension provides the following primitives:

- *Picture Pattern and Picture.* A geometric figure that may appear repeatedly in a diagram. Picture patterns are composed of points, lines, arcs and text. A Picture is a picture pattern with additional constraints such as a position.
- *Label.* Labels are used to represent attributes of entities, aggregates and relationships.
- *Diagram.* Diagrams are used to represent aggregates and correspond to individual drawing surfaces.
- *Icon.* An icon is used to represent an entity. An icon is represented as a rectangular area. Icons have a fixed size and may be annotated by pictures and labels.
- *Cluster.* A cluster represents a sequence of entities and is used to express presentation constraints (e.g. vertical or horizontal alignment). Clusters may be collapsed into a single icon.
- *Edge.* Edges are used to represent relationships. Edges may be annotated with pictures and labels.
- *Handle.* Handles are used to define the positions on an icon, where edges may be attached.

2.3.2.3 *Support for the concept of methodology*

Each methodology specification in MetaView defines a collection of diagram types. There is no support for the concept of methodology, especially in terms of process.

2.3.2.4 *Support for re-use*

MetaView does not promote re-use of methodology components, although it does support specialisation of entity types. Constraints are defined globally for each

methodology and cannot be re-used. There is no support for re-use of software engineering projects.

2.3.2.5 Problems and limitations

- An entity cannot be owned by more than one aggregate.
- An entity cannot be involved in more than one aggregation relationship. It can only be exploded to one type of aggregate.
- Correctness and completeness constraints are totally separate from the entities, aggregates and relationships. Constraints are also defined globally over all the entities, aggregates and relationships. This implies computation overhead.
- The formal nature of the meta-model implies that partially completed models cannot be built.
- The syntactic representation is totally integrated with the semantic representation. A change in the semantics implies a change in the graphical representation and vice versa. The cohesiveness of syntax and semantic descriptions is therefore reduced.
- There is no consideration of cognitive support (auto correction, feedback etc).
- An entity cannot be represented by more than one icon.
- A relationship cannot be represented by more than one edge.
- An icon cannot represent more than one entity or aggregate.
- Icons and pictures are of a fixed size.
- Diagrams are only views of aggregates.
- MetaView only supports 'pen and paper' notations. The syntax description is very simple and does not support facilities such as logical distortion.
- There is no support for process.
- There is no support for re-use of methodology descriptions or software projects.

2.3.3 Meta-Edit and MetaEdit+

MetaEdit and MetaEdit+ are modifiable CASE environments (Figure 1-3 - Classification hierarchy of CASE tool categories). These tools were developed as part of the

MetaPHOR project by the University of Jyväskylä, Technical Research Centre of Finland (VTT) and University of Oulu (Lyytinen *et al.*, 1994; MerridanMarketing, 1998; MetaCASE consulting, 1996a, b, 1998; Smolander *et al.*, 1991; Tolvanen and Lyytinen, 1993).

2.3.3.1 Underlying meta-model

The meta-model used in MetaEdit+ is called GOPRR (***G***raph, ***O***bjects, ***P***roperties, ***R***elationships, ***R***oles). It is an extension of the OPRR model used in MetaEdit (Smolander *et al.*, 1991). The OPRR meta-model is founded on "fixed mapping rules between modelling constructs and their graphical behaviours" (Smolander *et al.*, 1991).

The basic OPRR modelling constructs are:

- Objects. These are not objects in the object-orientated sense as they are passive. They are reminiscent of entity types.
- Properties, which are attributes of objects, relationships and roles.
- Relationships, which are associations between objects.
- Roles, which define the ways in which objects participate in specific relationships.

The GOPRR model adds the concept of Graph to the OPRR model. A graph denotes an aggregate that contains a set of objects, relationships, roles and other graphs. A graph also has its own properties and typically appears as a window. The graph concept has also been extended into a modelling unit called *Project.* A *Project* is used to manage the relationships between the collection of modelling languages in a particular methodology.

Objects can be arranged into specialisation hierarchies where 'sub-objects' inherit the relationships and properties of their 'super-objects'.

2.3.3.2 Representation of syntax

There is a one to one correspondence between GOPRR types (projects, graphs, objects, roles and relationships) and graphical representations (which MetaEdit+ calls symbols). Symbols are defined in terms of primitive shapes (ellipse, rectangle, rounded rectangle, line, polygon, text and bitmap). A symbol may have labels that correspond to the values of the properties of a GOPRR type.

2.3.3.3 Support for the concept of methodology

A methodology is mapped to the project concept in the GOPRR meta-model. There is no support for process.

2.3.3.4 Support for re-use

The GOPRR meta-model supports inheritance of GOPRR types. MetaEdit+ also supports a symbol library. The definition of an existing modelling language may be duplicated and modified. In practice this is only accidental re-use.

2.3.3.5 Problems and limitations

- The formal nature of the meta-model implies that partially completed models cannot be built.
- There is no support for the reuse of modelling results.
- Only accidental re-use of semantic descriptions is supported.
- An explosion of 'methodologies' and modelling languages. Each time a methodology engineer binds a semantic definition to a different syntax a new methodology is created.
- There is no support for process.
- There is no consideration of cognitive support (auto correction, feedback etc).
- Support for project (a type of graph) is an afterthought added to address lack of support of all aspects of a methodology.
- Syntax definition is a function of the semantic description because of the assumed one-to-one mapping between syntax and semantic elements.
- The syntactic representation is totally integrated with the semantic representation. A change in the semantics implies a change in the graphical representation and vice versa. The cohesiveness of syntax and semantic descriptions is therefore reduced.
- Symbols are of a fixed size and only defined in terms of primitive shapes. There are no facilities to describe symbols and connections that change size.
- Diagrams are only views of Graphs.
- MetaEdit and MetaEdit+ only support 'pen and paper' notations. The syntax description is very simple and does not support facilities such as logical distortion.

2.3.4 Alfabet

Alfabet is a commercial modifiable CASE environment (Figure 1-3 - Classification hierarchy of CASE tool categories) produced by mip GmbH & Co. Its primary focus is 'business modelling' and 'data modelling', although it does provide extensions for the support of UML (mip GmbH, 1998a-d).

> "Alfabet is a database-supported meta-modelling system with a powerful graphical user interface that can describe any kind of information model and analyse it with various methods. Alfabet offers two different user levels: The Developer level to develop models in a meta-modelling environment, and the User level to put these models into action."
>
> *From the ALFABET user manual (mip GmbH, 1998c)*

2.3.4.1 Underlying meta-model

The documentation for Alfabet does not make a specific reference to an underlying meta-model. The meta-model is implicitly related to the class-based database management system used by Alfabet.

> "Alfabet works on a class-based technology. This is an object-orientated approach suitable for modelling that has been developed by mip. This technology allows users to describe the deep structure of a model that is built from objects and their relationships (the meta-model), instead of filling predefined meta-models with data."
>
> *From the Alfabet user manual (mip GmbH, 1998c)*

The implicit Alfabet meta-model provides the following abstractions:

- *Class.* The Alfabet technology overview (mip GmbH, 1998b) equates class to abstract data-type. Classes contain properties and may be built into inheritance hierarchies.
- *Scalar type.* Examples include string and integer.

- *Multiple type.* According to the Alfabet technology overview (mip GmbH, 1998b) a multiple type is a list of classes. The instances of all of the classes are property values of the multiple type.

- *Container.* Containers are defined for all scalar and abstract types (classes).

- *Event.* Init, Put, Get and Clear events can be defined for properties.

2.3.4.2 Representation of syntax

Alfabet provides the following graphical primitives:

- *Node item.* Node items consist of various simple geometric shapes.

- *User item.* A user item is any combination of Node items.

- *Generator item.* These are predefined notation templates for common business applications (such as Gantt charts).

- *Special item.* These include lines, polygons and textboxes.

- *Link item.* Items that are used to connect generator and node items.

Graphical items may be associated with instances in the repository. The Alfabet manual (mip GmbH, 1998c) states, "in this case the graphical item represents a semantic instance". It is not clear how graphical items that are not associated with an instance in the repository are interpreted. Semantic items are represented by a small set of simple graphical primitives that may be scaled and combined. Alfabet does appear to provide notation frameworks (the generator items) for commonly used data modelling notations.

2.3.4.3 Support for the concept of methodology

Alfabet does not support the concept of methodology at all. The Alfabet Frequently Asked Questions states "The philosophy behind Alfabet puts great emphasis on the integration of important method or notation solutions" (mip GmbH, 1998a).

At best a methodology corresponds to a project in Alfabet. Each project is configured with a of set diagram types.

2.3.4.4 Support for re-use

Alfabet does not claim any support for re-use of projects or modelling languages, other than by accidental re-use.

2.3.4.5 Problems and limitations

- This tool is database driven, not methodology driven.
- There is no support for methodology or process. Alfabet only supports the definition of modelling languages and notations.
- Alfabet claims to be object-orientated. This is incorrect, as it is class based.
- Primary used to support data modelling languages, although support for UML is also claimed.
- Implicit meta-model.
- No support for re-use of syntax and semantic definitions.
- Alfabet only supports 'pen and paper' notations. The syntax description is very simple and does not support facilities such as logical distortion.
- There is no consideration of cognitive support (auto correction, feedback etc).

2.3.5 ToolBuilder

ToolBuilder is a commercial CASE Tool generator (Figure 1-3 - Classification hierarchy of CASE tool categories) created by Lincoln software (Alderson, 1991; Coxhead and Fisher *et al.*, 1994a, b; Coxhead *et al.*, 1994; Lincoln, 1994,1998).

ToolBuilder consists of:

- A method specification capture component called METHS.
- A run-time methods component called DEASEL.

DEASEL is a generic CASE tool offering fully integrated graphical and textual editing of data stored in the 'Lincoln repository'.

2.3.5.1 Underlying meta-model

Toolbuilder uses an extended entity relationship (EER) model as its meta-model. Entity types are built into specialisation hierarchies where subtypes inherit the relationships and

attributes of their super-types. This data model supports derived relationships and derived attributes. Triggers can be associated with attributes and relationships.

2.3.5.2 Representation of syntax

Toolbuilder supports two-dimensional, non-animated diagramming techniques. It considers each diagram consists of nodes (symbols) and links (connections).

The support for syntax has two components:

- The frame model. This corresponds to the visual presentation of the underlying data model and consists of a collection of diagrams.
- The notation for each diagram frame.

A set of basic shapes is provided from which more complex shapes may be defined. These shapes may be combined with other basic shapes, to create symbols and connections. Symbols and connections may have text fields associated with them. Toolbuilder supports the definition of the formatting (e.g. alignment) of text fields.

2.3.5.3 Support for the concept of methodology

Toolbuilder generates standalone CASE tools. Each CASE tool supports a single methodology. There is no support for software process or method.

2.3.5.4 Support for re-use

Toolbuilder only supports the generation of bespoke CASE tools, which all have separate repositories. There is no support for re-use of modelling results or of semantic descriptions.

2.3.5.5 Problems and limitations

- There is no support for process
- There is no support for re-use of modelling results.
- Only accidental re-use of semantic descriptions is supported.
- Toolbuilder only supports 'pen and paper' notations. The syntax description is very simple and does not support facilities such as logical distortion.

- A fixed mapping between syntax and semantics is implied.
- The semantic and syntax descriptions are tightly coupled. For example an entity's attributes may have an associated show trigger, which defines how the attributes are to be presented.
- Symbols are of a fixed size and only defined in terms of primitive shapes. There are no facilities to describe symbols and connections that change size.
- A total of five languages are used to define a CASE tool (LL, DDL, GDL, FDL and EASEL) (Alderson, 1991).
- There is no consideration of cognitive support (auto correction, feedback etc).

2.3.6 Graphical Designer Pro

Graphical Designer is a modifiable CASE environment (Figure 1-3 - Classification hierarchy of CASE tool categories) (ASD, 1995a, b, 1998).

Graphical Designer provides a single, function/procedure based, scripting language that is used to define the syntax and semantics of a methodology. The Graphical Designer language is used to define all aspects of a CASE tool, including report and code generation.

2.3.6.1 Underlying meta-model

A CASE tool is described in Graphical Designer as a set of functions that operate on symbols, attributes, roles and relationships. The meta-model used is the Object Property Role Relationship (OPRR) model, where Graphical Designer uses the terms Symbol, Attribute, Role and Relationship respectively.

2.3.6.2 Representation of syntax

Graphical Designer has a single description language that is used to describe the syntax and semantics of methodologies as well as the behaviour of Graphical Designer itself. There is a one to one mapping between syntax and semantic concepts.

2.3.6.3 Support for the concept of methodology

Graphical Designer is parameterised by a set of files per methodology. These files define the set of modelling languages available. Graphical Designer does not consider process or method at all.

2.3.6.4 Support for re-use

The only form of re-use supported by Graphical Designer is accidental re-use. There is no relation between the methodology descriptions in Graphical Designer. New methodologies must be effectively designed from scratch.

2.3.6.5 Problems and limitations

- Graphical Designer provides a function based language that has a high learning curve associated with it. The underlying meta-model is completely obscured because the scope of the language covers the syntax and semantics of methodologies as well as the behaviour of the tool itself.
- The semantic and syntax descriptions are totally integrated. A change in the semantics implies a change in the graphical representation and vice versa. The cohesiveness of syntax and semantic descriptions is therefore reduced.
- There is no support for process.
- There is no support for re-use of modelling results.
- Only accidental re-use of semantic descriptions is supported.
- There is no consideration of cognitive support (auto correction, feedback etc).
- Graphical Designer only supports 'pen and paper' notations. The syntax description is very simple and does not support facilities such as logical distortion.

2.4 Limitations of Current Meta-CASE Technology

The discussion of meta-CASE tools in section 2.3 has highlighted a range of limitations. These include:

- **Poor support for the concept of methodology**

  All meta-CASE tools interpret methodology as a collection of modelling languages. They make no attempt to support the concept of process or method.

- **Constraints are separate from the structural definition of methodology concepts**

  Most meta-CASE tools partition the semantic definition of a methodology into two parts: a) a data model and b) a global set of constraints that are applied to the elements of the data model. Often this can imply a significant overhead in terms of applying constraints, as they are evaluated for *all* instances.

- **Formal approach can be restrictive**

  The formal approach adopted by meta-CASE tools does not allow a user project to be in an incomplete/inconsistent state. This is a barrier to a creative, exploratory approach to development that software engineers naturally apply.

- **Syntax description is primitive**

  Virtually all meta-CASE tools derive syntax elements from a pre-defined set of graphical primitives. Notation elements are built by scaling and combining these primitive elements. Typically these notation elements do not resize dynamically as they are used. Most common symbols, with more than one compartment, are impossible to describe with such a strategy.

- **Fixed mapping between syntax and semantics**

  All meta-CASE tools assume that there is a fixed one-to-one mapping between syntax and semantic elements. It implies that the structure of the syntax elements is always the same as the structure of semantic elements. Whilst it is reasonable to expect a high structural homology between syntax and semantic descriptions, it is unnecessarily restrictive to assume the structure of each description is identical.

- **Coupling of syntax and semantic descriptions constrains each other**

  The coupling between the syntax description and semantic description in current meta-CASE tools is high. The high coupling can compromise the cohesiveness of the semantic and syntax descriptions. Moreover the structure of the syntax and semantic descriptions can affect each other. High coupling of syntax and semantic descriptions is also a barrier to their subsequent re-use.

- **No support for re-use of methodology description**

  The majority of meta-CASE tools do not give any consideration for the re-use of methodology descriptions. If re-use is supported it is only in the form of accidental re-use. Even those tools that support a form of specialisation do not place any emphasis on re-using methodology descriptions.

- **No relation between defined methodologies**

  This is a direct consequence of no support for the re-use of methodology descriptions and can result in a large collection of unrelated methodologies. These methodologies may in fact have a lot in common. In some tools this may also mean that one or more methodologies are in fact semantically the same, but simply have different syntax. This is a barrier to re-use because the CASE tool environment becomes a large collection of unrelated software engineering projects.

- **No support for re-use of software engineering projects**

  Meta-CASE tools do not consider the re-use of software projects developed with the methodologies that they support.

  For a CASE tool generator this is simply because each tool that is generated is considered in isolation. These tools may provide some form of import/export facilities, which is not sufficient to support anything other than accidental re-use.

  Modifiable CASE environments, however, have the *potential* to promote re-use. Unfortunately re-use is not even considered. The effective support for re-use, in a Modifiable CASE environment, is reliant on an explicit relation between the methodologies supported. This is currently not supported by Modifiable CASE environments.

- **Focus only on completeness and consistency checking**

  Meta-CASE tools only focus is determining if the rules of the various modelling languages have been violated. For example they do not consider supporting assistance during the development process, quality analysis or auto-correction. This is also related to the poor support of the concept of methodology.

2.5 Summary

This chapter has examined meta-modelling and meta-CASE technology. The four layer meta-modelling architecture has been presented and its relation to meta-CASE tools described. Applications of meta-modelling in software engineering field have been examined (OMG MOF, UML, COMMA, OML, OOram, CDIF and MDIF). A review of several representative meta-CASE tools (MetaView, MetaEdit+, Alfabet, Toolbuilder and Graphical Designer Pro) has been presented and limitations of meta-CASE tools, from a methodology representation perspective identified.

The limitations of CASE and meta-CASE technology are the basis from which the research presented in the remainder of this thesis has been derived.

Section II

Research Description

Chapter 3

Meta Object Orientated Tool

In our profession, precision and perfection are not a dispensable luxury, but a simple necessity.

Niklaus Wirth, 1997

3.1 Introduction

This chapter presents the philosophy and architecture of a new meta-case tool that has been developed as a result of this research, MOOT (*M*eta *O*bject *O*rientated *T*ool). The major goal of the MOOT research project is to build a useable, customisable CASE tool which provides a framework within which methodologies can be described. The sub-systems of MOOT that are related to the representation and processing of methodology descriptions are identified and issues related to the overall design and architecture of the new meta-CASE tool are discussed.

3.2 Method

The following is a high level description of the steps taken to develop MOOT.

1. Compare, contrast and evaluate existing CASE and meta-CASE tools to identify the limitations of CASE technology. The current state of CASE technology is outlined in chapter 1. A detailed comparison of meta-CASE tools is presented in chapter 2.

2. Define the rationale and goals of the MOOT project based on the identified limitations of current CASE technology.

3. Devise a representation strategy for methodology descriptions in MOOT. This research includes:

- The development of languages for the description of the syntax and semantics of software engineering methodologies.
- Devising a technique that supports late binding of syntax and semantic descriptions.

4. Analyse the notations commonly used by software engineering methodologies. Derive a new language (NDL) for the representation of methodology syntax.

5. Investigate a meta-systems approach based on an object-orientated meta-model. Derive a new language (SSL) for the representation of methodology semantics.

6. Investigate the binding between NDL and SSL. Derive a technique that supports late binding of NDL and SSL descriptions.

7. Derive a meta-model of the concept of methodology with the representation strategy defined in step 3. Implement the meta-model, with the language defined in step 5, as a library of re-usable semantic description components.

8. Derive a meta-model of concepts germane to all object-orientated methodologies with the representation strategy defined in step 3. Implement the meta-model, with the language defined in step 5, as a library of re-usable semantic description components.

9. Devise a means of efficiently processing methodology descriptions (implemented in the languages from steps 4 and 5).

10. Design the architecture of the new meta-CASE tool, MOOT.

11. Realise a prototype of the system proposed in step 10, which is suitable for assessing the efficacy of the representation scheme for methodology descriptions.

12. Validate the methodology representation strategy by modelling object-orientated methodologies and implementing support for design patterns.

3.3 Rationale and Goals of the MOOT Project

The goals of the MOOT project are defined based on the limitations of current CASE technology as identified in sections 1.6 and 2.4. These goals are:

1. **Support more than one methodology**

   Rationale: Software engineering companies need to utilise a number of different methodologies to support their work.

2. **Flexibility and customisation**

   Rationale: Software engineering companies often utilise in-house methodologies and/or their own extensions to commercial methodologies.

3. **Support the entire software development life-cycle**

   Rationale: The activities of a software engineering company encompass the entire software development life-cycle (SDLC), from requirements gathering through to the implementation and subsequent evolution of software systems. CASE tools should support all software development activities.

4. **Support re-use of software engineering projects**

   Rationale: Whilst it is true that object-orientated technology does not guarantee re-use it is accepted that one of the principle objectives of object-orientated technology is to enable re-use. Supporting re-use should be a key objective of a CASE tool that supports object-orientated methodologies.

5. **Support re-use of projects defined with different object-orientated methodologies**

   Rationale: The representation of potentially re-usable components, by different methodologies, should not be a barrier to their subsequent re-use. This goal is related to the support for re-use and the support of more than one methodology. Software engineering companies use many different methodologies and hence have a repository of potentially re-usable components, each of which may be represented differently.

6. **Separation of the syntax and semantic descriptions of methodologies**

   Rationale: The syntax and semantics of software engineering methodologies have different requirements in terms of the most appropriate modelling language for their description. Providing distinct modelling languages ensures that the descriptions of syntax and semantics are not constrained by each other. The coupling between syntax and semantic descriptions is minimised whilst their cohesion is maximised. In addition, supporting late binding of syntax and semantic descriptions increases their re-usability. The purpose of this approach is to maximise flexibility, adaptability and reusability.

7. **Support for re-use of methodology descriptions**

   Rationale: The descriptions of software engineering methodologies may have many components in common. Object-orientated methodologies, for example, have much in common that could be described by a set of methodology description components. New methodologies can be described by re-using and extending a set of existing methodology description components. These components may be sourced from existing methodology descriptions and from a pre-built library of core methodology description components. Maximising the re-use of semantic components between methodology descriptions is tightly coupled with the support for re-use in general.

The means by which the goals of the research project are addressed is summarised in Figure 3-1. This diagram illustrates how the various goals of the MOOT system have been addressed by some of the design decisions made regarding features of the MOOT system.

The left-hand side of Figure 3-1 lists the goals that have been identified. The right-hand side lists design decisions made regarding features of the MOOT system. The arrows illustrate the mapping between the goals and the design decisions. An arrow that starts or terminates on a box indicates that the mapping relates to all of the goals or design decisions contained within the box.

MOOT is a Modifiable CASE environment (see Figure 1-3 - Classification hierarchy of CASE tool categories). MOOT supports software engineers who apply a software

engineering methodology to describe a software artefact and also supports methodology engineers who create and modify definitions of software engineering methodologies. The overall aim of a system of this type is to support arbitrary methodologies. This addresses goals 1, 2 and 3.

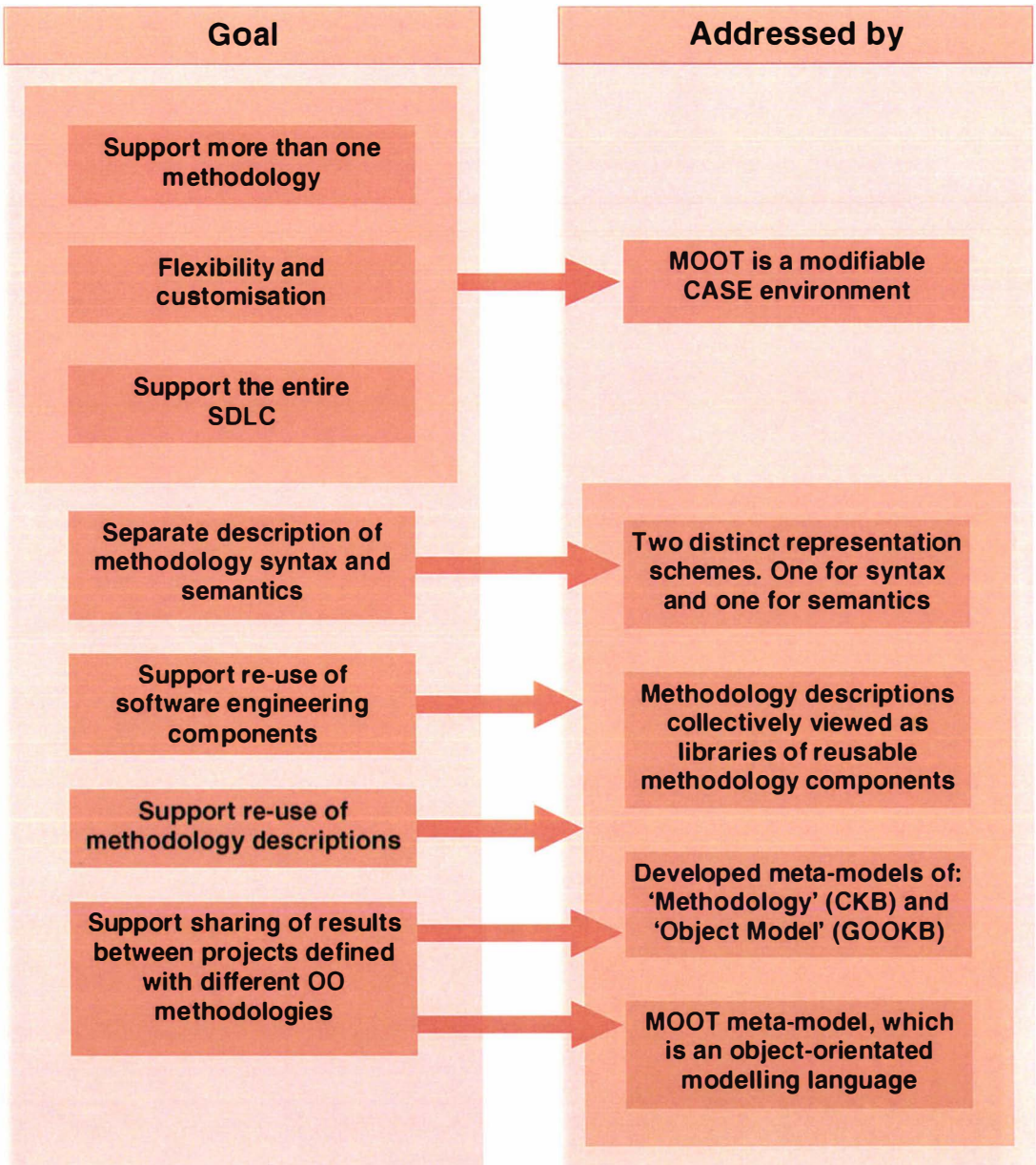


Figure 3-1 - Mapping between goals and design decisions made regarding MOOT

The underlying meta-model of MOOT is an object-orientated language[20]. The choice of an object-orientated language as the meta-model of MOOT supports the natural, efficient

[20] The MOOT meta-model is described in full in section 5.5.2 - MOOT Meta-Model.

and expressive realisation of object-orientated methodologies. The adoption of a representation strategy that directly enables re-use addresses goals 3, 4 and 7.

Re-use is further supported in MOOT with the development of two libraries of reusable methodology description components. The first has been named the Core Knowledge Base[21] (CKB). It implements a meta-model of the concept of 'Methodology'. The second has been named the Generic Object Orientated Knowledge Base (GOOKB). The GOOKB implements a meta-model of concepts that are germane to object-orientated methodologies and is a derivation of the CKB. The development of these two libraries of re-usable methodology description components addresses goals 4 and 7.

Methodologies are defined in MOOT as derivations of the CKB, the GOOKB and from other methodology definitions. The MOOT approach is to view the entire collection of methodology descriptions as a set of potentially re-usable methodology components. This approach further addresses goals 4 and 7.

All object-orientated methodologies support concepts such as class, object, message polymorphism and inheritance. Moreover these concepts are supported throughout the entire software development life-cycle (albeit with different levels of expressiveness). Concepts that are germane to all object-orientated methodologies are defined with the GOOKB. This specifically addresses goal 3.

MOOT utilises two separate modelling languages for the description of a methodology's syntax and semantics. The semantic representation strategy is an expression of the underlying MOOT meta-model. The syntax representation strategy is derived from an analysis of notations used by software engineering methodologies and the consideration of Human-Computer Interaction (HCI) principles. The binding of syntax and semantic descriptions, to compose a complete methodology description, is performed as late as possible. Utilising separate modelling languages and late binding of syntax and semantic descriptions addresses goal 5. This approach also means that common syntax and semantic descriptions need only be defined once, which addresses goal 7.

[21] The use of the term knowledge base is intentional. Ultimately the MOOT system will exhibit more intelligence and make use of Expert System techniques. Section 9.4 - Future Work covers aspects of this work.

The focus of the thesis is on the representation and execution of methodology descriptions by MOOT. This includes the representation of methodologies and software engineering projects, and the design of the CKB and GOOKB. A prototype of the MOOT system has been implemented in order to facilitate the investigation and validation of the approach taken to defining software development methodologies.

3.4 MOOT Methodology Descriptions

A methodology description in MOOT is composed of three parts: a description of the syntax, a description of the semantics and a description of the mapping between the syntax and semantics.

Two new methodology specification languages, NDL (***N***otation ***D***efinition ***L***anguage) and, SSL (***S***emantic ***S***pecification ***L***anguage) have been developed during this research. NDL and SSL allow the definition of the syntax and semantics of a methodology, respectively, in the MOOT system. Late binding of syntax and semantics descriptions is captured with a ***N***otation-***S***emantic ***M***apping (NSM) table.

Figure 3-2 shows the relation between syntax and semantic descriptions, the description of a particular methodology and a corresponding software project in the MOOT system.

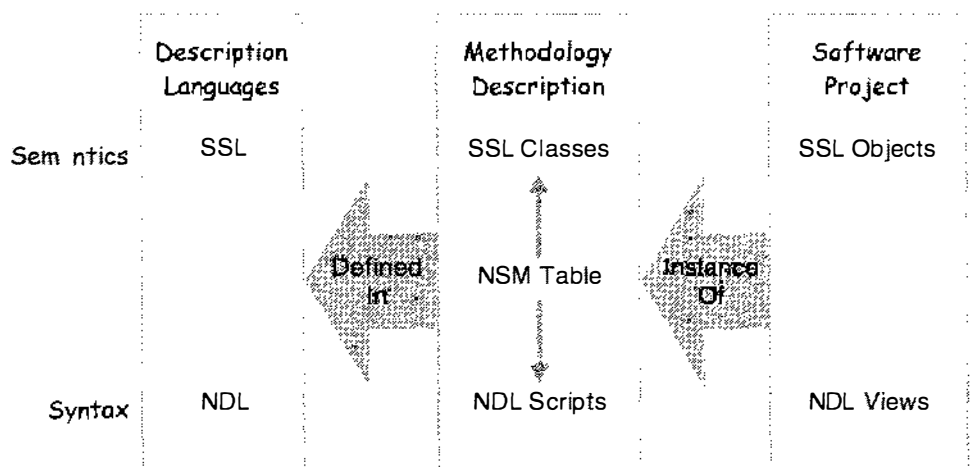


Figure 3-2 - The relation between software projects, methodology descriptions and the description languages in MOOT

NDL is a scripting language used to define the notation of methodology's modelling languages (Figure 3-2). NDL scripts describe how symbols and connections that appear in diagrams are rendered onto a computer display. An NDL description of a notation also provides facilities for binding actions to symbols and connections as well as logical

distortion (Adams, 1998; Clark, 1994; Ham, 1994; Mehandjiska, 1995b; Page *et al.*, 1994). A rendered image generated from an NDL script is called an NDL View (Figure 3-2). A detailed discussion of NDL is given in chapter 4.

SSL is an object-orientated language used to define the semantics of a methodology. This includes the modelling languages and methods supported, the process and the various documents that are produced by application of the methodology. A semantic description of a methodology consists of a collection of SSL classes (Figure 3-2). A software engineering project (developed with a particular methodology) consists of a collection of SSL objects (Figure 3-2). A detailed discussion of SSL and its design is given in chapter 5.

A Notation Semantic Mapping table defines the mapping between notation elements and semantic concepts (Figure 3-2) and is used to implement late binding of syntax and semantic descriptions. One role of the table is to translate 'logical actions' at the user interface, to the corresponding equivalent semantic actions and also to transform semantic actions back into the equivalent logical actions. Notation-semantic mapping is described in detail in chapter 7.

A methodology in MOOT is defined by a collection of NDL scripts and SSL classes. A software project in MOOT consists of a collection of NDL views and SSL objects (Figure 3-2). These views and objects are instances of the NDL scripts and SSL classes in the definition of the methodology used for the project. There is a one-to-many relation between each NDL script and NDL View and a one-to-many relation between each SSL class and SSL object.

The example in Figure 3-3 illustrates how a class diagram, which defines some of the classes for an abstract syntax tree, might be represented, using the MOOT approach. The modelling language used to generate the class diagram in Figure 3-3 consists of a notation and a semantic definition. The class diagram syntax (the notation) in Figure 3-3 is defined by NDL scripts. The concepts supported by the modelling language (class, inheritance relation, class diagram and so on) are defined by SSL classes. The software project consists of instances of the SSL classes (SSL objects) and NDL scripts (NDL Views).

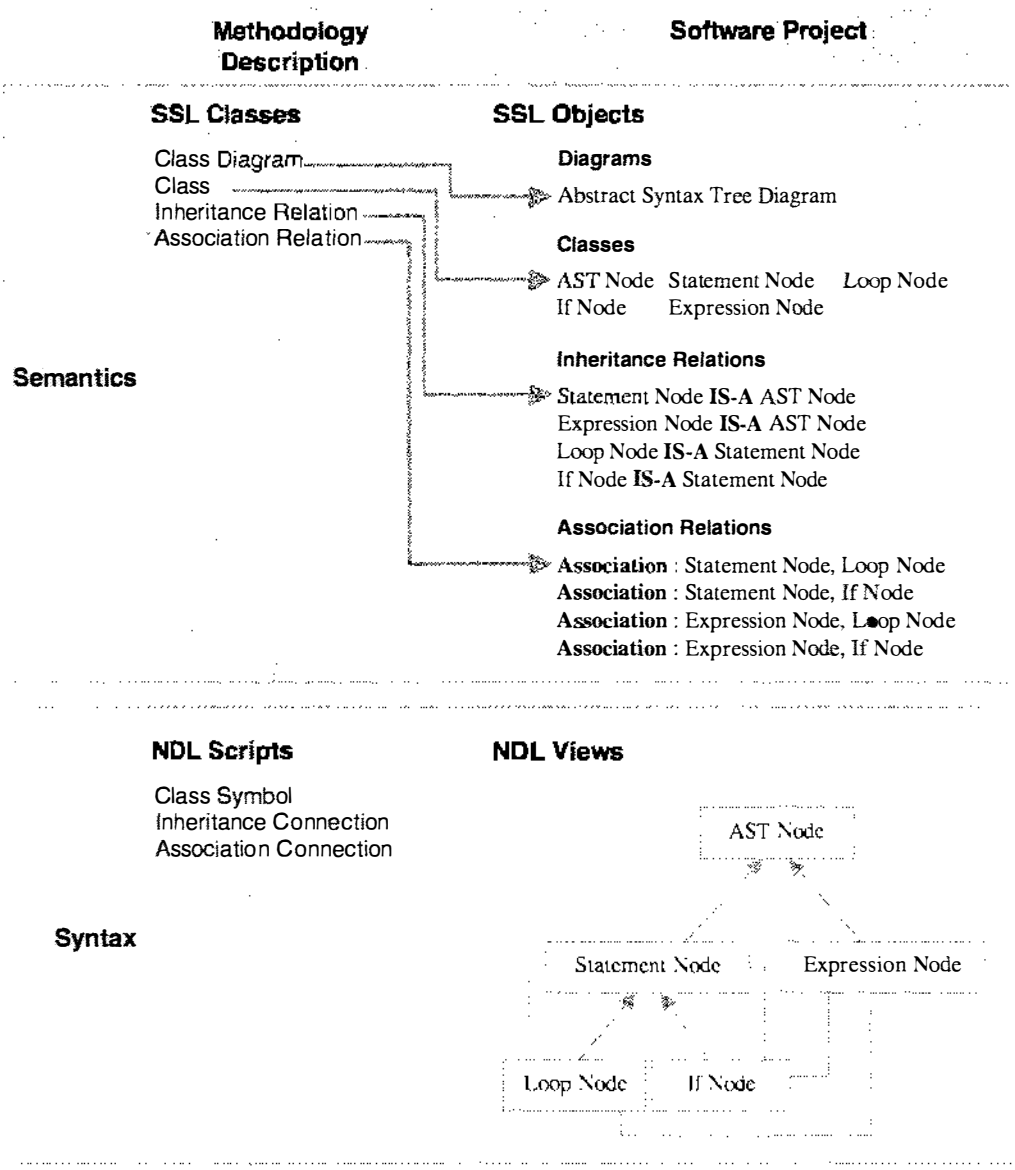


Figure 3-3 - Methodology descriptions and software engineering projects

The example in Figure 3-3 shows SSL objects representing the classes such as *AST Node* in the Abstract Syntax Tree class diagram. It also shows SSL objects representing the inheritance relations (e.g. *Statement Node* is a *AST Node*), associations (e.g. an association between *Statement Node* and *Loop Node*) and an SSL object that represents the diagram itself. An NDL script defines each of the different views that may be created (class symbols, inheritance connections and so on). Each NDL script may have many instances (for example, each rendered class symbol in the Abstract Syntax Tree diagram in Figure 3-3 is an instance of the 'class symbol' NDL script).

The proposed strategy for methodology descriptions in MOOT supports the goal of de-coupled syntax and semantics descriptions. Syntax and semantic descriptions are developed separately and bound together with an NSM table. A semantic description can be bound to many different syntax descriptions and a syntax description may be bound to many semantic descriptions. An NSM table defines each particular mapping between a semantic and syntax description.

There are many advantages of this approach:

- Semantic descriptions are not constrained by particular notations. No fixed mapping between elements in the semantic and syntax description is therefore necessary. Elements of a notation may correspond to one or more semantic elements and vice-versa.
- Methodology engineers can develop libraries of notations. In addition the notation used for a particular semantic description can be changed at any time.
- Methodology engineers can develop libraries of methodology semantic descriptions. New methodologies can therefore be defined as extensions of those already supported.
- Syntax and Semantic descriptions may be developed in isolation.

3.5 The CKB and GOOKB

Figure 3-4 shows how the CKB and GOOKB are related to methodologies in MOOT.

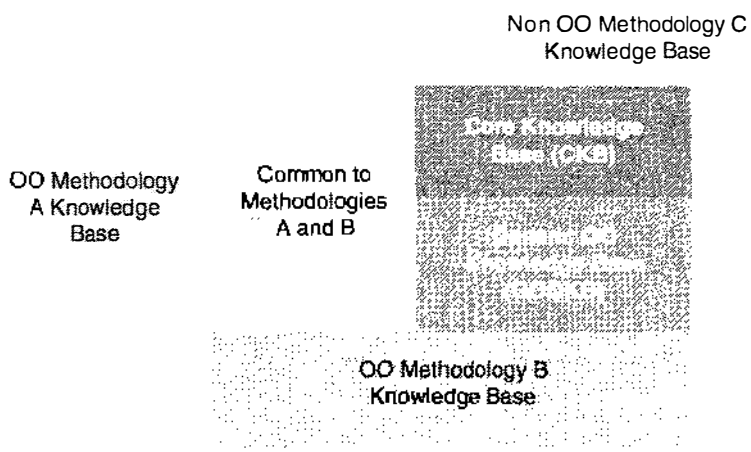


Figure 3-4 Knowledge bases in MOOT

OO Methodology A and *OO Methodology B* in Figure 3-4 are derived from the GOOKB (and by implication the CKB). They also have features in common. Non-OO methodologies only extend the CKB and may have features in common. Object-orientated methodologies may have common features with non object-orientated methodologies[22].

Figure 3-5 illustrates the relation between the CKB, the GOOKB, methodologies and software projects in MOOT.

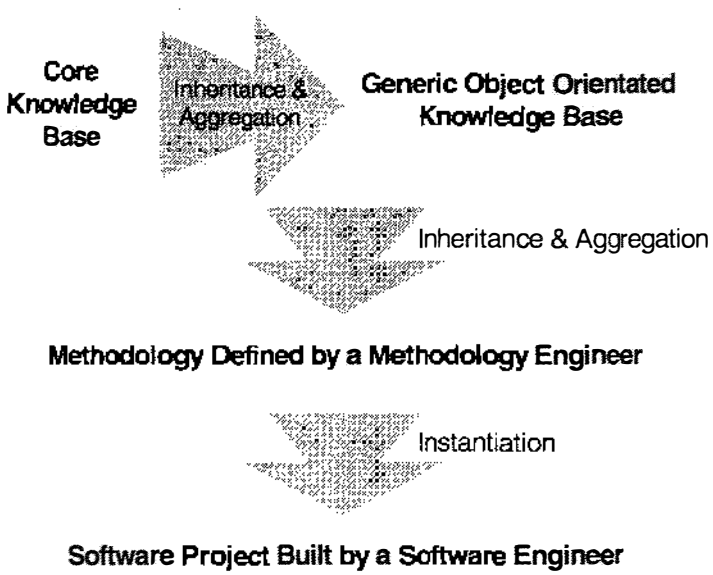


Figure 3-5 - The relation between the CKB, the GOOKB, methodologies and software engineering projects in MOOT

New methodologies in MOOT are derived from the CKB and the GOOKB using inheritance and aggregation. A methodology may also be defined in terms of previously defined methodologies using inheritance and aggregation. A MOOT methodology semantic definition consists of a collection of SSL classes derived from the CKB, GOOKB and potentially from other methodology definitions. A software project is constructed when a software engineer applies a methodology that has been defined in MOOT. The software project is an instance of the methodology used by the software engineer and consists of a collection of SSL objects, each of which is an instance of an SSL class in the methodology definition.

[22] Examples include Rumbaugh's use of Data Flow Diagrams from Structured Systems Analysis and the use of state transition diagrams in various object-orientated methodologies.

Figure 3-6 illustrates how MOOT relates to other meta-CASE tools in terms of the four layer meta-modelling architecture defined in Table 2-1 - Four layer meta-modelling architecture.

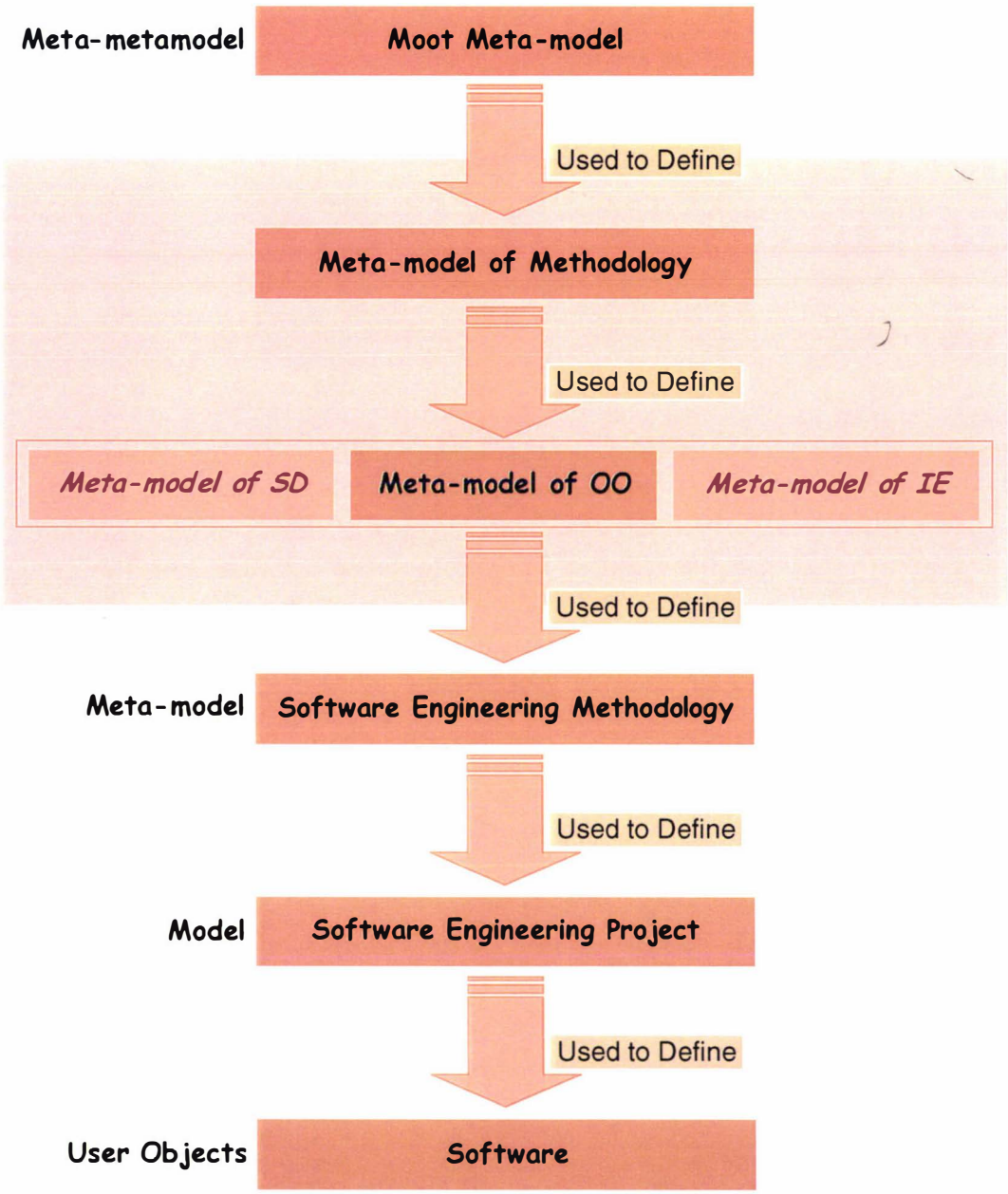


Figure 3-6 - Meta-modelling architecture

Existing meta-CASE tools define methodologies *solely* in terms of their meta-model. The MOOT approach, however, is quite different. MOOT introduces two additional layers between the topmost layers in the four layer meta-modelling architecture. Figure 3-6 shows how the MOOT meta-model is used to define a meta-model of methodology. The meta-model of methodology is implemented with the Core Knowledge Base (CKB). The

third layer in Figure 3-6 consists of meta-models that correspond to various approaches to the engineering of software. In the middle of the third layer is a meta-model of the object-orientated approach. This meta-model is implemented with the Generic Object Orientated Knowledge Base (GOOKB). On the left of layer three is a meta-model of structured development and on the right is a meta-model of information engineering[23].

The MOOT meta-model, therefore, is used to define various meta-models, which are in turn implemented as re-usable SSL class libraries. Methodologies in MOOT are defined as extensions of these libraries.

3.6 Addressing the Limitations of Meta-CASE tools

Existing Meta-CASE tools (as discussed in section 2.4 - Limitations of Current Meta-CASE Technology) suffer from limitations in the following areas:

I. Poor representation of the concept of 'methodology' and 'software process'

II. No relation between defined methodologies

III. No support for re-use of methodology descriptions

IV. No support for re-use of software engineering projects

V. High coupling of syntax and semantic descriptions. Subsequent lowering of the cohesion of syntax and semantic descriptions

VI. Syntax description is primitive

VII. Usability is poor

Figure 3-7 illustrates how the limitations of existing meta-CASE tools have been addressed by the MOOT approach. On the left-hand side is the list of limitations that have been previously identified. The right-hand side lists design decisions made regarding the features of the MOOT system. The arrows illustrate the mapping between the limitations and the properties of the MOOT system that address them. An arrow that starts or terminates on a box indicates that the mapping relates to all of the limitations or design decisions contained within the box.

---

[23] These two meta-models have not been implemented and are shown to illustrate the overall philosophy. This is further discussed in section 9.4 - Future Work.

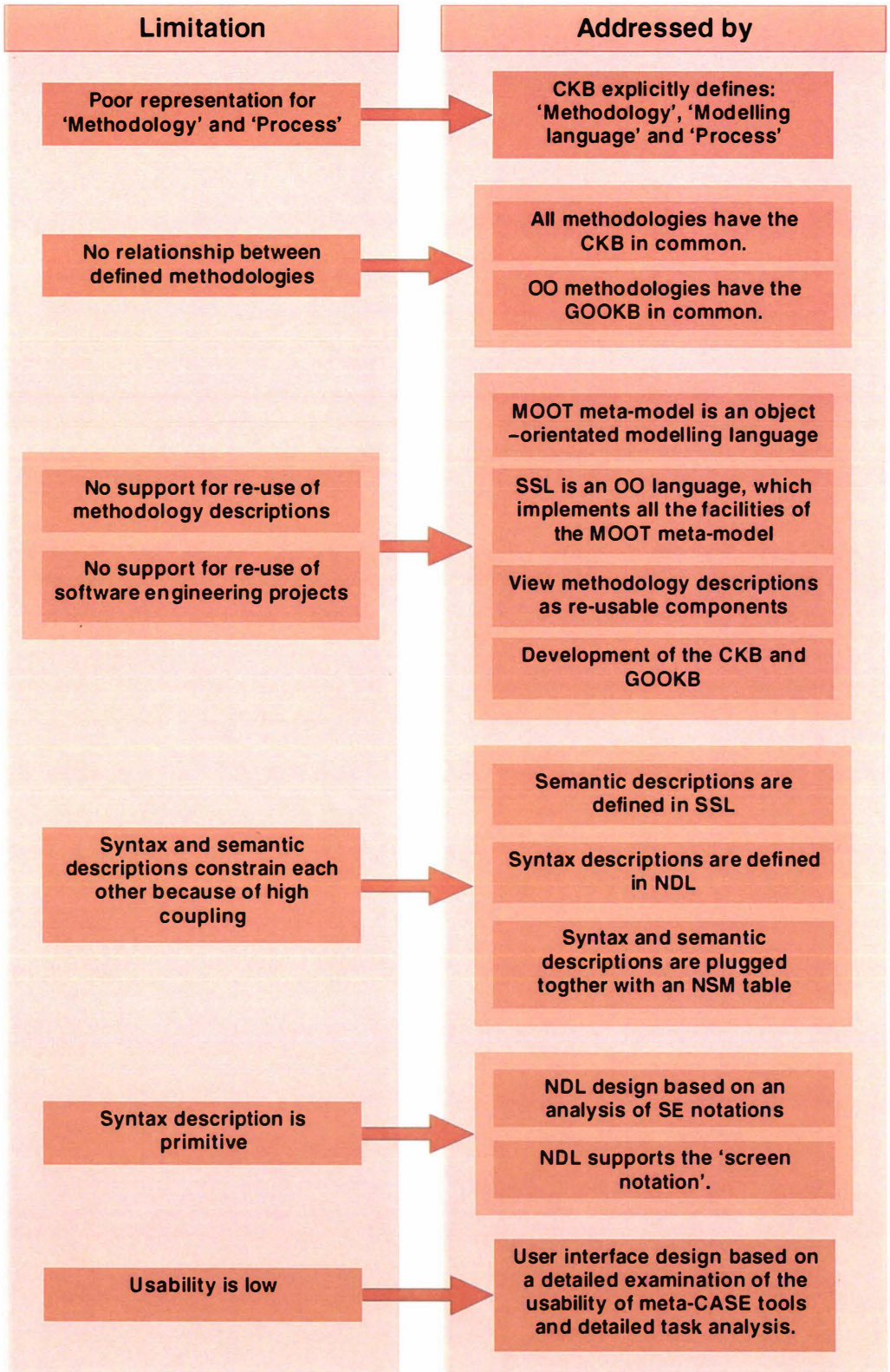


Figure 3-7 - Addressing the limitations of meta-CASE tools

The MOOT system addresses limitation I by explicitly supporting the concepts of methodology and software process within the derived Core Knowledge Base (CKB). The

CKB implements a meta-model of 'Methodology' and explicitly defines 'Methodology', 'Modelling Language' and 'Process'.

All methodologies in MOOT are derived from the CKB. MOOT also provides explicit support for object-orientated methodologies with the development of the Generic Object Orientated Knowledge Base (GOOKB). The GOOKB is derived from the CKB and implements a meta-model of concepts germane to all object-orientated methodologies. All object-orientated methodologies have the GOOKB in common. This addresses limitation II.

Methodology semantic descriptions (including the CKB and GOOKB) are defined in terms of the MOOT meta-model and implemented in SSL. The MOOT meta-model is an object-orientated modelling language and thus provides facilities such as classes, inheritance, message passing and polymorphism. SSL is an object-orientated language that implements all the facilities of the MOOT meta-model. This addresses limitation III.

The GOOKB and the CKB constitute a set of re-usable SSL classes from which all methodologies in MOOT are derived. Moreover the MOOT approach is to consider that all methodology descriptions consist of potentially re-usable components. This addresses limitation III.

The strategy for supporting re-usable methodology components means that there are relations between the different methodologies in MOOT. Object-orientated methodologies in particular always have the components in the GOOKB in common. Software projects can be re-used because they always share a common definition. This addresses limitation IV.

Limitation V has been addressed by the development of separate syntax and semantic representation schemes for methodologies (NDL – syntax and SSL – semantics). The association of syntax and semantic descriptions is achieved with the development of NSM tables. Reducing the coupling between syntax and semantic descriptions addresses limitations III. The separation of syntax and semantic descriptions in MOOT also means that the independent re-use of syntax descriptions is possible, which further addresses limitation III.

Limitation VI is addressed in two ways. Firstly, NDL is designed to support the description of interactive diagrams and provides facilities for the use of colour, logical distortion, hotspots and so on. NDL thus supports 'screen' notations rather than 'pencil and paper' notations. Secondly, the facilities NDL provides is based on the analysis and modelling of notations used in software engineering.

In brief[24], limitation VII is addressed by a detailed examination of the usability of meta-CASE tools, which has been conducted in association with other researchers. A CASE tool evaluation framework[25] has been developed, applied and documented in (Choi, 1996; Phillips *et al.*, 1998a). The design of the MOOT software engineer's user interface, based on this evaluation and on subsequent task analysis, is presented in (Adams, 1998; Philips *et al.*, 1998b, c).

3.7 Architecture of MOOT

MOOT has two distinct types of user. Software engineers utilise MOOT to build descriptions of software artefacts. Methodology engineers utilise MOOT to build descriptions of software engineering methodologies. MOOT supports each type of user by performing two distinct roles (MOOT as a CASE tool and MOOT as a methodology development tool). The two roles of the MOOT system are illustrated in Figure 3-8.

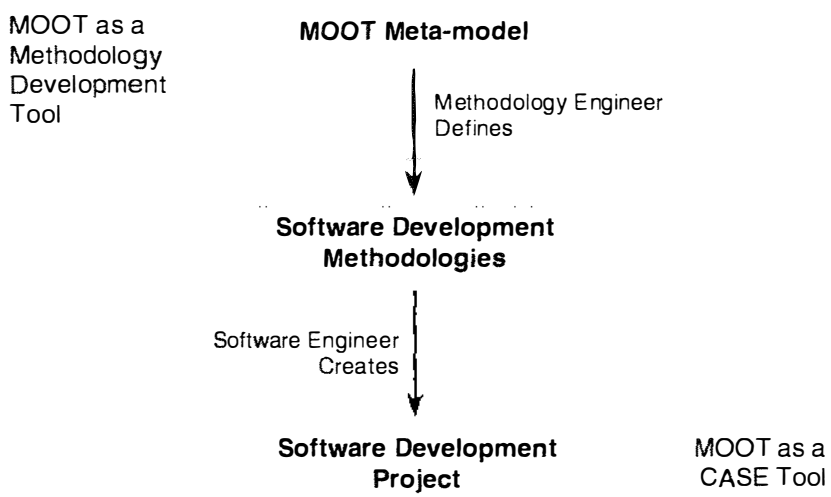


Figure 3-8 - The two roles of the MOOT system

[24] A detailed discussion of usability is outside the scope of the thesis. The reader is directed to (Adams, 1998; Choi, 1996; Gray 1995; Phillips *et al.*, 1998a, b, c) for more information.

[25] A high level overview of the evaluation framework is presented in appendix I.

The MOOT system is divided into two logical sub-systems (Figure 3-9) that correspond to the two roles of MOOT. These are the methodology development sub-system and the CASE tool sub-system.

The methodology development sub-system is an integrated tool-set allowing a methodology engineer to specify, modify and test methodology descriptions. Descriptions created using the methodology development sub-system are represented using SSL classes (for the semantic description) and NDL scripts (for the syntax description).

The CASE tool sub-system is the methodology CASE component of the MOOT environment. It is an integrated tool-set that allows a software engineer to develop software by applying methodologies described using the methodology development sub-system.

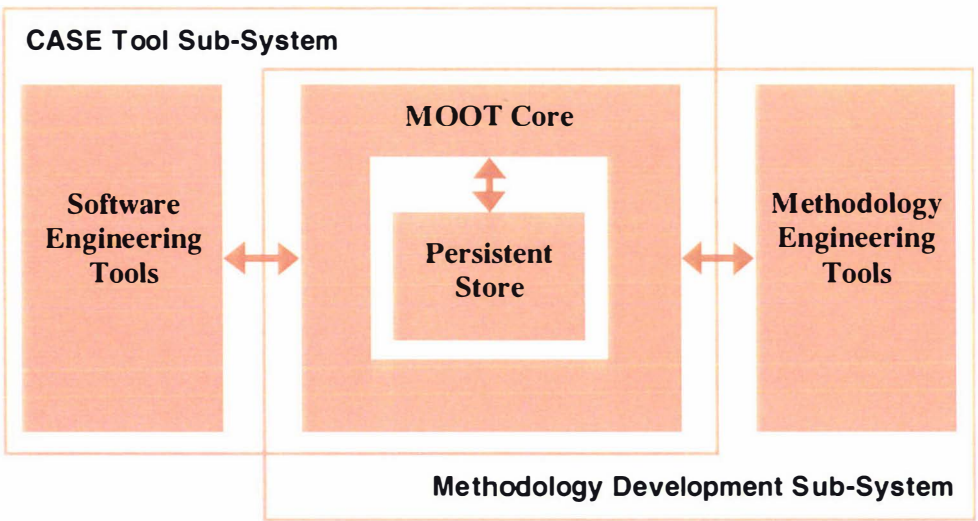


Figure 3-9 - Moot system

Both the CASE tool subsystem and the methodology development subsystem make use of the MOOT Core, which insulates the underlying, shared, repository (Persistent Store). Software engineering projects and software development methodologies are both stored in the Persistent Store.

The high-level system overview given in Figure 3-9 is further decomposed in Figure 3-10, which shows the derived architecture of the MOOT system. The Arrows in Figure 3-10 indicate that a communication pathway exists between two components.

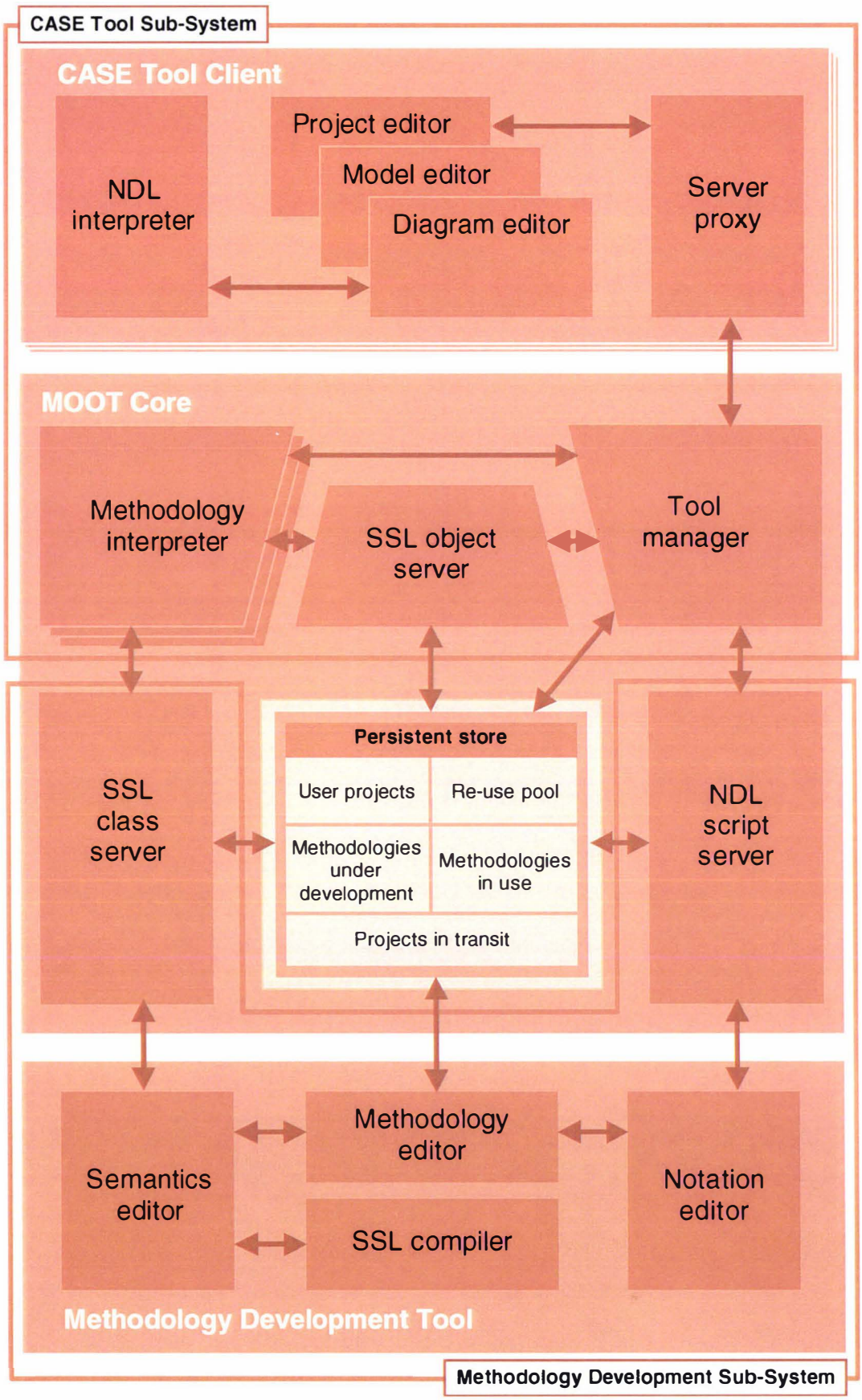


Figure 3-10 - Proposed, top level, system architecture

MOOT has a three-tier architecture where presentation, application logic and data are distributed across three levels, interconnected over a network (I-Kinetics, 1998). The presentation level corresponds to the CASE tool client and the methodology development tool. The application logic level corresponds to the MOOT core, which is responsible for isolating the persistent store (the data level).

Each of the components in Figure 3-10 is briefly discussed in the following sections.

3.7.1 CASE Tool Client

The CASE tool client provides the user interface of the CASE tool sub-system. It is a 'light weight' or thin client and is *only* responsible for the presentation of a software engineering project. It is essentially a user interface shell that is parameterised by NDL descriptions of modelling language notations. The CASE tool client provides a set of drawing tools that allow a software engineer to construct diagrams. The set of drawing tools available is based on a set of generic tools appropriate for the construction of arbitrary diagrams, and the notation elements that are defined in NDL specifications. A software engineer creates diagrams by selecting drawing tools that represent notation elements and placing instances of these onto a drawing canvas. The corresponding methodology semantic descriptions are managed by a corresponding instance of a methodology interpreter in the MOOT core. The CASE tool client is responsible for mapping physical user input to 'logical actions'. Actions that affect the meaning of the model being built (e.g. creating a connection) are propagated to the server. The client handles actions that do not affect the meaning of the model being built (such as resizing a symbol).

Multiple clients may interact with the CASE Tool server via the Tool Manager of the MOOT Core. The Tool Manager functions as a server, processing one thread of control for each CASE Tool client. The Tool Manager maintains an instance of the Methodology Interpreter for each software engineering project that is open in each client. The Tool Manager and the Methodology Interpreters are, in turn, clients of the Persistent Store.

The client is implemented in Java so MOOT may be used from any computer on a network that has a Java interpreter (SUN, 1998). The design and implementation of the CASE tool client has been completed in association with another researcher and is

outside the scope of this thesis. A detailed discussion of the CASE tool client, its design and implementation, HCI issues etc, is presented in (Adams, 1998; Phillips *et al.*, 1998b, c). Aspects of this work, relating to NDL and the execution of NDL scripts, is covered in detail in chapter 4.

3.7.2 Methodology Development Tool

The methodology development tool is used to maintain the collection of methodology descriptions. It provides a notation editor that is used to define notations and a semantics editor that is used to define the semantics of methodologies. The methodology editor is used to associate notation descriptions to semantic descriptions in order to provide complete methodology definitions.

The notation editor uses a visual programming approach where the user draws example 'pictures' of the notation[26]. The notation editor then generates an NDL description of the symbols and connections that comprise the notation, based on the examples drawn by the user. The notation editor itself is not in the scope of the thesis. Initial work on the notation editor tool is documented in (Ham, 1994; Mehandjiska *et al.*, 1995b).

The semantics editor is used to define the semantic specification of a methodology. This includes the various modelling languages, documents and the process supported by the methodology. The semantics editor generates SSL descriptions. The semantics editor itself is also outside the scope of the thesis.

SSL is compiled to a platform independent binary representation for reasons of efficiency. The SSL compiler translates SSL into SSL-BC (the platform independent binary representation) and is discussed in chapter 5.

The methodology editor is used to associate particular notations (defined in NDL) and methodology semantic definitions (defined in SSL) via NSM tables. The development of the methodology editor is also outside the scope of the thesis.

---

[26] The Notation Editor is similar to the more recent BuildByWire system described by Mugridge *et al.*, 1998; Warwick *et al.*, 1996. BuildByWire generates a collection of JavaBean components that correspond to a notation, as well as an editor JavaBean. The notation editor, in contrast, generates an NDL description of the notation only, which is subsequently interpreted by the CASE tool client.

3.7.3 MOOT Core

Tool Manager

The tool manager facilitates communication between the MOOT Core and the CASE tool clients. The tool manager is responsible for co-ordinating access to shared resources, and for monitoring the system's operation. There is a single instance of the tool manager operating, for a particular instance of the MOOT system. The tool manager is responsible for maintaining details specific to each client (such as the software engineering project that is open, the methodology in use and so on) and the corresponding methodology interpreter. Messages from the clients (such as: delete a class, add an operation or create a new state) are accepted by the tool manager and bound to a message to an SSL object and executed with a particular methodology interpreter.

Methodology Interpreter

Each CASE tool client is supported by an instance of the methodology interpreter. It is responsible for processing methodology semantic descriptions written in SSL. It applies the description of the active methodology, defined in SSL, to the user's project in response to logical actions at the user interface.

SSL is compiled to a platform independent binary representation (SSL-BC) for reasons of efficiency. The methodology interpreter executes the intermediate representation on a purpose built virtual machine (SSL-VM). SSL-BC, the SSL-VM and the SSL compiler are discussed in chapter 5.

Notation, SSL Class and SSL Object Servers

There is only one instance of each server executing at any time. Each server is responsible for isolating the persistent store from the rest of the system and for maintaining a cache. They all must ensure mutually exclusive access when appropriate. For example, the SSL Object server must ensure that an SSL object cannot be updated by more than one client at the same time.

Persistent Store

The persistent store is the repository for the MOOT system, both at the methodology description level, and at the user-project level. Methodologies are stored in two different partitions in the persistent store. Methodologies that have been developed and tested are

stored in the Methodologies-In-Use section. Methodologies-In-Use have been completely defined and tested, and are ready to be used to create user projects. These methodologies can not be modified, as this would affect the projects that use them. The Methodologies-In-Use section is read-only.

The Methodologies-Under-Development section contains methodologies that are in the process of being specified, tested and refined. It is not possible to use these methodologies to develop projects until they are deployed and become part of the Methodologies-In-Use section.

Software engineering projects are stored in two partitions in the persistent store. Software engineering projects (and portions of software engineering projects) that have been completed can be placed in the "re-use pool". These components are available to all other software engineering projects in the MOOT system to be re-used. The re-use pool is read only as re-usable components can only be extended, not modified. The User Projects area contains all software engineering projects that are in the process of being developed.

A Company may wish to distribute software projects, or parts of them, to clients without disclosure of their methodology. Methodology descriptions exported with a project are stored in the separate In-Transit area, and are not viewable on the target system.

3.8 The MOOT Prototype

The focus of the thesis is on the representation and execution of methodology descriptions by MOOT. Work has been carried out in the following areas:

- Development of the syntactic representation of software development methodologies which is addressed by the development of a new language, NDL. NDL and a prototype NDL interpreter are described in chapter 4.

- Derivation of the semantic representation of software development methodologies which is addressed with the development of the MOOT meta-model and a new language, SSL. The MOOT meta-model and SSL are described in chapter 5.

- Design and implementation of the methodology interpreter, which includes the development of the intermediate binary representation of SSL (SSL-BC), the design of a new virtual machine that SSL-BC executes on (the SSL-VM) and a compiler that translates SSL to SSL-BC. The development of SSL-BC, the SSL-VM and the SSL compiler are discussed in chapter 5.

- Design and implementation of two libraries of re-usable methodology semantic components, the Core knowledge Base and the Generic Object Orientated Knowledge BASE. The development of these libraries is described in chapter 6.

- Development of a technique that supports late binding of syntax and semantic descriptions (NSM tables). The function of NSM tables is discussed in chapter 7.

The support for re-use of software development methodologies and software engineering projects is discussed throughout chapters 4, 5, 6 and 7.

A prototype of MOOT has been implemented in order to facilitate the investigation of the approach to defining software development methodologies. The architecture of the MOOT prototype is shown in Figure 3-11. All further discussion of MOOT in the thesis is in relation to this prototype.

The current implementation of the MOOT core is a single server. All of the components of the MOOT core execute in a single process, rather than being distributed over a network. The Server accepts connections from multiple clients. The SSL compiler currently accesses the persistent store directly. The persistent store is implemented as a collection of files.

The components of the MOOT core and the SSL compiler are discussed in chapters 4, 5, 6 and 7. The Java CASE tool client is based on the NDL interpreter discussed in chapter 4 and is implemented in association with another researcher (Adams, 1998; Phillips *et al.*, 1998b-c).

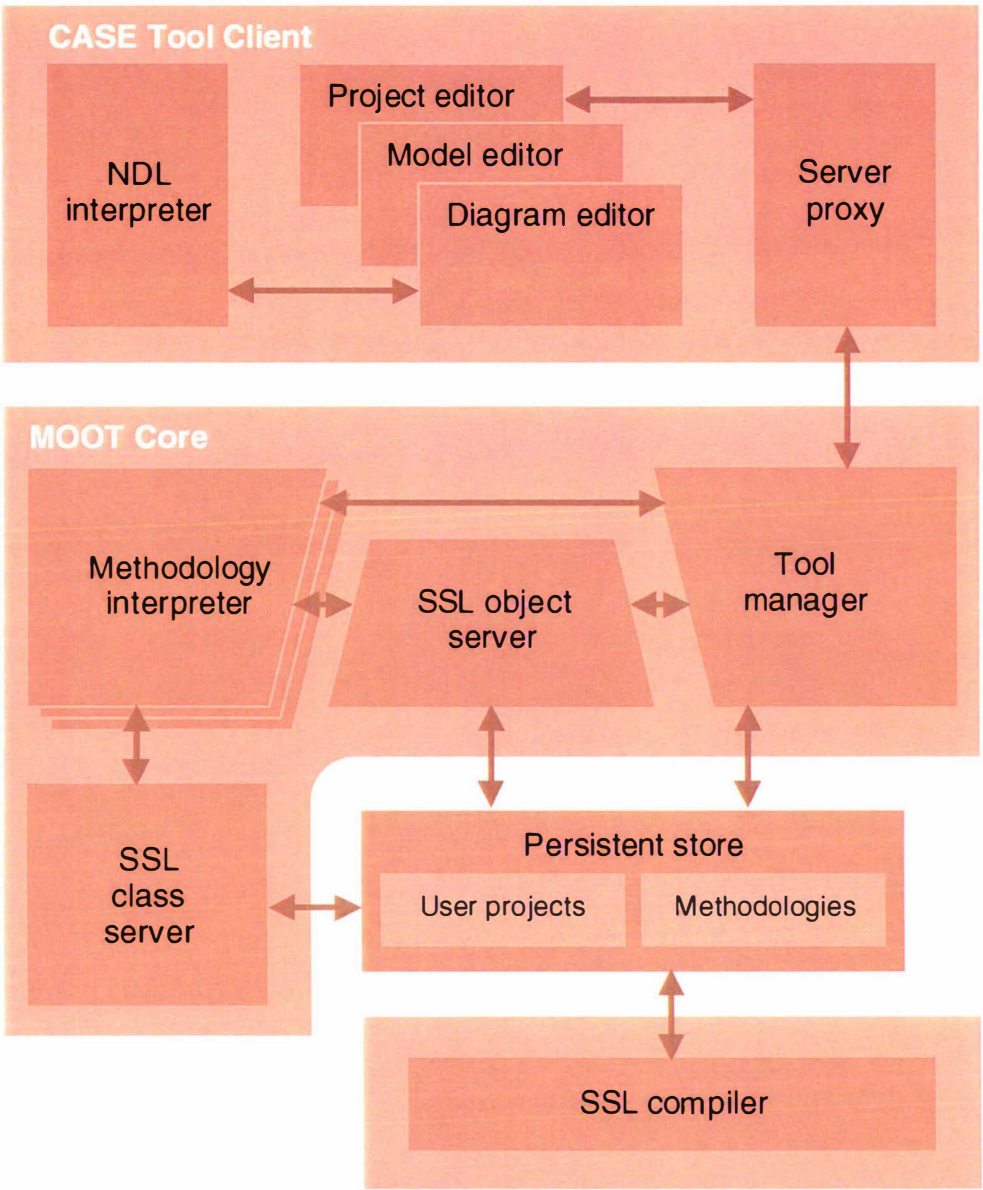


Figure 3-11 - Architecture of the MOOT prototype

3.9 Summary

This chapter has presented the approach taken in the research to address the limitations of methodology CASE tools and meta-CASE tools. This included:

- A proposed architecture of MOOT system. This architecture describes a distributed, three layer, database centric system. The architecture has been designed to effectively support the two categories of user for the MOOT system – software engineers and methodology engineers. The persistent store at the lowest layer stores methodologies and software engineering projects. The second layer consists of a collection of

distributed components that isolate the persistent store. The top layer consists of thin CASE tools clients and methodology specification tools.

- An outline of MOOT methodology descriptions. The syntax and semantics of methodologies are described completely separately in the MOOT system. Two new languages have been developed during this research for this purpose. The ***N***otation ***D***efinition ***L***anguage (NDL) is used to define syntax and the ***S***emantic ***S***pecification ***L***anguage (SSL) is used to define semantics. A complete methodology description, in the MOOT system, is made by associating an NDL and SSL description with a ***N***otation ***S***emantic ***M***apping (NSM) table.

- An outline of the ***C***ore ***K***nowledge ***B***ase and the ***G***eneric ***O***bject ***O***rientated ***K***nowledge ***B***ase. These are two libraries of re-usable methodology semantic components that are implemented in SSL. All methodologies have the CKB in common. Object-orientated methodologies also have the GOOKB in common.

- The description of the architecture of a prototype of MOOT that has been built during the research detailed in this thesis.

Chapter 4

Notation Definition Language

By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems

Booch 1991

4.1 Introduction

This chapter investigates the framework within which the syntax of a methodology is defined in the MOOT system. A new language has been developed to support the description of the visual syntax of the modelling languages supported by a methodology. The derived language (*N*otation *D*efinition *L*anguage - NDL), is presented along with the implementation of a prototype system used to assess the language.

4.2 Method

The following is an outline of the steps followed during the development of NDL.

1. The graphical notations of software engineering methodologies are explored and their components identified. The modelling languages considered in the analysis are from the software engineering literature. *Some* of the modelling languages considered include:
   - UML class diagrams (Booch and Rumbaugh, 1995; Jacobson *et al.*, 1996; OMG, 1997g; Rational, 1997b)
   - Coad and Yourdon class diagrams (Coad and Yourdon, 1990, 1991a, b; Coad and Nicola, 1993)
   - Data flow diagrams (Rumbaugh *et al.*, 1991; Whitten *et al.*, 1994)
   - State transition diagrams (Feylock, 1977; Booch, 1991, 1994)
2. The requirements for a language that allows the description of arbitrary notations are derived based on the analysis of notations in step 1. The language must support all

the components of visual notations of the various modelling languages used in software engineering.

3. A new language is derived which satisfies these requirements. This new language is called NDL (***N***otation ***D***efinition ***L***anguage). NDL is required to support more than the static reproduction of a notation on a computer display. It must also support many facilities often not considered by methodology developers such as logical distortion and the use of colour.

4. A means of efficiently processing NDL descriptions is developed and implemented.

4.3 Models and Notations

A notation is the visual syntax used to document a model. A particular modelling language may be used for many different purposes. It is therefore possible for the same modelling language to have more than one notation. It is also possible for a single notation to be used for many different modelling languages. Hence a many-to-many relation exists between the concepts modelling language and notation.

The majority of notations supported by the modelling languages common in software engineering methodologies are graphs containing nodes (symbols) connected by paths (connections).

A notation consists of:

- Symbols that represent semantic concepts
- Connections that represent semantic relations between concepts
- Text associated with the symbols and connections
- Constraints that specify the way symbols and connections are created and manipulated

Examples of symbols include Coad and Yourdon's Class&Object, Booch's Bubble and Rumbaugh's Process Bubbles. Examples of connections include Gen-Spec relations in Coad and Yourdon, Using relations in Booch and Associations in Rumbaugh. Some

symbols are compositions of other symbols; examples are subject areas (Coad and Yourdon), class categories (Booch) and packages (UML).

An Example: The State Transition Diagram

State transition diagrams are utilised in diverse areas of computer science. Feylock (1977) uses state transition diagrams as a representational basis for Computer Assisted Instruction systems. Booch (1991, 1994) utilises state transition diagrams to model the dynamic behaviour of objects. The state transition diagram used by Feylock and Booch both have the following properties:

- A single start state
- Multiple end states
- Transitions between states. A transition is labelled with an event that causes it to occur. A transition may be optionally labelled with an action that is carried out when the transition occurs

The semantics of the state transition diagram used by Feylock and Booch is the same, as they both use the Mealy model (Booch, 1991) where actions are bound to the events of the transitions. Two key differences exist in the utilisation of state transition diagrams by Booch and Feylock:

1. State transition diagrams are used for very different purposes. Feylock uses state transition diagrams as a representational basis for Computer Assisted Instruction systems. Booch utilises state transition diagrams to model the dynamic behaviour of objects.
2. The notation of the state transition diagrams is different.

Figure 4-1 is an example of two state transition diagrams, which represent the operation of drawing a rubber-band line[27]. The topmost diagram uses Booch's notation and the bottom uses Feylock's notation. The underlying meaning of the two diagrams is identical although the notation used in each is different.

[27] To draw a rubber band line the user first selects the start point on the drawing surface by depressing a button on the mouse. A rubber band line is drawn from the start point to the current position when the mouse is moved. The actual line is drawn when the mouse button is released.

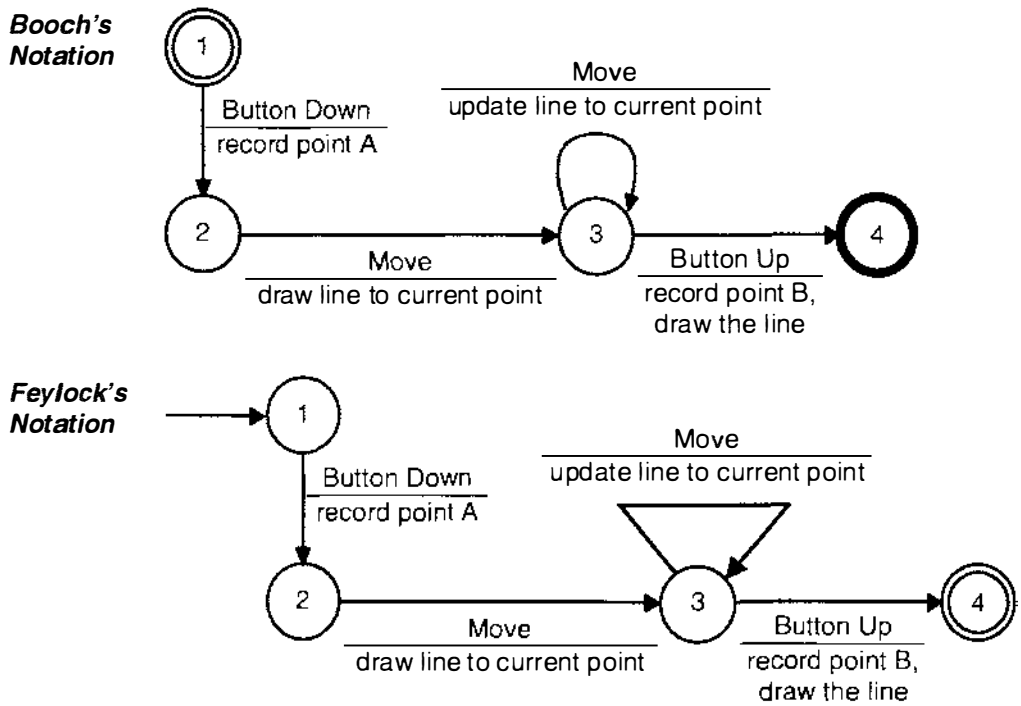


Figure 4-1 - A state transition diagram drawn in the notation of Booch and Feylock

The meaning that is being conveyed in both diagrams of Figure 4-1 is:

State 1 is a start state, **state 2** and **state 3** are normal states and **state 4** is a stop state

If there is a ***Button Down*** event in ***state 1*** then

go to ***state 2*** and ***record point A***

If there is a ***Move*** event in ***state 2*** then

go to ***state 3*** and ***draw line to current point***

If there is a ***Move*** event in ***state 3*** then

go to ***state 3*** and ***update line to current point***

If there is a ***Button Up*** event in ***state 3*** then

go to ***state 4*** and ***record point B, draw the line***

The representation of start and stop states is different in both notations. Booch uses a special symbol for both the start and stop states. A start state, in Feylock's notation, has an incoming transition that does not originate from another state. Booch represents a start state with a double circle. The symbol Feylock uses for a stop state is used for a start state in Booch. A transition is composed of straight line segments in Feylock's notation and curves in Booch's notation.

This simple example illustrates the many-to-many relation that exists between the concepts modelling language and notation. It also serves to demonstrate that a notation defines the syntax of a particular representation of a model and is not the same as the model itself.

It is clear that NDL must allow a notation to be defined in a way such that the notation is not tightly coupled to the modelling that is represented. For example the mechanism should allow the notations of Feylock and Booch state transition diagrams to be defined and associated with a single semantic description of the State Transition Diagram. In terms of this research this means that the semantics of state transition diagrams would be defined once in SSL (see chapter 5) and NDL would be used to define the notations of Feylock and Booch.

4.4 Analysis of Notations

The analysis of notations focuses only on the visual syntax, and not on the meaning the modelling languages are capable of conveying (the semantics). It is important to avoid a simple static view of notations during the analysis and note that computer presentations of models need not be restricted to simple 'pictures'. They may include, for example, facilities for logical distortion and animation (Apperley and Chester, 1995; Smith and Anderson, 1996)[28]. Only two-dimensional notations are considered in the analysis as three-dimensional layout and navigation is currently not practical on desktop machines. This limitation has also been adopted by UML, for the same reason:

> "Note that the UML notation is basically 2-dimensional. Some shapes are 2-dimensional projections of 3-d shapes (such as cubes) but they are still rendered as icons on a 2-dimensional surface. In the near future 3-dimensional layout and navigation may be possible on desktop machines but it is currently not practical."
>
> *From the UML Notation Guide, version 1.0 (Rational, 1997)*

[28] A detailed consideration of human computer interaction in the context of CASE tools is not the focus of the thesis. The reader is directed to (Adams, 1998; Amulet, 1998; Apperley and Chester, 1995; Brough, 1992; Choi, 1996; Gray, 1995; McWhirter, 1998; McWhirter and Nutt, 1994; Minas and Viehstaedt, 1995; Mugridge *et al.*, 1998; MultiView, 1998; Myers *et al.*, 1997; Phillips *et al.*, 1998a-c; Purchase, 1998; Read and Marlin, 1996, 1998; Warwick *et al.*, 1996) for more information on HCI issues related to CASE and meta-CASE tools.

The analysis of notations is presented by consideration of: Symbols, Connections, Docking Areas, Groups, Presentation and Actions. Each is discussed in turn.

4.4.1 Symbols

Figure 4-2 is an example of a class diagram, taken from Gamma *et al.* (1995), related to the Visitor pattern. The same diagram is drawn with the UML, Coad and Yourdon and Booch notations.

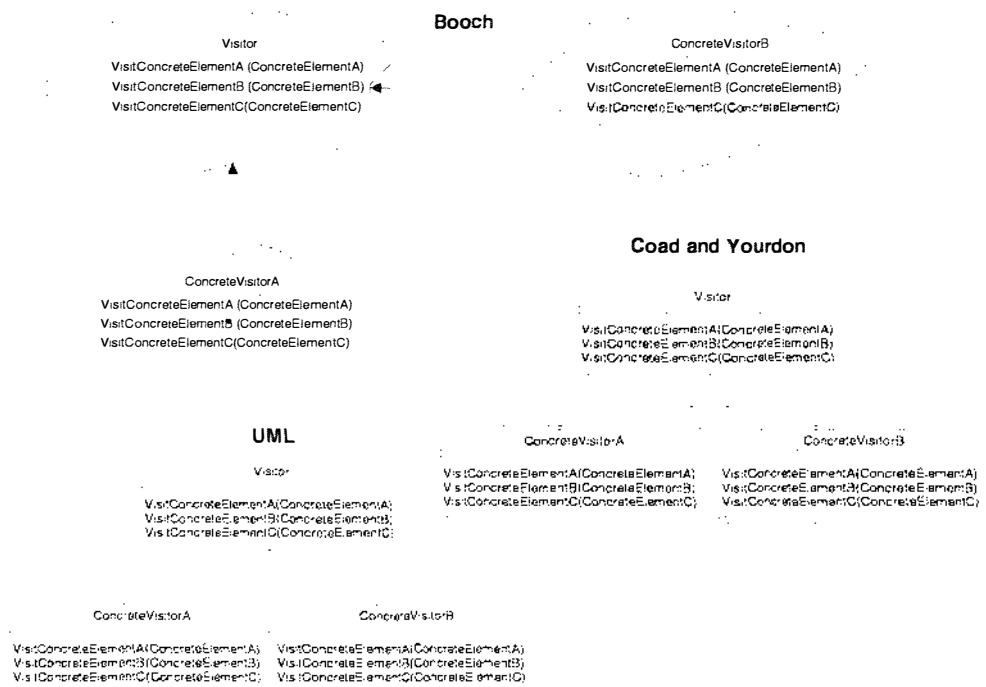


Figure 4-2 - A simple diagram drawn with UML, Coad and Yourdon and Booch notations

Symbols are composed of lines and arcs and enclose text fields. Symbols have a well defined boundary or border that is typically visually represented. The boundaries of all the symbols in Figure 4-2 are explicitly shown as a series of lines and arcs that encompass them. In many instances the boundary is coincident with the position to which connections may adhere themselves. The example in Figure 4-2 illustrates one exception to this general rule; inheritance connections penetrate the boundary of the Coad and Yourdon Class&Object symbol.

Text fields describe properties of the concept the symbol represents (such as class name and operations for the classes in Figure 4-2). Typically the height and width of symbols vary depending on the content of the text fields. Symbols may expand to contain other

things, such as lists of strings or other symbols. Many symbols are divided into compartments. A symbol often contains a field that represents a property related to the identity of the concept depicted. The class name fields in Figure 4-2 are an example of such a field.

Figure 4-3 shows three different UML class symbols. The overall size of the UML class symbol (height and width) is related to the size of the text it encloses.

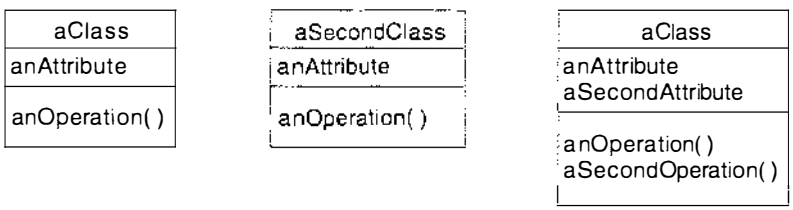


Figure 4-3 - Three examples of a UML class symbol

The size and position of each subpart of the UML class symbols in Figure 4-3 depends on the enclosed text, and on other parts of the symbol. For example the overall width of the symbol is related to the maximum length of the class name, attribute and operation compartments. The position of the class name text field is always centred in the symbol and is also related to the widths of the three text fields. The height of the symbol is related to the sum of the heights of the individual text fields. Figure 4-4 shows a topographical description of an UML class symbol based on these observations.

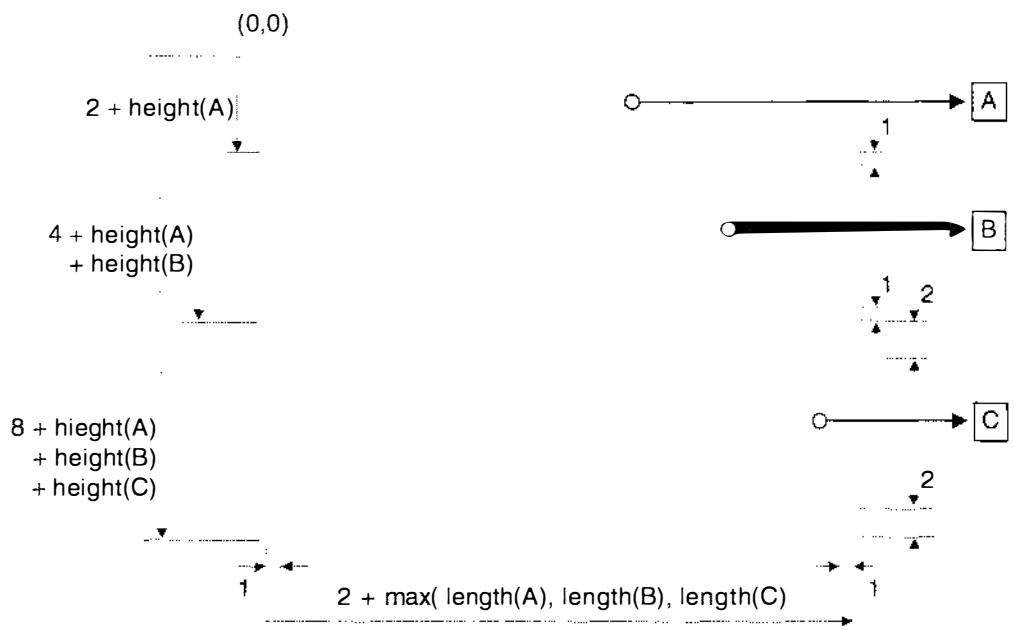


Figure 4-4 - Topographical description of a UML class

The start and endpoints of the line segments in Figure 4-4 are all functions of the sizes of text-fields A, B and C (which enclose the class name, the attributes and the operations respectively). For example the overall height and width of the symbol could be defined by the following expressions.

$$SymbolWidth = 2 + \max(length(A), length(B), length(C))$$

$$SymbolHeight = 8 + height(A) + height(B) + height(C)$$

Many symbols have sub-parts in common. In Coad and Yourdon the Class&Object symbol is the same as the Class symbol with an additional bounding round rectangle (the first two symbols in Figure 4-5). The Booch Class, Parameterised Class and Instantiated Class all have the Booch bubble in common (the last three symbols in Figure 4-5).

Figure 4-5 - Coad and Yourdon and Booch symbols showing common sub-parts

4.4.2 Connections

Figure 4-6 shows two example connections. The first is a Coad and Yourdon inheritance connection. This connection has a special symbol (the half circle) and consists of a series of recta-linear line segments. The second shows a transition connection from a Booch state transition diagram. It has an arrow-head at one terminus of the connection and it also has some associated text (the event-action pair for the transition).

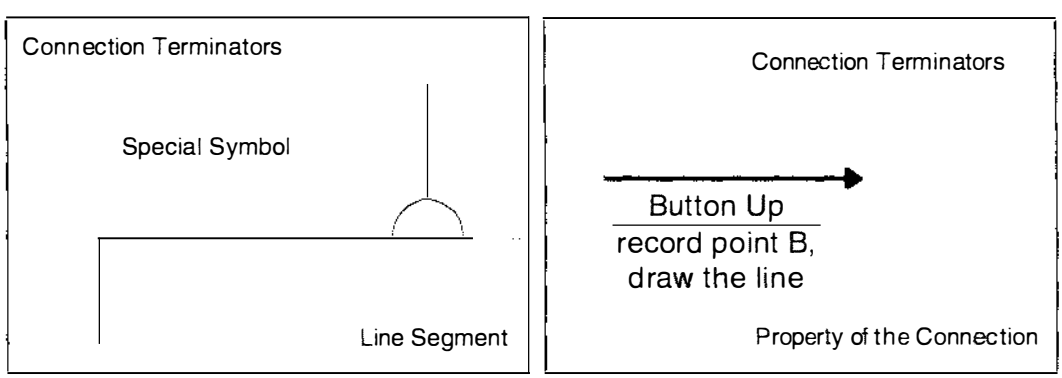


Figure 4-6 - Two example connections

Connections are a visual depiction of a relation between one or more concepts[29] and are a composition of three parts: connection terminators, an optional special connection symbol and a series of interconnecting line segments and arcs. Connections do not exist in isolation and must attach to at least one symbol. Text may be associated with a connection to define concepts such as cardinality or role names. In addition connections may have other symbols as annotations (for example the triangle in the whole-part and semi-circle in the inheritance relations in the Coad and Yourdon notation). This definition of connection does not constrain the manner in which a connection is constructed or drawn. UML has adopted a more restricted definition of connection than that described here. The UML notation guide states:

> "Paths[30] are sequences of line segments whose endpoints are attached. Conceptually a path is a single topological entity, although its segments may be manipulated graphically. A segment may not exist apart from its path. Paths are always attached to graphic symbols at both ends (no dangling lines). Paths may have *terminators*, that is, icons that appear in some sequence on the end of the path and that qualify the meaning of the path symbol."
>
> *From the UML Notation Guide, version 1.0 (Rational, 1997)*

The UML notation guide does not consider that a connection may have one or more floating endpoints. A start state, in Feylock's notation for example, has an incoming transition that does not originate from another state (see the example in Figure 4-1). It is better to state that a connection must be associated with at least one symbol. Furthermore the phrase 'Paths are always attached to graphic symbols at both ends' also implies connections can only occur between two symbols. Rumbaugh's ternary relation, for example, belies this assumption.

Some connections visually appear to be grouped in a diagram. Figure 4-7 shows an inheritance connection in UML. The single tree-like connection actually represents two

[29] A connection may express a relation that a concept has with itself.

[30] Path is the equivalent UML term for connection.

separate inheritance relations[31]. The Gen-Spec connection in the Coad and Yourdon notation is another example of 'grouping' connections.

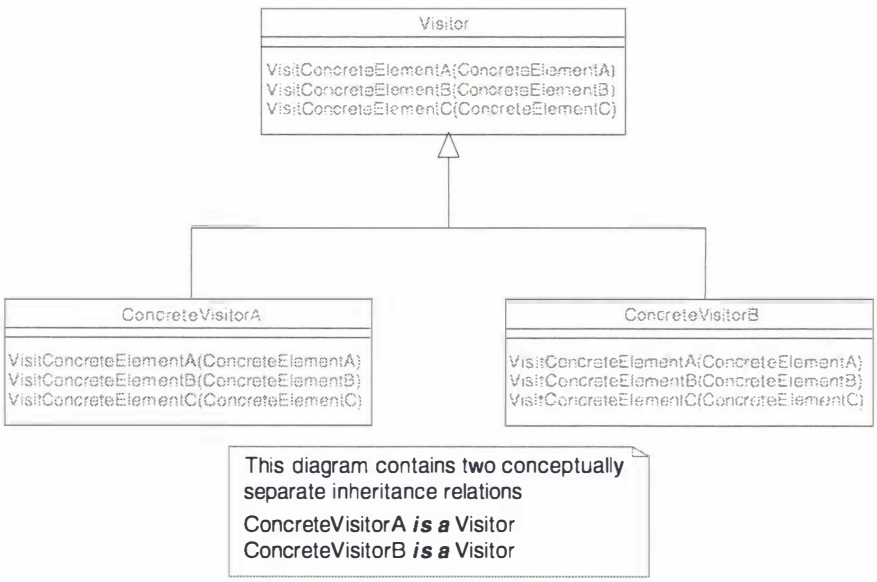


Figure 4-7 - Inheritance connection in UML

It is clear that the mapping between semantic relations and connections is not necessarily one-to-one. This example also emphasises that the appearance of a connection is a presentation issue only. The UML notation guide supports this view:

> "In some relationships (such as aggregation and generalisation) several paths of the same kind may connect to a single symbol. In some circumstances (described for the particular relationship) the line segments connected to the symbol can be combined into a single line segment, so that the path from that symbol branches into several paths in a kind of tree. This is purely a graphical presentation option; conceptually the individual paths are distinct."
>
> *From the UML Notation Guide, version 1.0 (Rational software, 1997)*

In many instances the orientation of a connection is constrained. Some notations support recta-linear line segments whilst others prefer smooth curves. Consider the UML sequence diagram in Figure 4-8. Connections in the UML sequence diagram are constrained to being horizontal only (except for the special case of a message to self). The message name and sequence number, as a group, is centred on the connection.

[31] UML does allow multiple, separate, inheritance connections as well.

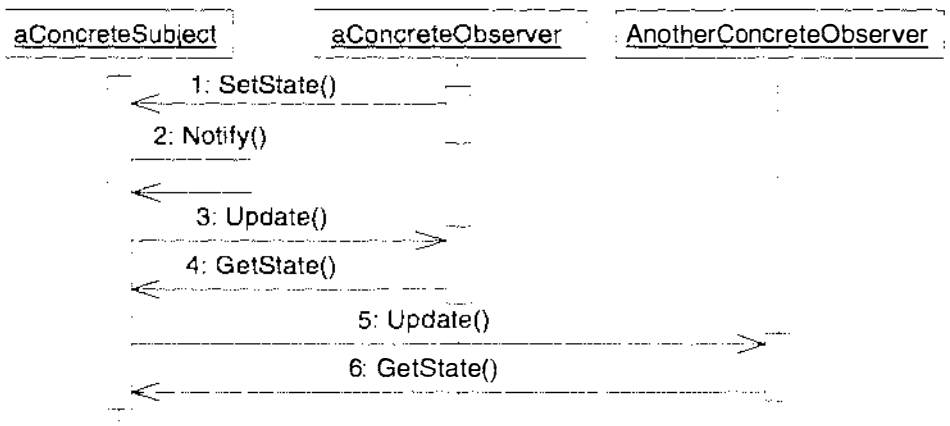


Figure 4-8 An example UML sequence diagram[32]

The sequence diagram is an interesting example where the coupling between syntax and semantics is very high, as the relative vertical position of the message invocations has meaning. This is discussed further in section 9.4.

4.4.3 Docking Areas

Many notations constrain the valid positions a connection may attach (or dock) itself to a symbol. The inheritance connection in Coad and Yourdon may only attach to the top and bottom of Class&Object symbols for example. In addition whole-part connections only attach to the sides of Class&Object symbols. The valid connection point (or docking area) between a connection and a symbol is therefore also part of the notation.

Connections to the Actor and Use Case symbols in Jacobson's OOSE methodology (Jacobson *et al.*, 1993) adhere to the boundary of the symbol. Figure 4-9 shows a simple Use-Case diagram with the boundaries of the symbols shown in grey. The Actor is also a prime example of a symbol whose boundary is not explicitly rendered.

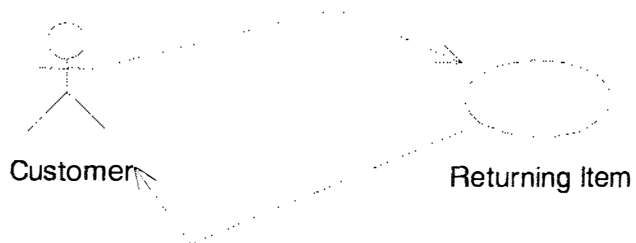


Figure 4-9 - A Jacobson Use Case diagram[33]

[32] The Sequence Diagram is taken from Gamma (Gamma *et al.*, 1995).

[33] The Use Case Diagram is a diagram fragment taken from (Jacobson *et al.*, 1993).

The UML notation guide states:

> "Paths are connected to two-dimensional symbols by terminating the path on the boundary of the symbol. Dragging or deleting a 2-d symbol affects its contents and any paths connected to it."
>
> *From the UML Notation Guide, version 1.0 (Rational, 1997a)*

It is clear that UML considers that the docking area coexists with the boundary of a symbol. Whilst this is typical of graphical notations it is not universal. The boundary of a symbol and the docking areas do not have to overlap.

Consider the composite pattern (Gamma *et al.*, 1995) drawn using the notation of Coad and Yourdon (Figure 4-10).

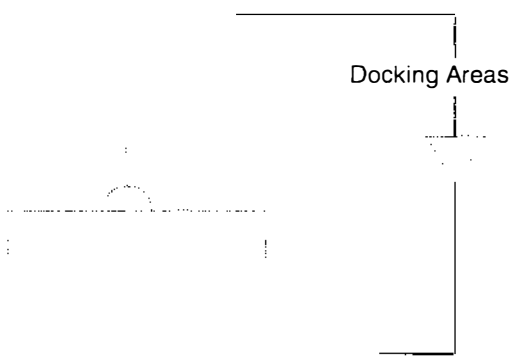


Figure 4-10 Docking areas on Coad and Yourdon Class&Object symbols

In the Coad and Yourdon notation whole-part connections (and instance connections) adhere to the outside round rectangle (which represents objects)[34]. The inheritance relation may only adhere to the innermost round rectangle (as inheritance is a relation among classes). Neither of these connections may adhere to the curved portion of the Class&Object symbol. Connections are always rectilinear and orthogonal in Coad and Yourdon.

[34] It may also adhere to the inner round rectangle, but only for an abstract class.

4.4.4 Groups

Symbols may also appear to be compositions of other symbols. These symbols are interesting as their shape and size depends on one or more other symbols (each of which is dependent on its own properties) as well as properties of their own. The subject area symbol in Coad and Yourdon is an excellent example. Figure 4-11 shows two possible states for a Coad and Yourdon subject area, in a diagram that describes the composite pattern (Gamma *et al.*, 1995).

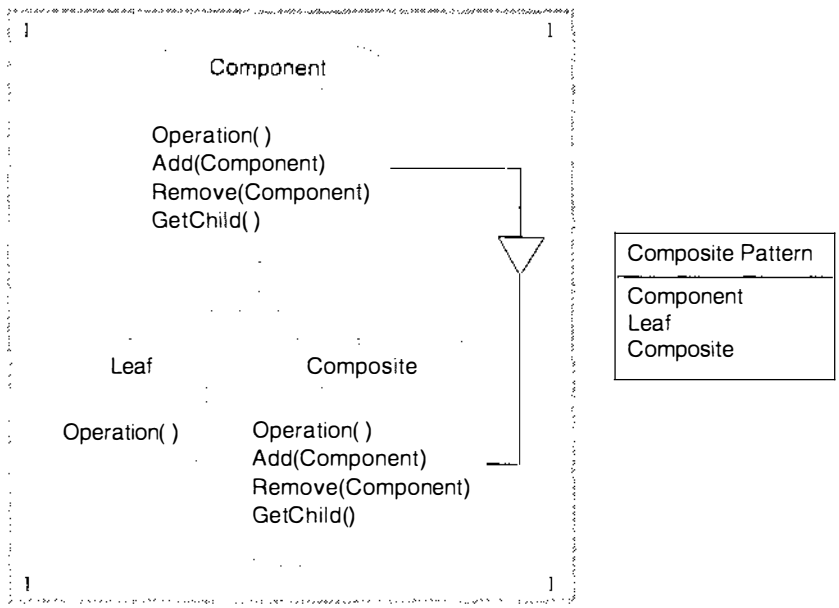


Figure 4-11 Coad and Yourdon Subject Area: expanded (left) and collapsed (right)

On the left-hand side of Figure 4-11 is a small class hierarchy surrounded by a grey border that delineates the subject area. The subject area may be collapsed into the single symbol shown on the right-hand side in Figure 4-11. The term used here for a notation element of this type is a composite symbol.

Composite symbols should not be confused with symbols that may be 'exploded' into another canvas or drawing surface. That is an example of a simple symbol that is linked to separate diagram or model. For example a UML package may be exploded into a separate class diagram. A process in a data flow diagram may be exploded into either a process specification or another data flow diagram. The various states of a composite symbol, such as the Coad and Yourdon subject area, must all appear on the same drawing surface.

4.4.5 Presentation

One of the common problems associated with any computer representation of complex data, is the relatively small window through which an information space can be viewed. CASE tools, which generally provide multiple orthogonal representations of a model being developed, share this problem. The small window effect gives rise to difficulties in locating a given item of information (navigation), in interpreting an item once it has been located, and in relating a given item to others, if that item cannot be seen in its full context.

A range of distortion-oriented presentation techniques have evolved to overcome some of these difficulties, (Apperley and Chester, 1995, Leung and Apperley, 1993, 1994; Leung *et al.*, 1995; Smith and Anderson, 1996). The common feature of these techniques is to allow a user to examine a local area in detail (e.g. a number of classes with their attributes and operations), whilst presenting a global view in order to provide an overall context and facilitate navigation.

Many CASE tools support distortion orientated presentation by allowing portions of a diagram or symbol to be elided (this is an example of logical distortion). For example Rational Rose allows various compartments of a symbol to be hidden (Figure 4-12).

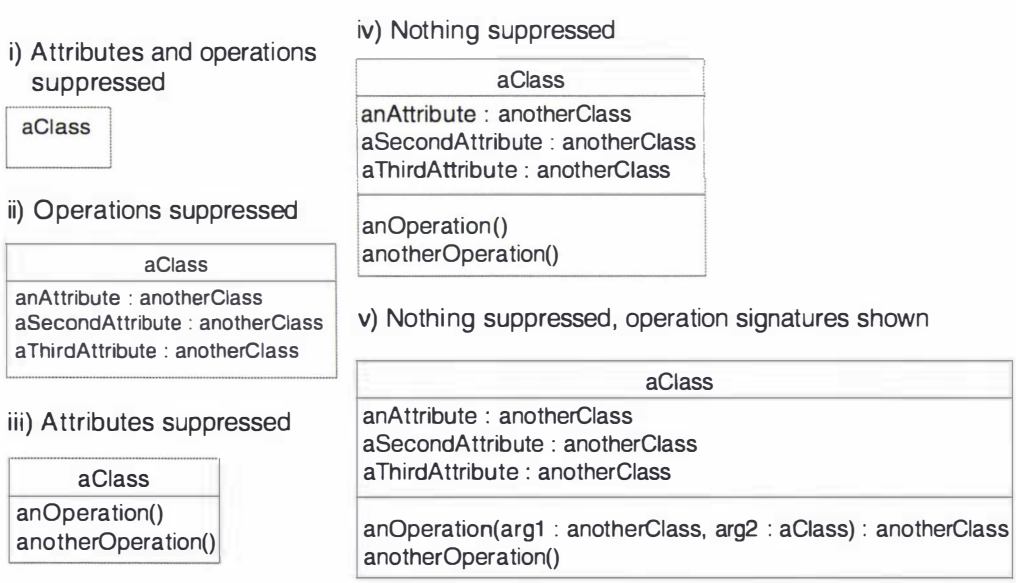


Figure 4-12 - A UML class expressed with varying levels of detail

Figure 4-12 shows the same UML class symbol in various levels of detail ranging from (i) attributes and operations being suppressed to (v) all compartments visible, with operation signatures displayed.

Facilities for logical distortion are not commonly part of the definition of the notation of a methodology. One notable exception to the rule is UML that includes a description of 'Presentation options' in the UML notation guide.

> "**Presentation options**: Describes various options in presenting the model information, such as the ability to suppress or filter information, alternate ways of showing things, and suggestions for alternate ways of showing information within a tool. Dynamic tools need the freedom to present information in various ways and we do not want to restrict this excessively. In some sense we are defining the 'paper notation' that printed documents show rather than the 'screen notation' ... Note that a tool is not supposed to pick one of the presentation options and implement it; tools should give the users the option of selecting among various presentation options, including some that are not described in this document."
>
> *From the UML Notation Guide, version 1.0 (Rational, 1997a)*

The explicit consideration of 'presentation options' by UML is a significant development in the evolution of CASE technology. It signals the recognition by methodologists of the importance of CASE support for software engineering methodologies, presentation and human computer interaction issues.

4.4.6 Actions

Actions correspond to the tasks a user performs at the user interface whilst developing a model. Some actions may affect the semantics of a model (such as deleting and editing) and some only the syntax (such as formatting, querying and resizing).

Methodologists do not consider the concept of actions. Their primary concern is to define the static pen-and-paper notation for their methodologies. The 'screen' notation, however, need not be static and can therefore include a description of 'hotspots' (or active areas) on the symbols and connections. Further it may be possible for a notation to

include a definition of actions related to these hotspots. Two possibilities are illustrated in Figure 4-13. In the first example the user has selected the text area, which causes an update box to appear. In the second example, selection of a hotspot causes the symbol to modify its appearance to show more information.

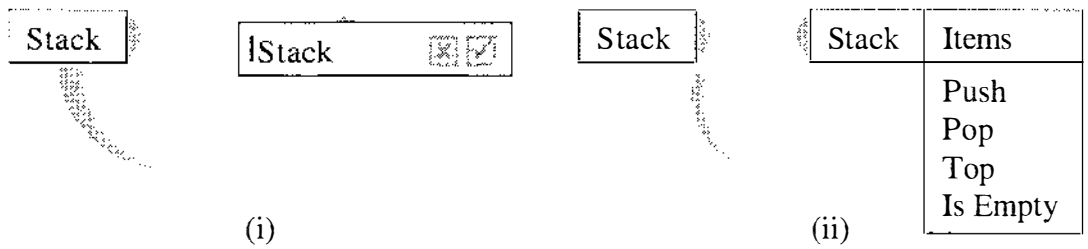


Figure 4-13 - Two example active areas

The use and visual appearance of hotspots is only limited by the imagination of the notation designer. Two primitive actions that must be supported are update actions (example (i) in Figure 4-13) and transition actions (example (ii) in Figure 4-13).

4.5 Notation Definition Language

One of the design philosophies of the MOOT methodology representation strategy is to separate the description of the syntax and semantics of a methodology as much as practical. NDL is only used to define the syntax of the modelling languages supported by a methodology. Similarly SSL is only used to define the semantics of the modelling languages supported by a methodology. The MOOT approach is to have a single 'editor' (the CASE tool client), which is dynamically parameterised by NDL notation descriptions of the syntax[35].

4.5.1 Requirements of NDL

NDL must provide the necessary facilities to describe how symbols and connections may be rendered and manipulated to describe the types of notations discussed thus far. NDL must support the 'screen' notation by supporting:

1. Graphical primitives such as lines, arcs, text, regions, fill and pattern styles, fonts and font size, colour. These are the building blocks of symbols and connections

[35] This can be contrasted to systems such as DiaGen (Minas and Viehstaedt, 1995), MutiView (Marlin *et al.*, 1993; Marlin, 1996; MultiView, 1998; Read and Marlin, 1996, 1998) and BuildByWire (Mugridge *et al.*, 1998; Warwick *et al.*, 1996) that focus on generating bespoke editors for a particular notation.

2. Relations between subparts of a notation element. For example the size and position of a line may depend on the length of one or more text fields
3. Grouping. This allows shapes common to several symbols and/or connections to be defined once
4. Connection docking areas and annotations
5. Logical distortion
6. Hotspots/Active areas
7. Transition and update actions

4.5.2 Design of NDL

A template strategy has been designed to support the elements of a notation. The NDL templates are blueprints for creating notation elements. NDL provides template types that correspond to each type of notation element. For example, symbols are defined with symbol templates, connections with connection templates and lines with line templates. Figure 4-14 presents a UML class symbol and illustrates how NDL templates represent the corresponding notation elements.

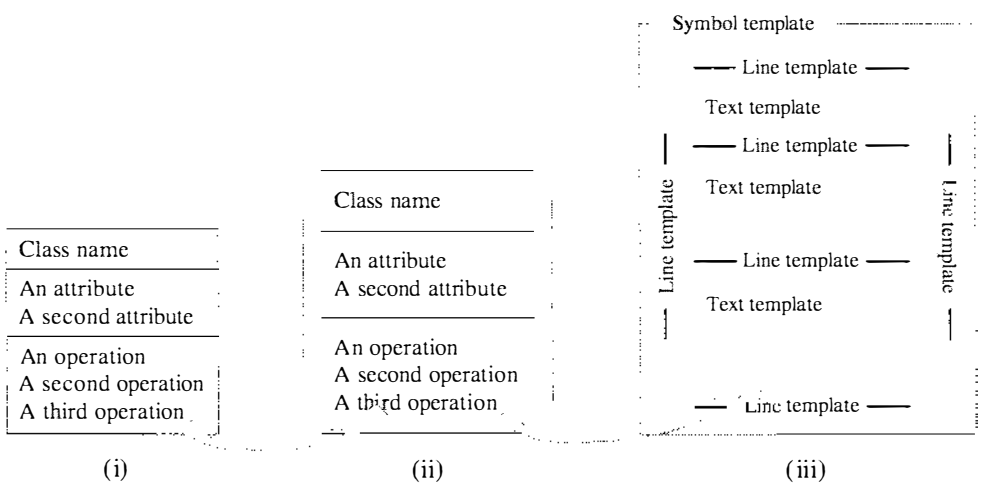


Figure 4-14 - (i) A symbol (ii) exploded Symbol (iii) templates

Figure 4-14 (ii) shows the notation elements that comprise the UML class symbol in Figure 4-14 (i). Figure 4-14 (iii) shows the one-to-one mapping between templates and the notation elements that they describe. In addition Figure 4-14 demonstrates that the composition of templates in a symbol template parallels the composition of notation elements in a symbol.

A complete NDL description of a notation consists of a collection of templates to define the symbols and connections of that notation. The example in Figure 4-15 illustrates how a symbol template is applied to generate a symbol in an arbitrary notation.

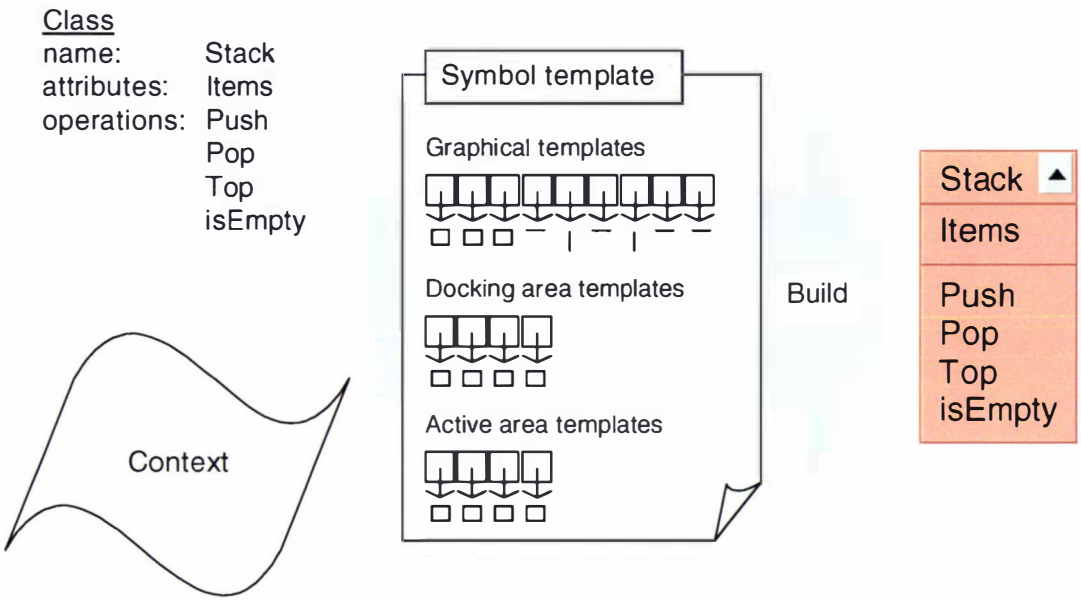


Figure 4-15 - Applying a template

The symbol template in Figure 4-15 describes a class symbol. In this example the symbol template creates a symbol when it is provided the properties of an instance of a concept (a class in this example) and a context. The size and position of the lines, text fields, hotspots and docking areas depend on the properties of the concept. In addition the generated symbol is also dependent on the context within which it is to be rendered (a Macintosh, or a system running X-windows for example). In general a template produces a notation element when provided a concept and a context.

The context in Figure 4-15 is an abstraction of the environment within which symbols and connections are rendered and acts as an interface between a notation specification and a drawing surface. It abstracts the dependency between the underlying graphical system (e.g. the Macintosh toolbox, or X-windows) and the notation specification. One of the responsibilities of the context, for example, is calculating the size, in picture elements, of a string of characters.

Each notation defines a set of identifiers (NDL ID) that correspond to the properties of concepts that are needed in the notation. In the example in Figure 4-15 the identifiers are name, attributes and operations. These identifiers only exist within the context of the

notation and in no way constrain the way in which these properties are represented in SSL (Semantic Specification Language). Mapping NDL identifiers to the values of SSL properties is a responsibility of the NSM table and is discussed in detail in chapter 6.

The advantage of this design is twofold:

1. The description of notation elements is separated from the properties of the semantic concepts they represent.
2. The description of notation elements is separated from the environment within which the symbols and connections are rendered.

Currently a minimal set of graphical primitives (lines, arcs and text boxes) is supported by NDL. This minimal subset has been chosen, as it is sufficient for constructing notation elements and determining the efficacy of the proposed approach to defining the syntax of a methodology.

4.5.3 Describing Symbols in NDL

A UML class symbol will be defined in NDL to assist illustrate how NDL is used to define symbols in general. See appendix II for a complete definition of NDL syntax.

Figure 4-16 shows a topographical description of a UML class symbol. The symbol has three fields (A, B and C) where text may appear. The height and width of these text fields depend on the text they contain.

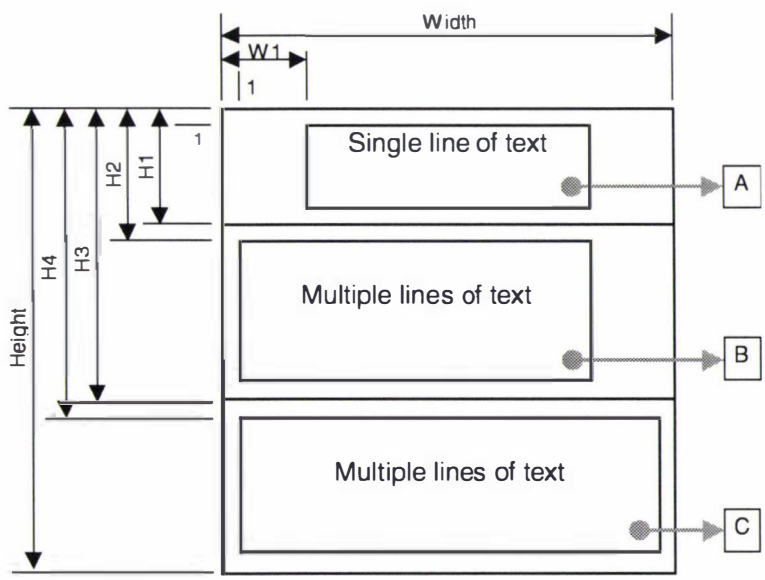


Figure 4-16 - Topographical description of a UML class symbol

Each text box in Figure 4-16 has two properties, a height and a width. NDL provides two functions (*height* and *width*) that are used to dynamically determine the size of a text field. The value of *H1*, *H2*, *H3*, *H4* and *Height* can be calculated by using the *height* and *width* functions. All NDL expressions are in reverse polish.

```
H1 = + 1 height( A )
H2 = + 1 H1
H3 = + + 1 H2 height( B )
H4 = + 1 H3
Height = + + 1 H4 height( C )
```

The overall width of the symbol in Figure 4-16 is dependent on the widths of all three text fields. NDL supports a function, *max*, which returns the maximum value of its arguments. The width of the symbol in Figure 4-16 can be calculated in the following way:

```
Width = + 2 max( width( A ), width( B ), width( C ) )
```

The position of the class name text field in Figure 4-16 can be derived from the overall width of the symbol and the width of the class name field in the following way:

```
W1 = div - Width width( A ) 2
```

So far this example has shown that the basic arithmetic operations (add, subtract, multiply and divide) are all supported by NDL. It has shown that two functions (*height* and *width*) are used to represent the dynamic properties of text fields. It has also demonstrated that the *max* function is used to capture relations amongst sub-parts of a symbol.

The UML symbol in Figure 4-16 is comprised of three lines and three text boxes. Two of the lines separate the various compartments and the third is a poly-line that represents the boundary of the symbol. NDL supports statements that correspond to each type of primitive notation element. These are represented in NDL in the following way:

```
LINE (0,0)(Width,0)(Width,Height)(0,Height)(0,0)
LINE (0,H1)(Width,H1)
LINE (0,H3)(Width,H3)
TEXT A (W1,1)
LISTTEXT B (1,H2)
LISTTEXT C (1,H4)
```

This example shows the *line*, *text* and *listtext* statements respectively[36]. Lines must define at least two points (the first line in this example defines five). The text statement defines a text field that may only contain a single line of text. The listtext statement defines a text field that may contain multiple lines of text. Both the text and listtext statements introduce an NDL ID for that field that may be used elsewhere to reference that field.

A symbol may also define one or more hotspots or active areas. Figure 4-17 is an extended topographical description of the UML class symbol of Figure 4-16, with four active areas.

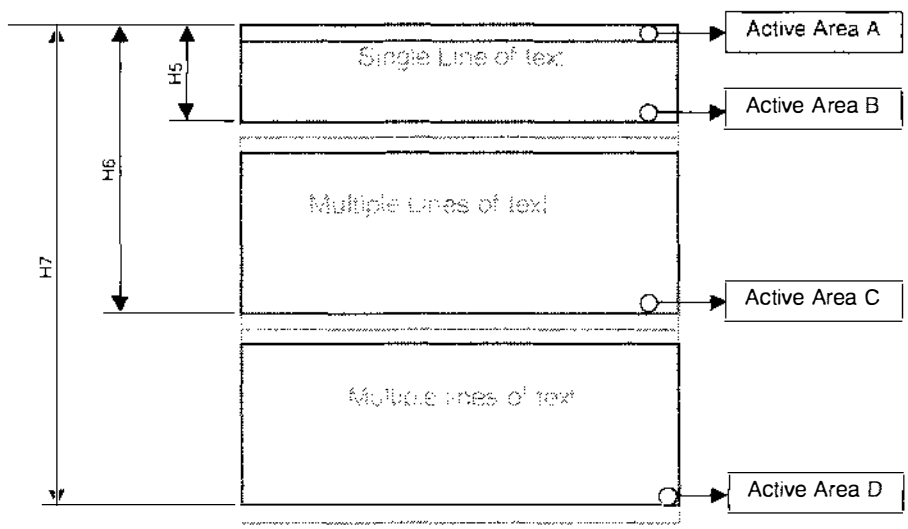


Figure 4-17 – A UML class symbol with active areas

Two default actions, that may be associated with an active area, are supported by NDL. A *Transition* action specifies that the symbol should be rendered with another template (where the symbol shows less information, for example). An *Update* action specifies that a change has been requested of one more text areas. These four areas are defined in NDL in the following way:

```
ACTIVE AREA (0,0)(Width,1) TRANSITION TO <TargetTemplate>
ACTIVE AREA (0,1)(Width,H5) UPDATE A
ACTIVE AREA (0,H2)(Width,H6) UPDATE B
ACTIVE AREA (0,H4)(Width,H7) UPDATE C
```

The steps that are carried out in response to an action are the responsibility of the user interface, not the notation. The notation only defines where actions are generated. Active

[36] NDL also supports an arc statement.

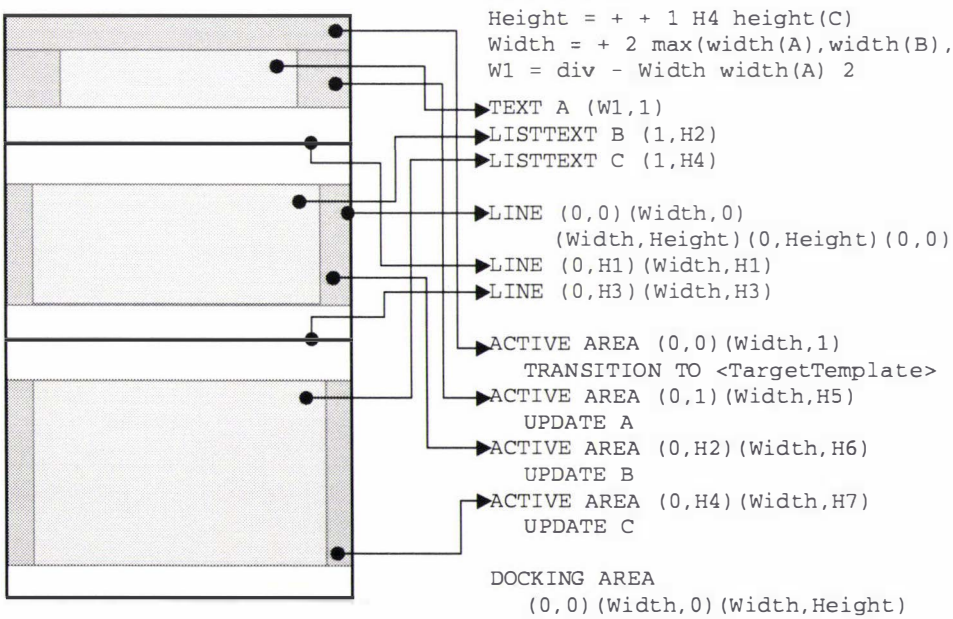
areas are currently defined as rectangles for the purpose of the initial research. In general an active area will be defined as a region.

A symbol may also define one or more docking areas. A docking area defines a place on a symbol that connections may attach themselves to. The UML class symbol of Figure 4-16 and Figure 4-17 may accept connections at any point around its perimeter. A single docking area is therefore sufficient to define the UML class symbol. A docking area is defined in NDL in the following way[37]:

```
LINE DA (0,0)(Width,0)(Width,Height)(0,Height)(0,0)
```

This docking area defines a poly-line that is coincident with the boundary of the UML class symbol. NDL actually supports three types of docking area. Each will be discussed in more detail in section 4.5.5 - Docking Areas.

A complete template that describes a UML class symbol is given in Figure 4-18.



```
Symbol_Template UML_CLASS_SYMBOL
{
  H1 = + 1 height(A)
  H2 = + 1 H1
  H3 = + + 1 H2 height(B)
  H4 = + 1 H3
  Height = + + 1 H4 height(C)
  Width = + 2 max(width(A),width(B),width(C))
  W1 = div - Width width(A) 2
  TEXT A (W1,1)
  LISTTEXT B (1,H2)
  LISTTEXT C (1,H4)

  LINE (0,0)(Width,0)
       (Width,Height)(0,Height)(0,0)
  LINE (0,H1)(Width,H1)
  LINE (0,H3)(Width,H3)

  ACTIVE AREA (0,0)(Width,1)
     TRANSITION TO <TargetTemplate>
  ACTIVE AREA (0,1)(Width,H5)
     UPDATE A
  ACTIVE AREA (0,H2)(Width,H6)
     UPDATE B
  ACTIVE AREA (0,H4)(Width,H7)
     UPDATE C

  DOCKING AREA
     (0,0)(Width,0)(Width,Height)
     (0,Height)(0,0)
}
```

Figure 4-18 - Template describing a UML class symbol

[37] This docking area template has been simplified for the sake of discussion. A complete description of docking area templates in made in 4.5.5 - Docking Areas.

A, B and C are NDL IDs that are unique within the context of the notations as is the name given to the template. The arrows show the mapping between notation elements and the templates that define them.

4.5.4 Support for Grouping

NDL provides group templates to support the definition of common sub-components of symbols and connections. Group templates are used to define icons that may appear on connections or as annotations on a symbol. Consider the pseudo Coad and Yourdon notation of Figure 4-19 where there are two versions of the class and Class&Object symbols.

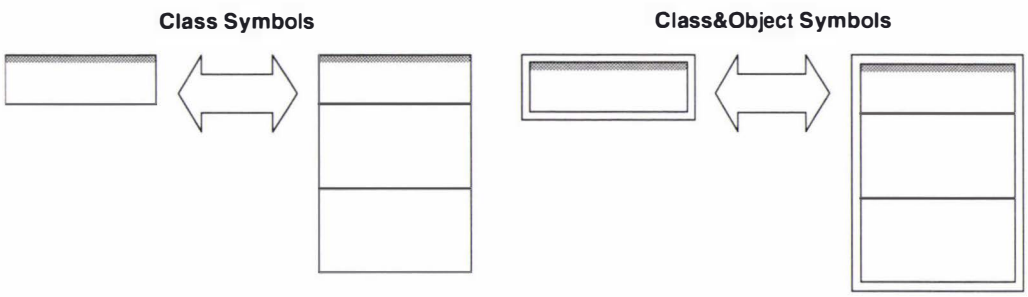


Figure 4-19 - Coad and Yourdon class and Class&Object symbols

The two versions of each symbol show different levels of detail. One only shows the class name whilst the other shows all three compartments. They grey area at the top of each symbol is an active area which causes a transition from one form of the symbol to the other. It would be possible to create this notation in NDL with four completely separate symbol templates. This would, however exhibit some redundancy in the descriptions. Figure 4-20 shows the pseudo Coad and Yourdon notation of Figure 4-19 with the identification of common sub-parts. An NDL definition of these symbols can be simply achieved by using NDL group templates.

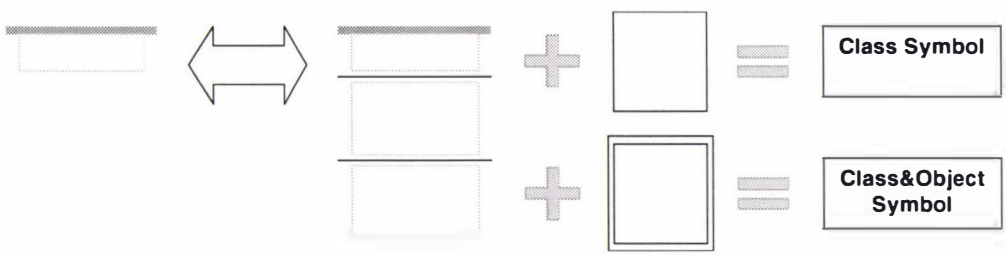


Figure 4-20 - Identified common sub-parts in Coad and Yourdon's notation

Group templates are defined in exactly the same way as symbol templates except they may not define docking areas. Figure 4-21 shows NDL definitions of two group templates complete with active areas.

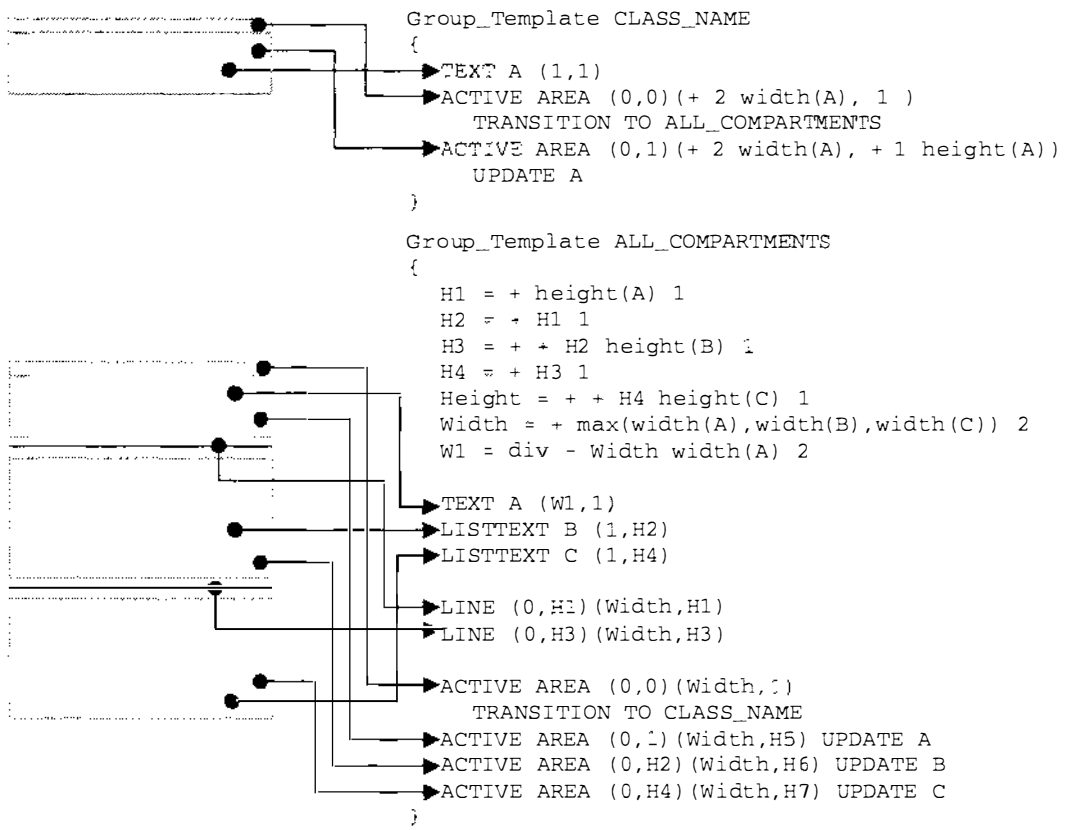


Figure 4-21 Group templates

The group templates in Figure 4-21 can be used to create the symbols in Figure 4-19. Figure 4-22 shows an NDL definition of the Coad and Yourdon class symbol that uses the group templates in Figure 4-21.

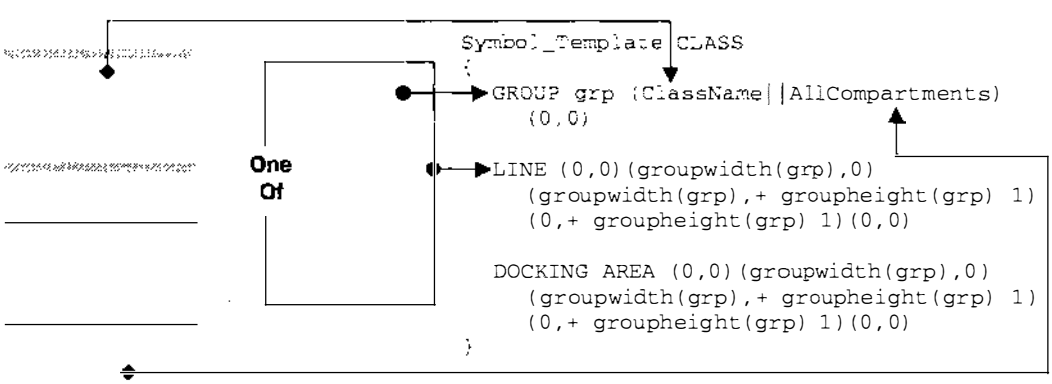


Figure 4-22 - Coad and Yourdon class symbol

Symbol templates can additionally contain a group reference statement. Each group reference statement contains a list of all possible group templates that may constitute its state. The first group template in the list is the default.

The class symbol template in Figure 4-22 consists of a single group (*grp*) and a line for the border. The first template listed in the group reference statement of Figure 4-22 is the *ClassName* group template. This means that a new Coad and Yourdon class symbol would only show the class name field when it is initially created. If a notation designer wished all three compartments to be visible by default they would place the *AllCompartments* group template first in the list.

The docking area in Figure 4-22 is co-incident with the border as in the previous examples. Both the border and the docking area are defined using the *groupheight* and *groupwidth* functions. These functions are used to dynamically determine the size of a group reference. They imply a rectangular boundary around all groups, which may be unnecessarily restrictive. The empirical evidence gained from using NDL has not proven this to be so.

4.5.5 Docking Areas

Docking areas represent the positions on a symbol that a connection can attach itself to. NDL supports three docking areas of different shape. These are point, line and arc docking areas.

All docking areas allow the following to be constrained:

- The number of permissible connections.
- The types of connection that may be attached.
- The direction connections can approach from, in order to attach.

The last constraint is implemented by defining what is 'inside' and what is 'outside' of a docking area. The purpose of this constraint is to avoid a connection crossing the interior of a symbol, to attach at a docking area.

Point Docking Area

A point docking area represents a single point on a symbol that can accept a connection. Figure 4-23 shows two examples of connections attached at a point on a symbol.

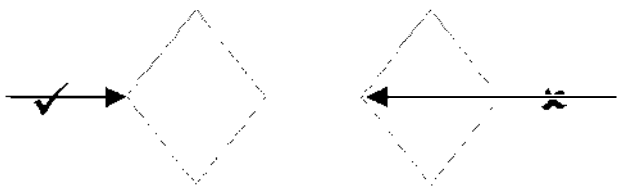


Figure 4-23 - Docking at a point

The example in the left-hand side of Figure 4-23 shows a desirable connection. The example in the right-hand side of Figure 4-23 shows an undesirable connection that crosses the symbol. Figure 4-24 shows a symbol that has a point docking area on the top-left corner.

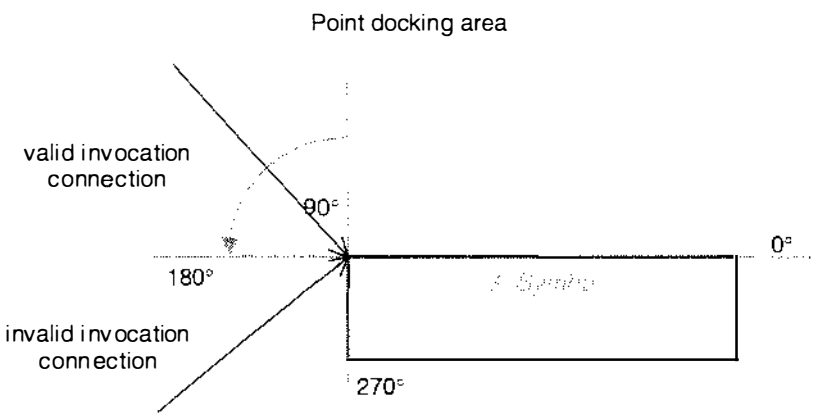


Figure 4-24 - Anatomy of a point docking area

A fragment of NDL code that could be used to describe the docking area in Figure 4-24 is given below.

```
POINT DA (0,0) 1 CONNECTION ARC(90,90)(invocation)
```

The first property of a point docking area is its position. In this example (0,0) coincides with the top-left corner of the symbol. The next property specifies the maximum number of connections that may be attached to this docking area. In this example only a single connection may be attached at a time. A value of *u* is used to specify that any number of connections may attach at this point. The next property defines the connection arc, through which all valid connections must pass. A connection arc is specified as a start angle – extent pair. The co-ordinate system for connection arcs is shown in Figure 4-24.

The final property is a list of connection types that may attach at this point. Each element in the list is the name of a connection template (connection templates are discussed in section 4.5.6). If this list is empty then any connection may attach at this point. In this example only connections of type *invocation* may attach at this docking point.

Line Docking Area

A line docking area represents a line on a symbol that can accept connections. Figure 4-25 shows a symbol with two connections that have attached along its left-hand side.

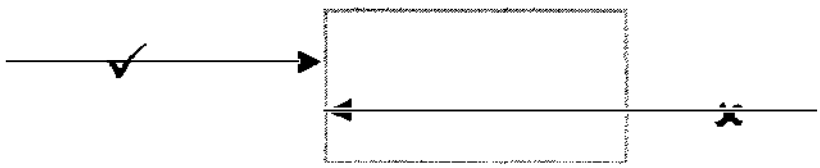


Figure 4-25 - Docking on a line

The connection coming from the left in Figure 4-25 is a desirable connection. The connection coming from the right in Figure 4-25 is undesirable as it crosses the symbol. Figure 4-26 shows a symbol with a line docking area on it left-hand side.

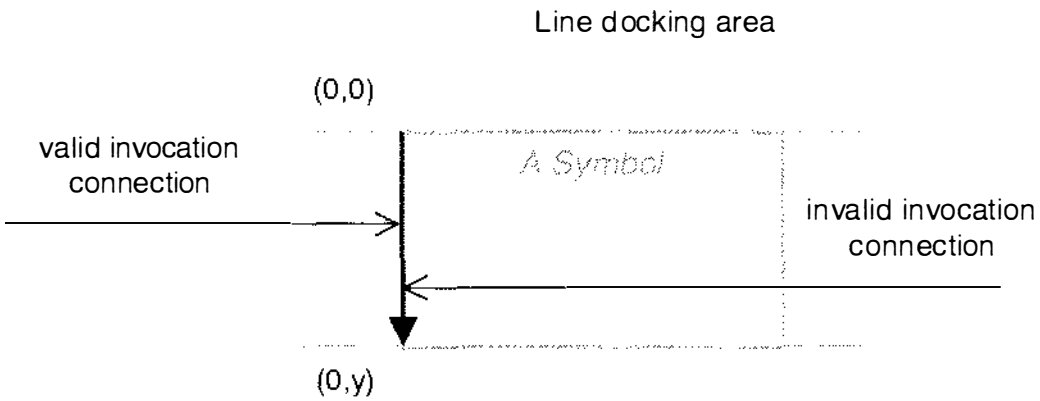


Figure 4-26 - Anatomy of a line docking area

A fragment of NDL code that describes the docking area in Figure 4-26 is given below.

```
LINE DA (0,0)(0,y) u 5 (invocation)
```

The first property of a line docking area is its position, which is represented by a series of points. In this example the line docking area coincides with the left-hand side of the symbol. The direction of the line docking area is used to encode the valid direction from which a connection may come to attach to the docking area. Figure 4-27 shows how this is achieved.

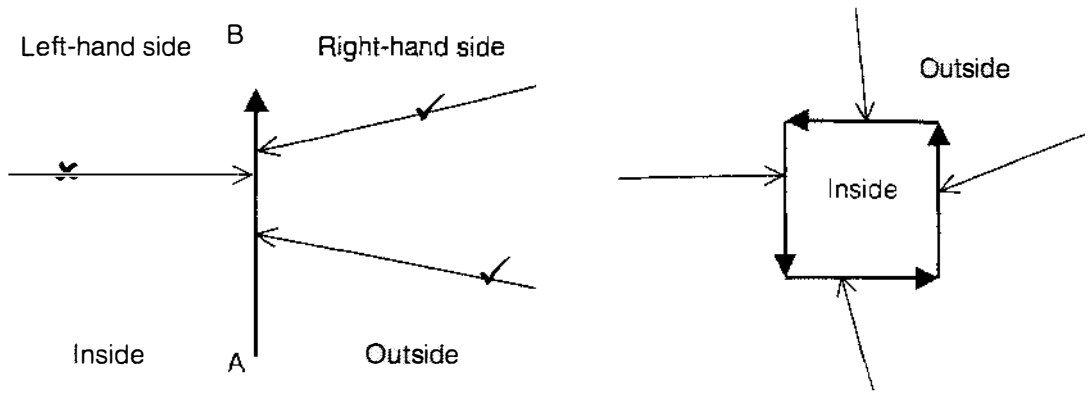


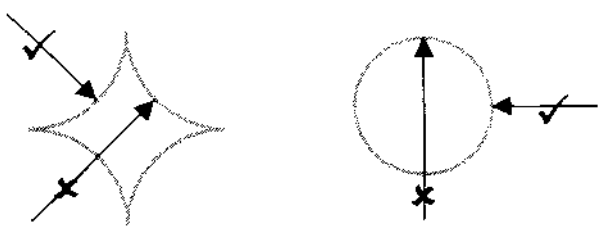
Figure 4-27 - Representing valid directions for a line docking area

The right-hand side of line segment *AB*, as viewed looking along the line from *A* to *B*, is taken as the outside of the line (Figure 4-27). The left-hand side of the line (and the line itself) is taken as the inside of the side. Only connections that approach from the *outside* of a line docking area may be attached. The right-hand side of Figure 4-27 shows how this property of line docking areas can be used to approximate the inside and outside of a symbol[38].

The next property of a line docking area specifies the maximum number of connections that may be attached. The value *u*, in this example, means that the number of connections is unconstrained. The next property specifies a minimum inter-connection distance. The final property is a list of connection types that may attach to the line docking area. In this example only connections of type *invocation* may attach at this line docking area.

Arc Docking Area

An arc docking area describes a curve along which connections may attach themselves. Figure 4-28 shows some examples of how an arc might be used as a docking area.



(i) Using the concave side (ii) Using the convex side

Figure 4-28 - Docking on an arc

[38] It is only an approximation because the sum of the outside regions of all the line docking areas could intersect with the interior of a symbol that has concavities

The connections that cross over the symbols in Figure 4-28 are undesirable. Figure 4-29 shows how an arc docking area can be used. This particular example shows a single arc docking area that is coincident with the boundary of a circular symbol.

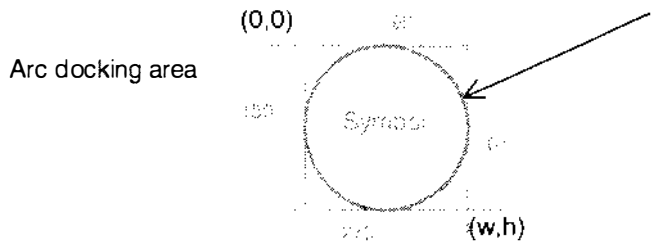
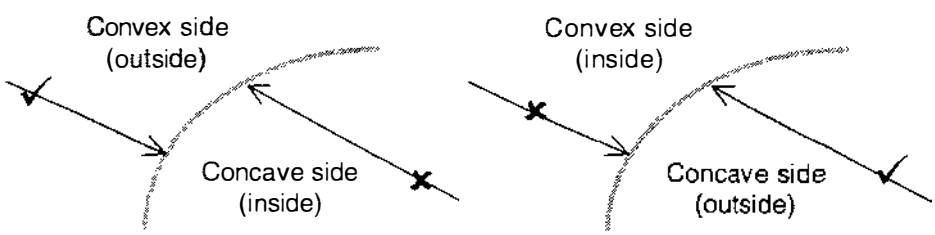


Figure 4-29 Anatomy of an arc docking area

A fragment of NDL code that could be used to describe the docking area in Figure 4-29 is given below.

```
ARC DA (0,0)(w,h) (0,360) CONVEX u 5 (invocation)
```

The first property of an Arc docking area defines its position and shape. In this example the arc is bounded by a box from (0,0) to (w,h). The shape of the arc is defined by a start angle – extent pair. In this example (0,360) describes a circle. The next property is used to define the direction connections may approach the arc and attach themselves. Figure 4-30 shows how the two possible values for this property (*Convex* and *Concave*) define the inside and outside of an arc docking area.



(i) 'Convex' Arc docking area (ii) 'Concave' Arc docking area

Figure 4-30 - Representing valid directions for an arc docking area

A value of 'convex' means that the convex side of the arc is to be interpreted as its outside (example (i) in Figure 4-30). A value if 'concave' means that the concave side of the arc is to be interpreted as its outside (example (ii) in Figure 4-30).

The next two properties of an arc docking area specify the maximum number of connections that may be attached and a minimum inter-connection distance. The final property is a list of connection types that may attach to the arc docking area.

4.5.6 Describing Connections in NDL

Connections in NDL are composed of a single, optional, connection symbol template and a collection of connection terminator templates. Each type of template will be discussed in turn, followed by the definition of an NDL connection template.

Figure 4-31 shows an example of Coad and Yourdon's notation with a Gen-Spec connection and a message connection. Both types of connection will be defined in NDL to assist illustrate how NDL is used to define connections in general.

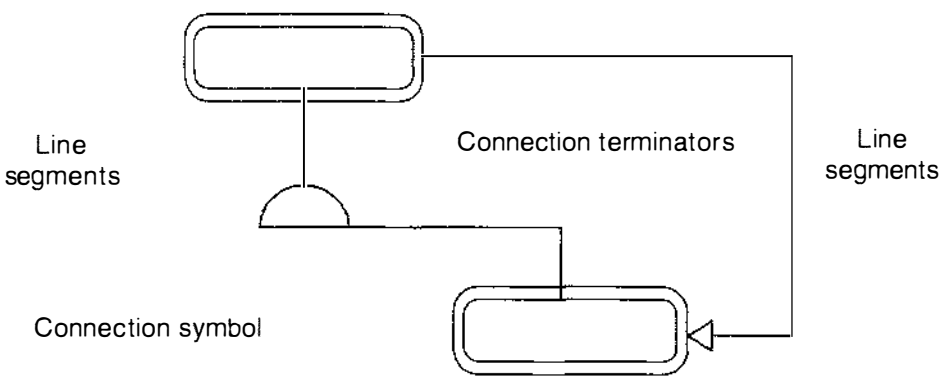


Figure 4-31 Two example connections

A Coad and Yourdon Gen-Spec connection consists of two connection terminators, a special connection symbol and some line segments (Figure 4-31). A message connection consists of two connection terminators (one of which is an arrow head) and a collection of line segments.

Connection Symbols

An NDL connection symbol template for the Coad and Yourdon Gen-Spec connection symbol is given in Figure 4-32.

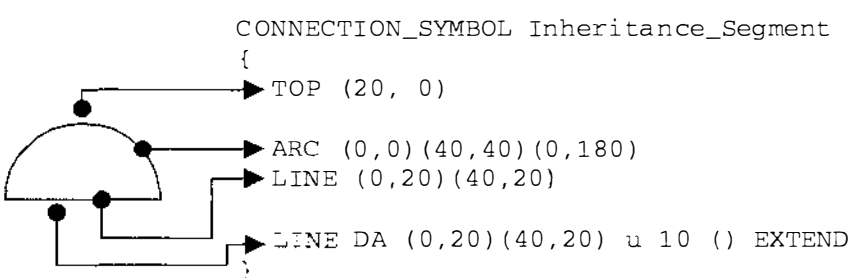
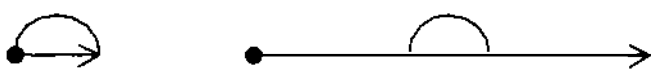


Figure 4-32 - Connection symbol template

A connection symbol has one incoming line segment. In the case of a Coad and Yourdon Gen-Spec connection this is from a super-class. It may have several outgoing line segments. In the case of a Coad and Yourdon Gen-Spec connection these lines go towards sub-classes. The first statement of the connection symbol template specifies where the connection symbol attaches itself to the incoming line segment. The arc and line template statements define the shape of the connection symbol. The last statement of the connection symbol template is a line docking area from which all the outgoing line segments start. A line docking area for a connection symbol has an additional property that defines whether the line docking area is permitted to change its length, to support more line segments. Figure 4-33 illustrates this property with two examples.



(i) (ii)

Figure 4-33 - Coad and Yourdon connection symbol line docking area (i) with a single connection (ii) with multiple connections

The connection symbol line docking area is shown by an arrow on the inheritance connection symbol in Figure 4-33. In the example on the left-hand side of in Figure 4-33 the line docking area is big enough to maintain a single connection. In the example on the right-hand of in Figure 4-33 side the line docking area has been extended past the boundaries of the connection symbol. Connection symbol line docking areas will automatically extend in this way if the last property of the line docking area has the value *Extend*.

Connection Terminators

Figure 4-34 shows NDL connection terminator templates for the different types of connection terminator in the Coad and Yourdon Gen-Spec and message connections.

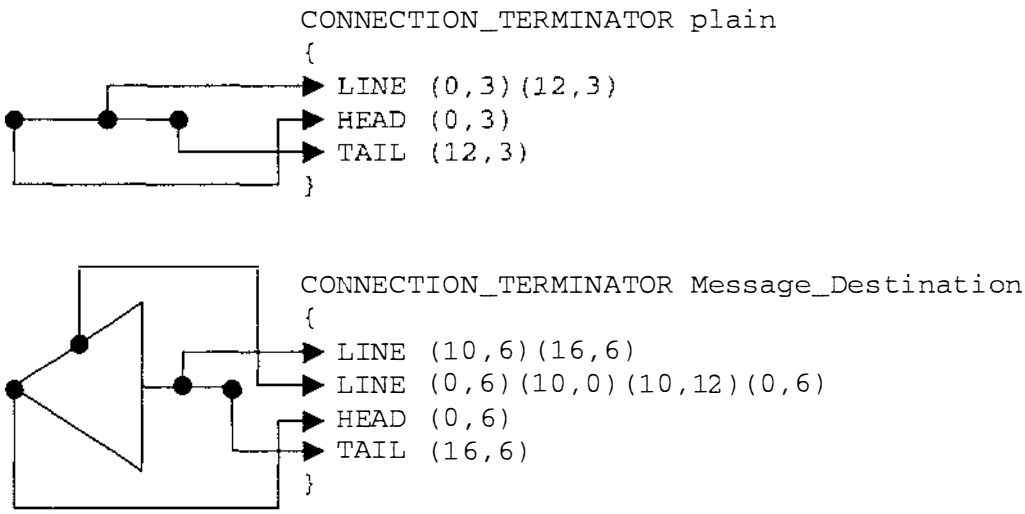


Figure 4-34 - Connection terminator templates for Coad and Yourdon Gen-Spec and message connections

A connection terminator consists of a collection of primitive template segments (in this example lines and arcs). The last two statements define the head and tail positions on the connection terminator. The head position is where the terminator will attach to a docking area. The tail position is where the terminator attaches to a line segment.

Connection Template

Each connection in a notation is defined by a separate connection template. A connection template specifies the arity[39] of the connection, an optional connection symbol template and a list of terminator templates. NDL connection templates that implement Coad and Yourdon Gen-Spec and message connections are given in Figure 4-35.

```
CONNECTION_TEMPLATE
Inheritance_Connection
{
  ARITY 2
  CONNECTION_SYMBOL
    Inheritance_Segment
  TERMINATOR (plain,plain)
}
```

```
CONNECTION_TEMPLATE
Message_Connection
{
  ARITY 2
  TERMINATOR
    (plain,Message_Destination)
}
```

Figure 4-35 - NDL connection templates for Coad and Yourdon Gen-Spec and message connections.

The order of the terminator templates in the terminator list corresponds to the order with which the portions of the connection are created. The message connection template in

[39] The arity of a connection specifies the number of symbols that may be involved in the connection.

Figure 4-35, for example, specifies the sequence *plain* followed by *Message_Destination*. This means that a *plain* terminator is used at the beginning of a message connection and a *Message_Destination* terminator is used at the end of a message connection.

4.6 NDL Interpreter

An NDL interpreter has been built to verify that the approach taken with NDL is efficacious. Each notation is written in NDL and stored in a file. The grammar of NDL is presented in appendix II. The interpreter is a simple drawing tool that allows a user to place symbols and connections that are defined in a notation file. A high level architecture is given in Figure 4-36.

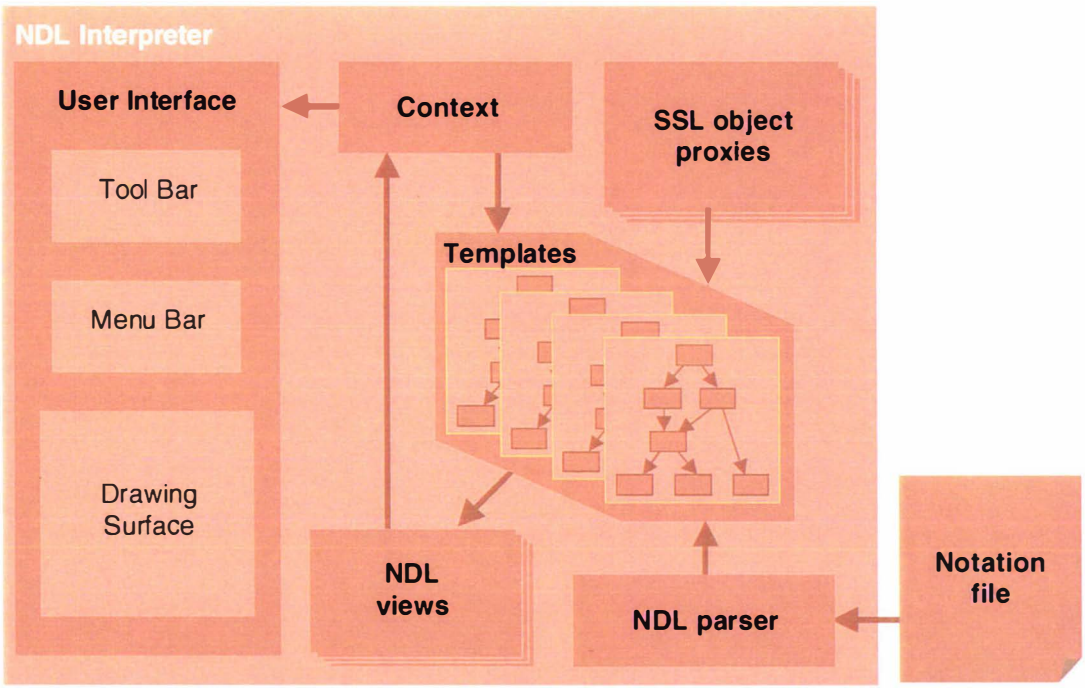


Figure 4-36 - Components of the NDL interpreter

The NDL interpreter parses a notation file and builds an abstract syntax tree (which is an instance of the Composite pattern (Gamma *et al.*, 1995)) for each *Template*. These abstract syntax trees are also instances of the Interpreter pattern as they can execute themselves. Each *Template* requires an *SSL object proxy* and a *Context* to generate its corresponding notation element.

An *SSL object proxy* is a stub that takes the place of the MOOT server in this prototype. Each proxy encapsulates a map of NDL identifiers and properties. For example the map in an *SSL object proxy* representing a class would contain three elements, which would

correspond to the NDL identifiers '*classname*', '*attributes*' and '*operations*'. SSL object proxies perform no semantic validation.

The *NDL views* in Figure 4-36 are the symbols and connections that have been generated from templates. An *NDL view* is a collection of primitive notation elements (such as lines, active areas and text fields). *NDL views* know how to draw themselves on a drawing surface with the assistance of a *Context* object.

The *Context* is an instance of the Visitor pattern (Gamma *et al.*, 1995). It hides the properties of the drawing surface (for example how long a string actually is, in drawing units, on the drawing surface) from the interpreting mechanism. *Templates* use this behaviour when generating notation elements. The *Context* also provides facilities for drawing primitive notation elements on a drawing surface.

The NDL interpreter as a whole can be ported to a different windowing interface environment by updating the specific interface elements (windows toolbars etc) and the *Context*. The *Context* is implemented as an abstract super-class, which defines the interface needed by *Templates* and *NDL views*. Implementations of the interpreter specialise *Context* as appropriate.

The user interface is implemented in tcl, the notation file in NDL and the rest of the components in C++. The CASE tool client (Figure 3-11 - Architecture of the MOOT prototype) is based on the design of the NDL interpreter.

4.7 Design of the NDL Interpreter

4.7.1 Representing Expressions

All templates are defined in terms of a series of expressions (see Figure 4-4, Figure 4-16, Figure 4-17). The types of expression supported in NDL include:

- arithmetic expressions and numerical constants
- maximum and minimum function
- height and length functions
- groupheight and grouplength functions

The different types of expression are represented in the inheritance hierarchy given in Figure 4-37.

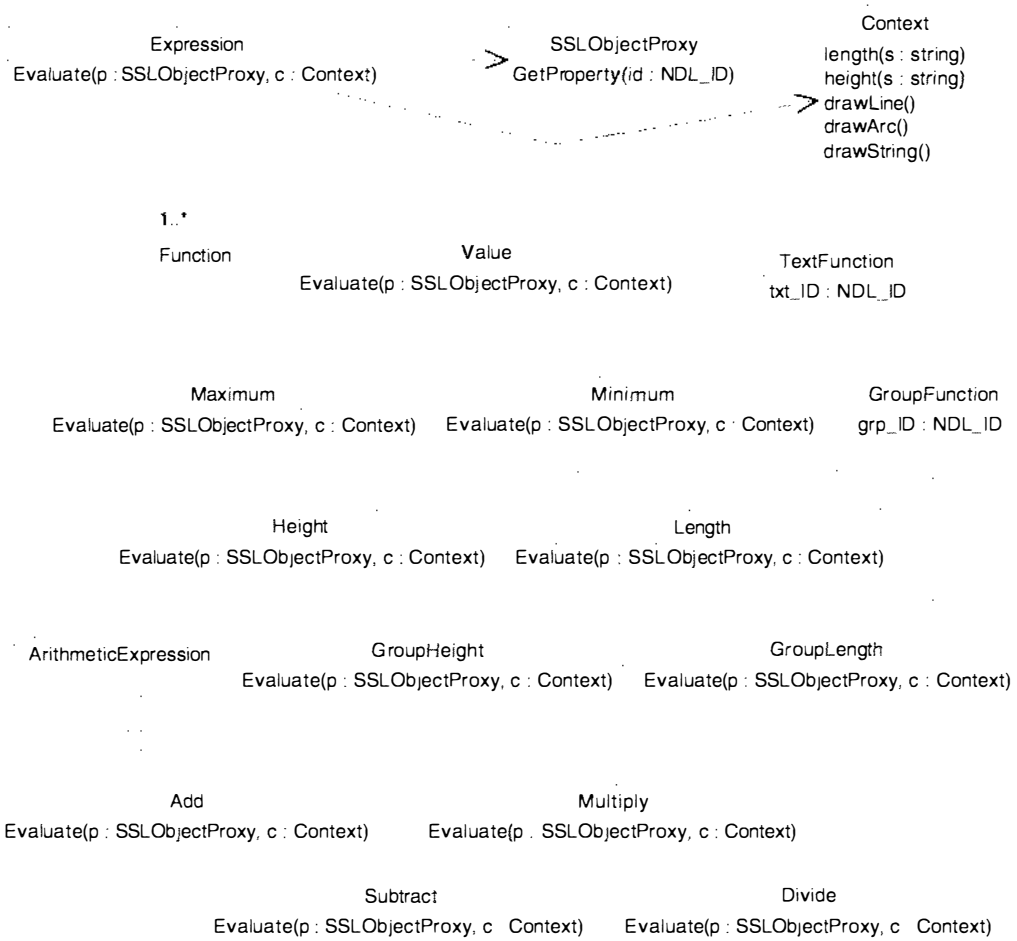


Figure 4-37 - The Expression class hierarchy

All *expression* objects respond to the message evaluate with arguments of an *SSL object proxy* and a *Context*. Arithmetic expressions include all basic operations involving two operands (which are both expressions) and an operator. The arithmetic expressions supported in the initial prototype include *addition*, *subtraction*, *multiplication* and *division*.

Text functions calculate either the height or width of a string or a list of strings. A text function knows the NDL name of the property it is to be applied to. The width and height of a text item depends on the context it is viewed in. This includes the particular font, the font size for the block of text. Text functions delegate the responsibility for performing their calculation to a *Context* object. The *Context* object can calculate the physical size (in drawing units) of a string.

Four functions are used to capture constraints between sub-parts of a view. The *GroupHeight* and *GroupLength* functions calculate the physical size (in drawing units of a group reference. The *Maximum* and *Minimum* functions calculate the maximum and minimum value of their argument expressions respectively.

4.7.2 Segment Templates

Segment templates correspond to the primitive notation elements such as lines and arcs. They are implemented in the interpreter with the class hierarchy given in Figure 4-38.

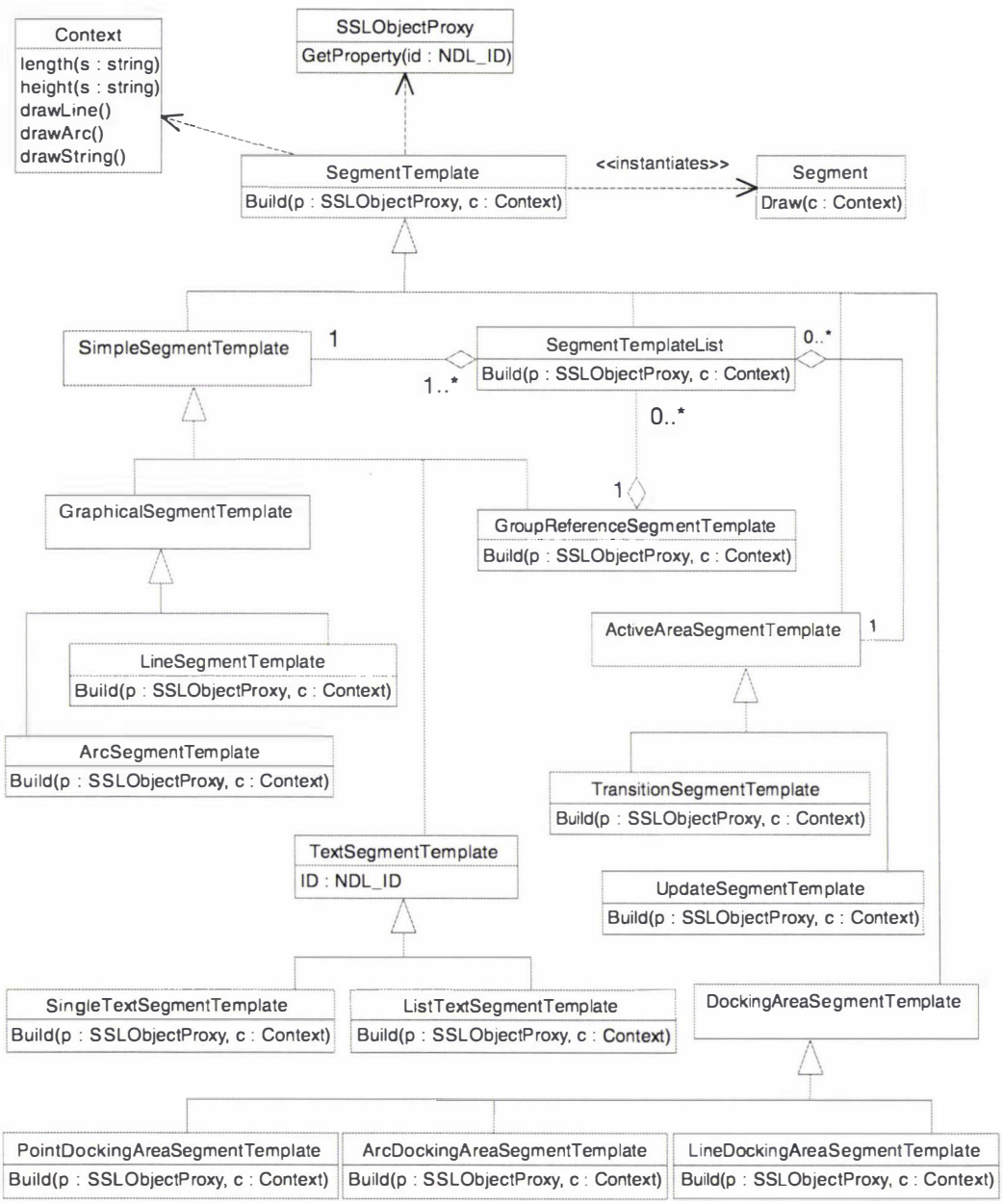


Figure 4-38 - Template segment hierarchy

The leaf classes in the inheritance hierarchy in Figure 4-38 correspond to the different components of a view that are supported (lines, arcs, text, active areas and docking areas). The *SegmentTemplate* class defines an operation called build, which takes an *SSL object proxy* and a *Context* as arguments. A *SegmentTemplate* object responds to the *Build* message by creating an instance of the *Segment* class. Segments are the primitive components of views and correspond to notation elements such as lines, arcs and text fields. Instances of *ArcSegmentTemplate* and *LineSegmentTemplate* construct arc and lines respectively. A *SingleTextSegmentTemplate* defines a single line of text. A *ListTextSegmentTemplate* builds a list of text items. *Segments* know how to draw themselves with the assistance of a *Context* object. An inheritance hierarchy of segment classes corresponding to the segment template hierarchy is also defined, but not shown for brevity.

4.7.3 Group Templates

Group templates are implemented by the classes *GroupReferenceSegmentTemplate* and *SegmentTemplateList* (Figure 4-38). An instance of *GroupReferenceSegmentTemplate* encapsulates a reference to a segment template list (which contains a collection of segment templates). An instance of *SegmentTemplateList* may also contain instances of *GroupReferenceSegmentTemplate*. A group template may, therefore, be defined as a collection of simple segment templates and other group templates.

A template segment list also contains a collection of active area segment templates. The two types of active area (transition and update) are supported with the classes *TransitionSegmentTemplate* and *UpdateSegmentTemplate* (Figure 4-38). Active area segment templates know which template they belong too. A Transition template segment additionally knows which template to transform into. An update segment template knows which properties are to be updated.

4.7.4 Connection and Symbol Templates

The components of a notation are supported with the classes in Figure 4-39. A notation is composed of templates. An instance of class *Template* responds to the message *Build* by creating an instance of the class *View*. The *Template* class defines the view construction protocol that is implemented by its sub-classes.

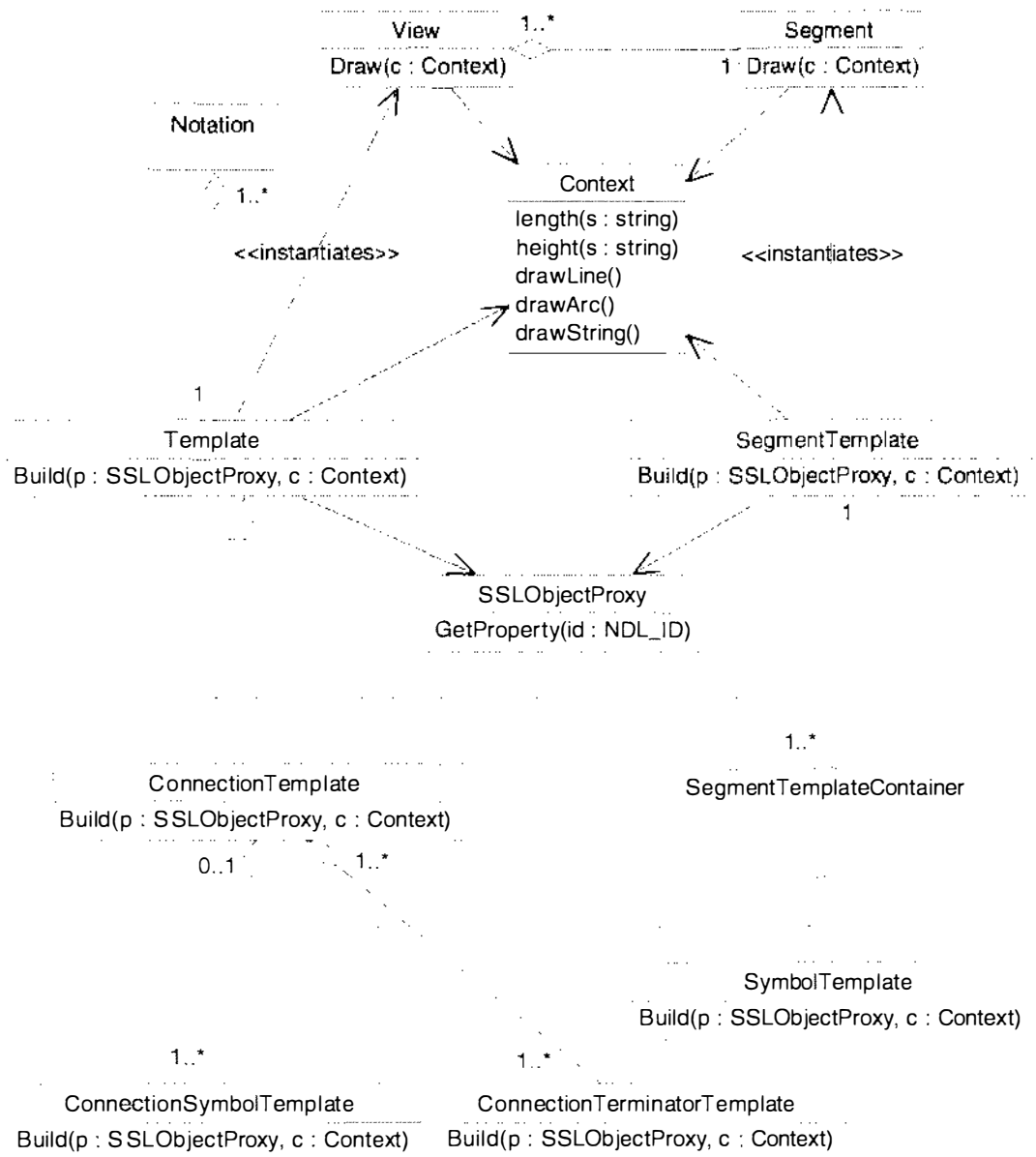


Figure 4-39 - The different types of template

The immediate sub-class of *Template* is *SegmentTemplateContainer*. This abstract super-class maintains a collection of segment templates. The sub-classes of *SegmentTemplateContainer* are *ConnectionSymbolTemplate*, *ConnectionTerminatorTemplate* and *SymbolTemplate*.

Instances of *ConnectionSymbolTemplate* describe connection symbols (such as the triangle in the UML inheritance connection and the semi-circle in the Coad and Yourdon Gen-Spec connection). Instances of *ConnectionTerminatorTemplate* describe the terminators that appear at the ends of connections. Finally, instances of *SymbolTemplate* describe symbols such as

process bubbles in a data flow diagram, classes on a class diagram and states on a state transition diagram.

Templates for building connections are represented by the class *ConnectionTemplate*. A connection template is composed of a collection of connection terminator templates and an optional connection symbol template.

4.8 Implementation of the NDL Interpreter

The NDL interpreter has been implemented on a Sun SparcSERVER 1000e running Solaris 2.5 using SparcWorks C++ 2.0, Tcl 7.3, Tk 3.6 and xf 2.3. Tcl is a general-purpose interpreted programming language. Tk is an extension to Tcl that supports graphical windowing applications. Xf is an interface development tool that allows the construction of applications based on Tcl and Tk. Together these tools allow the rapid construction of graphical interfaces.

Figure 4-40 shows two snapshots of the system processing NDL descriptions of the Rumbaugh instance and object diagram. Figure 4-41 shows a snapshot of the system processing an NDL description of the Coad and Yourdon class diagram.

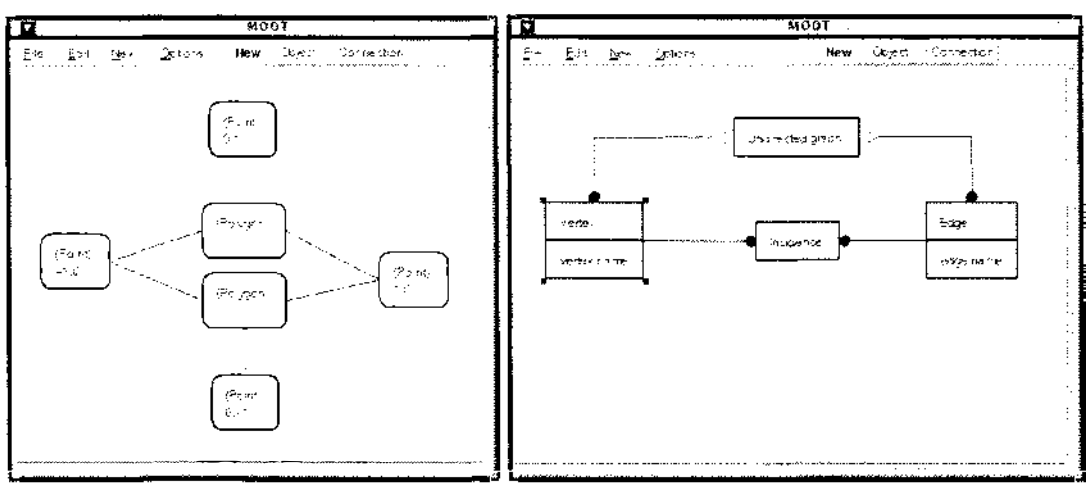


Figure 4-40 - NDL interpreter using an NDL description of the Rumbaugh instance and object diagram

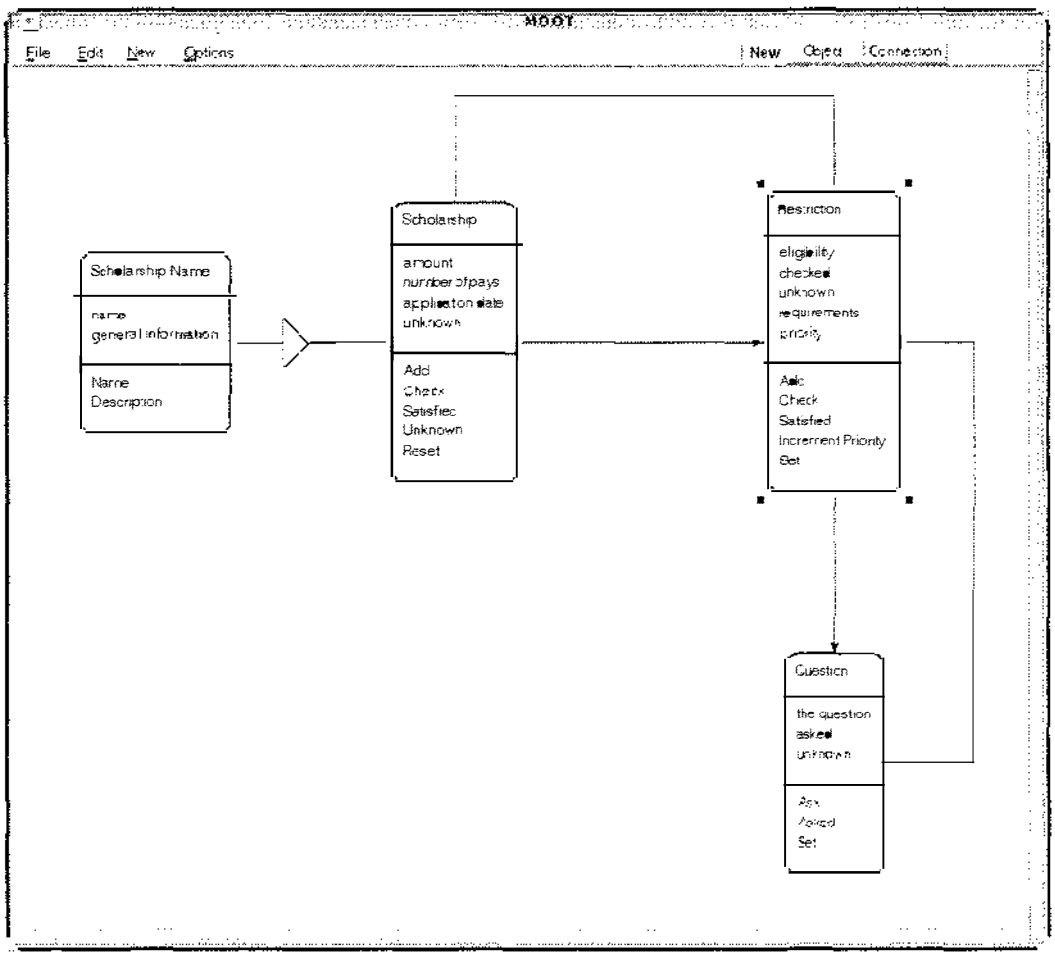


Figure 4.41 - NDL interpreter using an NDL description of the Coad and Yourdon class diagram

4.9 Summary

This chapter has presented the development of NDL. It began with an overview of graphical notations that are used in software engineering methodologies. A clear distinction was drawn between the syntax of a model and its meaning. A notation only facilitates the communication of meaning. It is not the 'meaning' itself. The requirements and design of NDL were discussed and a prototype NDL interpreter, from which the development of the CASE tool client is based, was presented.

Chapter 5

Semantic Specification Language

Good languages not only rest on mathematical concepts which make logical reasoning about programs possible, but also on a small number of concepts and rules that can freely be combined. If the definition of a language requires fat manuals of hundred pages and more, and if the definition refers to a mechanical model of execution (i.e. to a computer), this must be taken as a sure symptom of inadequacy.

Niklaus Wirth 1997

5.1 Introduction

This chapter presents the design and philosophy of a new language that is used to implement methodology semantic descriptions in the MOOT system. The new language has been named SSL (***S***emantic ***S***pecification ***L***anguage). The major goal of development of SSL is to derive a language that directly supports the MOOT meta-model, provides facilities for re-use of methodology descriptions and is suitable for a programmer to use. An aspect of this research is the development of an efficient and portable mechanism for executing SSL.

5.2 Method

The following is a high level description of the steps taken during the design and development of SSL.

1. Derive the requirements for a language that allows the description of methodology semantics.
2. Investigate existing languages. Clarify the goals and design a new language, SSL.
3. Derive an execution strategy for SSL. Consider space-time efficiency and platform independence of the execution strategy.

4. Design an intermediate, platform independent binary representation for SSL (SSL-BC).

5. Design and implement a new virtual machine after consideration of other virtual machines - SSL-VM (***SSL V***irtual ***M***achine).

6. Develop and implement a compiler that translates SSL into SSL-BC.

7. Test SSL, and the SSL-VM with some simple examples.

5.3 Rationale and Goals of SSL

The goals of SSL are derived from some of the limitations of meta-CASE tools as described in chapter 2. They address limitations related to the use, number and separation of the specification languages used by existing meta-CASE tools. These goals are:

1. **Integrate the description of structure and behaviour**

   Previous meta-CASE tools provide two or more separate languages for the specification of methodologies. One is used to define structure and the second to define constraints on the structure (a form of behaviour). There are several problems with this approach: a) there are multiple languages for the same task b) the coupling of methodology semantic specifications increases and c) the cohesion of methodology semantic specifications decreases.

2. **Support more than completeness and consistency checking**

   Current meta-CASE tools only focus on checking the rules of the various modelling languages. There is no consideration of things such as auto-correction, quality analysis or guidelines. The behavioural aspects of SSL can include more than checking constraints, and be used to implement auto-correction etc.

3. **Emphasise 'programming the semantics' rather than formally defining them**

   Most meta-CASE tools provide either an extremely formal set of languages or a large application programmer interface (API).

   Formal languages can be difficult to understand and use. The use of formal languages also means that supporting inconsistent models is often not possible. This is a barrier to an exploratory approach to design that software engineers naturally use.

Meta-CASE APIs obscure the underlying meta-model and place no emphasis on specification. A large portion of the API is also generally related to the user interface and the underlying repository. This means the API itself provides facilities that are at different levels of abstraction with respect to the meta-CASE tool.

If SSL provides some of the facilities of a general purpose programming language then it will be more flexible and comprehensible. SSL should be sufficiently flexible that programmers feel comfortable using it, yet it should never imply it is a general purpose programming language.

4. **Support re-use**

Existing meta-CASE tools do not place any emphasis on re-use of methodology descriptions. They only support accidental re-use, where existing methodology descriptions may be duplicated and then changed. There are several problems with this approach: a) it is wasteful in terms of resources and development effort; b) there is no clear relation between methodology descriptions that are similar; c) support for re-use of software engineering projects is difficult; d) a very large and unstructured pool of methodology descriptions exist.

5. **Space/time efficiency**

SSL must support methodology descriptions whose execution is efficient in terms of space and time. This includes the language and its run-time representation.

6. **Platform independence**

Both methodology specifications and user projects must be completely portable across platforms. Methodology descriptions and software engineering projects can then be distributed to other users of MOOT without translation.

7. **Hide persistence of SSL objects**

Software engineering projects are represented by collections of SSL objects in the persistent store. This fact should be completely hidden from users of SSL. Object persistence is transparently addressed by the SSL-VM.

Figure 5-1 illustrates how the various goals of SSL have been addressed by some of the design decisions made regarding the features SSL.

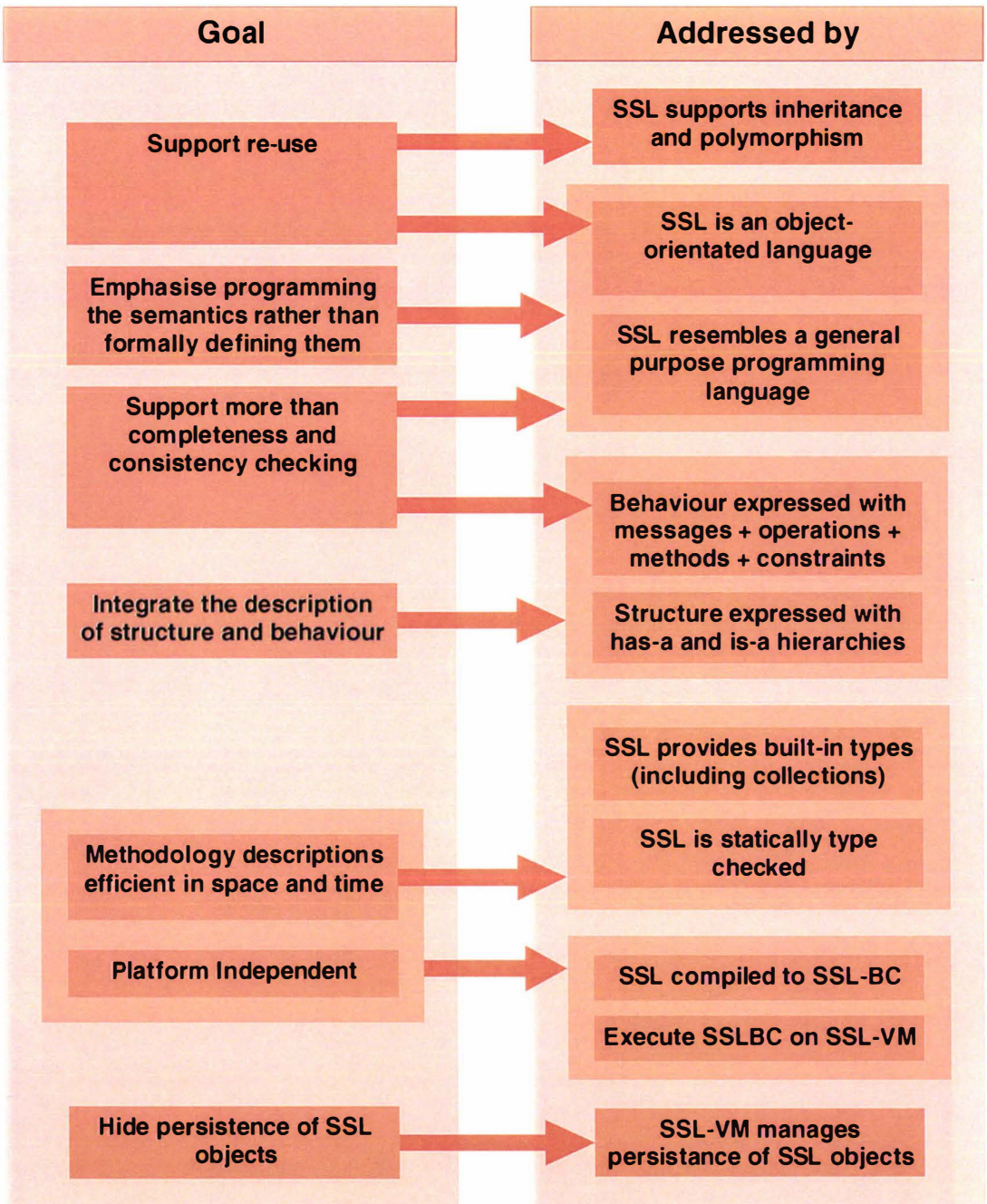


Figure 5-1 - Mapping between goals and design decisions made regarding features of SSL.

5.4 Requirements of SSL

There are two types of requirements for SSL. The first type (MOOT specific requirements) is related to supporting the MOOT philosophy for the description of

methodology semantics. The second type (SSL specific requirements) is related to addressing the limitations of other meta-CASE specification languages.

MOOT Specific Requirements

- Provide all the facilities of the MOOT meta-model. The MOOT meta-model is an object-orientated meta-model. SSL must, therefore, be an object-orientated language.
- Support the description of behaviour. This requirement is satisfied by the decision to develop SSL as an object-orientated language. Behaviour is supported with message passing, methods and constraints.
- Suitable for 'programmers' to use. SSL provides some of the facilities of a general-purpose object-orientated programming language. For example it provides classes and supports sequence, selection and repetition. It does not provide facilities such as input/output.
- Used as a specification language. The choice of facilities that SSL supports must ensure that it is not considered to be a general-purpose programming language.
- Support the expression of constraints. Each SSL class may define a constraint. A constraint is a boolean expression that is a function of the state of an object.
- Support re-use. SSL is an object-orientated language. Re-use is supported with inheritance and polymorphism.

SSL Specific Requirements

- Provide the following basic primitive types: Real, Integer, Boolean and String. SSL supports primitive types to facilitate time-efficient execution of SSL.
- Provide facilities for collating sequences of items. The support should be as simple as possible and at least permit adding and removing items as well as the traversal of a sequence.
- Address the global namespace pollution common in other meta-CASE tools.
- Provide a clean separation of interface and implementation. The public interface of SSL classes only consists of a collection of operations. The attributes and methods are not accessible to other classes.

- Support multiple entry points. Conceptually the execution of an SSL description may start at any operation.

SSL does not require:

- Input/output facilities. Supporting input and output is the responsibility of NDL.
- Concurrency[40]. Supporting concurrency in SSL could have several negative effects: a) the language becomes more complex; b) the language resembles a general purpose programming language; c) the MOOT meta-model becomes obscured.

5.5 Semantic Specification Language

5.5.1 Overview

SSL is an object-orientated language, with extensions to explicitly support the description of methodologies. It is an executable specification language whose primary purpose is to provide all the facilities of the MOOT meta-model. SSL is strongly typed, statically typed checked, implements late binding, provides a module system and supports a simple automatic memory management system.

The execution profile expected of SSL is:

- A large number of messages
- Each message will take a small amount of time to process
- Frequent creation and destruction of objects

5.5.2 MOOT Meta-Model

A model of the MOOT meta-model (a meta-metamodel) has been derived and is shown in Figure 5-2.

SSL Classes

SSL classes have an interface[41], a collection of attributes and a collection of methods. Multiple inheritance is supported; an SSL class can inherit from one or more super-

[40] Concurrency will be addressed in section 9.4 Future Work

[41] The current meta-model supports a one-to-one mapping between a class and its interface.

classes. The interface of an SSL class consists of the list of operations and defines the name of the class. Operations are overloaded based on the order, number and type of parameters in the parameter list. Each method corresponds to one of the operations in the interface of the class. A method consists of a collection of statements. The types of statement support sequence, selection, iteration and assignment.

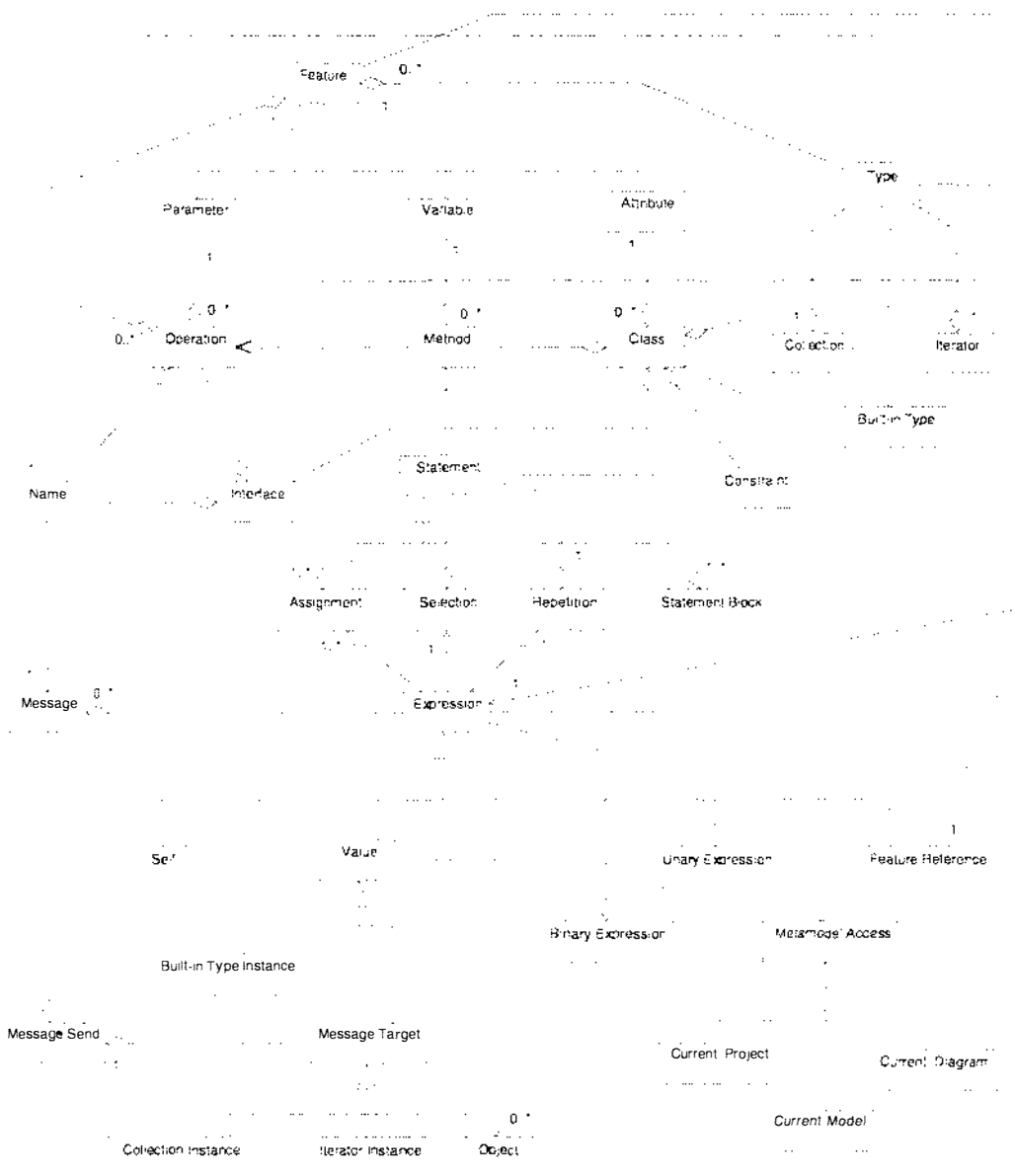


Figure 5-2 - MOOT meta-metamodel

Constraints

In addition to attributes and operations, each class may define a constraint for its instances (an invariant which is a function of an object's state). The constraint is

evaluated after an object receives and processes a message. If the constraint is violated the state of the object is restored to the state it was in prior to the message being invoked.

SSL Objects

An object contains a set of values, which correspond to the attributes defined by its class. It also contains a collection of objects that correspond to the super-classes of its class. Values can be instances of built-in types (integers, strings and so on), iterator instances, collection instances and objects. The state of an SSL object only changes as a result of accepting and processing messages.

Extensions to support the description of Methodologies

The MOOT meta-model defines a set of built-in variables called *current_project*, *current_model* and *current_diagram*. The values of these variables define the context the user is in as they carry out actions at the user interface. They are analogous to the *self* in Smalltalk and *this* in C++. Figure 5-3 shows how the values of these variables define the context (the project, model and diagram) the user is in whenever they perform an action.

The user has selected an active area on a symbol in Figure 5-3. As discussed in chapter 4, an action is generated and propagated to the MOOT core (see Figure 3-11 - Architecture of the MOOT prototype). *Current_model* is a reference to the model that is the context from which the action occurred. *Current_diagram* is a reference to the corresponding diagram of the *currnt_model*. Finally *current_project* is a reference to the software engineering project.

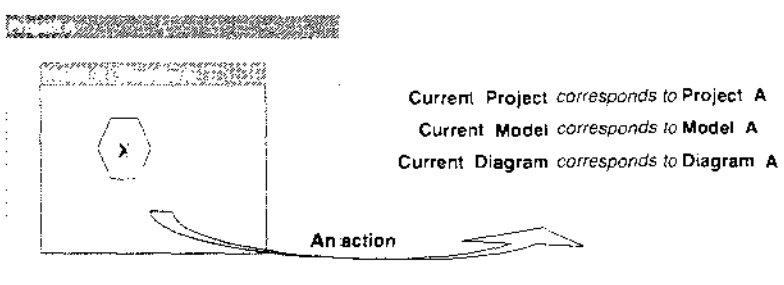


Figure 5-3 - The built-in SSL variables

Expressions

The MOOT meta-model provides boolean, integer, real and string fundamental types. The collection and iterator types together provide support for sequences of items. Expressions include values, self, the built-in SSL variables, unary and binary expressions

and message-send expressions. Messages may be sent to an object, a collection instance and an iterator instance. Each message has a name and a set of arguments. The message name corresponds to the name of an operation.

5.5.3 Module System

All SSL classes belong to a module. All module names must be unique in the scope of the CASE environment. Modules are implemented in two parts in SSL. Interface modules provide a public list of class interface definitions. Each class interface only defines the class name, a set of operations and any super-classes. Implementation modules provide the corresponding implementation for each class. This includes the attributes and methods implemented by each class. The SSL module is similar to a Booch Class Category, a C++ namespace, an Ada95 package and a category in the Smalltalk programming environment. There are two major differences:

- SSL modules are separated into interface and implementation modules
- SSL interface modules only define a collection of class interfaces

5.5.4 Memory Management

SSL provides a simple, automatic, memory management system. It is a simple adaptation of the reference counting algorithm (Jones and Lins, 1996). Each SSL object maintains a count of the number of other objects that reference it. SSL objects are given an initial count of 1, as they are created. The count is incremented for each new reference to the object and decremented each time a reference is broken. SSL objects delete themselves, once their reference count reaches zero. This scheme has been adopted because:

- It distributes the memory management overhead by interleaving the garbage collection with the execution of SSL.
- The response time, with respect to execution of SSL, is regular.
- The execution profile expected of SSL (high number of requests for computation and small execution time of each computation) suggests that the memory overhead of storing reference counts and the computation overhead from updating reference counts would not be an issue.

SSL objects are created by sending a create message to a class[42]. There must be at least one create operation implemented for each class. Create operations are implicitly meta-level operations whose sole purpose is to provide an appropriate initial state for SSL objects. Create operations may be overloaded.

Each class may define a single destroy operation. Once the number of references to an SSL object reaches zero a destroy message is automatically sent to it and the SSL object is released. Destroy messages are automatically sent to the objects that correspond to the super-classes of its class.

5.5.5 Messages

A message in SSL, as in other object-orientated languages, represents a request to perform an operation. A message has two parts: a message selector and an argument list. The message selector corresponds to the name of an operation to be performed. The argument list is a collection of SSL objects, collections, iterators and simple values required to perform the operation.

All messages in SSL are dynamically bound. Late binding is implemented via a method lookup table per class to avoid the run-time overhead of searching the inheritance hierarchy for an appropriate method to bind to a message.

5.6 Semantic Specification Language Definition

The following discussion uses a simple implementation of the Sieve of Eratosthenes[43] in SSL to aid the illustration of the facilities SSL provides. The syntax of SSL is presented by using examples in the remainder of the chapter. The SSL grammar is presented in appendix III. Two complete SSL implementations of the Sieve of Eratosthenes are given in appendix IV.

SSL provides the following simple built-in types: Integer, Real, Boolean and String. It also provides two parameterised types: collection and iterator. The definition of a class introduces a new type. New types are also added by providing concrete parameters for the collection and iterator types.

[42] This is currently the only message that may be sent to a class.

SSL variables, whose type corresponds to a class[44], are similar to variables in Smalltalk and Java and contain a reference to an SSL object. These variables are initialised to a special value (*no_object*) before their first use. Variables of a collection type also contain a reference. Variables of an iterator type and variables of a simple type contain values.

Variables of a class type may contain a reference to an SSL object defined by the class type of the variable itself. It may also contain a reference to an object defined by any of the sub-classes of the variable's class. The only messages which may be sent via a variable of class type are those that are defined in the interface of the class of the variable, or one of its super-classes. This restriction is imposed because SSL is statically type checked.

5.6.1 Collections

SSL provides built in polymorphic collection and iterator types. These two types operate together to provide sequences of elements of an arbitrary type. SSL collections support insertion deletion and traversal of collections. No particular ordering of items in a collection can be assumed. The interfaces of the *Collection* and *Iterator* types are given in Figure 5-4.

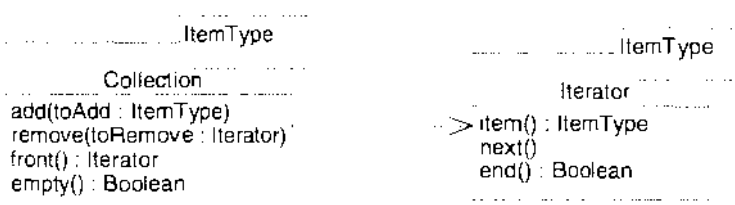


Figure 5-4 SSL collection and iterator types

A collection item type may be any of the built-in SSL types (including iterators and collections) as well as SSL objects. The interface of the SSL collection and iterator types, as shown in Figure 5-4, is the minimum needed to support collections.

5.6.2 Simple Expressions

The types of expressions in SSL include: simple values, special SSL values, arithmetic expressions, relational expressions, boolean expressions, string expressions, scope resolution expressions, create object expressions and message send expressions.

---

[43] The Sieve of Eratosthenes is a well-known and extremely elegant algorithm for calculating prime numbers.

[44] Such variables are subsequently referred to as 'variables of class type.'

Simple Values

SSL values may be any of the following:

- A constant. This includes constant integer values (e.g. 10), real values (e.g. 10.2), boolean values (true or false) and strings (e.g. "a string")
- The result of a message
- A reference to an attribute, a local variable of a method or a message argument
- Self, which is a reference to the current object
- One of the built-in, pre-defined variables. These are current_project, current_model and current_diagram

The following special values are also defined in SSL.

`True`	Boolean true
`False`	Boolean false
`No_object`	Empty reference to an object

Arithmetic Expressions

SSL supports the following arithmetic operators:

`+`	Addition	`/`	Division
`-`	Subtraction	`Div`	Integer division
`*`	Multiplication	`Mod`	Integer modulus

All operators are overloaded for the basic built-in arithmetic types except the *div* and *mod* operators, which are only defined for integers.

Relational and Boolean Expressions

SSL supports the following relational operators:

`<`	Less than	`<=`	Less than or equal to
`>`	Greater than	`=`	Equal to
`>=`	Greater than or equal to	`<>`	Not equal to

The values of all built-in types may be compared with the relational operators. Both of the values compared must be of equivalent type. The type of a relational expression is boolean. The following boolean operators are also supported:

`And`	Logical conjunction
`Or`	Logical disjunction
`Not`	Logical negation

These operators may only be used with boolean values and expressions.

String Expressions

Strings concatenation is performed with the overloaded addition (+) operator. The relational operators may also be used with strings, as expected.

Scope Resolution Expressions

: :	Module scope resolution operator
:	Class scope resolution operator

The module scope resolution operator is used to qualify a class name with its owning module to overcome class name clashes. The class scope resolution operator is used to qualify a message name with a class to overcome message name clashes. For example two modules may define a class with the same name. They can be referred to by fully qualifying the class name with the module name (for example *aModuleName::aClassName*). The class scope resolution operator can be used in an analogous way (for example *aClassName:anOperationName*).

Message Send Expressions

.	Message send operator

A message-send expression is composed of three parts: an object (the message receiver), the message-send operator and a message. The message consists of a message name and a list of arguments. The message-send operator is used to bind the message to a particular operation of the receiving object. The type of a message-send expression is that of the result type of the requested operation. A message-send expression can be used anywhere a value of its type may be used[45] (e.g. on the right hand side of an assignment, or as an argument to another message). Message-send expressions are evaluated eagerly[46]; all of the actual arguments are evaluated before the message is sent.

Create Object Expressions

Objects are created in SSL by sending a create message to a class. The result is the creation of an instance of the class the message is sent to. Each class may implement

---

[45] Currently messages that return more than one object as a result may only appear on the right hand side of an assignment statement This restriction is in place to speed up the implementation.

[46] All arguments are evaluated first, before the message-send expression is evaluated.

several create operations. They are overloaded by the order and type of the arguments. The following example illustrates how objects are created.

```
// aList is of type list
aList = list.create();
aList.cons( listItem.create( 10 ) );
// anotherList is of type list
anotherList = list.create( aList );
// anItem is of type listItem
anItem = listItem.create( 20 );
anotherList.cons( anItem );
```

5.6.3 Interface Module

An interface module provides a public collection of class interfaces. All interface modules must have unique names. The names of classes defined within each interface module need only be unique within the scope of the module.

Each module starts with the *module* keyword and is followed by a name. A 'uses clause' declares the modules and classes that are used in a module. Class names are always qualified by the name of the module they are defined in. Using a class name without qualification is a shortcut for identifying a class defined within the current module.

```
module moduleName;
uses otherModule::otherClass, anotherModule;
```

This uses clause specifies that the class *otherClass,* and any of its sub-classes, may be used within *moduleName,* It also declares that any of the classes defined by the module *anotherModule* may be used. A 'uses clause' can be used to introduce a local name (an alias) for a class. The scope of the alias is the module the alias is defined in.

```
module MyMethodology_Model_Elements;
uses OOM_A_Model_Elements::class = OO_A_class;
uses OOM_B_Model_Elements::class = OO_B_class;
```

In this example two modules that both define a class called *class* are used. Two local aliases (*OO_A_class* and *OO_B_class*) are introduced as a syntactic convenience.

5.6.4 Class Interface Definition

A class interface defines the set of operations that may be performed by an instance of a class. It consists of a class name, an optional list of super-classes and a list of operations.

```
className : superClass, anotherSuperclass
{
   integer operationOne()
   integer operationOne( integer X )
   operationTwo()
   operationTwo( integer X, integer Y )
}
```

In this example the class *className* has two super-classes (*superClass*, *anotherSuperclass*). It also defines four operations. The operations are overloaded based on the operation name and on the order and type of the arguments.

5.6.5 Implementation Module

Each interface module has an associated implementation module, which defines the implementation of each class listed in its corresponding interface module.

The implementation module starts with the keyword *module* and is followed by its name. The name of the implementation module is the same as its corresponding interface module. Implementation modules may also have zero or more uses lists. The rest of the module consists of a list of class definitions.

5.6.6 Class Definition

Each SSL class definition in the implementation module corresponds to an SSL class interface in the interface module. An SSL class definition consists of a class name followed by the definition of the attributes, methods and an optional constraint. The following is an example that shows a definition of a list class (from the Sieve of Eratosthenes example in appendix IV), where the method bodies are empty.

```
list
{
   attributes
      listnode l;
   operations
      new()                    {}
      cons( integer value )    {}
      listIterator front()     {}
      tail()                   {}
      boolean isEmpty()        {}
}
```

Figure 5-5 Partial SSL implementation of a list class

The *attributes* section consists of zero or more attributes. The type of an attribute may either be a built-in SSL type, a class, a collection or an iterator. The aggregation relation in Figure IV-1 - Sieve of Eratosthenes version 1, is captured by the *listnode* attribute *l* in Figure 5-5. The operations section lists the implementation of the operations (the methods) in the class interface. Each SSL class may optionally define a single constraint, which is a list of boolean expressions that are functions of the state of an SSL object. Evaluating the constraint for an object includes evaluating the constraints defined in the class of the object and in each super-class.

5.6.7 Methods

The methods for an SSL class are defined inside the body of the class. Each method definition has five parts: a result type, a name, a formal argument list, a local variable list and a body.

The formal argument list follows the method name and is a comma-separated list of type -argument name pairs. All arguments are passed by value. Variables of simple built-in SSL types and iterators contain values whilst variables of class and collection types contain references. The formal argument list is optionally followed by the definition of any local variables that are used in the body of the method. Finally, the method body consists of a sequence of SSL statements.

SSL methods may return zero or more objects as a result. In the following example the class *aClass* defines four methods, each of which returns a different number of objects as a result.

```
aClass
{
   attributes
   operations
      methodOne() {}
      integer methodTwo() {}
      (integer,integer) methodThree() {}
      (integer,integer,integer) methodFour() {}
}
```

A complete SSL class implementation of the list class in Figure IV-1 - Sieve of Eratosthenes version 1, is given in Figure 5-6.

```
list
{
   attributes

      listnode l;

   operations

      new() { l = no_object; }

      cons( integer value )
         listitem i;
      {
         i = listitem.create( value );
         l = listnode.create( i, l );
      }

      listIterator front() {
         return listiterator.create( l );
      }

      tail() {
         if( not ( l = no_object ) )
         {
            l = l.next();
         }
      }

      boolean isEmpty() { return l = no_object; }
}
```

Figure 5-6 - SSL implementation of the list class

Create Methods

The purpose of create methods is to provide an appropriate initial state for newly created objects. An object is created, in SSL, by sending a create message to a class. The create message names one of the create operations in the interface of the class. This causes the corresponding create method, defined by the class, to be executed.

The default[47] behaviour of sending a create message, with no arguments, is to return a new instance, which has all attributes of class and collection type initialised to *no_object*. This default behaviour can be replaced by implementing a create operation that takes no arguments.

Each class may implement multiple overloaded create operations. Create operations are overloaded by order and type of arguments. The example in Figure 5-7 shows the

[47] A default implementation of the 'no arguments' create operation is automatically provided if one does not exist.

implementation of a class (*aClass*) that defines two create operations. The first initialises the attribute *anInt* to zero. The second initialises *anInt* with the value of its single integer argument.

```
aClass
{
   attributes

      integer anInt;

   operations

      create()
      {
         anInt = 0;
      }

      create( integer I )
           =>   superClass.create( I ),
                anotherSuperClass.create( I + 10 );
      {
         anInt = I;
      }
};
```

Figure 5-7 Implementing SSL create operations

The super-classes of a class are initialised by sending a create message to each super-class. The second create method in Figure 5-7 shows how these create messages are specified (the *super-class create list*). The SSL compiler is responsible for ensuring that the *super-class create list* is correct.

Destroy Methods

A destroy message is automatically sent to an object when the number of references to it reaches zero. Destroy messages received by an object with one or more references are ignored.

A class may only implement the destroy operation once. The purpose of the destroy method is to permit an object to perform any necessary tasks before it is released[48]. The default[49] behaviour of the destroy operation is to assign the special value, *no_object*, to all variables of class and collection type. The destroy operation must be explicitly implemented if any other behaviour is required.

[48] This would include tasks such as informing other objects of its impending demise.

[49] The SSL compiler provides a default implementation of the destroy operation only if one does not exist.

5.6.8 Statements

The following statements are available in SSL:

Message send	Return	Assignment
If	Loop	Debug_Print

Message Send Statement

The message send statement is a special case of a message-send expression where the message being sent does not return any instances as a result.

Return Statement

The return statement signals the end of execution of a method. The value returned must match the return type of the method. A method may contain more than one return statement. Methods that do not have a return type do not require a return statement.

Assignment Statement

The assignment statement is used to update the value of an l-value. L-values include attributes, method parameters and variables local to a method. The right hand side of an assignment statement is an expression. The result of evaluating the expression, the r-value, is used to update the l-value. The types of the l-value and r-value must be compatible. An l-value may also be a tuple, for assignment of the result of a message that returns more than one result. In the following example an object is sent three different messages that each return three values as a result.

```
(X,Y,Z) = aPoint.getCartesianOrdinates();
(r,theta,Z) = aPoint.getCylindricalOrdinates();
(r,theta,phi) = aPoint.getSphericalOrdinates();
```

Tuples are an additional type in SSL that is currently only supported with the assignment statement and return types of methods.

If Statement

The if statement consists of the *if* reserved word, a condition, a statement block and an optional else part that consists of the *else* keyword and a statement block. The condition must be a boolean expression. If the condition evaluates to *true*, then the if statement block is executed. If the condition evaluates to *false*, and there is an else part of the statement, then the else statement block is executed.

Loop Statement

The loop statement consists of the *loop* reserved word followed by a loop body. The loop body consists of a statement block that contains a single endloop clause. The endloop clause consists of the reserved words *endloop when*, followed by a boolean expression. The endloop clause may appear anywhere in the body of the loop statement. The loop body is repeatedly executed until the endloop condition is true. Execution continues from the statement following the loop. Figure 5-8 shows the implementation of the *findprimes* method of the sieve class in Figure IV-1 - Sieve of Eratosthenes version 1.

```
// find the prime numbers contained in the list ints
findPrimes()
   integer step, upperLimit;
   listiterator l;
   listitem i;
{
   step = 2;
   upperLimit = top div 2;
   loop
   {
      l = skip( ints.front(), step - 2);
      if( not l.end() )
      {
         i = l.item();
         if( i.isprime() )
         {
            mark( l, step );
         }
      }
      step = step + 1;
      endloop when( step = upperLimit );
   }
}
```

Figure 5-8 Example loop and if statements

Debug_Print Statement

The debug_print statement exists for debugging purposes during the development of SSL. It was added since SSL itself does not need to support any input/output. The debug_print statement evaluates its single argument and displays the result onto the standard error stream. Both the expression type and the result are displayed.

5.7 SSL Compiler

The SSL compiler translates SSL interface and implementation modules into ***SSL B***yte ***C***ode (SSL-BC). The main components of the SSL compiler are presented in Figure 5-9.

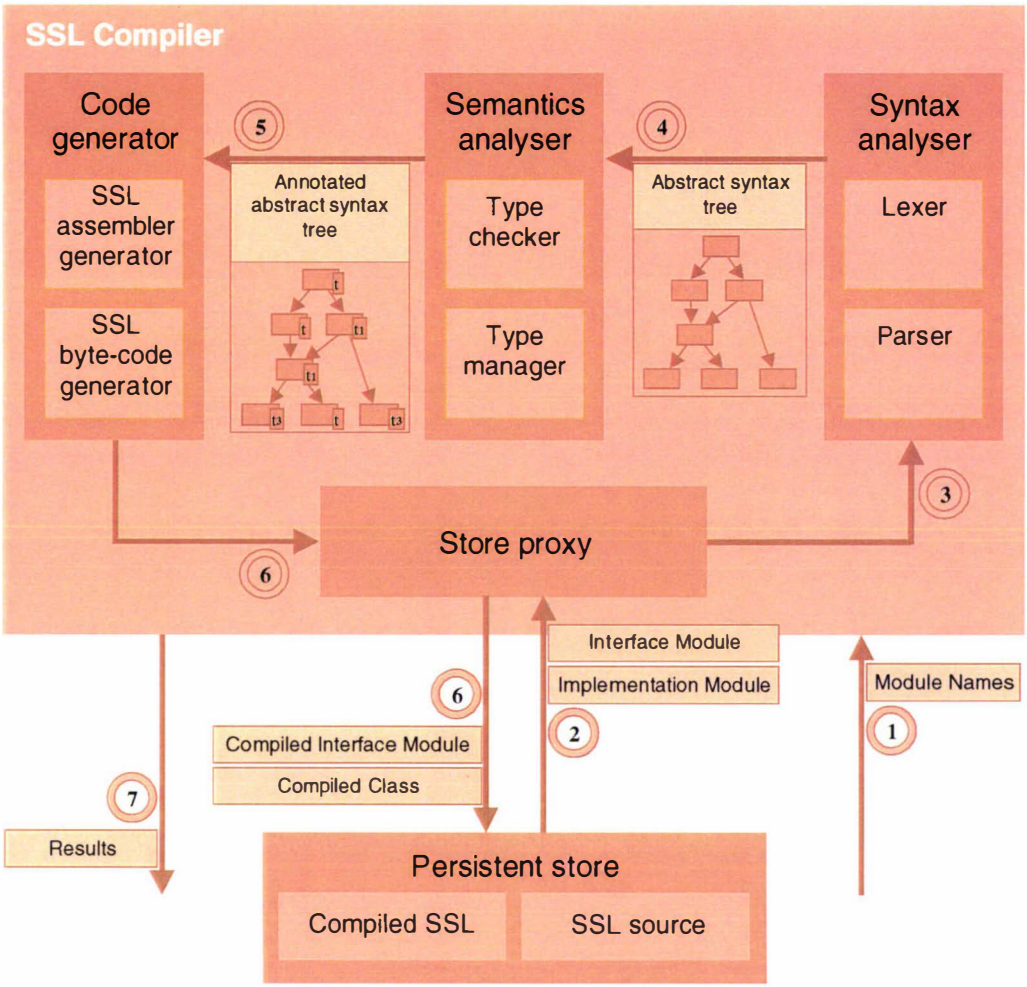


Figure 5-9 - SSL compiler

SSL source code and compiled SSL are both maintained in the persistent store. The persistent store provides version control facilities and ensures mutually exclusive access to SSL classes. The Store Proxy isolates the compiler from the persistent store.

The syntax analyser is responsible for performing lexical analysis and syntax checking. This component was built using PCCTS (Purdue Compiler Construction Toolkit) (Parr, 1997; PCCTS, 1998). The parser is an LL(k) parser, which dynamically adjusts the look-ahead depth (k) as it parses.

The semantics analyser is responsible for performing type checking. It also builds and checks the method lookup table for each class.

The code generator is responsible for translating SSL into SSL-BC for each class and compiled interface modules. The code generator can also be used to print the annotated abstract syntax tree and to produce SSL assembler[50].

Figure 5-9 also shows the steps in the compilation process. These are:

1. The compiler is invoked. It accepts a list of module names to be compiled as input. Each module is compiled in two phases. In the first phase steps 2 – 6 are performed for the interface module. In the second phase steps 2 – 6 are performed on the corresponding implementation module.
2. The compiler determines if the interface and implementation module source code has been updated since the last time it was compiled. It does this by asking the store proxy to compare the modification dates of the source code and compiled module. If the source need to be compiled the compiler asks the store proxy to retrieve the interface and implementation module source code from the store.
3. The lexer and parser then process the module source code and build an abstract syntax tree. Any lexical and syntax errors are reported.
4. The type checker traverses the abstract syntax tree and annotates each node in the tree with type information. It also generates method lookup tables from class interface definitions in the interface module. This includes checking for operations that cannot be disambiguated from each other and detecting the inheritance of the same operation from two or more super-classes. Type errors, ambiguous operations etc are reported.
5. The code generator traverses the annotated abstract syntax tree generated by the semantics analyser and generates SSL-BC code for each class and interface module.
6. The compiler then asks the store proxy to place the compiled SSL into the persistent store.
7. Finally the results are reported. This will be a simple list of all the interface modules and implementation modules that were successfully compiled.

A description of the implementation of the SSL compiler is given in appendix VI.

---

[50] SSL assembler is a simple human readable version of SSL-BC that is provided for debugging purposes. It is not discussed further in the thesis.

5.8 Executing SSL

Figure 5-10 shows a more detailed view of the components of the methodology interpreter and the tool manger.

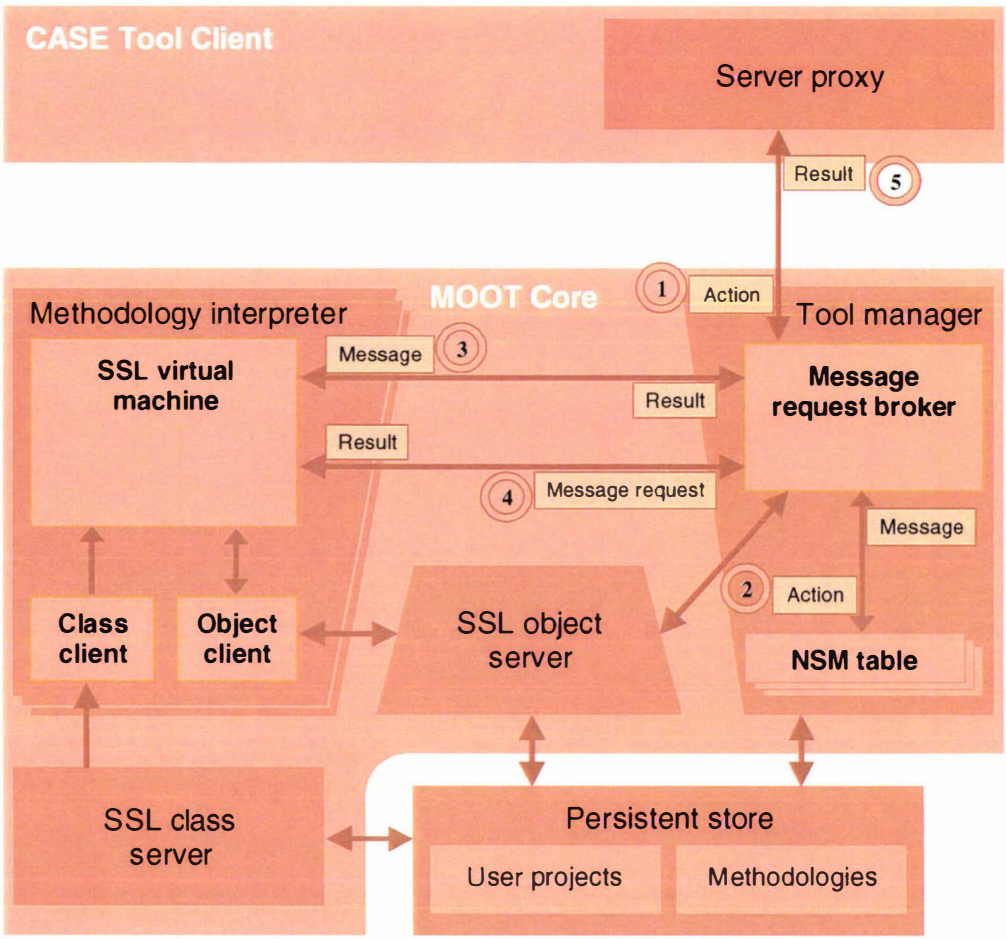


Figure 5-10 - Processing actions

There is one tool manager for the overall system, which acts as a server for multiple virtual machines. The tool manager insulates the rest of the system from the persistent store and the CASE tool clients from their corresponding virtual machines.

The tool manager maintains an instance of the virtual machine for every active user. Each instance of the SSL-VM has only one thread of control. This means that incoming messages are queued until the SSL-VM completes processing the current message.

The Message Request Broker accepts message requests from clients and messages sent as a result of the interpretation of a method. It is responsible for ensuring mutually exclusive

access to objects (through object level locking). It is also responsible for detecting deadlock situations and resolving them in a manner transparent to the message senders.

The SSL class server maintains a cache of SSL classes. If a requested class is not present in the cache it is retrieved from the persistent store. This class replaces the least recently requested class in the cache, if the cache is full.

The SSL object server is responsible for caching SSL objects in a manner similar to the SSL class server. It provides a transparent reference counting mechanism and garbage collection for SSL objects in the persistent store. It also maintains consistency between copies of the same object.

The SSL class client maintains a cache of SSL classes in a manner similar to the SSL class server. The difference is that the class client is executing as part of the same process as the virtual machine. The purpose of this cache is to minimise inter-process communication.

The SSL object client manages temporary (non-persistent) SSL objects. Any updates to or requests for persistent objects are passed directly to the SSL object server. Temporary objects that are assigned to attributes of persistent objects become persistent themselves, and are passed to the SSL object server to be placed in the persistent store. The Methodology Interpreter does not distinguish between temporary, local and persistent objects. Responsibility for all accesses and updates is delegated to the SSL object client, which determines if the operation needs to be passed to the SSL object server or dealt with locally.

The following lists the steps taken in processing an action (see Figure 5-10):

1. The server proxy in the CASE tool client propagates an action at the user interface to the tool manger. The server proxy implements the communication protocol between the CASE tool client and the MOOT core. The message request broker initially translates the user action into a corresponding semantic action.
2. The message request broker delegates the responsibility for finding this semantic action to an NSM table. The NSM table returns a message as a result. Note that each methodology has its own NSM table and that the role of the table is to provide a

mapping between notations and semantics descriptions. NSM tables are the topic of chapter 7.

3. The message is propagated to the SSL Virtual Machine. The SSL-VM binds the message to a particular method and executes it.

4. Any messages that are sent during the execution of the method are initially propagated to the message request broker. The message request broker is responsible for ensuring mutually exclusive access to objects and for detecting deadlock situations.

5. The result of the initial action is returned to the CASE Tool client once the corresponding message found in step 1 has been processed.

The message bandwidth between the message request broker and the methodology interpreter is high as the execution of SSL methods typically cause many messages to be sent. Whilst not an issue for the prototype[51], it is an important consideration for the final MOOT system.

The proposed architecture (Figure 3-10 - Proposed, top level, system architecture) could be implemented in many different ways. For example each component could execute as separate processes and could conceivably execute on different machines. It is more likely that the persistent store, the SSL class server and the SSL object server will execute as separate processes, possibly on separate machines. The tool manger and methodology interpreters will most likely execute as a single process, possibly on a separate machine, with one thread of control used for each methodology interpreter.

5.9 SSL Virtual Machine

Much work has been done previously on virtual machines for object orientated-programming languages. Two examples are Smalltalk (Deutsch and Schiffman, 1984; Goldberg and Robson, 1983) and Java (Lindholm and Yellin, 1997). The requirements for the SSL-VM were found to differ significantly from the virtual machines adopted in other programming language systems. The differences are:

- Each operation in the interface of a class can be used as an entry point.

[51] The prototype has been primarily built to test the efficacy of the MOOT methodology representation scheme. The prototype implements the MOOT core and persistent store as a single process, without the use of threads.

- There is only a single thread of control required in the SSL-VM. However multiple instances of the SSL-VM can be active at the same time processing messages from a common pool of SSL objects.
- Support for persistent objects and object-level locking is required.
- SSL is designed to be a specification language, and thus does not require many of the facilities of general-purpose languages.
- Existing virtual machines are too low-level, in terms of abstraction.

Appendix VII describes the implementation of the SSL-VM.

5.9.1 Requirements of the SSL Virtual Machine

The SSL-VM is required to provide support for a multi-user environment. The SSL-VM must ensure that separate updates are not being performed on the same object at the same time (i.e. that mutual exclusion is guaranteed, at the object level). This allows the possibility of a model being open for writing, but individual components in it being read-only (locked). A collorary of this is that the SSL-VM must be able to detect and resolve deadlock situations.

5.9.2 Architecture of the SSL Virtual Machine

The SSL Virtual Machine has a stack based architecture (Figure 5-11).

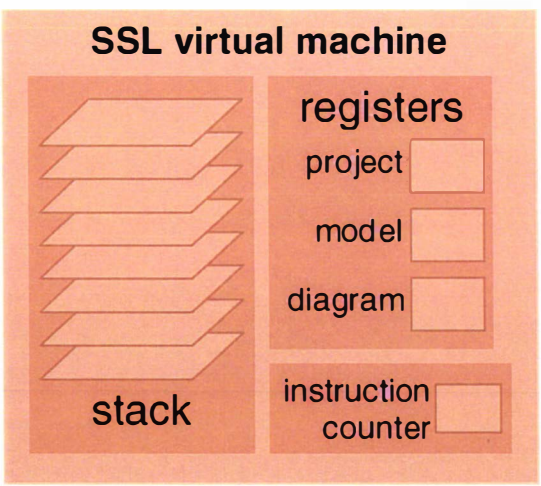


Figure 5-11 - Architecture of the SSL virtual machine

The stack stores message arguments and results, which allows nested message calls. It is also used to perform expression evaluation. All pushes onto the stack are balanced by

pops off the stack. The SSL-VM has an instruction counter which always points to the next SSL-BC instruction. The instruction counter is modified via instruction execution. It has three special registers (methodology registers) that each contain the SSL ID of the current project, model and diagram. These registers correspond to the SSL variables *current_project*, *current_model* and *current_diagram* respectively.

The SSL-VM provides explicit support for all the types available in SSL. All values are stored with the most significant byte first. The sizes for the SSL-VM types correspond to the sizes of equivalent types on the development architectures used (various 32-bit platforms). No final, long term, decisions about the sizes of these types have been made The SSL-VM types are shown in Table 5-1.

Type	Size	Comment
Boolean	One byte	Value is stored in the least significant bit
Integer	Four bytes	
Real	Eight bytes	
String	-	Null terminated sequence of bytes
Collection	Four bytes	The unique SSL ID of a collection
Iterator	Eight bytes	An SSL ID of a collection and an offset into the collection
Object reference	Four bytes	The unique SSL ID of an object

Table 5-1 SSL VM types

5.9.3 SSL Virtual Machine Instruction Set

The SSL-VM instruction set includes instructions for stack operations, arithmetic operations, string operations, comparison operations, conditional branching, issuing message calls and manipulating collections. Instructions may operate on the stack, local variables, message arguments, object attributes, the instruction counter or methodology registers.

There are 29 instructions (Table 5-2). A complete list of all SSL-BC instructions, with explanations is given in appendix V.

Instructions on the SSL-VM have an address mode and a type mode. The address mode specifies the location of any operands.

There are three address modes:

- Implicit (no operand for this instruction)
- Immediate (operand follows the instruction)
- Indirect (a reference to the operand follows the instruction)

The type mode is used to specify the data type that the instruction will operate on. The type modes supported on the SSL-VM include: boolean, integer, real, string, collection, iterator and object reference.

DBG	Debug print	RTN	Return from message
PSH	Push item onto stack	POP	Pop item from stack
ADD	Add	SUB	Subtract
MUL	Multiply	DIV	Divide
MOD	Integer modulus	NEG	Negation
CNV	Convert type	AND	Logical and
OR	Logical or	NOT	Logical negation
EQ	Equal	NEQ	Not equal
LSS	Less than	GRT	Greater than
BRT	Branch if true	BRF	Branch if false
MGS	Message send	CMG	Create message
SMG	Scoped message send	FNT	Front of collection
END	End of collection	ITM	Item from collection
PRJ	Current project	MDL	Current model
DGM	Current diagram		

Table 5-2 SSL-BC instruction set

5.9.4 Internal Representation of Classes, Objects and Methods

All SSL classes and SSL objects have a unique ID. The creation of the unique IDs is the role of the persistent store. The design of the internal representation of SSL classes and objects relies heavily on the proxy pattern (Gamma *et al.*, 1995). The SSL class proxy

encapsulates a reference to the Methodology interpreter class client and an SSL class ID. The SSL object proxy encapsulates a reference to the Methodology interpreter object client and an SSL object ID. All references to SSL objects and SSL classes are managed through proxies. The reference counting mechanism is implemented via the proxies.

Representation of SSL classes

Figure 5-12 shows the components of an SSL class. An SSL class consists of a unique ID, a description of its attributes, a vector of super-classes, a method lookup table and a method table.

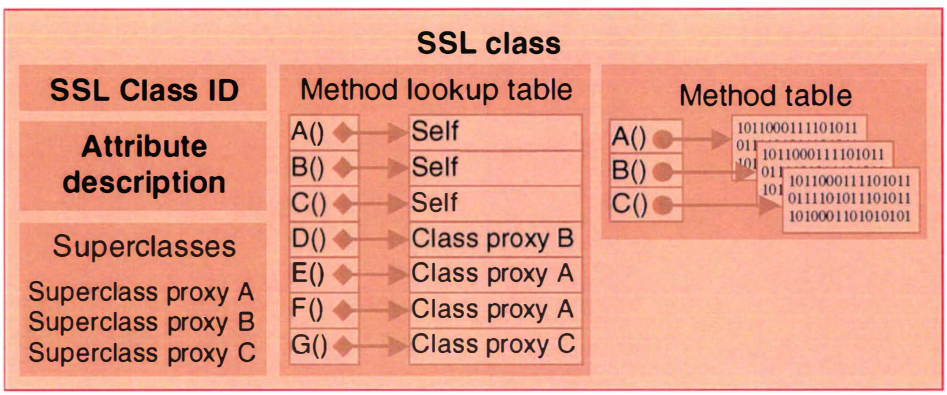


Figure 5-12 - SSL class

The SSL class ID corresponds to the unique fully qualified name of the class. An ID is used instead of the complete class name to minimise the memory required to store SSL classes and SSL class proxies (which also contain an SSL class ID). The attribute description defines the number of attributes of each type the class has. Each class also contains a vector of all direct and indirect super-classes of the class. The super-class vector is a flattened version of the inheritance lattice with respect to the class and is generated by the SSL compiler. This approach implies some redundancy but simplifies class instantiation and the implementation of late binding. The method lookup table is a map of operations and SSL class proxies. The table contains all operations (including those inherited from super classes) accessible from the class. Each operation is mapped to an SSL class proxy that identifies where that operation is implemented. Finally the method table is a map of operations and methods. This table contains all the operations implemented (i.e. the methods) in this class.

There are several advantages to this design:

- Accessing a leaf class in an inheritance hierarchy will not cause the retrieval of the entire inheritance hierarchy from the persistent store. The use of SSL class proxies ensures that a class will only be retrieved when it is actually needed.
- The method lookup table simplifies the implementation of late binding. The onus of building the lookup table, however, is on the compiler.

Representation of SSL Methods

Figure 5-13 shows the components of an SSL method. Each method has a name, arguments, local variables and a method body.

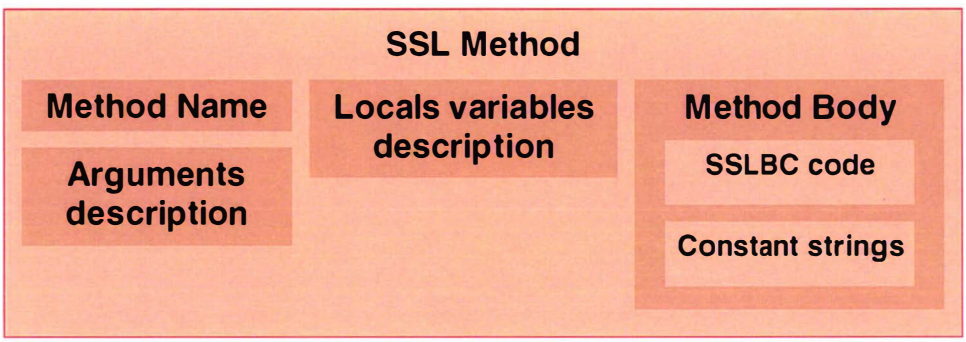


Figure 5-13 - SSL method

The method name is the same as the name of the operation that the method implements. The argument description defines the number of arguments of each type the method has. The local variable description defines the number of local variables of each type that the method uses. The method body consists of two parts: a block of SSL-BC code and a list of constant strings. The constant string list contains constant literal strings and class names that are used in the method. The block of SSL-BC code references a constant string in the string list via an absolute offset to the first character in the string.

Representation of SSL objects

Figure 5-14 shows the components of an SSL object. It consists of a unique ID, a proxy for its class, its state and object proxies corresponding to the super classes of its class. The unique ID defines the identity of the object. Creating new IDs is the responsibility of the persistent store. The class of the SSL object is represented by an SSL class proxy. The state of the object corresponds to the values of the attributes defined by the class of the

object. The super-state of the object is a vector of SSL object proxies that correspond to instances of the direct and indirect super classes of the class of the object.

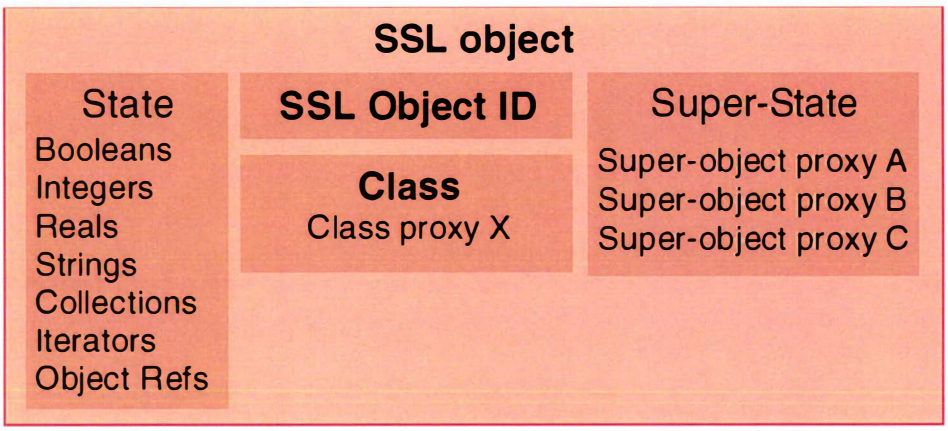


Figure 5-14 - SSL object

5.9.5 Processing Messages on the Virtual Machine

Figure 5-15 is a more detailed view of the methodology interpreter shown in Figure 3-10, Figure 3-11 and Figure 5-10. It depicts the steps taken and the components involved in processing a message with the methodology interpreter and SSL virtual machine. The components involved include the SSL interpreter, the SSL Virtual Machine, SSL classes and SSL objects. The reader is directed to appendix VII for a detailed description of the implementation of the SSL virtual machine.

The SSL interpreter is responsible for managing the execution of a method on the virtual machine. It does this by executing the SSL-BC instructions contained in a method body.

The steps taken to process a message are illustrated in Figure 5-15.

1. The SSL interpreter receives a message and a proxy to the SSL object to which the message has been sent. The interpreter obtains a reference to the SSL object via its proxy. It then pushes the SSL ID of the object onto the stack of the Virtual machine (this is the implicit 'self' argument).
2. The SSL interpreter requests the object to accept the message. The result of the object accepting the message will be: a) a method suitable for processing the message and b) the state of the object. The state of the object is part of the context the message will be processed in.

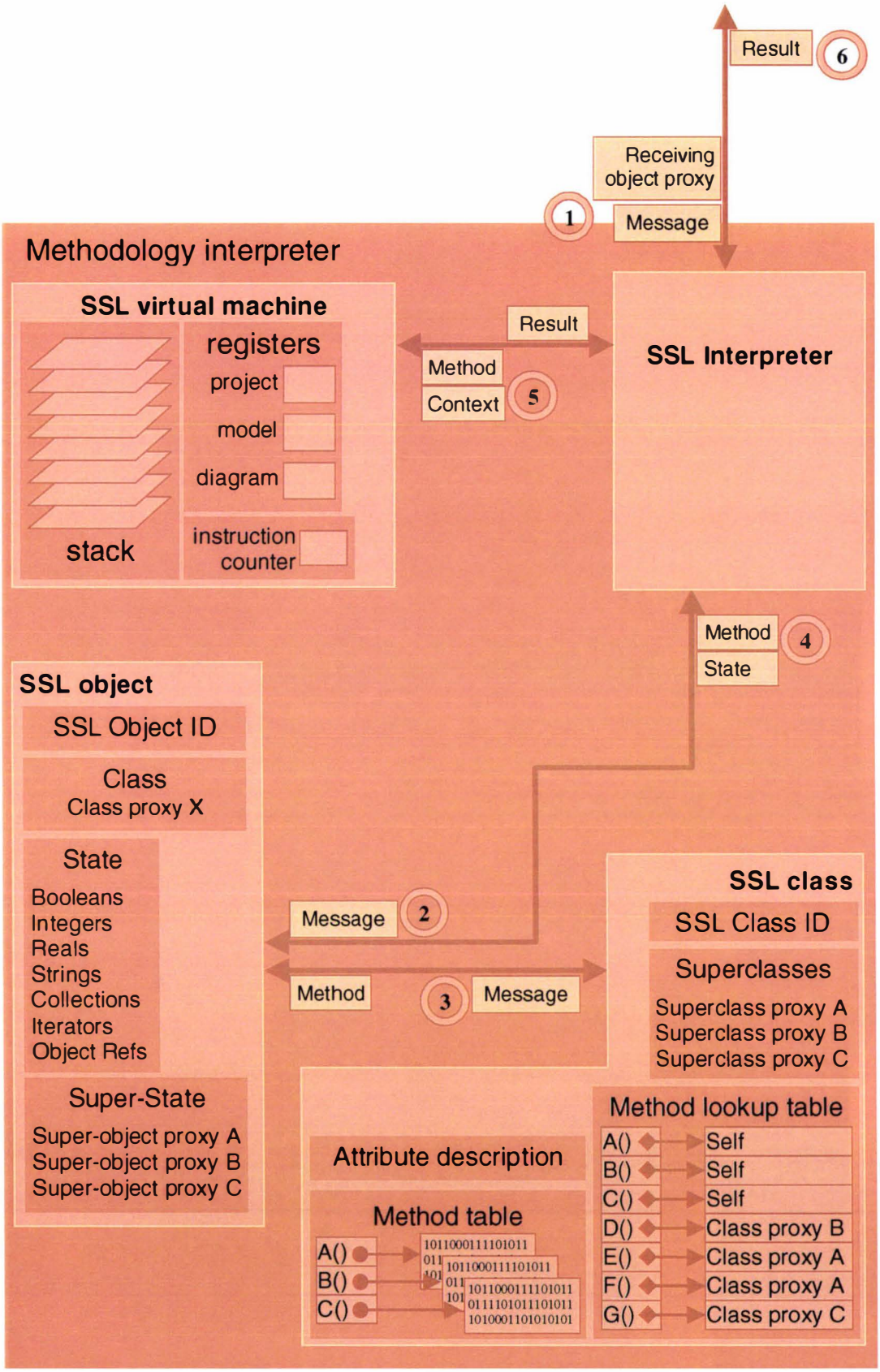


Figure 5-15 - Processing messages on the SSL-VM

3. In order to find an appropriate method the object asks its class to bind the message to an appropriate method. The object first obtains a reference to its class via the SSL class proxy it contains. It then delegates the method binding responsibility to its class. The class uses its method lookup table to search for a proxy to the class that defines an appropriate method. If the proxy refers to a super class then the method will be fetched from the super-class method table. Otherwise the method is fetched from the local method table. Note that the code generation and type-checking steps of the compilation process are responsible for ensuring that there will always be a method to bind to a message.

4. The SSL interpreter receives the method and the state of the object as a result of steps 2 and 3. It builds the context within which the method will be executed. This context is composed of three parts: a) the actual message arguments b) space for any local variables the method requires and c) the state of the object. The method and context are then used to execute the method.

5. The SSL-VM performs a fetch-decode-execute cycle where each of the instructions in the method body is executed on the virtual machine in turn. These instructions will cause changes in the Virtual machine instruction counter, stack, and in the context.

6. The method execution finishes once a RTN SSL-BC instruction is executed. The SSL interpreter then evaluates the object constraint to see if the object is still valid. If the constraint is satisfied then the object is updated with the new state that is contained in the context. Finally the result of the message is returned to the tool manager.

5.10 Summary

This chapter has described the development of SSL. The goals of SSL and the M●OT meta-model (the facilities of which SSL provides) were discussed.

SSL is an object-orientated language that supports a subset of the facilities of a general purpose programming language. It is a statically type checked language that provides clean separation between 'class interface' and 'class implementation'. SSL supports dynamic binding, multiple inheritance, built-in primitive types, polymorphic collection and iterator types and provides a module system.

Chapter 6

The Core Knowledge Base and Generic Object Orientated Knowledge Base

Myth #9: Software re-use will just happen.

Tracz 1988

6.1 Introduction

This chapter presents two libraries of re-usable methodology semantic description components that have been developed as part of this research. The libraries are called the ***C***ore ***K***nowledge ***B***ase (CKB) and the ***G***eneric ***O***bject ***O***rientated ***K***nowledge ***B***ase (GOOKB). The primary objective of these two libraries is to provide a pool of re-usable components that methodology semantic descriptions will be defined as extensions of. There are two major goals to be realised by this approach. Firstly, the effort required to define new methodologies is reduced. Secondly, all methodology definitions share a common sub-set, which provides distinct advantages in terms of reasoning about the methodologies and re-using software engineering results.

6.2 Context of the Core Knowledge Base and the Generic Object Orientated Knowledge Base

The information processed by MOOT can be classified into three groups:

- Meta-descriptions of 'methodology' and software engineering approaches (meta-models of methodology, object-orientated development, information engineering etc.).
- Descriptions of methodologies built using the methodology development sub-system (specific methodologies).
- Descriptions of software built using the CASE tool sub-system (user projects).

These categories of information are arranged in three tiers, as shown in Figure 6-1.

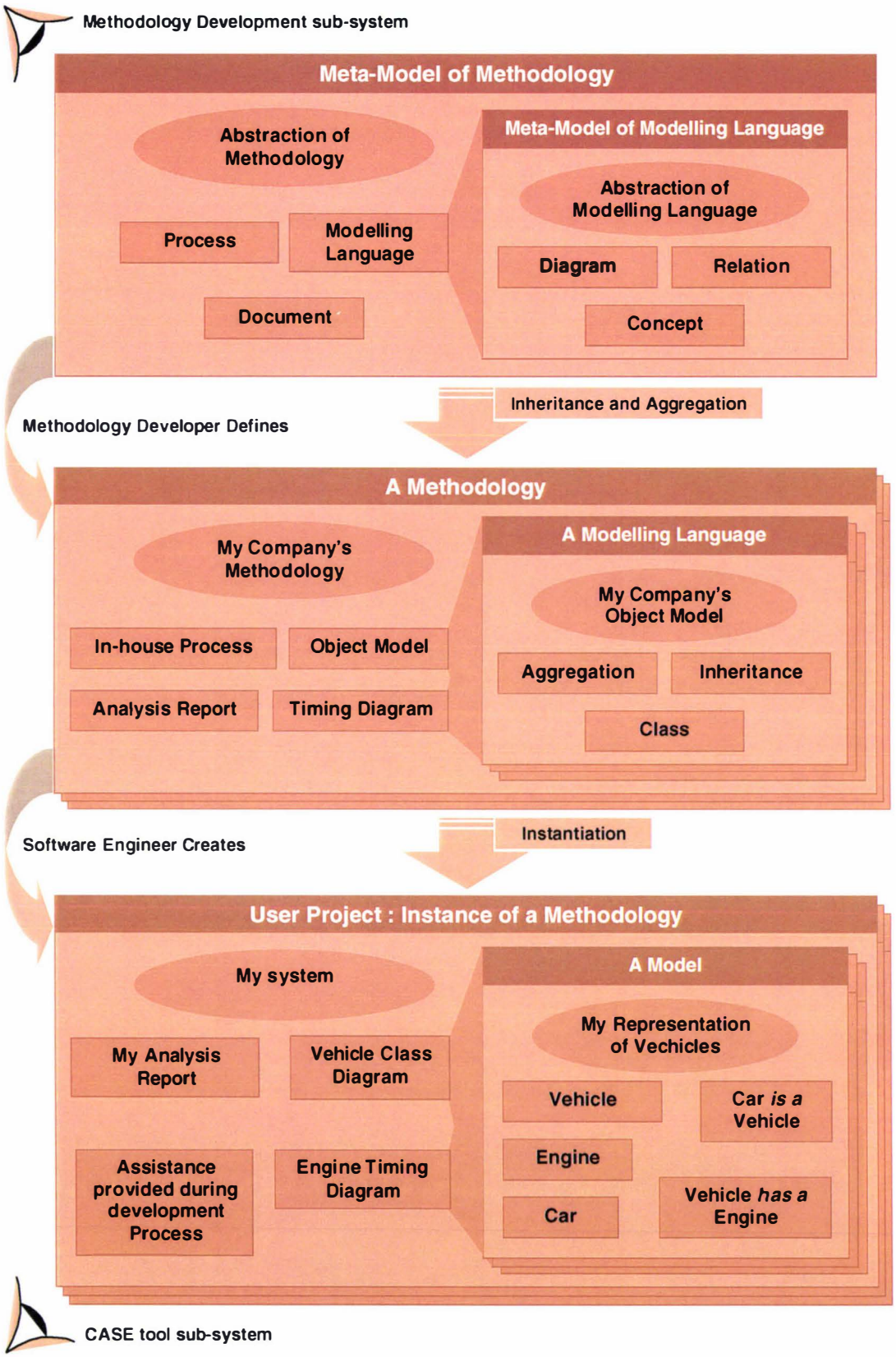


Figure 6-1 - The three tier structure of the information processed by MOOT

The methodology engineer's view is from the top where methodologies in tier two are defined in terms of tier one. This view is provided by MOOT's methodology engineering sub-system. The software engineer's view is from the bottom where a user project in tier three is defined in terms of tier two. This view is provided by the CASE tool sub-system.

In the top tier of the structure depicted in Figure 6-1 is the meta-model of methodology described by the Core Knowledge Base (CKB). The classes at this level define an abstraction of methodologies. The Generic Object Orientated Knowledge Base (GOOKB), which is an extension of the CKB for object-orientated methodologies, is also defined in tier one[52].

Methodology engineers create their own methodologies in tier two by sub-classing SSL classes within tier one. They may also inherit from other classes previously defined in tier two. All methodologies in tier two have classes from the CKB defined in tier one in common as they are all directly or indirectly defined in terms of it.

Software engineers build descriptions of software artefacts at tier three by instantiating the SSL classes defined in tier two. Thus the semantic content of a user project consists of a collection of SSL objects.

6.3 Development of the Core Knowledge Base

The Core Knowledge Base has been designed by adopting a meta-modelling approach.

6.3.1 Meta-Model of Methodology

Each methodology has a collection of modelling languages, documents and provides a process.

A software engineer uses the modelling languages supported by a methodology to express and investigate the relevant abstractions in the problem domain. Various modelling languages are available to the software engineer to use. Each modelling language has an associated method, which at least is 'do not break the rules of the modelling language.' It may also include quality guidelines and direction for how a modelling language is best

[52] Other software engineering approaches can also be meta-modelled and supported in tier one. This work is discussed in section 9.4 - Future Work.

applied to build models of software. The method subsumes the guidelines, suggestions and strategies that may be contained in the description of the modelling language. The evolution of software development methodologies has resulted in many methodologies providing the same modelling language with variation in interpretation, application or appearance (Henderson-Sellers, 1996).

Documents are produced during the process of applying a methodology to a particular problem. These documents may vary in terms of scope, content and their intended audience. The structure of these documents is not necessarily defined by a particular methodology. A Company may choose to adopt an in-house standard for the documentation or may choose to use a more widely used document standard.

A software engineering process is a suggested framework that the software developer applies whilst building a software artefact. The process may define the order with which models of the software are derived and may also provide quality guidelines. Ideally the process provides a systematic approach to constructing models. It may include guidelines regarding the suitability of modelling languages for particular tasks and suggestions and strategies for problem solving using the methodology. The process of a methodology is more than a suggested software development life-cycle. It also subsumes the guidelines, suggestions and strategies that may be contained in the description of the methodology.

Figure 6-2 shows a meta-model of methodology. Each *Methodology* has a *Process*, zero or more *Documents* and one or more *Modelling Languages*. Each *Process*, *Document* and *Modelling Language* may be used in more than one *Methodology*.

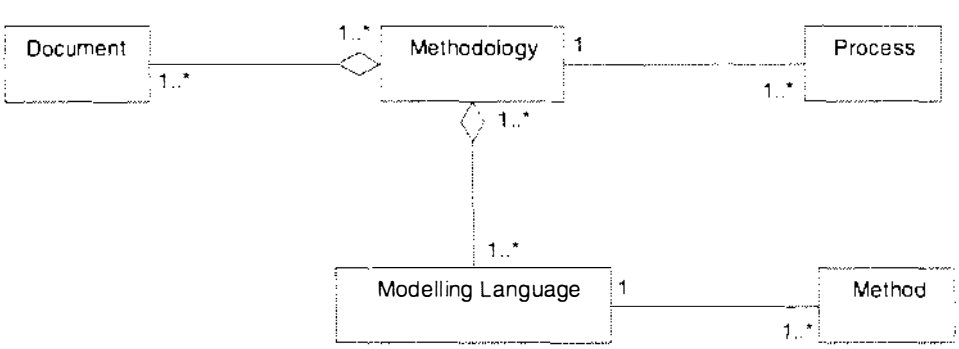


Figure 6-2 Methodology meta-model

The relation between *Methodology* and *Process* and between *Modelling Language* and *Method* has been modelled with an association in Figure 6-2. Henderson-Sellers (1996) notes "We

have seen that, on the one hand, a process has three constituent parts, one of which is methodology and, on the other hand, that a methodology must contain a process. These two relationships between methodology and process are, on the face of it, contradictory. Which is right? Well they both are!" It is expected that software engineering processes may be attached to more than one methodology and methods to more than one modelling language. Moreover it is possible that a modelling language may be used in association with a different method, when used to model different classes of problem. For example the method used to apply a state transition diagram in the context of Booch object-orientated design, is likely to be different to the method used to apply the same modelling language as a representational basis for Computer Assisted Instruction systems (Feylock, 1977). The research to date has yet to consider meta-modelling of software process and method in detail. Such research closely is related to the cognitive support of software engineering discussed in section 9.4 - Future Work.

Models do not exist in isolation. Different models can be used to investigate different dimensions of a problem. There may be relations between parts of a model and relations between different models in a software engineering project. For example a package on an UML class diagram may be exploded into a separate UML class diagram. Transitions correspond to the possible paths of navigation in a methodology. The classes *Intra-Model Transition* and *Inter-Model Transition* in Figure 6-3 represent these navigation paths.

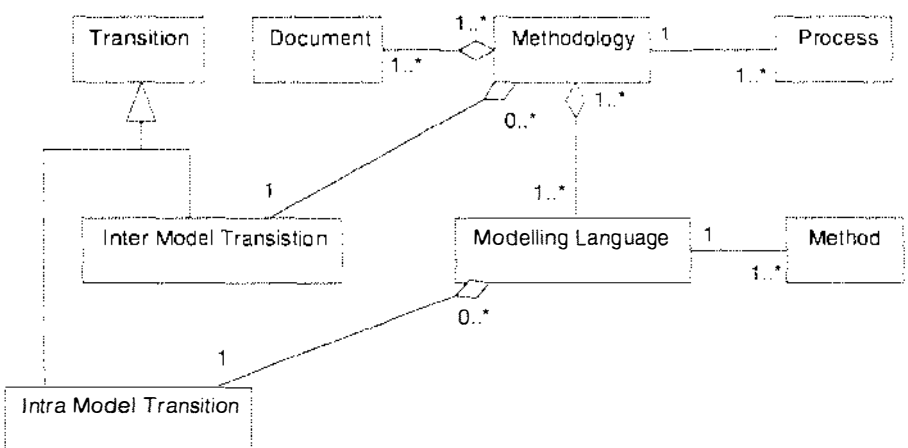


Figure 6-3 - Transitions

6.3.2 Meta-Model of Modelling Language

A modelling language provides one or more diagrams (for example a DFD model consists of a context diagram, DFD diagrams as well as process specifications). Diagrams

may contain zero or more model elements. Specialised model elements include concept (such as a class), relation (such as an association between two classes) and composite (such as a Coad and Yourdon Subject Area). Model elements must exist in at least one diagram but may be used in several others. Figure 6-4 shows a meta-model of modelling language.

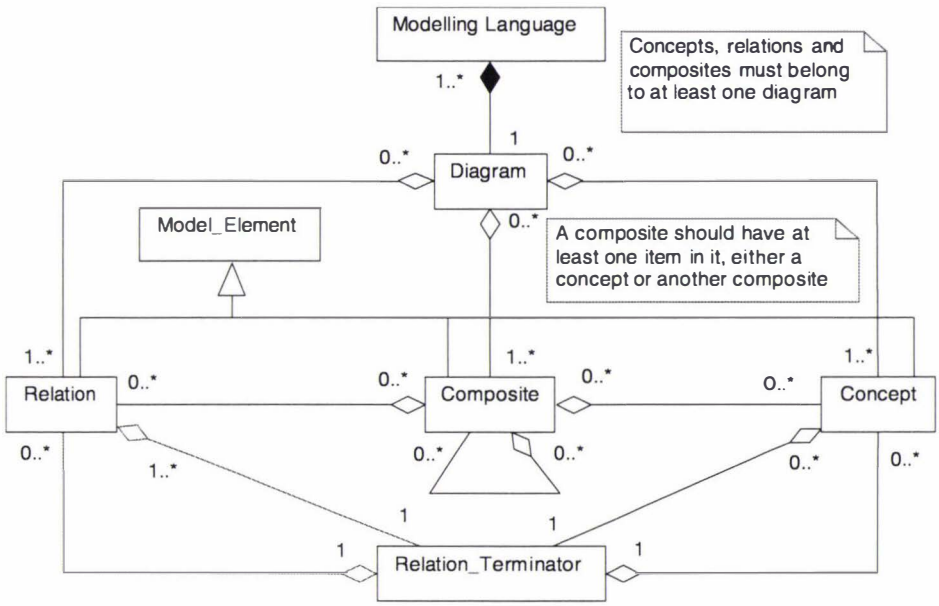


Figure 6-4 - Meta-model of modelling language

A modelling language consists of one or more diagrams. Each diagram is part of one modelling language. An instance of *Diagram* contains a collection of instances of the modelling elements that are supported by the modelling language. A composite is a group of concepts and relations and other composites. Each modelling element must be used in at least one diagram.

Relations have been represented with two classes, *Relation* and *Relation Terminator* in Figure 6-4. The *Relation Terminator* class models the end points of a relation. It may be sub-classed to implement specialist roles (e.g. whole, part, message sender, message receiver etc). Each instance of *Relation Terminator* knows the relation it is part of and the concept it attaches itself to. A *Relation* object contains instances of *Relation Terminator* for each endpoint of the relation. Figure 6-5 illustrates the structures involved in representing relations with an example that uses the classes in Figure 6-4.

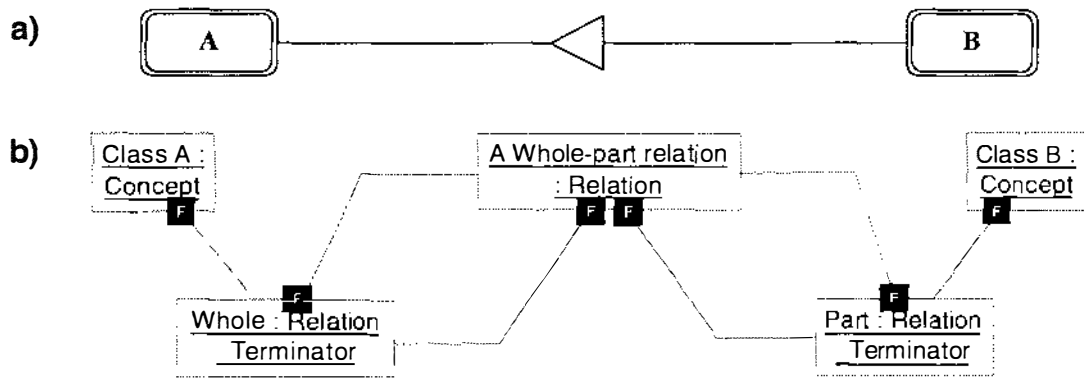


Figure 6-5 Representing a whole-part relation

Figure 6-5 shows a whole-part relation expressed using Coad and Yourdon. This is a directed binary relation where one class (*A* in Figure 6-5) takes the role of 'whole' and the other class (*B* in Figure 6-5) takes the role of 'part'. The two classes are represented by instances of *Concept*. Each end of the relation is represented by an instance of *Relation Terminator*. The whole-part relation itself is represented by an instance of *Relation*. Figure 6-5 describes the whole-part relation only in terms of the classes in Figure 6-4. In practice descendants of classes defined in the GOOKB, or in extensions of the GOOKB, would be used to represent such a relation. The object structure, however, would be the same.

Figure 6-6 shows how a simple diagram that represents the composite pattern (Gamma *et al.*, 1995) could be represented with instances of the classes in Figure 6-4.

The Coad and Yourdon class diagram in Figure 6-6 (a) involves three concepts (the classes *A*, *B* and *C*) and three relations (an inheritance relation between class *A* and class *B*, an inheritance relation between class *A* and *C*, and a whole-part relation between class *A* and *C*). All of the concepts and relations belong to a diagram, which in turn is part of a 'Coad and Yourdon class diagram' model.

Figure 6-6 (b)(i) shows the objects involved in representing the inheritance relation between class *A* and class *B* in Figure 6-6 (a). Each end of the relation is represented with an instance of *Relation Terminator*[53]. The inheritance relation itself is represented with an instance of *Relation*. Figure 6-6 (b)(ii) shows a similar collection of objects that represents the inheritance relation between class *A* and class *C*. The object structure highlighted in Figure 6-6 (b)(iii) represents the whole-part relation between class *A* and class *C*.

[53] The names of classes in subsequent diagrams relating to the CKB and GOOKB will be prefixed with the name of the knowledge base they originate from

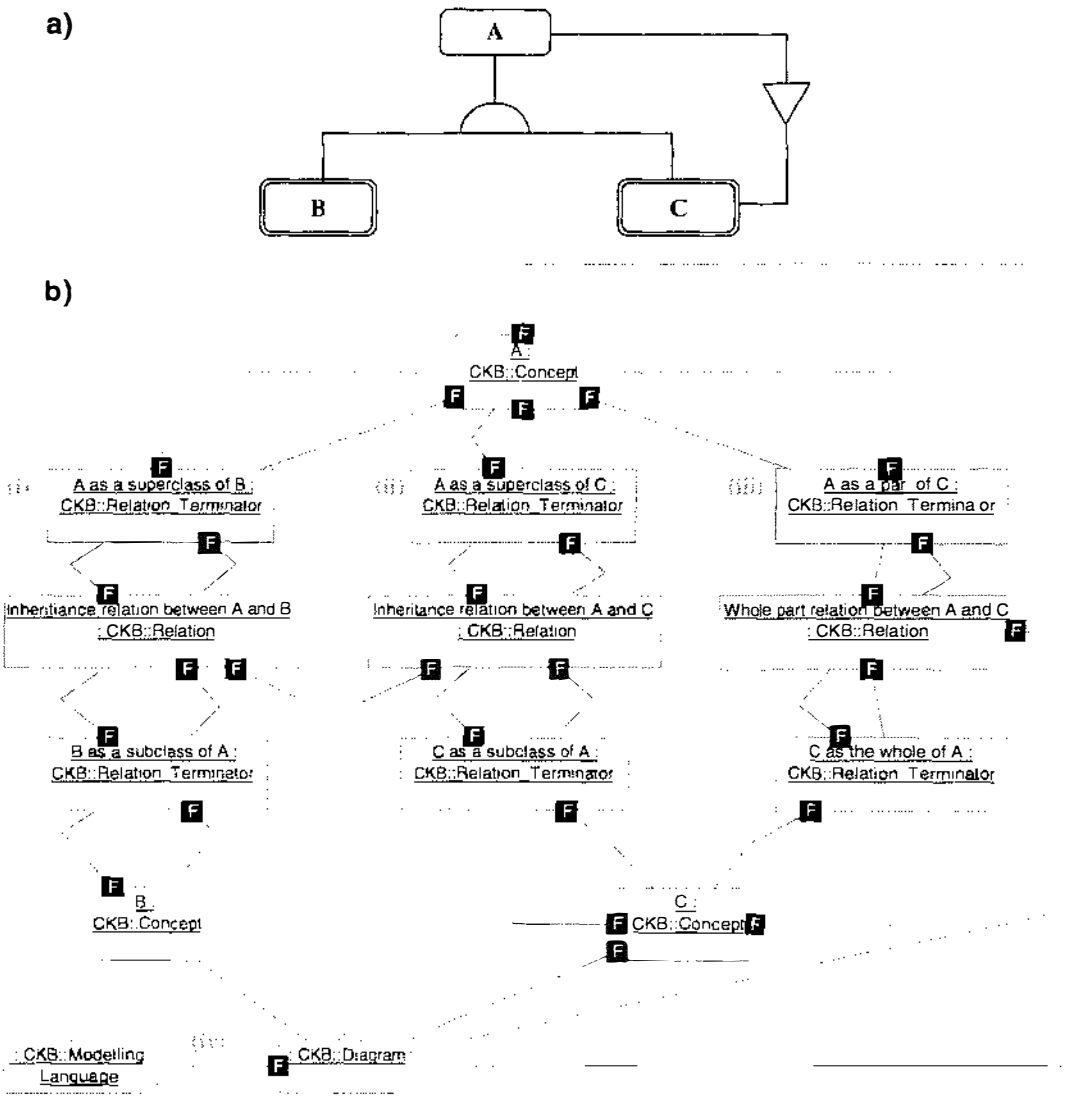


Figure 6-6 Representing a class diagram with instances of classes from the CKB

The collection of *Relation Terminator* objects that is maintained by each instance of the *Concept* class in Figure 6-6 corresponds to the roles that each concept plays in the diagram of Figure 6-6 (a). For example, object *A* in Figure 6-6 (b) has links to three *Relation Terminator* objects. They represent the roles class *A* plays in the diagram of Figure 6-6 (a) (*A* as a *super-class of B*, *A* as a *super-class of C*, *A* as a *part of C*).

The Coad and Yourdon 'class diagram' model of Figure 6-6 (a) is represented by the two objects in Figure 6-6 (b)(iv). The instance of *Modelling Language* represents the whole model. This object has a link to a single instance of *Diagram*. The *Diagram* object maintains links to the three *Concept* objects, and the three *Directed Binary Relation* objects.

Figure 6-7 shows the meta-model of modelling language (Figure 6-4) in more detail. The Critic class in Figure 6-7 is discussed in section 6.3.3.

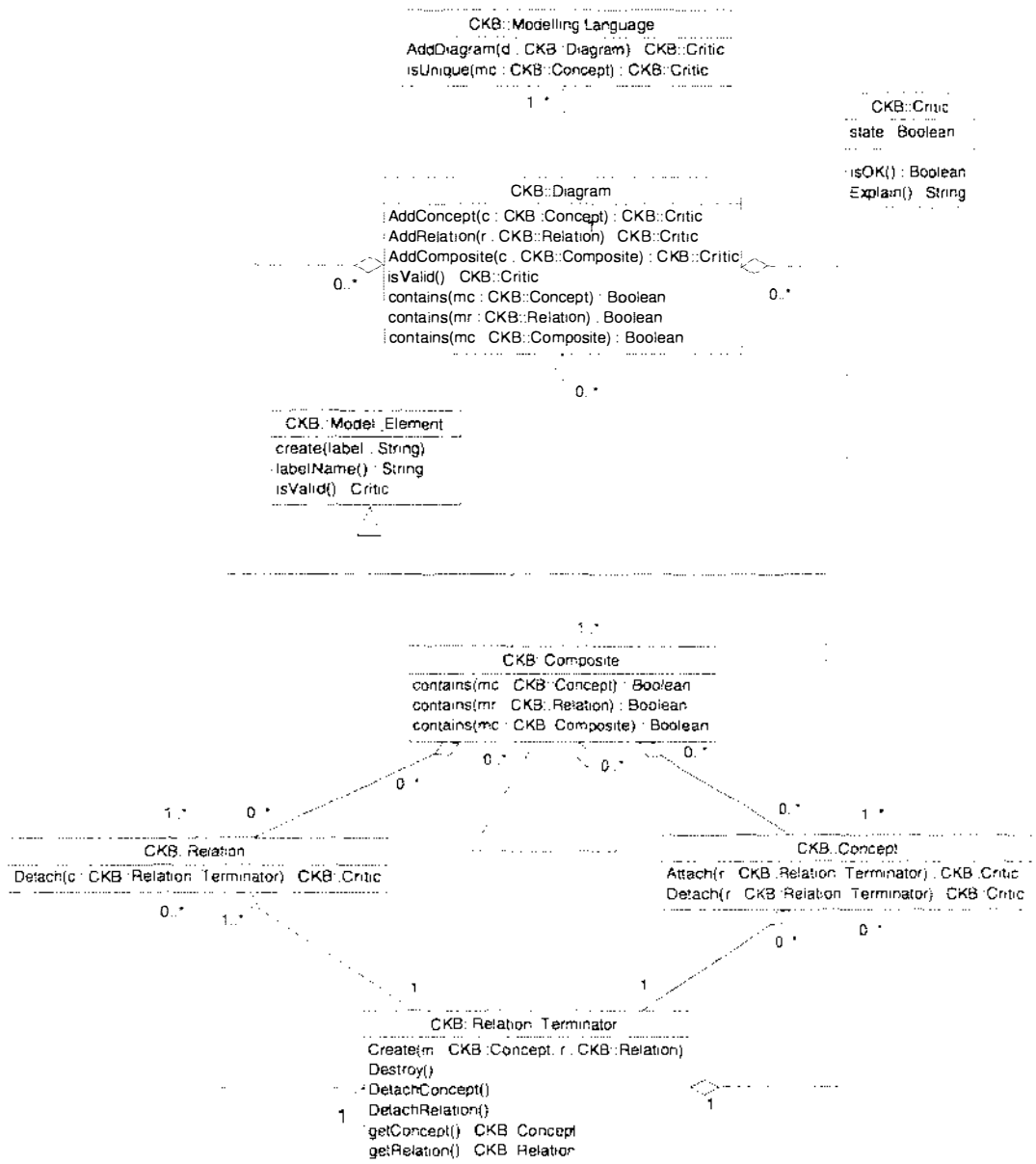


Figure 6-7 Detailed meta-model of modelling language

The meta-model in Figure 6-7 has been specialised in Figure 6-8 with additional classes that represent the various types of relations that may exist. They have been classified in terms of the number of concepts involved in the relation and the direction of the relation.

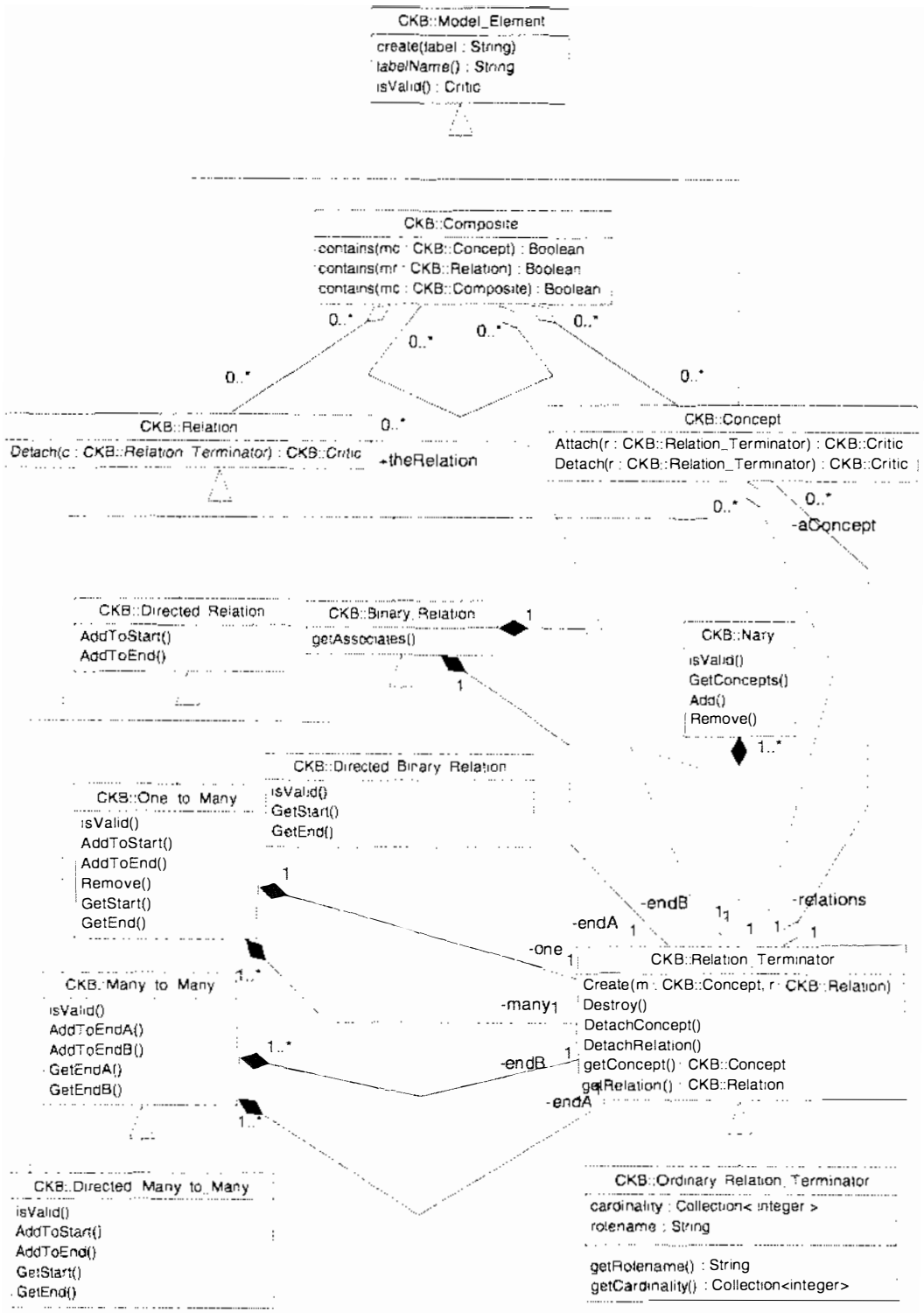


Figure 6-8 : Extended meta model of modelling language

The relations supported in Figure 6-8 include *Binary Relation*, *Directed Binary Relation*, *One to Many Relation*, *Many to Many Relation*, *Directed Many to Many Relation* and *Nary Relation*. *Relation Terminator* has been sub-classed to provide a terminator that additionally provides a role name and cardinality. Figure 6-9 shows the classes in the Core Knowledge Base.

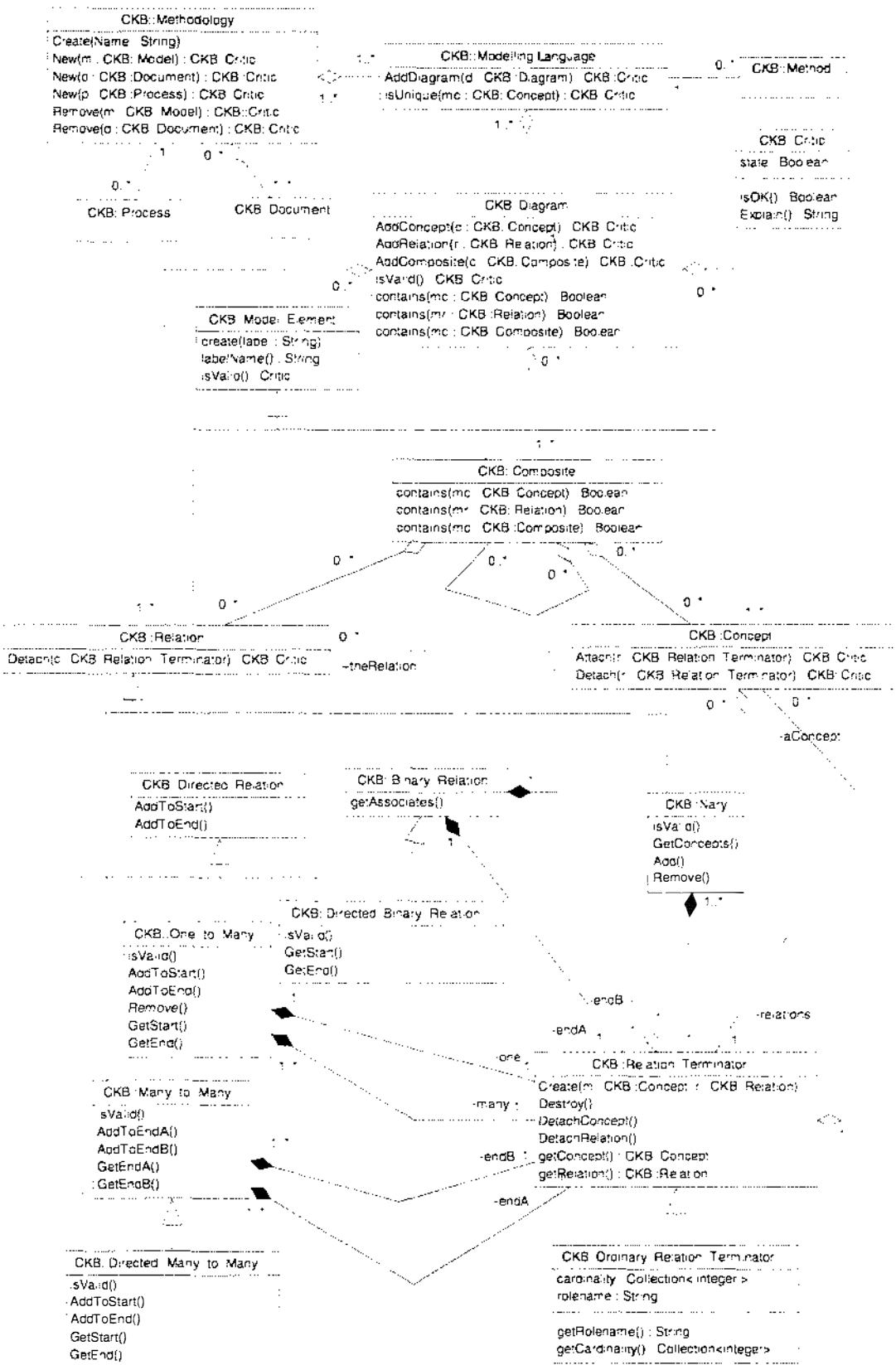


Figure 6-9 - Core Knowledge Base

6.3.3 Handling Exceptional Situations

The user performs many logical actions whilst applying the methodology process to build a description of their software artefact. These actions are related to tasks such as creating and updating models, searching the current and existing projects, checking their work for correctness, completeness, quality and so on. The *situation* in Figure 6-10 corresponds to the response to a request made by the user.

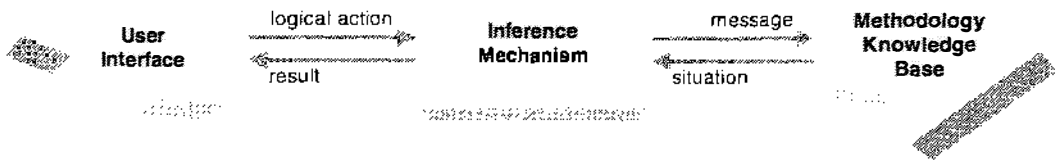


Figure 6-10 - Situations

What occurs in response to the generation of a situation is under the control of the inference mechanism and is a function of the situation itself. It may be that a simple warning is passed to the user as a result or that some form of auto-correction is applied. For example consider the situation where a class is created with a name that is in use by another class. It is up to the methodology engineer to allow or disallow this situation. They may decide that this is a fatal error and disallow it. Or they may simply change the requested name automatically and report the situation to the user.

Some of the situations and responses that might occur include:

- reporting an erroneous state to the user
- suggesting corrective action to the user
- performing auto-correction
- providing comments about the suitability of the users project, model or model element
- providing a link to an aspect of the methodology process or the method of a modelling language

The inference mechanism in Figure 6-10 is outside the scope of the definition of methodologies that existing meta-CASE tools provide. The ARGO project (Robbins *et al.*, 1996, 1997, 1998) has considered some of these issues with the development of a

methodology dependent CASE tool called ARGO/UML. The ARGO project refers to the facilities described here as 'cognitive support' for software engineers. Implementing these facilities in the CKB was relegated to future work once the existence of the ARGO/UML project was identified.

Simple explanation facilities have been implemented with the Critic class in the CKB. The majority of the operations in the CKB return a result that is an instance of the *Critic* class. Critics are used to signal the result of an operation in the CKB. The Critic class hierarchy given in Figure 6-11 is an instance of the composite pattern (Gamma *et al.*, 1995).

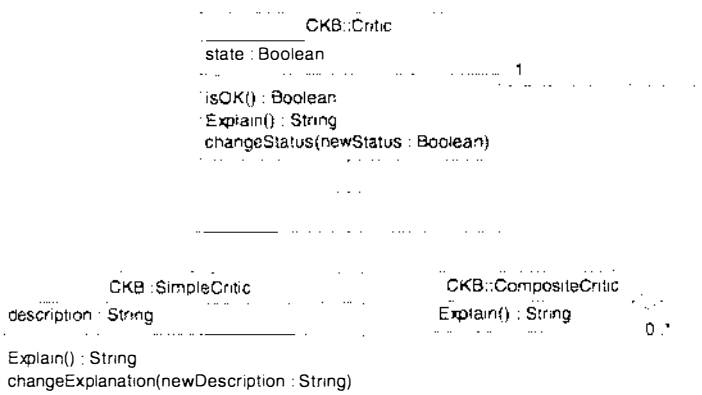


Figure 6-11 - Critics

The abstract class *Critic* in Figure 6-11 encapsulates a boolean flag that signals the success or failure of an operation. The *SimpleCritic* class extends the Critic class with a string that contains an explanation. The explanation could be the reason an operation was unsuccessful or perhaps some feedback relating to the success of an operation. The *CompositeCritic* class maintains a collection of other critic objects. Composite critics are used in situations where the success of an operation is dependent on the result of several sub-operations. The explanation given by a composite critic is the concatenation of its component critic's explanations.

6.4 Development of the Generic Object Orientated Knowledge Base

This ***G***eneric ***O***bject ***O***rientated ***K***nowledge ***B***ase (GOOKB) is an extension of the CKB that contains classes that represent concepts germane to object-orientated methodologies.

All object-orientated methodologies encompass the concepts of encapsulation, information hiding and hierarchical decomposition, and are founded on the concepts of

classes, objects, inheritance, message passing and polymorphism. The nature of the object model is consistent across what is traditionally described as the four phases of the software development life-cycle: analysis, design, implementation and maintenance. There is a basic core of commonality between all object-orientated analysis and design methodologies due to this consistency even though each methodology has its own variations in its expression of the 'object model.'

> "The OMG Object Model defines a core set of requirements that must be supported in any system that complies with the Object Model standard. The set of required capabilities is called the 'Core Object Model' "
> *(QED, 1992)*

This statement indicates that object-orientated methodologies have properties that are generic and can be modelled with the generic object-orientated knowledge base.

The method used to derive the classes in the GOOKB was designed by considering existing comparisons of object-orientated methodologies. The objective was to identify potential methods for the comparative analysis and subsequent meta-modelling of object-orientated methodologies.

This research pre-dates the COMMA project, the development of UML and submissions to the OMG OA&DF, all of which have a similar objective – understanding the common aspects of object-orientated methodologies.

6.4.1 Object-Orientated Methodology Comparisons

Many object-orientated methodology comparisons have been conducted in the past. Notable research includes (Arnold *et al.*, 1991; Brinkkemper *et al.*, 1998; de Champeaux and Faure, 1992; Cribbs *et al.*, 1992; Fichman and Kemerer, 1992; Fung *et al.*, 1997; Henderson-Sellers and Bulthuis, 1996a, b, 1997; Henderson-Sellers and Firesmith, 1997a; Hong *et al.*, 1993; Hutt, 1994; Loy, 199[illegible]; Monarchi and Puhr, 1992; Object Agency, 1998; Rumbaugh *et al.*, 1991; Sharble and Cohen, 1993; Taylor, 1998; van den Goor *et al.*, 1992; Wirfs-Brock and Johnson, 199[illegible]; Yourdon and Argila, 1996).

Existing methodology comparisons were analysed, evaluated and contrasted. A taxonomy of object-orientated methodology comparisons was subsequently derived as a result (Dasari *et al.*, 1995; Mehandjiska *et al.*, 1996a-c) and is presented in Figure 6-12.

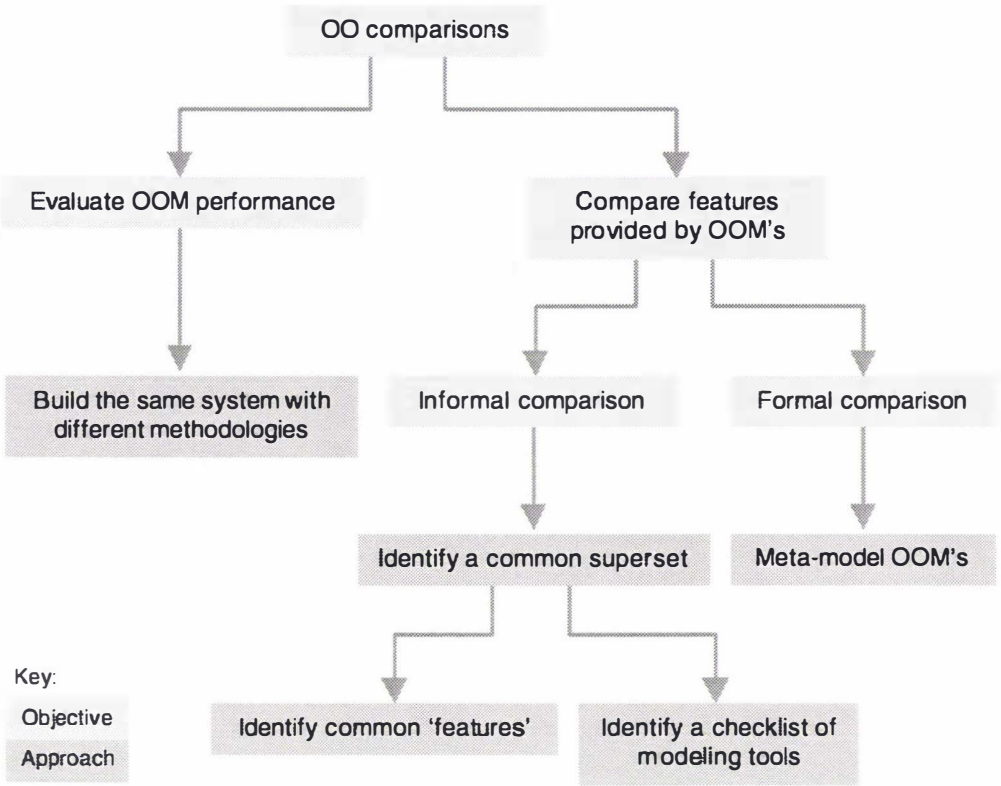


Figure 6-12 - Taxonomy of object-orientated methodology comparisons

The purpose of methodology comparisons that have been conducted in the past was to either evaluate the performance of different methodologies or to compare and contrast their features. Attempts were also made to establish a common understanding of object technology.

Typically the performance of an object-orientated methodology has been evaluated by modelling a single problem with multiple methodologies and comparing the resulting analysis, design and implementation models.

The features provided by object-orientated methodologies have been compared formally by meta-modelling or informally by identifying common modelling tools and features. Comparisons of object-orientated methodologies have been classified according to the approach taken to the comparison and the objectives of the comparison.

The consideration of object-orientated methodology comparisons has highlighted the following issues:

- Each methodology has its own set of definitions, processes, notations and tools. Concepts may be named differently for each methodology and the richness of support of a concept may vary between methodologies.

- Comparisons that involve a simple matching of 'terms' are inaccurate as many methodologies use the same 'term' but with distinct interpretation. Ideally a methodology comparison should involve matching the *definition* of 'terms' as many methodologies support the same concepts with different names.

- Comparing methodologies 'two by two' is time consuming, as the number of discrete methodology comparisons (Nc) for a set of Nm methodologies is large (Figure 6-13). For example, 1225 discrete comparisons would be required to evaluate the fifty (Muller, 1997) object-orientated methodologies that existed by 1995.

$$Nc = \sum_{n=1}^{n=Nm}(n-1) \quad \text{or} \quad Nc = \frac{Nm}{2}(Nm-1)$$

Figure 6-13 Number of comparisons for Nm methodologies

- Researchers evaluating methodologies are often biased in their review results as they attempt to evaluate methodologies within the context of their own development background.

- Many comparisons initially involve identifying a generic list of properties that a methodology should support. Methodologies are then compared to this generic list of properties. Different researchers may choose a different set of representative properties and may use different definitions for those properties. Different comparison results are produced due to the difference in the choice of representative concepts and definitions.

6.4.2 Method used to Design the Generic Object Orientated Knowledge Base

Based on the review of the methodology comparisons it was decided that the method used to identify the components in the generic Object Orientated Knowledge Base would:

- Use a formal meta-modelling approach. This is an obvious decision given the desired result is a meta-model.

- Use a small sub-set of methodologies. Whilst meta-modelling every object-orientated methodology would produce an appropriate meta-model it was decided that a small subset would be sufficient. Existing comparisons support the view that there is a high degree of similarity in the interpretation of object-orientated principles amongst object-orientated methodologies.

- Be carried out relative to a set of methodology independent object-orientated terms. This was done to avoid a two-by-two approach to meta-modelling. The terms chosen constitute a first-guess at the components expected in the meta-model.

The method adopted is:

1. Identify candidate generic concepts defined by the Object Management Group (QED, 1992; OMG, 1991, 1992).

2. Identify equivalent OMG concepts in a subset of object-orientated methodologies. Each methodology defines and uses distinct terms for the fundamental object-orientated concepts. The concepts identified by the OMG provide a consistent vocabulary that is not methodology specific.

3. Model the identified concepts in a single homogenous object-orientated meta-model.

4. Identify the portion of the meta-model that is not methodology specific.

5. Re-define the generic portion of the meta-model as an extension of the Core Knowledge Base.

6. Implement the GOOKB in SSL.

In-house experience and knowledge of object-orientated methodologies, especially those that were new (such as UML and OPEN) and were not considered in the literature dealing with methodology comparisons, was used throughout the process.

The results of the meta-modelling work in steps 1 – 4 is presented in (Mehandjiska *et al.*, 1996a-c). The remainder of this section will cover the last two steps and present the initial design of the GOOKB as an extension of the CKB.

6.4.3 Generic Object Orientated Knowledge Base

Figure 6-14 shows the classes in the GOOKB that are used to represent the object-orientated concepts of class and object.

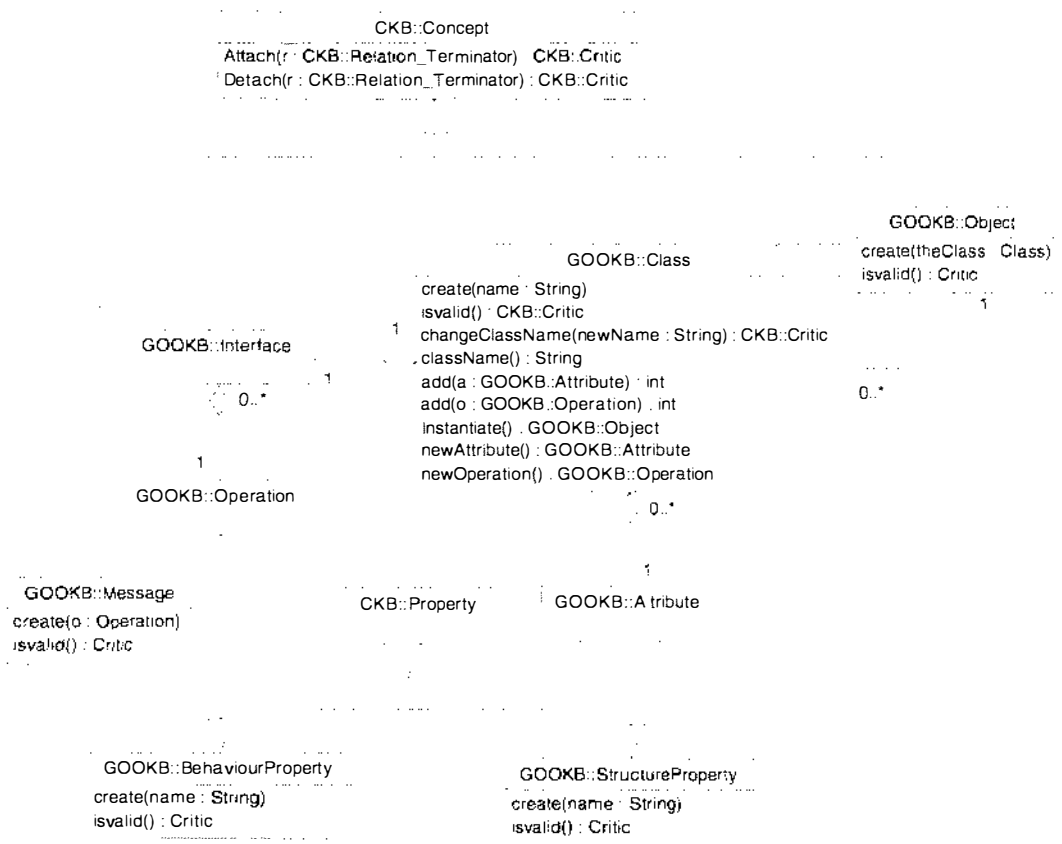


Figure 6-14 - Representing classes and objects

The GOOKB introduces four sub-classes of *Concept*. These are *Object*, *Class*, *Interface* and *Message* (Figure 6-14). It also introduces two direct sub-classes of *Property*. These are *BehaviourProperty* and *StructureProperty*. *Operation* and *Attribute* are defined as sub-classes of *BehaviourProperty* and *StructureProperty* respectively. An *Interface* consists of a collection of

Operations. A *Class* has a single *Interface* and has zero or more *Attributes.* A *Class* may have zero or more instances. Each *Object* is an instance of a single *Class.*

Figure 6-15 shows the classes in the GOOKB that are used to represent inheritance, aggregation and association.

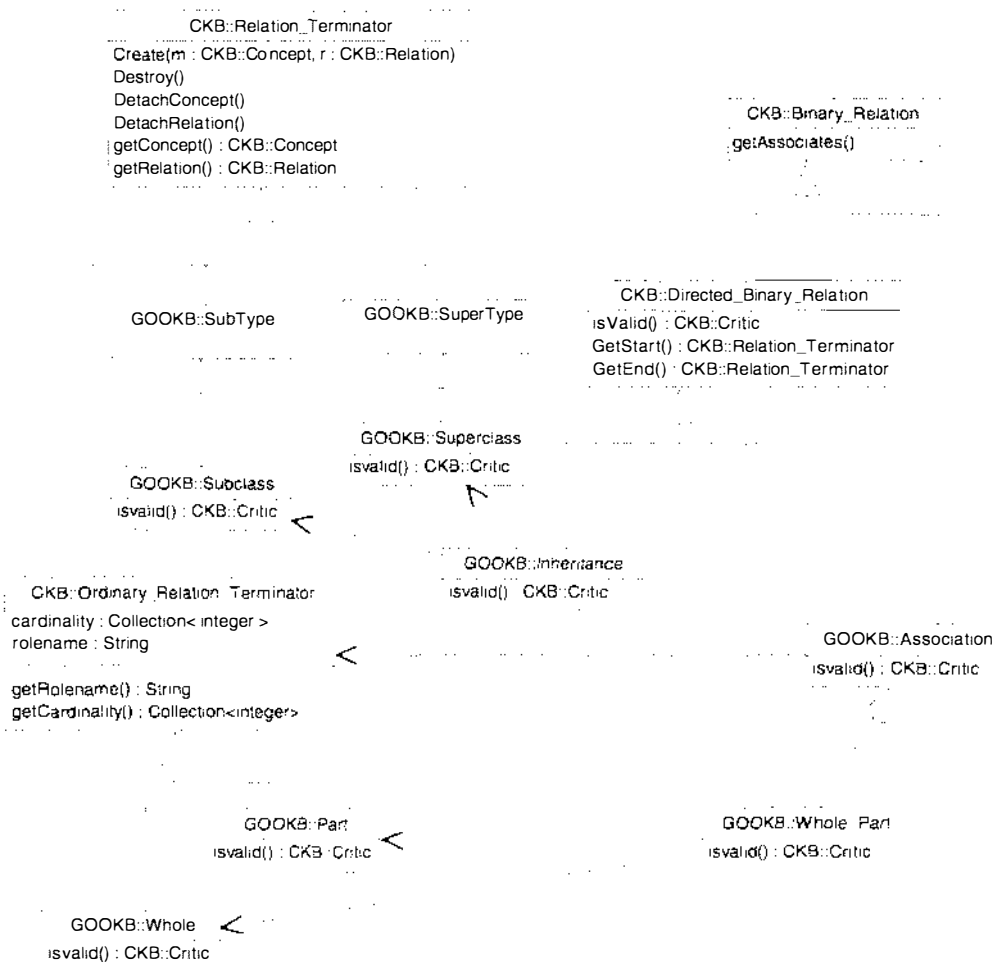


Figure 6-15 - Representing object orientated relations

Association is defined as a sub-class of *Binary Relation.* Each end of an association is an instance of *Ordinary Relation Terminator* (or an instance of a sub-class). The *Ordinary Relation Terminator* defines a role name and cardinality. *Inheritance* is defined as a type of *Directed Binary Relation. Subclass* and *Superclass* represent the terminuses of an inheritance relation. Aggregation has been defined as a *Directed Binary Relation* and as a specialised *association. Whole* and *Part* represent the terminuses of an aggregation relation.

Figure 6-16 shows all of the classes that are defined by the GOOKB.

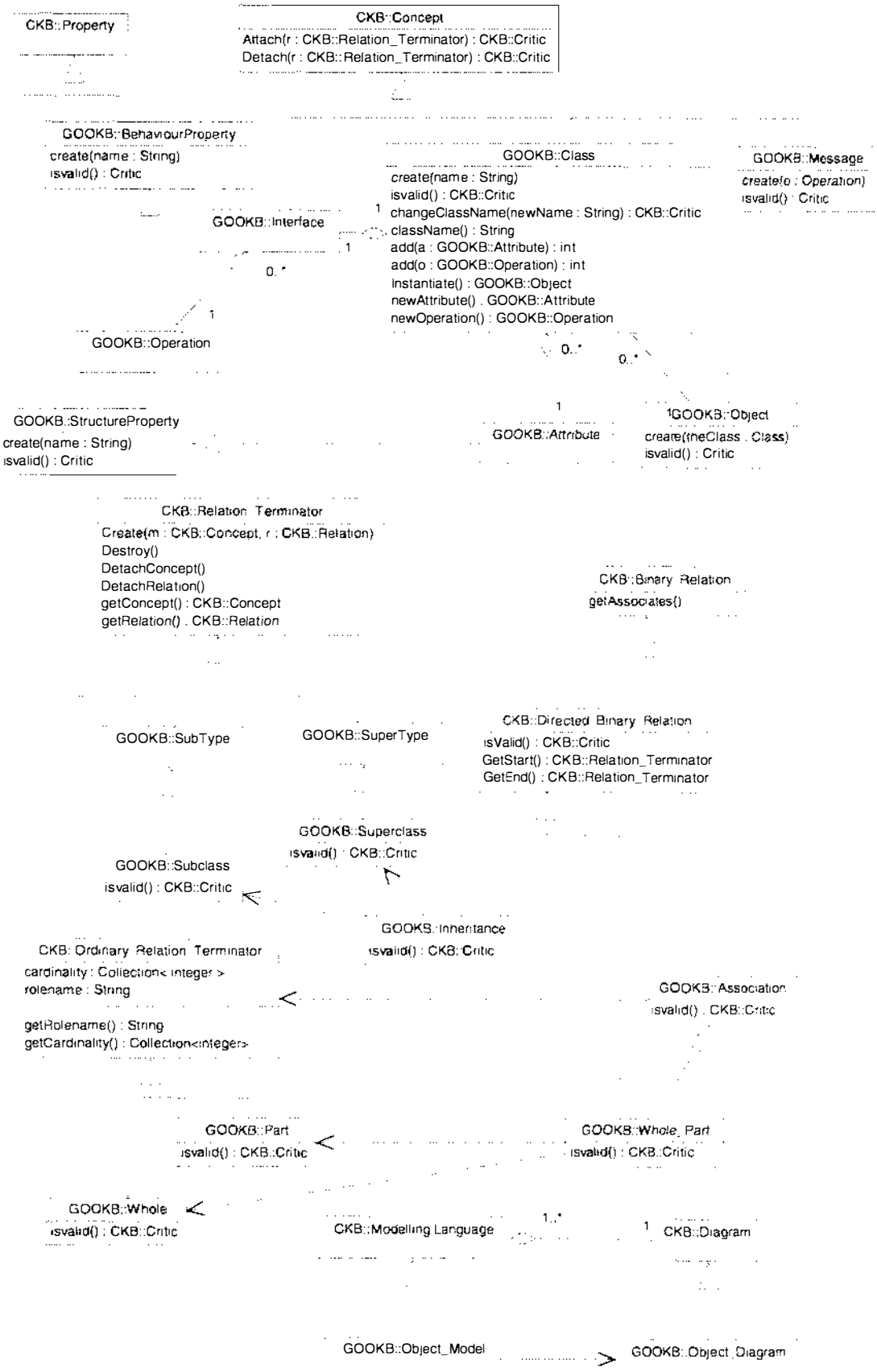


Figure 6-16 - The Generic Object Orientated Knowledge Base

Figure 6-17 (a) shows a Coad and Yourdon class diagram that represents the composite pattern (Gamma *et al.*, 1995). Figure 6-6 previously showed how instances of classes in the CKB could be used to represent this model. Figure 6-17 (b) shows how instances of the classes in the GOOKB could also be used. The object structure described in Figure 6-6 (b) and Figure 6-17 (b) is identical. The only difference is the class of the objects involved.

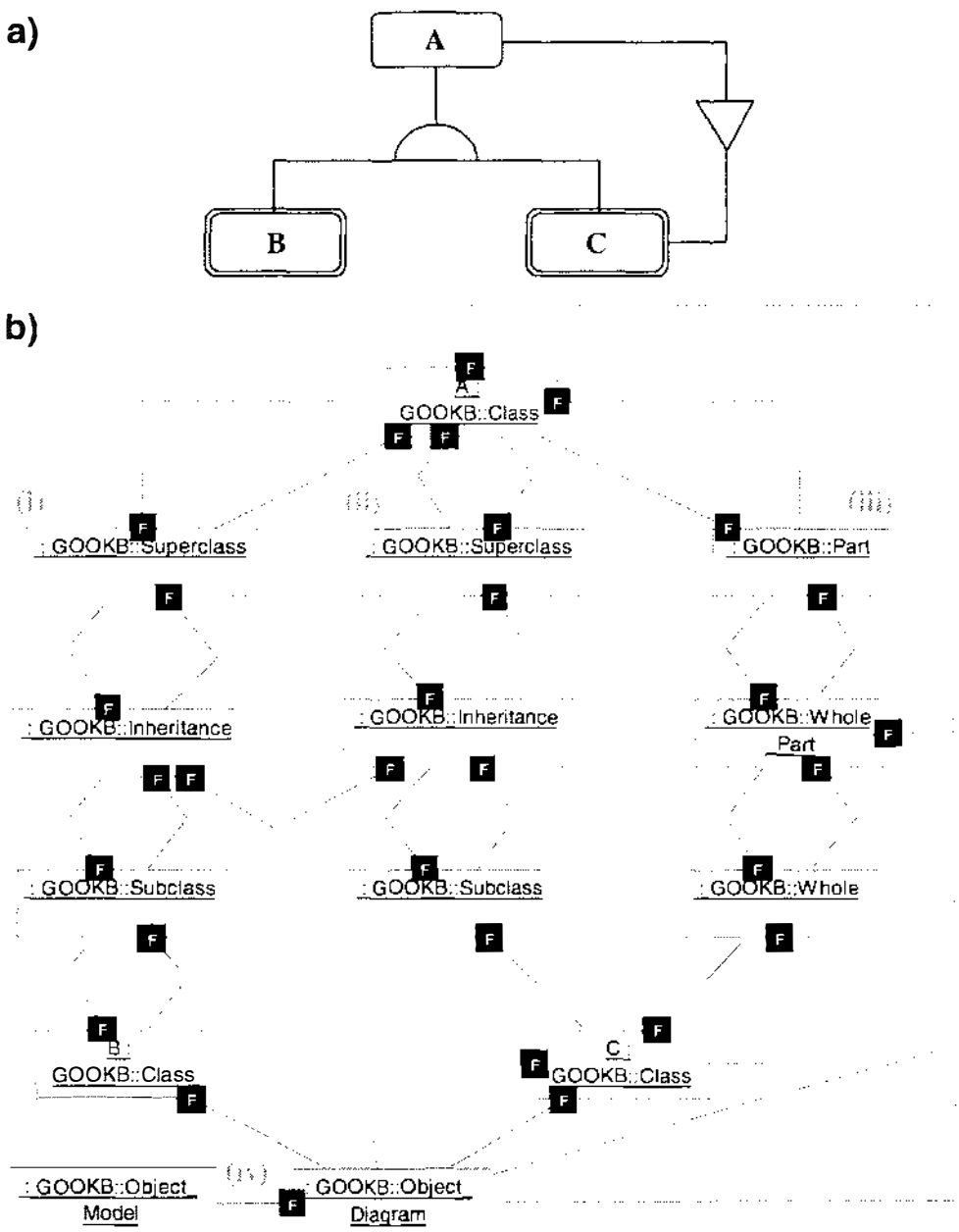


Figure 6-17 - Representing an object model with classes from the GOOKB

The objects in Figure 6-17 (b)(i) represent the inheritance relation between class *A* and class *B* in Figure 6-17 (a). The objects in Figure 6-17 (b)(ii) represent the inheritance relation between class *A* and class *C* in Figure 6-17 (a). The objects in Figure 6-17 (b)(iii) represent the whole-part relation between class *A* and class *C* in Figure 6-17 (a).

6.5 Implementing the Knowledge Bases

A convention has been adopted for the SSL module structure used to implement the knowledge bases. Each methodology knowledge base is partitioned into the following modules, where *KB Name* is the name of the knowledge base.

KB_Name	*KB_Name*_Model
*KB_Name*_Model_Element	*KB_Name*_Transistion
*KB_Name*_Document	*KB_Name*_Process
*KB_Name*_Critic	

Figure 6-18 shows the SSL module structure of the CKB and GOOKB.

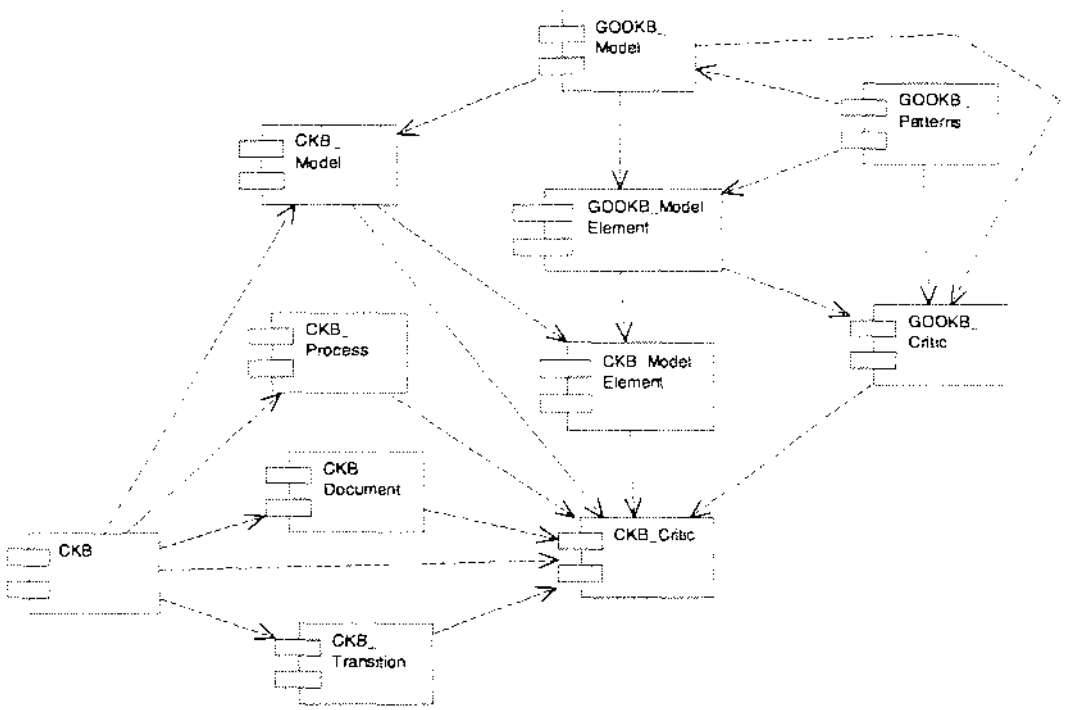


Figure 6-18 - Module structure of the CKB and GOOKB

The GOOKB extends the *CKB_Model*, *CKB_ModelElement*, *CKB_Transition* and *CKB_Critic* modules. The *GOOKB_Patterns* module provides support for patterns and is described in section 8.3 - Supporting Patterns.

6.6 Summary

This chapter has presented the development of the Core Knowledge Base (CKB) and the Generic Object Orientated Knowledge Base (GOOKB). The CKB was derived using a meta-modelling modelling approach and implements a meta-model of methodology. It also provides simple facilities for cognitive support. The GOOKB was derived by meta-modelling and implements a meta-model of concepts that are manifest with all object-orientated methodologies

One of the significant benefits of the Object-Oriented paradigm is the support for re-use. Re-use of methodology components is supported through the inheritance and aggregation mechanisms of the SSL class descriptions. The support of re-use in MOOT is fundamentally different to that of other Meta-CASE environments, which only support accidental re-use. The re-use strategy of MOOT is a reflection of the underlying meta-model (Mehandjiska *et al.*, 1995a, 1996a-c).

Chapter 7

Realising Methodologies and Software Engineering Projects in MOOT

We all agree that your theory is crazy, but is it crazy enough?

Niels Bohr

7.1 Introduction

This chapter presents the design and philosophy of the mechanisms for realising methodology descriptions and software engineering projects in the MOOT system. There are two aspects to this research: a) the development of a communication protocol between the MOOT core and CASE tool clients and b) the de-coupling mechanisms that have been developed to support late binding of syntax and semantic methodology descriptions. The syntax-semantic de-coupling is achieved in two parts: a) A table of methodology descriptions, which has been named the ***M***ethodol***o***gy ***D***escription ***T***able (MDT) and b) a mapping table, which is named the ***N***otation ***S***emantic ***M***apping (NSM) table. This chapter presents a high-level description of the communication between CASE tool clients and the MOOT core. The MDT and NSM tables are also described.

7.2 Interaction Between CASE Tool Clients and the MOOT Core

The interaction that occurs between a CASE tool client and the MOOT core can be classified, based on the direction of the interaction.

CASE Tool Client → MOOT Core

The communication in this direction corresponds to a software engineer trying to perform a task. This includes:

- Logging-in and logging-out
- Manipulating software engineering projects, models and diagrams

- Creating, deleting and updating notation elements in diagrams

Each action is implemented as a request that is sent from a CASE tool client to the MOOT core.

MOOT Core → CASE Tool Client

The MOOT core is responsible for processing requests from CASE tool clients. It is also responsible for determining if any other CASE tool client should be notified of the result of a successful request. Communication in this direction includes:

- *Responses to requests generated by CASE tool clients*
  Responses correspond to the MOOT core informing CASE tool clients of the success or failure of satisfying a request. Each and every request is matched by a response.
- *Directives from the MOOT core to CASE tool clients*
  Directives support the broadcast of information to CASE tool clients. This ensures that clients are aware of important events that have caused a change in the state of the software engineering projects they are using. Directives are matched by an acknowledgement by clients. Directives are assumed to be successful if received[54].

Figure 7-1 shows some of the requests, responses and directives that are transferred between CASE tool clients and the MOOT core. The requests and directives in Figure 7-1 have been subdivided into project-level requests and directives and model-level requests and directives.

7.2.1 CASE Tool Client Requests

The general requests, in Figure 7-1, generated by the client include:

- Getting a list of all the available methodologies, so a software engineer may select one and create a new software engineering project.
- Getting a list of all the available software engineering projects, so a software engineer may open an existing project.

[54] The MOOT core manages the semantic state of a software engineering project, whilst a CASE tool client manages the syntactic state. The MOOT core, therefore, only generates directives that correspond to a correct semantic state.

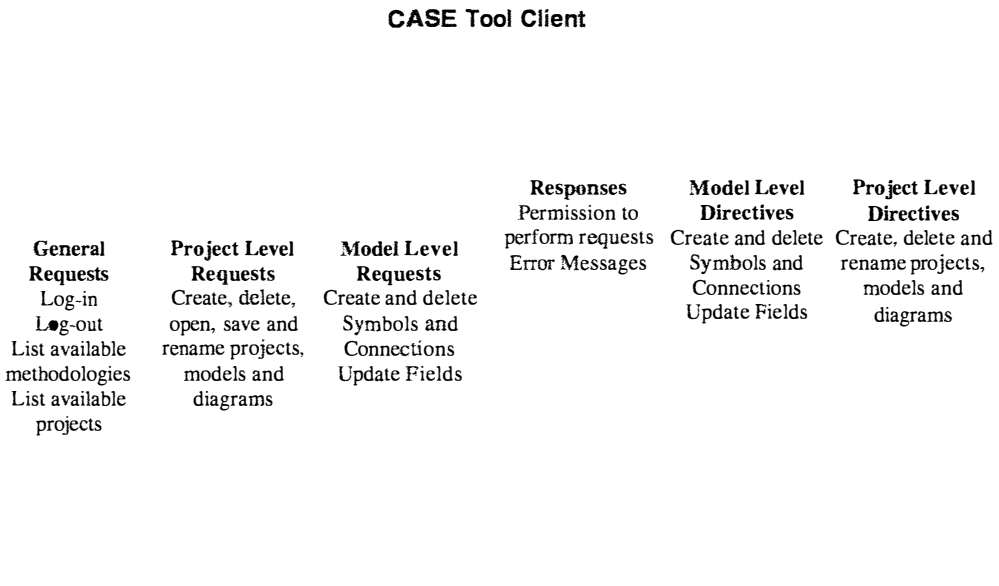


Figure 7-1 The communication between CASE tool clients and the MOOT core

Project-level requests correspond to actions (at the CASE tool client) on whole projects, models and diagrams. This includes creating, deleting, opening, closing and renaming software engineering projects, models and diagrams.

General and project level requests are satisfied by the MOOT core with the assistance of the Methodology description table (MDT). The MDT table is discussed in detail in section 7.3.

Model-level requests correspond to actions, at the CASE tool client, on the elements of diagrams. This includes:

- Placing new symbols and connections
- Deleting symbols and connections
- Updating text fields

The MOOT core uses an NSM table to translate these requests into semantic actions on the collection of SSL objects that define the state of the user's software engineering project. NSM Tables are discussed in detail in section 7.4.

7.2.2 MOOT Core Directives and Responses

Project-level directives correspond to the MOOT core broadcasting the results of successful project-level requests to other CASE tool clients. This includes broadcasting the creation, deletion and renaming of software engineering projects, models and diagrams to CASE tool clients.

Model-level directives correspond to broadcasting two types of result to CASE tool clients. The first type is the result of successful model-level requests. The seconds relates to actions performed by the MOOT core, which have knock-on effects that must be propagated to CASE tool clients. The model-level directives (shown in Figure 7-1), generated by the MOOT core include:

- Directing a CASE tool client to create a new symbol or connection on a diagram that corresponds to semantic elements (SSL objects) created by the MOOT core.
- Directing a CASE tool client to delete a symbol or connection from a diagram that corresponds to semantic elements (SSL objects) removed by the MOOT core.
- Directing a CASE tool client to update a text field in a diagram based on the change in state of semantic elements (SSL objects) by the MOOT core.

Project-level directives are generated by the MOOT core with the assistance of the methodology description table. Model-level directives are generated by the MOOT core with the assistance of NSM tables.

The responses, in Figure 7-1, are generated by the MOOT core as a result of processing a request. There are two types of response:

- Permission to carry out general, project-level and model-level requests. Responses of this type include any relevant information that the CASE tool client requires.
- Errors that correspond to disallowing a general, project-level and model-level request. An error response is always accompanied by an explanation.

A communication protocol has been defined that supports the requests a CASE tool client may make of the MOOT core and the directives and responses the MOOT core

sends to CASE tool clients. It is simple hand-shaking protocol that has been implemented on top of TCP/IP. A description of the protocol can be found in (Adams, 1998).

7.3 Methodology Description Table

The Methodology Description Table (MDT) provides a list of methodologies supported by MOOT and corresponds to an index of the methodologies in the persistent store. Each element in the table specifies:

- The SSL classes that define the methodology
- The modelling languages supported by the methodology
- The diagrams supported by each modelling language
- The notation, defined in NDL, to use for each modelling language
- An NSM table

7.3.1 Composition of the Methodology Description Table

Figure 7-2 shows the composition of the methodology description table. It is a map of Methodology descriptions that is indexed by methodology name. Each element of the table corresponds to a methodology in the MOOT system. The structure of the MDT corresponds to the upper level of the Core Knowledge Base (CKB). A methodology has one or more modelling languages, one or more documents and a process. Each modelling language provides one or more diagrams.

Each methodology in Figure 7-2 has a name that is unique within the MOOT system. The methodology name is a descriptive string that identifies the methodology. Each methodology description in the MDT references:

- A methodology type name that corresponds to its semantic definition
- An SSL class that defines its semantics
- An NSM table that defines the mapping between syntax and semantics
- A list of model descriptions

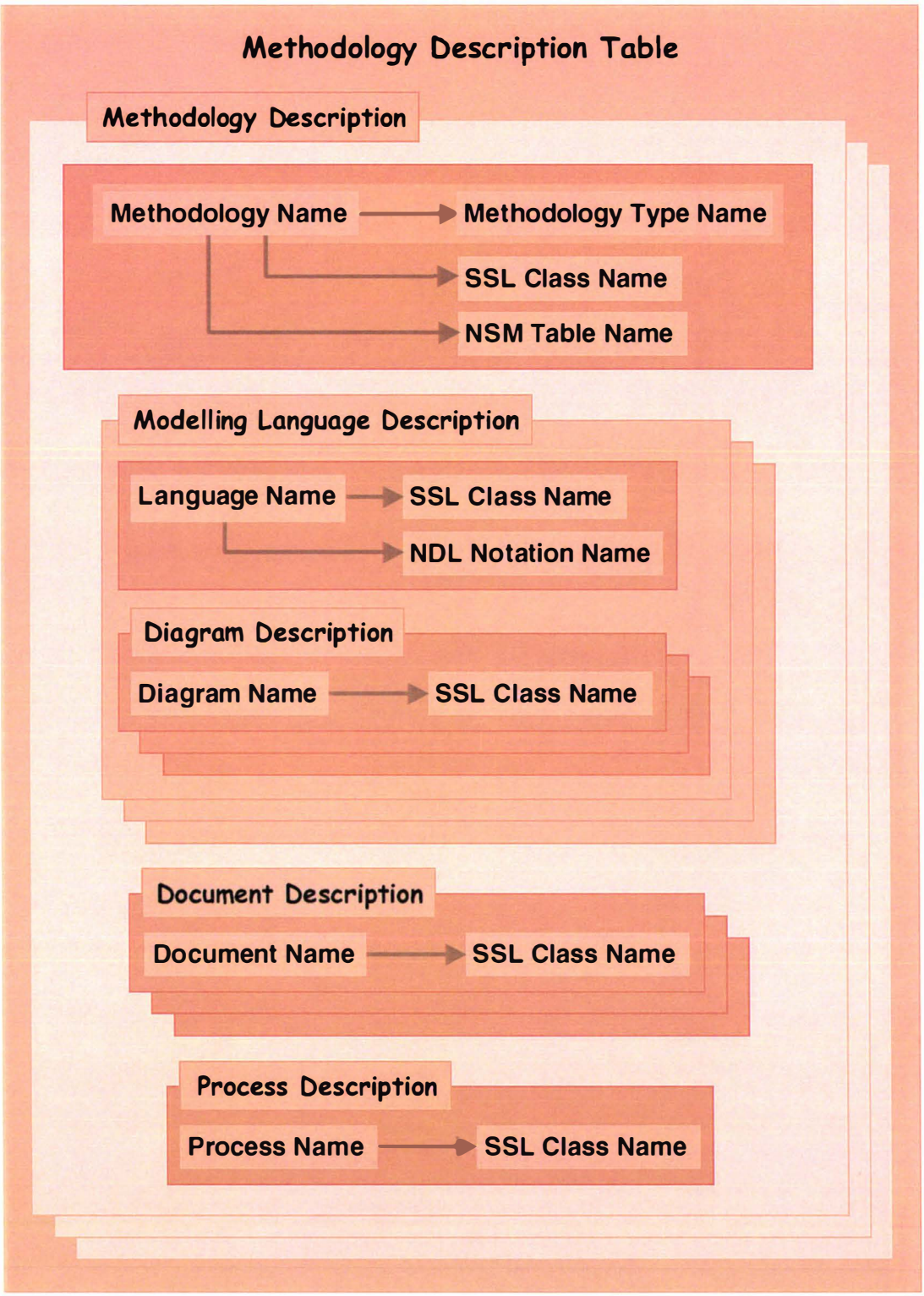


Figure 7-2 - Methodology Description Table

The methodology type name is a descriptive string that identifies the semantic definition of a methodology[55]. Each SSL class that corresponds to a methodology definition (that is

[55] The methodology type name is currently automatically derived from an SSL class name.

the *Methodology* SSL class defined in the CKB, or one of its sub-classes) has a corresponding methodology type name. The name that a software engineer will see, for a methodology, is the conjunction of the methodology name and the methodology type name. Consider the situation where the MOOT system has two variants of UML with the same semantic definition, but with different syntax. The UML semantic definition has a descriptive type name (say 'UML'). Each methodology will also have a descriptive name (say 'Company X' and 'Standard'). The names that the software engineer will see, at the CASE tool client, would be 'Company X (UML)' and 'Standard (UML)'. The purpose of the methodology type name is to make it explicit to the software engineer that the two methodologies are both semantically identical. This approach also avoids the problem with other meta-CASE tools where a 'same semantics' with 'different notation' implies a 'new methodology'. In MOOT this situation is simply viewed as 'same methodology' but 'different notation'.

Each modelling language description has a name that is unique within the context of its methodology description. This name is a descriptive string that identifies the modelling language and is the name that a software engineer will see, at the CASE tool client. Each modelling language description references:

- An SSL class that defines its semantics
- An NDL notation description that defines its syntax
- A list of diagram descriptions

The modelling language description list defines the set of modelling languages that are available with this methodology. It is possible that the list may contain a subset of all the modelling languages defined in the semantic description. The name of each diagram description is unique within the context of the modelling language description. This name is a descriptive string that identifies the diagram and is the name that a software engineer will see at the CASE tool client. The diagram description maps the diagram name to the SSL class that defines its semantics.

Each methodology may have one or more documents. The name of each document description is unique within the context of the methodology description. This name is a

descriptive string that identifies the document. Document descriptions map document names to the SSL classes that define them.

The last element in a MDT entry is a process description. It consists of a descriptive name that identifies the process and an SSL class that defines it. Any references made to the process, at the CASE tool client user interface, will be made with respect to the process name.

This thesis has not concerned itself with detailed modelling of documents and software development processes (see section 9.4 - Future Work). The coupling between a methodology and its documents and process is much lower than that between a methodology and its models. It is expected, therefore, that methodologies with the same semantic definition may be able to have different documents and processes.

7.3.2 Applying the Methodology Description Table

Figure 7-3 describes the scenario of opening a new software engineering project. For the purposes of this discussion the network communication between the *Client* object and *ClientManager* object is represented by the interchange of messages.

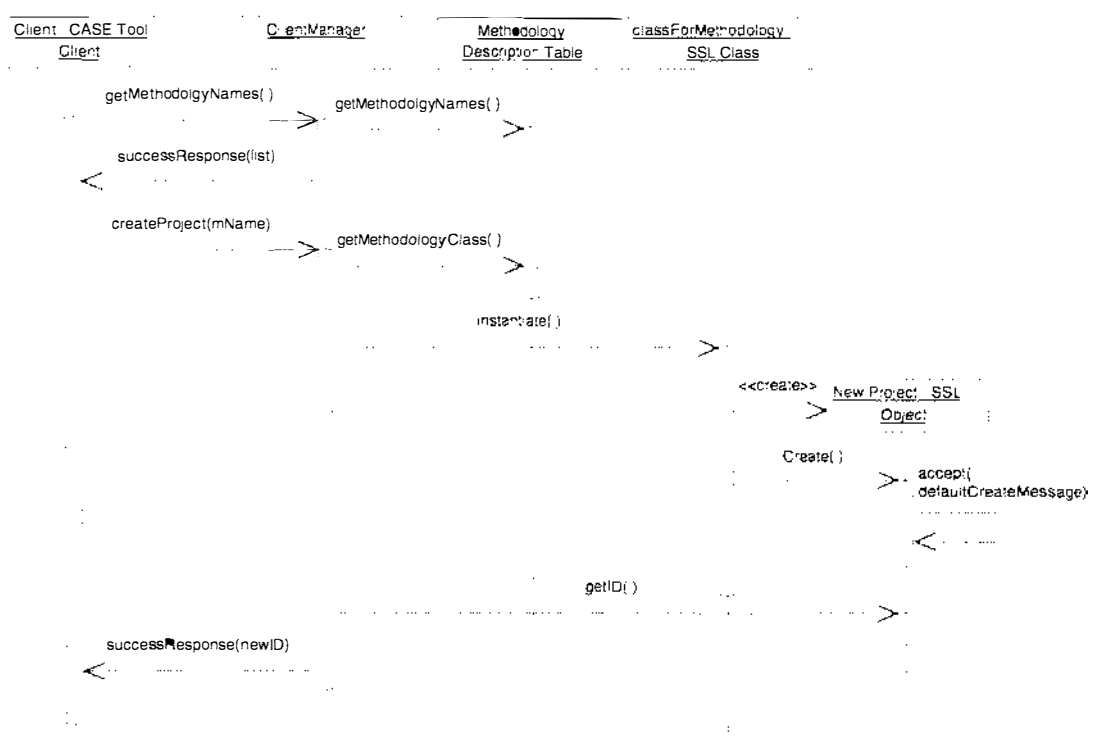


Figure 7-3 Creating a new software engineering project

ClientManager is a singleton object (Gamma *et al.*, 1995) that is part of the tool manager (see Figure 3-11 - Architecture of the MOOT prototype). It is responsible for managing communication between the MOOT core and CASE tool clients.

Initially a software engineer creates a new software engineering project. The first sub-task performed is to choose a particular methodology. The *Client* initially requests a list of methodology names from the MOOT core. The *ClientManager* object delegates the responsibility of generating this list to the MDT. The *ClientManager* sends a *successResponse* to the *Client*, with the list of names as an argument. The software engineer then selects the methodology they wish to use from this list.

The *Client* then requests that the MOOT core creates a new project. It provides the methodology name, selected by the software engineer, as an argument of a request sent to the *ClientManager*. The *ClientManager* initially must determine the SSL Class that corresponds to this methodology by interrogating the MDT. Once it has the appropriate SSL class, it then creates an instance of it. The resulting SSL object (*NewProject* in Figure 7-3) is the root of the new software engineering project. The last step of the process is to return the unique SSL ID of *NewProject* to the client as an argument of a *successResponse* message. Any future references the *Client* makes to the new software engineering project will include this SSL ID as an argument.

7.4 Notation Semantic Mapping Tables

The Notation Semantic Mapping (NSM) table defines the mapping between notation elements and semantic concepts (see Figure 3-2 - The relation between software projects, methodology descriptions and the description languages in MOOT).

7.4.1 NDL vs. SSL

The syntax of a methodology is defined using the Notation Definition Language (NDL). NDL is a scripting language that is based on composition of a fixed set of template types. Template types are parameterised by a set of text fields, each of which has a unique NDL ID. The syntax definition of a methodology consists of a set of NDL scripts. The visual representation of a software engineering project consists of a collection of NDL views, grouped into a collection of diagrams.

The semantics of a methodology is defined using the Semantic Specification Language (SSL). SSL is an object-orientated language based on classes, inheritance, aggregation, association, polymorphism and message passing. The semantic definition of a methodology consists of a set of SSL classes. The state of a software engineering project consists of a collection of SSL objects. The MOOT core processes SSL by executing SSL methods on the SSL virtual machine. It does this in response to the tasks the software engineer performs using the CASE tool client.

The NSM table is responsible for defining a particular mapping between a syntax and semantic definition. Table 7-1 shows the correspondence between elements that support syntax and elements that support semantics in MOOT.

	Syntax	**Semantics**
Project Structure Elements	Project editor of the CASE tool client	Instance of the *Methodology* SSL class (and sub-classes)
	Model editor of the CASE tool client	Instance of the *Modelling Language* SSL class (and sub-classes)
	Diagram editor of the CASE tool client	Instance of the *Diagram* SSL class (and sub-classes)
Structural Elements	NDL template	SSL class
	NDL view	SSL object
	Text fields	Attributes of SSL classes
Dynamic Elements	Creating and destroying NDL views	Creating and destroying SSL objects
	Values in Text fields	SSL object state
	Actions on NDL views	Messages to SSL objects

Table 7-1 - Correspondance between syntax and semantic elements

The mapping of the structural and dynamic elements shown in Table 7-1 is supported by NSM tables. The mapping of project structure elements is supported with the Methodology Description Table.

7.4.2 Composition of NSM Tables

An NSM table consists of six associative arrays (maps). Each map supports one aspect of the mapping between an NDL description and an SSL description.

The names of the six maps contained in an NSM table are:

1. 'Create concept' map
2. 'Create relation' map
3. 'Add' map
4. 'Action' map
5. 'SSL object creation' map
6. 'SSL object update' map

Create Concept Map

The create concept map (Figure 7-4) defines the mapping between NDL symbol templates and SSL classes.

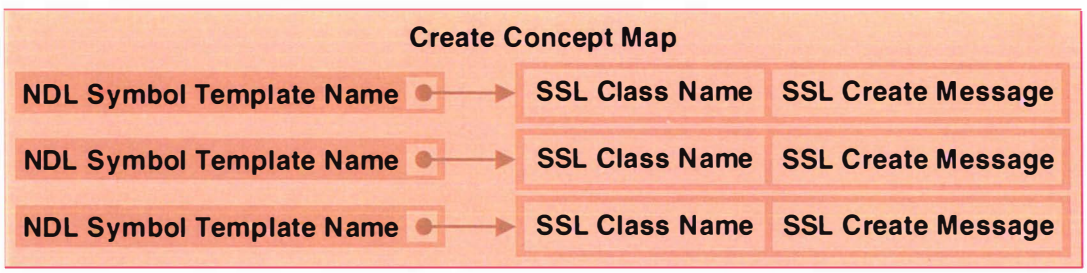


Figure 7-4 - The create concept map

The left-hand side is a list of NDL symbol template names. The right-hand side is a list of SSL class names and SSL create message pairs. A request from the client to create a particular symbol (that is represented by an NDL symbol template) is satisfied by instantiating the corresponding SSL class. The corresponding create message is sent to this class to initialise the new SSL object.

Each NDL symbol template may only appear once in the map. The current implementation of the NSM table assumes a one-to-one mapping between an NDL symbol template and an SSL class. Only SSL classes that correspond to an NDL symbol template will appear in this table.

Create Relation Map

The create relation map (Figure 7-5) defines the mapping between NDL connection templates and SSL classes.

The left-hand side of Figure 7-5 is a list of NDL connection template names. The right-hand side is a list of SSL class names and SSL create message pairs. A request from the client to create a particular connection (that is represented by an NDL connection template) is satisfied by instantiating the corresponding SSL class.

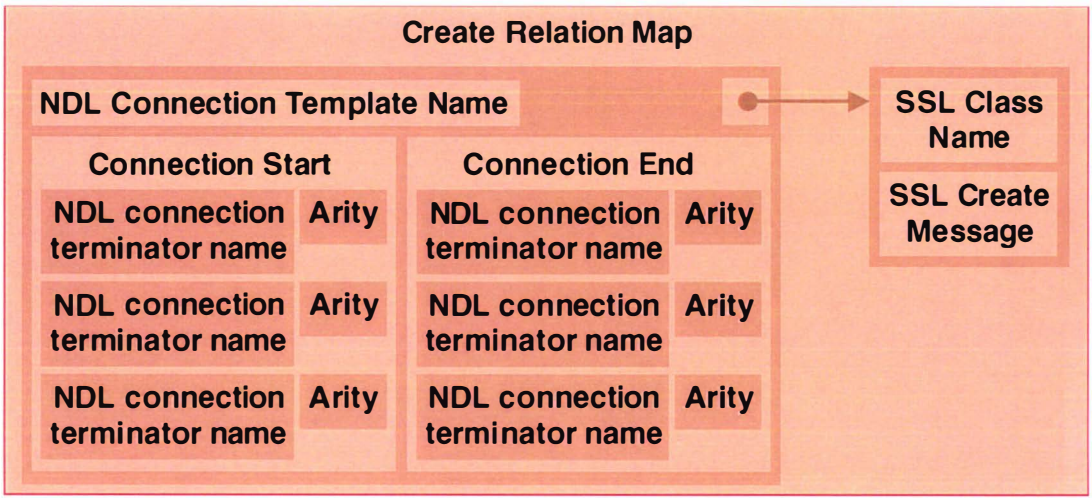


Figure 7-5 - The create relation map

Each create relation entry in the create relation map (Figure 7-5) is composed of one or two parts. The connection start defines items that may appear at the beginning of a connection. The connection end defines items that may appear at the end of a connection. This structure implies the convention that all relations have a 'start' and an 'end'. Nary connections (which do not have a start or end) are represented with the connection start. The distinction between start and end parts is important for directed relations, but not for bi-directional relations.

Each item in the connection start and connection end parts is composed of an NDL template of a connection terminator and an arity. The arity defines how many connection terminators may be involved in a connection.

The current implementation of the NSM table assumes a one-to-one mapping between an NDL connection template and an SSL class. Only SSL classes that correspond to an NDL connection template will appear in this table.

Add Map

The add map (Figure 7-6) is used to add models to projects, diagrams to models and concepts and relations to diagrams.

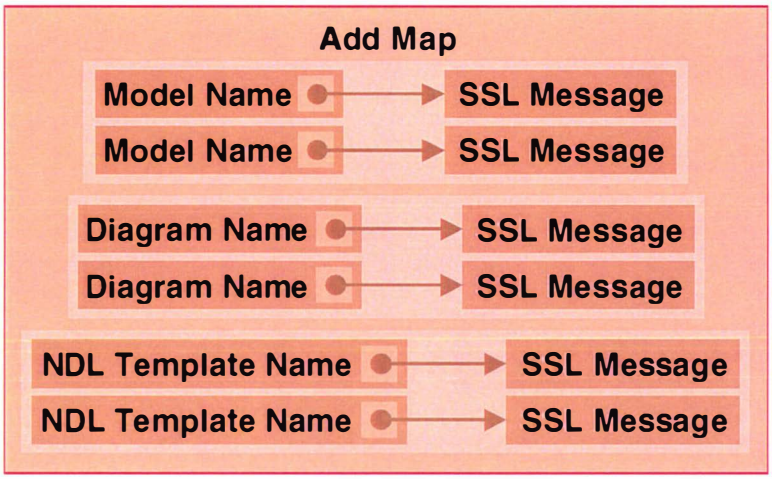


Figure 7-6 - The add map

Models and diagrams are created with the assistance of the MDT, which provides a similar service as the create concept and create relation maps. The add map is used to find the SSL message that is used to add models to projects and diagrams to models. The first two maps, in Figure 7-6, provide a mapping between model and diagram names and the appropriate SSL message.

Creating a new concept or relation is achieved in two steps. The first is to create a corresponding SSL object. This is supported with the create concept and create relation maps. The second step is to add the new SSL object to a diagram. The add map is used to find the SSL message that is used to add an item, that corresponds to a particular NDL template, to a diagram. The message is sent to the SSL object that represents the diagram, with the item as an argument.

Action Map

The action map (Figure 7-7) helps translate 'logical actions' at the software engineer user interface, to the corresponding equivalent semantic actions.

NDL text fields can be considered as the visual representation of the properties of concepts and relations (such as the role name on an association connection, or the class

name in a class symbol). The properties correspond to attributes of SSL classes. The values of the properties correspond to the state of SSL objects.

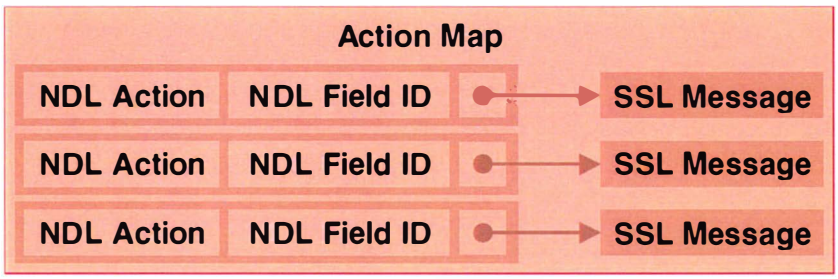


Figure 7-7 - The action map

An NDL action on a particular field is mapped to an SSL message. NDL actions do not reference the state of SSL objects directly, as this would break encapsulation. This mechanism supports the binding of arbitrary NDL actions to SSL messages.

SSL Object Creation Map

The SSL object creation map (Figure 7-8) performs the reverse operation of the create concept and create relation maps.

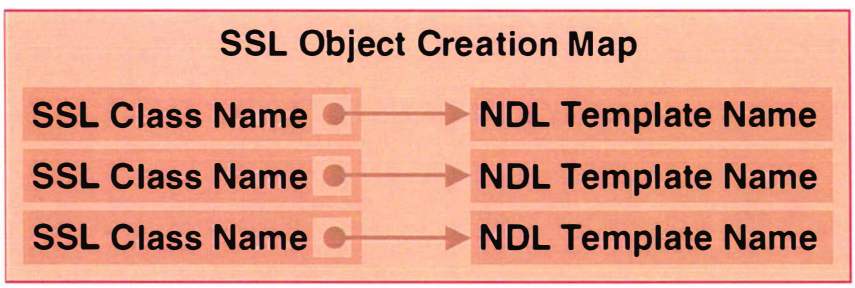


Figure 7-8 - The SSL object creation map

The SSL object creation map identifies those SSL classes whose instances should be reflected by visual representations in clients. It is used when the server creates SSL objects, without a request from a client. For example an SSL object representing a context diagram in a Data Flow Diagram (DFD) model may automatically create an SSL object for the system process. SSL classes in this category are a subset of all the SSL classes in a methodology description.

SSL Object Update Map

The SSL object update map (Figure 7-9) performs the reverse operation of the action map for updates of the state of an SSL object. It maps SSL messages to one or more NDL fields (each of which has a unique identification number – an NDL ID). When an

SSL object on the server sends a message that appears on the left-hand side of the map, the NDL fields on the right-hand side need to be updated with new values. Such changes are broadcast by the MOOT core to affected clients.

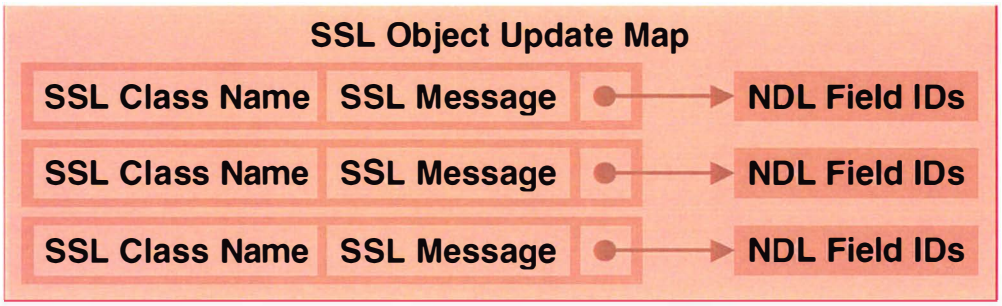


Figure 7-9 - The SSL object update map

7.4.3 Applying NSM Tables

The complete NSM table is shown in Figure 7-10. The arrows indicate sub-parts of the table that are related to each other, in terms of satisfying requests and satisfying the issuing of directives. The lower section shows the NSM table elements that provide reverse operations.

The top four maps are related to communication from CASE tool clients to the MOOT core (the requests), whilst the bottom two maps are related to communication in the reverse direction (the directives).

The create concept and create relation maps are both used to translate NDL templates into SSL classes and support the creation of new symbols and connections on diagrams. The SSL object creation map provides the reverse operation of translating SSL classes into NDL templates.

The add map is used to specify how new concepts and relations are added to a diagram.

The action map is used to translate logical actions at the CASE tool client into semantic actions (messages to SSL objects). The SSL object update map provides the reverse function of the action map, for update actions.

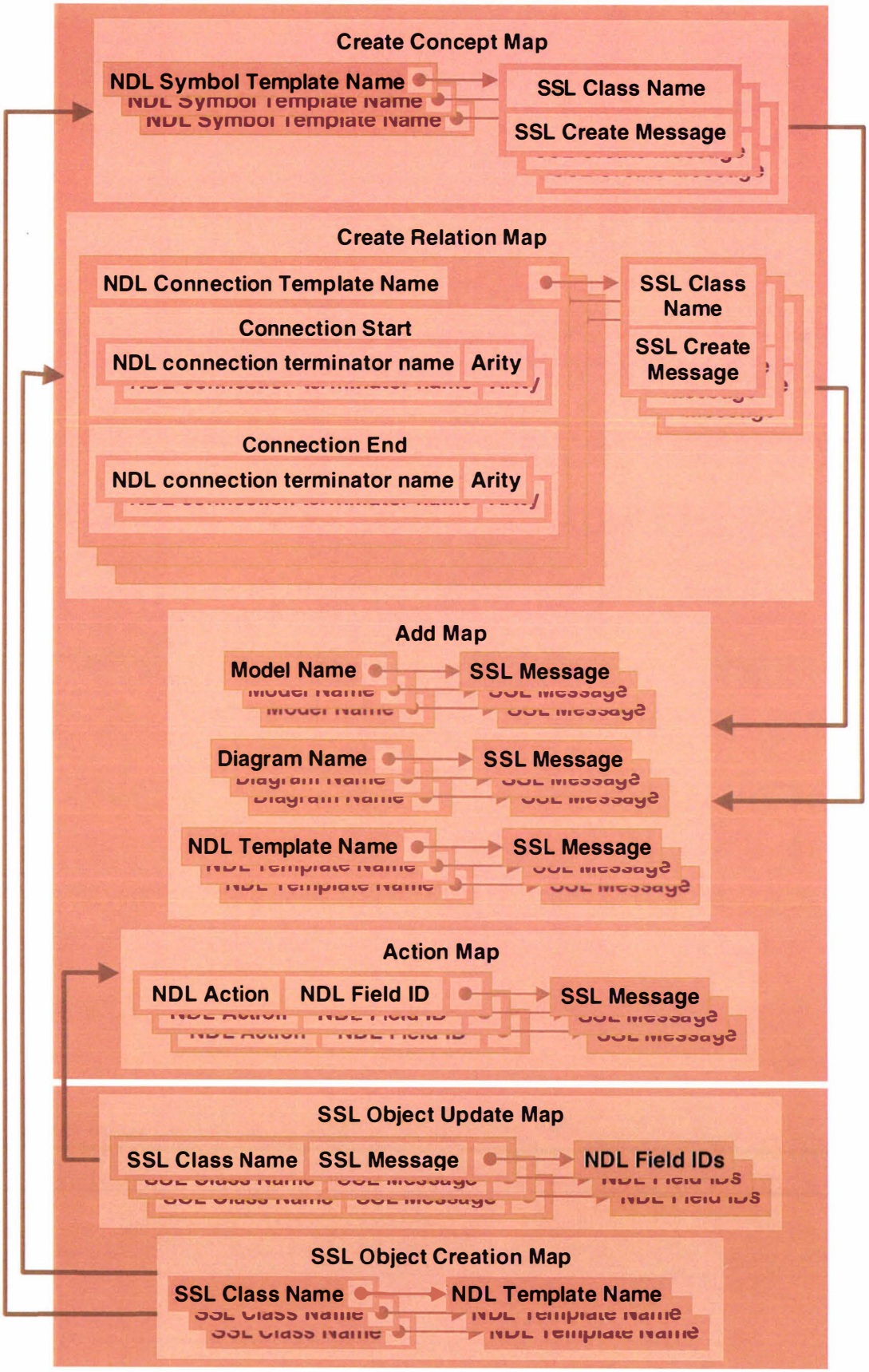


Figure 7-10 - The Notation Semantic Mapping Table

Several scenarios that illustrate how NSM tables are used follow. These are: creating a new model; creating a new concept; the successful update of a text field; a failed attempt to update a text field and the propagation of a server side update to CASE tool clients.

Creating a New Model

The scenario in Figure 7-11 (creating a new model) illustrates the use of the NSM table add map. It also further shows the use of the Methodology Description Table (MDT).

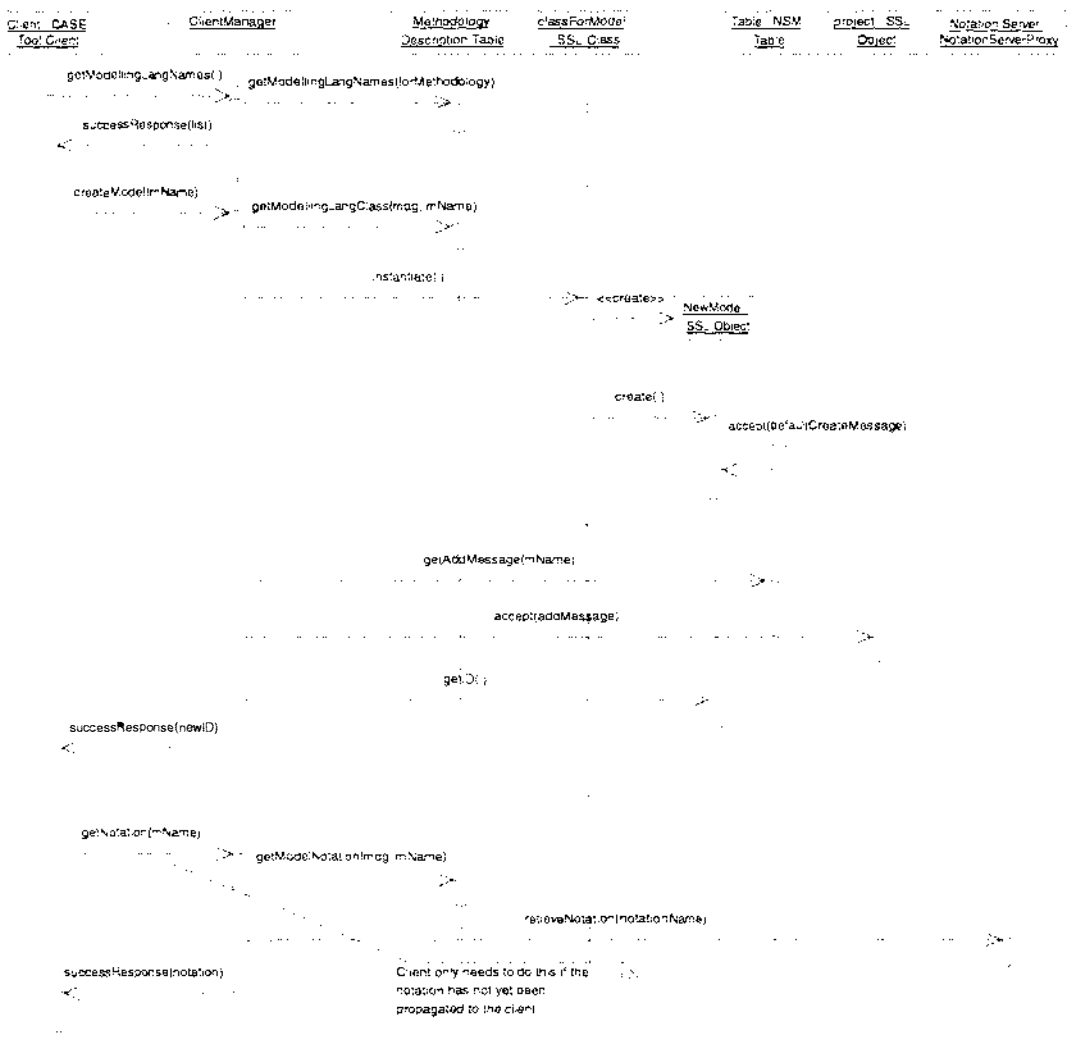


Figure 7-11 - Creating a new model

The software engineer has already created a new software engineering project, or opened an existing one, and has decided to create a new model. The *Client* requests a list of modelling language names from the MOOT core[56]. The *ClientManager* delegates this

[56] The *Client* actually requests this list when the user selects a methodology. It is shown here so all the steps involved in creating a new model can be explained in the same context.

responsibility to the MDT. The argument to the *getModellingLangNames* message corresponds to the methodology of the user's project[57]. The *ClientManager* then sends a *successResponse* message to the *Client* with a list of modelling language names as an argument.

Once the software engineer has chosen the modelling language to use, the Client sends a *createModel* message to the *ClientManager*, with the modelling language name as an argument. The *ClientManager* uses the MDT to find the appropriate SSL class for this modelling language (the *classForModel* object in Figure 7-11). It then creates an instance of this class (*NewModel* in Figure 7-11) by sending *classForModel* an *instantiate* message.

The *ClientManager* now uses the NSM table object (*Table* in Figure 7-11) to find the SSL message which is used to add the newly created model (*NewModel* in Figure 7-11) to the current project (the *project* object in Figure 7-11). The add message is then sent to the *project* object with *NewModel* as an argument. The *ClientManager* then sends a *successResponse* message to the *Client* with the unique SSL ID of the newly created model as an argument.

The *Client* also needs the notation that corresponds to the new model. If it does not already have the notation it sends a *getNotation* request to the *ClientManager*, with the modelling language name as an argument. The *ClientManager* determines the NDL script that is required by interrogating the MDT. It then requests the notation from the *NotationServer* object[58]. The *ClientManager* finally sends a *successResponse* message to the Client, with the notation as an argument.

Creating a New Concept

The scenario captured by Figure 7-12 illustrates the use of the NSM table create concept map. This scenario occurs whenever the software engineer places a new concept into a diagram (e.g. a new class on a class diagram, or a new state on a state transition diagram).

[57] The *ClientManager* actually maintains a vector of *ClientProxy* objects (one for each connected client). Each ClientProxy object maintains details such as the user and the active project. The *ClientProxy* objects have been omitted from these diagrams for brevity.

[58] The MOOT prototype implements this as a direct request to the persistent store.

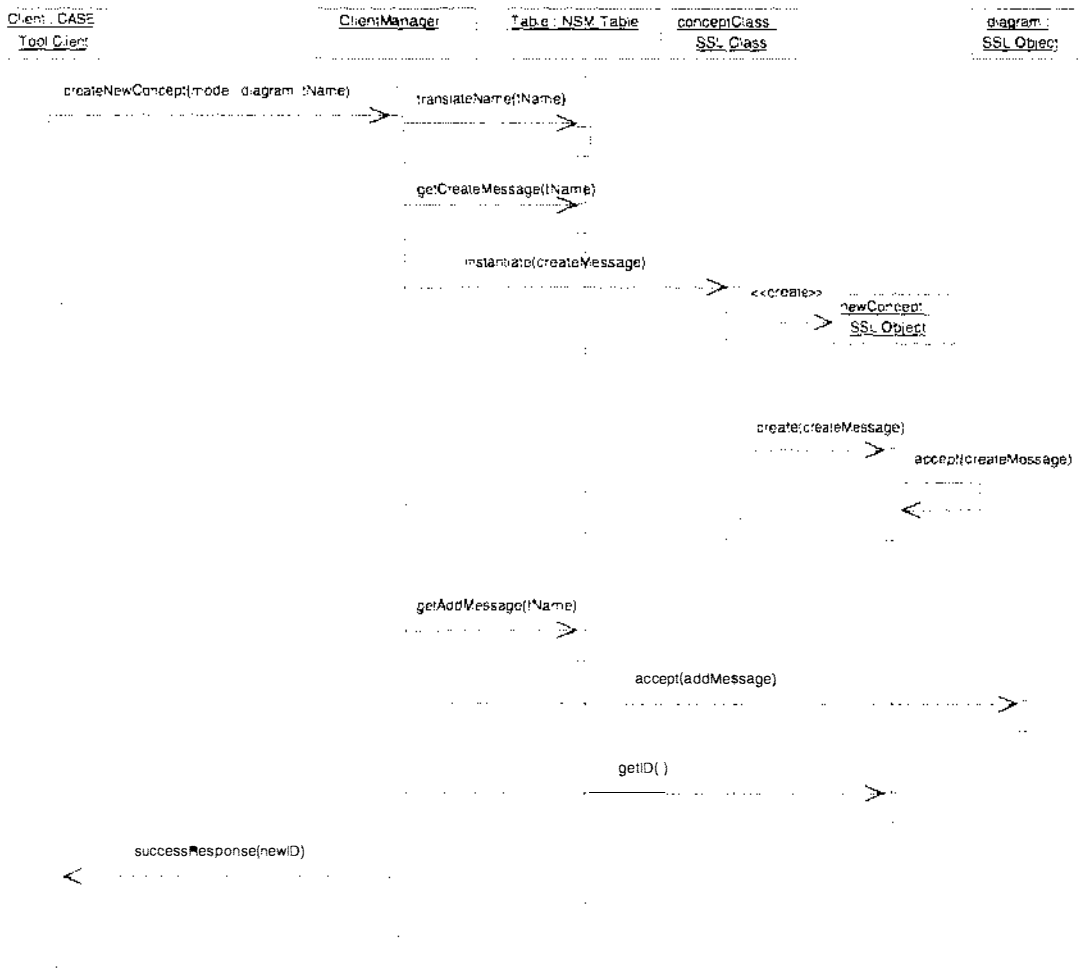


Figure 7-12 - Creating a new concept

The Client sends a *createNewConcept* message to the *ClientManager*. The arguments to this message include the SSL IDs of the model and diagram[59] where the concept is to be placed and the name of an NDL symbol template. The *ClientManager* performs two tasks on receipt of the *createNewConcept* message. It first asks the NSM table (the *Table* object in Figure 7-12) to translate the NDL symbol template name into a corresponding SSL class name (*conceptClass* in Figure 7-12). It then asks the NSM table for a create message. The create message is used to initialise the new concept (*newConcept* in Figure 7-12). Once the object has been created the *ClientManager* asks the NSM table for an add message. The add message is used to add the new concept to the diagram (*diagram* in Figure 7-12) identified in the original *createNewConcept* message.

[59] It is assumed that a client can only access a single project at any time. In practice all messages from the *Client* also include the SSL ID of the project. The tuple: project ID, model ID, diagram ID uniquely identifies the context within which an action occurs.

The *ClientManager* then sends a *successResponse* message to the *Client* with the unique SSL ID of the new concept as an argument. In this example all steps are successful. In general each step may fail and result in a *failResponse* message being sent to the *Client*. An example that includes a *failresponse* is given on page 197.

The Successful Update of a Text Field

The next two scenarios demonstrate the use of the NSM table action maps. In Figure 7-13 the software engineer has changed the value of a text field on a diagram. Text fields correspond to the properties of concepts and relations. This scenario corresponds, for example, to changing an attribute name in a class diagram or renaming a store in a data flow diagram.

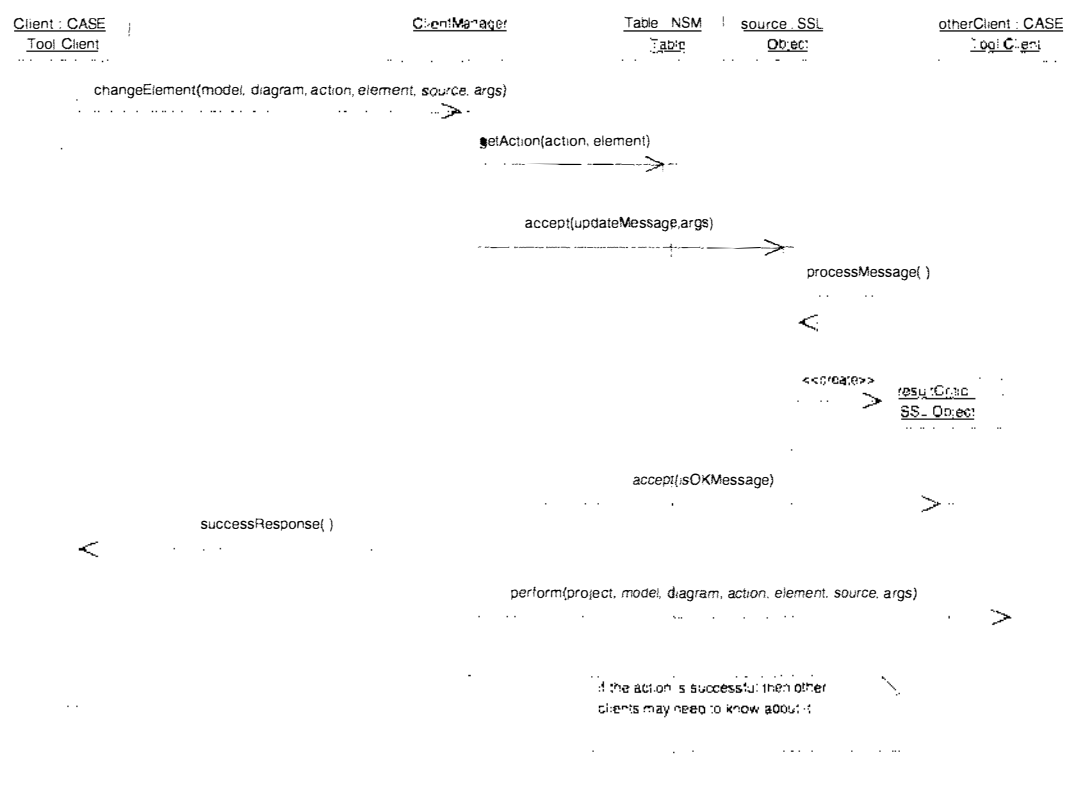


Figure 7-13 - Successful update of a field

Initially the client sends a *changeElement* message to the *ClientManager*. The arguments to this message include the SSL IDs of the model and diagram on which the updated field is placed. It also includes IDs that identify the action to be performed, the text field that has been altered and the SSL ID of the concept or relation that owns the text field. The model, diagram, element and source IDs uniquely identify the field to be updated. The action ID defines what is to be done to the field.

The *ClientManager* asks the NSM table to translate the action and element ID into an SSL message (*updateMessage* in Figure 7-13). The update message is then sent to the object that corresponds to the source argument of the original message (*source* in Figure 7-13).

Figure 7-13 shows that the update message has resulted in the creation of a critic object (*resultCritic* in Figure 7-13). The *resultCritic* SSL object is then interrogated to determine the result of the update message. If the *resultCritic* indicates a success the *ClientManager* sends a *successResponse* to the Client to indicate that it can commit the update. The *ClientManager* can now broadcast the change to any other CASE tool clients that may be interested (*otherClient* in Figure 7-13).

A Failed Attempt to Update a Text Field

Figure 7-14 shows the same scenario as Figure 7-13 except that the result of processing *updateMessage* indicates that the update is not valid. For example changing the name of an attribute so it is the same as another might be considered illegal.

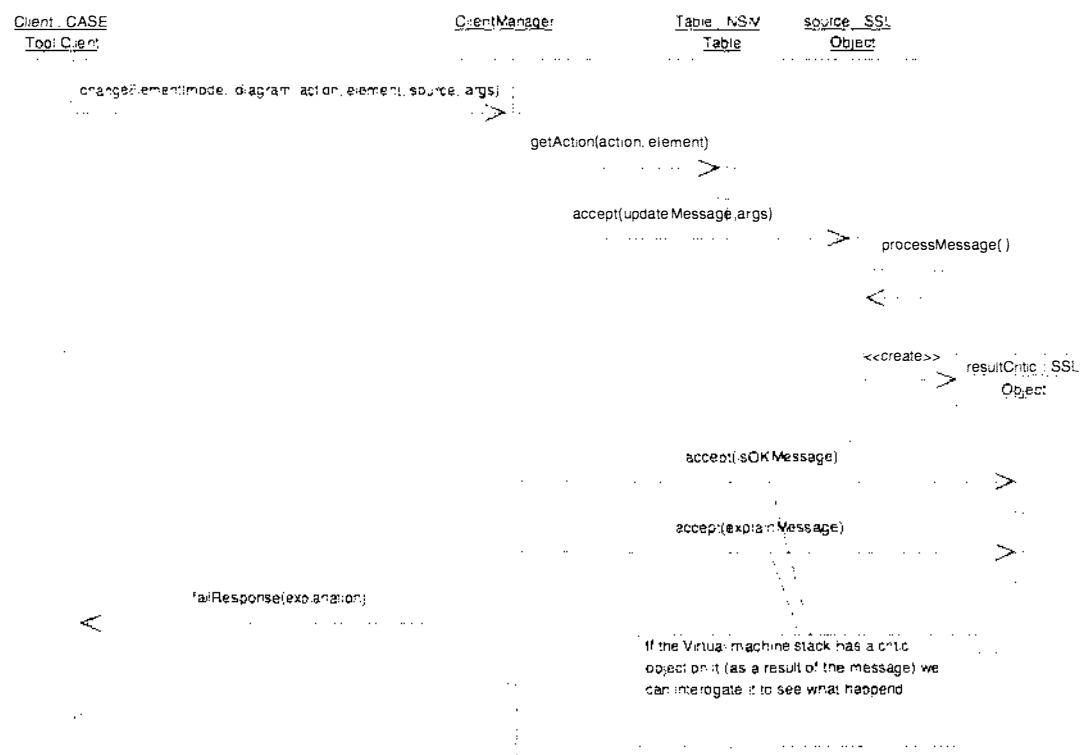


Figure 7-14 Failed attempt to update a field

The *resultCritic* object in Figure 7-14 has indicated that an error occurred (the *isOKMessage* message has returned false). The *ClientManager* interrogates the *resultCritic* object for an

explanation. A *failResponse* message is sent to the Client with the explanation as an argument. The Client can then present the explanation to the software engineer[60].

Propagation of a Server Side Update to Other Clients

The final example (Figure 7-15) illustrates the use of the SSL object update map.

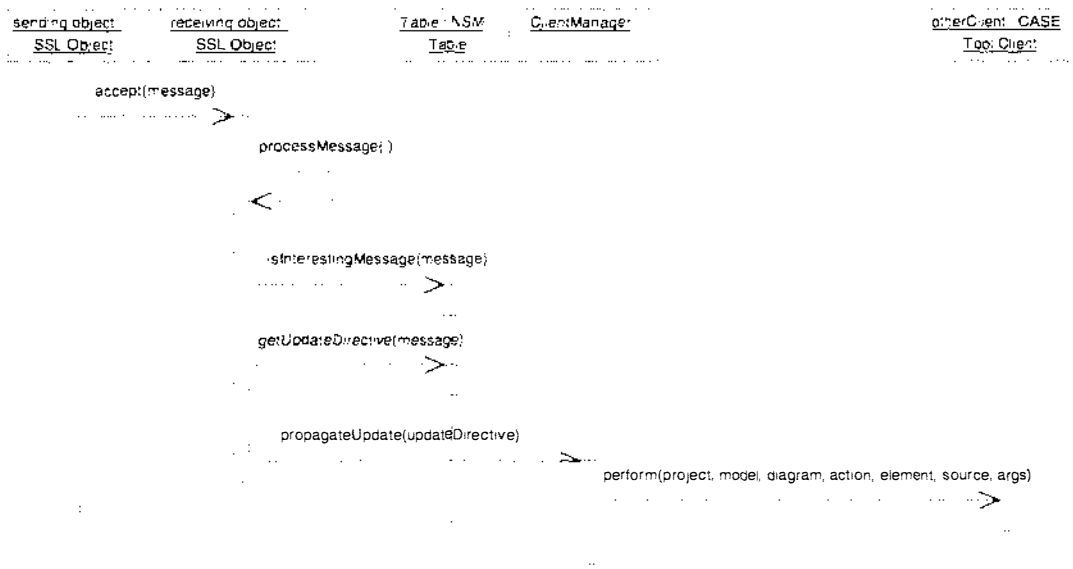


Figure 7-15 - Propagating server side update

The SSL object *sending object* in Figure 7-15 has sent a message (*message* in Figure 7-15) to another SSL object (*receiving object*). Once *receiving object* has finished processing the message it then interrogates the NSM table to see if the message might cause an update that is of interest to CASE tool clients. The message is translated into a collection of NDL field IDs by the NSM table. *Receiving object* then asks the *ClientManager* to propagate update directives to any CASE tool clients that may be interested. The *ClientManager* sends a perform message to all interested clients (*otherClient* in Figure 7-15). The perform message includes the SSL ID of the source object and the element ID of the field. The *Client* is responsible for updating all views that correspond to the source object.

[60] In the prototype implementation of the CASE tool client explanations are collated into a separate feedback window so the software engineer can trace what they have done.

7.5 Summary

This chapter has discussed the realisation of methodologies in the MOOT system and illustrated how syntax and semantic descriptions are associated with each other to form complete methodology descriptions. The syntax-semantic association involved:

- The relation between a software engineering project, in terms of its models, diagrams and documents, to the methodology used to create it.
- The relation between the syntax and semantic descriptions expressed with NDL and SSL.
- The relation between logical actions performed using the CASE tool client and semantic actions processed by the MOOT core.

The three aspects of the syntax-semantic mapping have been supported by:

- The Methodology Description Table (MDT). Each element of the MDT corresponds to a methodology. The structure of the MDT corresponds to the upper level of the Core Knowledge Base (a methodology has a collection of modelling languages, a collection of documents and a process. Each modelling language has a collection of diagrams).
- Notation-Semantic Mapping tables. Each NSM table defines the mapping between a NDL notation description and SSL semantic description. This includes mapping NDL templates to SSL classes, NDL Views to SSL objects and logical actions on NDL views to messages to SSL objects.
- The communication protocol between the CASE tool client and MOOT core. This protocol is based on the requests and directives that are passed between the CASE tool client and the MOOT core.

Section III

Results, Discussion and Review

Chapter 8

Validating the MOOT Approach

Give me a fruitful error any time, full of seeds, bursting with its own corrections. You can keep your sterile truths for yourself.

Vilfredo Pareto

8.1 Introduction

The research described in this chapter is representative of work carried out to validate the initial MOOT prototype and to investigate the efficacy of the MOOT approach. A range of results that illustrate the application of the principles, techniques and ideas propounded in this thesis are presented. This includes:

- The implementation of the Coad and Yourdon methodology (Coad and Yourdon, 1990, 1991a, b). The description includes fragments of NDL code, SSL code, a portion of an NSM table and an entry from the Methodology Description Table.
- An extension of the Generic Object Orientated Knowledge Base, which implements support for patterns (Gamma *et al.*, 1995).
- Defining the core UML meta-model as an extension of the CKB and GOOKB.
- Development of the semantics editor using MOOT. Two modelling languages are proposed for this purpose – the SSL module structure modelling language and the SSL method modelling language. The semantics of these modelling languages are defined as an extension of the Core Knowledge Base (CKB) and the Generic Object Orientated Knowledge Base (GOOKB).
- Development of the Joosten workflow methodology (Joosten, 1995).

8.2 Defining the Coad and Yourdon Methodology

The Coad and Yourdon methodology can be readily defined as an extension of the Generic Object Orientated Knowledge Base (GOOKB). The GOOKB already defines SSL classes for class, attribute, operation, inheritance, whole-part and association. The only additional relation that must be added is message connection.

Figure 8-1 shows the SSL classes that have been added to implement Coad and Yourdon.

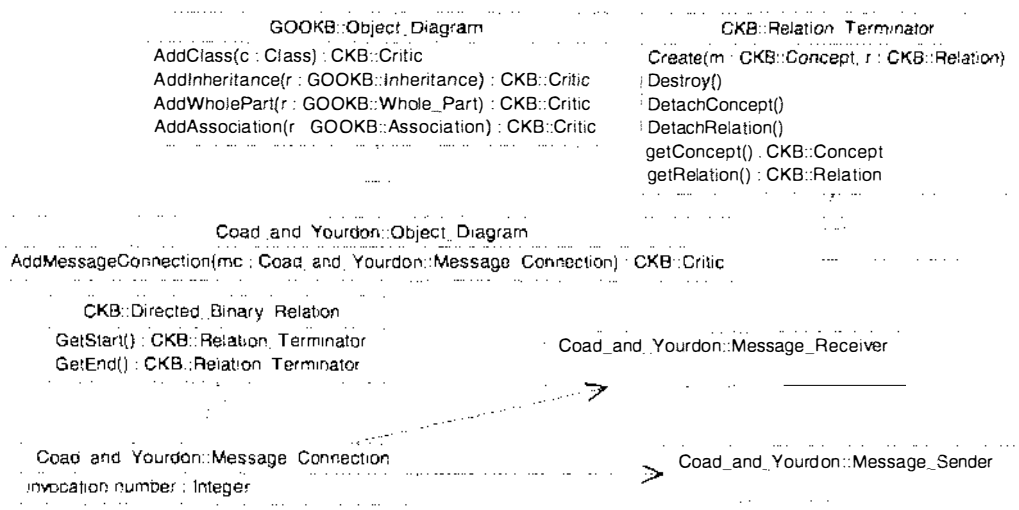


Figure 8-1 - Supporting the Coad and Yourdon

The *Message Connection* class in Figure 8-1 is a type of *Directed Binary Relation.* Its end points are modelled with two sub-classes of *Relation Terminator* - *Message Receiver* and *Message Sender. Object Diagram* has been sub-classed and extended with behaviour for handling message connections. This example illustrates two important points:

- The definition of Coad and Yourdon is straight forward. Whilst it can be said that Coad and Yourdon is one of the simplest methodologies, the semantic definition of Coad and Yourdon only required four new classes.

- The SSL module system prevents an explosion of SSL class names. For example the two *Object Diagram* classes in Figure 8-1 are disambiguated by the knowledge base they are defined in (*GOOKB* and *Coad and Yourdon*).

Figure 8-2 shows an entry in the Methodology Description Table (MDT) for Coad and Yourdon. The first line is the name that a software engineer will see when using the CASE tool client. The second line is the SSL class that is instantiated when a new Coad

and Yourdon project is created. Line three specifies how many modelling languages are supported. In this example Coad and Yourdon supports a single modelling language. Lines four to nine define the single modelling language. Line four is the descriptive name for the modelling language that the software engineer will see. The next line is the corresponding SSL class that is instantiated when a new model is created that uses this modelling language. Line six is the name of an NDL notation to be used for the modelling language. Line seven specifies that this modelling language consists of a single type of diagram. Line eight contains a descriptive name that the software engineer sees, for this diagram and line nine is the corresponding SSL class.

```
Line 1.  coad and yourdon
Line 2.  ckb:methodology
Line 3.  1
Line 4.  coad and yourdon class diagram
Line 5.  gookb_model:object_model
Line 6.  coadyourdon_classdiagram
Line 7.  1
Line 8.  class diagram
Line 9.  coadyourdon_model:object_diagram
```

Figure 8-2 - Methodology description table for Coad and Yourdon

Figure 8-3 shows a snapshot of the CASE tool client 'select methodology' dialogue box.

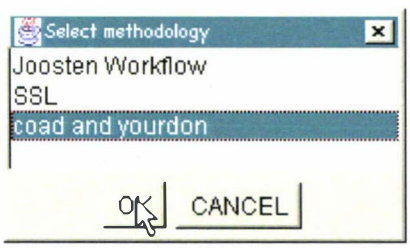


Figure 8-3 - The select methodology dialogue box

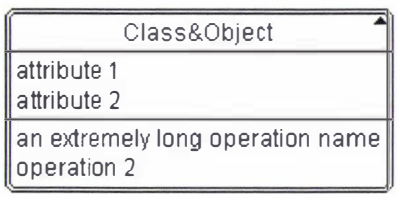
The example in Figure 8-3 shows that the software engineer has three methodologies to choose from, when creating a new project[61]. The last item in the list (*coad and yourdon*) corresponds to the entry in the MDT in Figure 8-2. The text displayed in this dialogue box is from line one of the Coad and Yourdon entry in the MDT.

The NDL description to be used for the Coad and Yourdon class diagram modelling language is given in line 6 of the Coad and Yourdon entry in the MDT. This NDL

[61] The other entries will be discussed in subsequent sections of this chapter.

description contains definitions of the symbols and connections that are used for the syntax of the Coad and Yourdon class diagram modelling language. Figure 8-4 (a) shows a Coad and Yourdon Class&Object symbol rendered by the CASE tool client. The corresponding NDL specification, from which the Class&Object symbol is generated, is given in Figure 8-4 (b).

a)



b)

```
Symbol_Template class
{
   hght = + 4 grpHeight(grp) # 0: classheight
   wdth = + 4 grpWidth(grp)  # 1: classwidth
   dy = - hght 5             # 2: dockingY
   dx = - wdth 5             # 3: dockingX

   # (i) components
   GROUP grp (_classnamegroup||_classgroup) AT (2, 2)
   LINE (5, 0)(dx, 0)
   LINE (wdth, 5)(wdth, hght)
   LINE (dx, hght)(5, hght))
   LINE (0, dy)(0, 5)
   ARC (0, 0)(10, 10), 90, 90
   ARC (- wdth 10, 0)(wdth, 10), 0, 90
   ARC (0,- hght 10)(10, hght), 180, 90
   ARC (- wdth 10,- hght 10)(wdth, hght), 270, 90

   BOUNDING RECT (wdth,hght)

   # (ii) docking areas
   # instance, whole-part, message connection
   LINE DA 0 (0,5)(0, dy) u 10
             (instance whole-part message)
   LINE DA 0 (5,hght)(dx,hght) u 10
             (instance whole-part message)
   LINE DA 0 (wdth,dy)(wdth,5) u 10
             (instance whole-part message)
   LINE DA 0 (dx,0)(5,0) u 10
             (instance whole-part message)

   #inheritance
   LINE DA 0 (2,5)(2,dy) u 10 (inheritance)
   LINE DA 0 (5,- hght 2)(dx,- hght 2) u 10 (inheritance)
   LINE DA 0 (- wdth 2,dy)(- wdth 2,5) u 10 (inheritance)
   LINE DA 0 (dx, 2)(5, 2) u 10 (inheritance)

   DEFAULT TEXT (grp,9) # text field contained in grp
}
```

Figure 8-4 - Symbol template for the Coad and Yourdon Class&Object symbol

Figure 8-4 (b)(i) consists of a collection of segment templates[62]. The first segment template is a group reference. The default group template (*_classnamegroup*) defines the three inner compartments and a surrounding round rectangle. Clicking on the arrow in the upper right-hand corner of the symbol in Figure 8-4 (a) will cause this group to change to an instance of *_classgroup*. The remaining eight segment templates define the outer round rectangle of the symbol (which denotes the 'objects' in Coad and Yourdon).

Figure 8-4 (b)(ii) lists eight docking areas. The first four define where whole-part, instance and message connections may be attached. These types of connection may be attached anywhere on the outer round rectangle of the symbol (except on the curves). The second group of four docking areas define that inheritance connections can only attach at the inner round rectangle (but, again, not on the curves).

Figure 8-5 (a) shows an example Coad and Yourdon message connection rendered by the CASE tool client. The corresponding NDL specification, from which the message connection is generated, is given in Figure 8-5 (b) and Figure 8-5 (c).

Three types of template are used to define connections in NDL (connection template, connection symbol template and connection terminator template). Figure 8-5 (b) shows NDL connection terminator templates for the Coad and Yourdon message connection. The first (*_default*) describes a single line. This connection terminator template is used for connections that do not have a symbol at their end points (that is the inheritance, instance and whole-part connections in Coad and Yourdon). The second connection terminator template defines a simple arrow head. The arrow head can be seen attached to the *Destination* class in Figure 8-5 (a).

Figure 8-5 (c) shows the NDL definition for the Coad and Yourdon message connection. The line templates and the bounding rectangle are used to draw an icon for a button that will appear on the CASE tool client toolbar[63]. The next NDL statement specifies that the message connection is a binary connection. The terminator list specifies the connection terminator templates that are used for each end point of the connection[64]. In this example

---

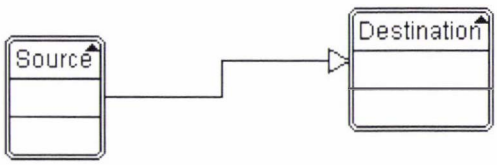
[62] The '#' symbol is used to indicate a comment in NDL.

[63] Icons for symbol templates are currently automatically generated by rendering the symbol onto the toolbar button.

[64] The connection symbol template entry is optional. It appears before the terminator list, if needed.

the *_default* connection terminator template is used at the beginning of the connection and the *_messageDestination* connection terminator template is used at the end.

a)



b)

```
Connection_Terminator _default
{
   LINE (0,3)(12,3)
   BOUNDING RECT (12,6)
   HEAD (0,3)
   TAIL (12,3)
}

Connection_Terminator _messageDestination
{
   LINE (10,6)(12, 6)
   LINE (0,6)(10,0)(10,12)(0,6)
   BOUNDING RECT (12,12)
   HEAD (0,6)
   TAIL (12,6)
}
```

c)

```
Connection_Template message
{
   # toolbar icon
   LINE (3,3)(12,3)
   LINE (0,3)(3,0)(3,3)(0,3)
   BOUNDING RECT (12,6)
   ARITY 2
   TERMINATOR _default _messageDestination
}
```

Figure 8-5 - Representing the Coad and Yourdon message connection

A portion of the Notation Semantic Mapping (NSM) table used to associate the syntax and semantic descriptions for the implementation of Coad and Yourdon is given in Figure 8-6.

Figure 8-6 (a) is the action map. The number 0 indicates an update action. The second number is a unique NDL ID that corresponds to a text field. The last part is the SSL message that corresponds to the action. The first entry in the action map, for example, specifies that the SSL message *changeclassname-S#* will be used to implement an update

action on the text field with ID 9 (the field with ID 9 contains the class name in this example).

```
a)
0 9  changeclassname-S#
0 12 addattribute-S#
0 15 addoperation-S#

b)
class            gookb_modelelement:class      create-#
abstract_class   gookb_modelelement:class      create-B#

inheritance gookb_modelelement:inheritance                2
     _default                2
     create-Vgookb_modelelement:class#Vgookb_modelelement:class##
whole-part  gookb_modelelement:whole_part                 2
     _default                2
     create-Vgookb_modelelement:class#Vgookb_modelelement:class##
instance    gookb_modelelement:association                2
     _default                2
     create-Vgookb_modelelement:class#Vgookb_modelelement:class##
message     coadyourdon_modelelement:messageconnection    2
     _default                1
     _messageDestination     1
     create-Vgookb_modelelement:class#Vgookb_modelelement:class##

c)
class            addclass-Vgookb_modelelement:class##
abstract_class   addclass-Vgookb_modelelement:class##
inheritance
   addinheritance-Vgookb_modelelement:inheritance##
whole-part
   addwholepart-Vgookb_modelelement:whole_part##
instance
   addassociation-Vgookb_modelelement:association##
message
   addrelationship-Vcoadyourdon_modelelement:messageconnection##
```

Figure 8-6 NSM table for Coad and Yourdon

Figure 8-6 (b) contains the create concept and create relation maps. The Coad and Yourdon Class and Class&Object symbols (line one and two respectively of Figure 8-6 (b)) are both represented by an instance of *Class* in the GOOKB. The two different create messages are used to ensure that Class symbols represent abstract classes, whilst Class&Object symbols represent concrete classes.

Figure 8-6 (c) contains a portion of the add map. The six entries in the add map define the SSL message that is used to add an item that corresponds to a particular NDL

template, to a diagram. This message is sent to the SSL object that represents the diagram, with the item as an argument.

Figure 8-7 shows the implementation of the *addAttribute* operation (an SSL method) that is referenced in the second line of the action map in Figure 8-6 (a).

```
// implmentation of operation addattribute for
// gookb_modelelement:class

critic addAttribute( string a )
   Attribute toAdd, current;
   ComplexCritic addCritic;
   SimpleCritic s;
   Iterator[Attribute] attr;
{
   // a) create a new attribute
   toAdd = self.newAttribute( a );
   addCritic = ComplexCritic.create( toAdd.isValid() );

   // b) check to see if it is a duplicate
   attr = theAttributes.front();
   loop
   {
      endloop when( attr.end() );
      current = attr.item();
      if( toAdd.isSameAs( current ) )
      {
         addCritic.add(
              SimpleCritic.create( false,
                 "there is already an attribute called "+a )
            );
      }
      attr.next();
   }
   if( addCritic.isOK() ) { theAttributes.add( toAdd ); }
   return addCritic;
}
```

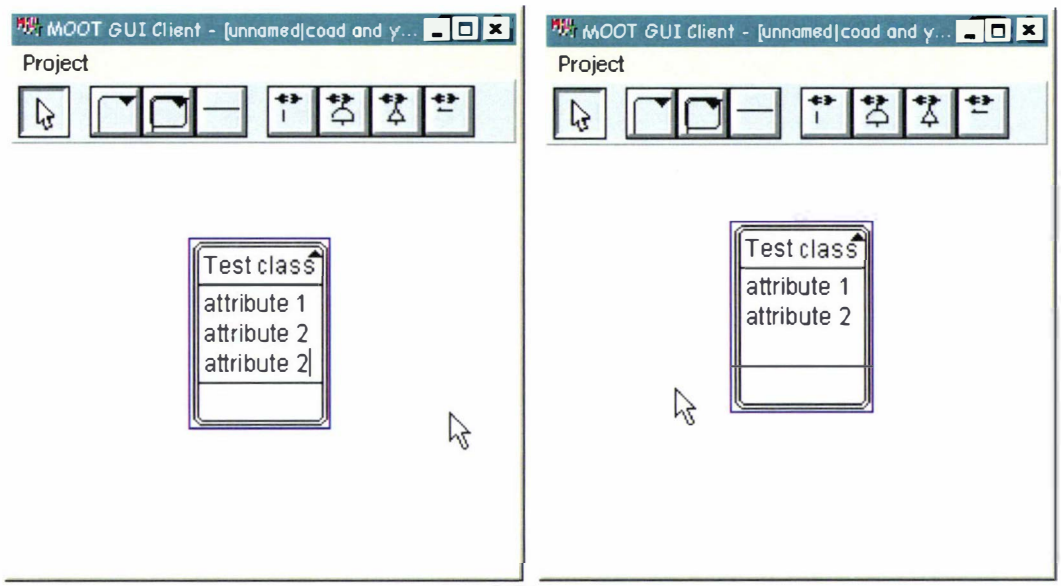
Figure 8-7 Implementation of the addAttribute operation

The *addAttribute* method given in Figure 8-7 is a method of the SSL class '*Class*', defined in the G●OKB. It is executed whenever a new attribute is added to the class.

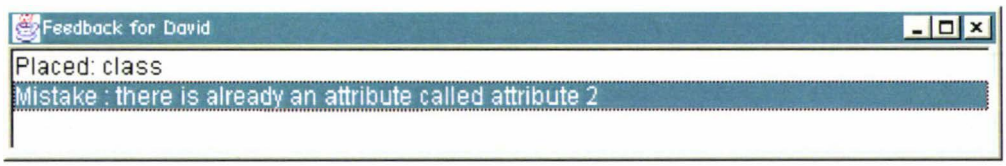
In Figure 8-7 (a), a new SSL object (*toAdd*) is created to represent the new attribute. This is achieved by sending the *newAttribute* message to the *self* object[65]. The new SSL object is then sent an *isValid* message, which returns an instance of *Critic* as a result.

[65] The *newAttribute* operation may be overridden in sub-classes to generate an instance of an appropriate SSL class.

The code fragment in Figure 8-7 (b) determines if the newly created attribute is unique within the context of the class. It does this by iterating over the collection of attributes in the class (*theAttributes*) and comparing each one to the newly created attribute. If the new attribute (*toAdd*) is found to be a duplicate a *Critic* object is created, with an appropriate explanation. Figure 8-8 illustrates these steps by showing an example of the CASE tool client in action.



(a) the user tries to duplicate an attribute **(b)** the action has been disallowed



(c) the explanation generated by the MOOT core

Figure 8-8 - Adding an attribute

In Figure 8-8 (a) the software engineer has created a single class. The first message in the feedback window (Figure 8-8 (c)) corresponds to the successful creation of the class. The user has then started to add several attributes. The software engineer is entering a third attribute in Figure 8-8 (a), but has mistakenly duplicated the previous attribute name. When the software engineer presses the enter key, or de-selects the class, a request to create a new attribute is propagated from the CASE tool client to the MOOT core. The MOOT core uses the NSM table in Figure 8-6 to determine that an *addAttribute* message must be sent to the SSL object that corresponds to *Test class* in Figure 8-8 (a). The result of processing the *addAttribute* message will be a *Critic* object that indicates an error has

occurred. The resulting explanation is returned to the CASE tool client with a fail response. Figure 8-8 (b) shows that the CASE tool client has cleared the offending text field and added the explanation to the feedback window in Figure 8-8 (c).

Figure 8-9 shows a screen snapshot of MOOT used to capture a model of Object-Orientated Analysis (page 205 of Coad and Yourdon (1991a)).

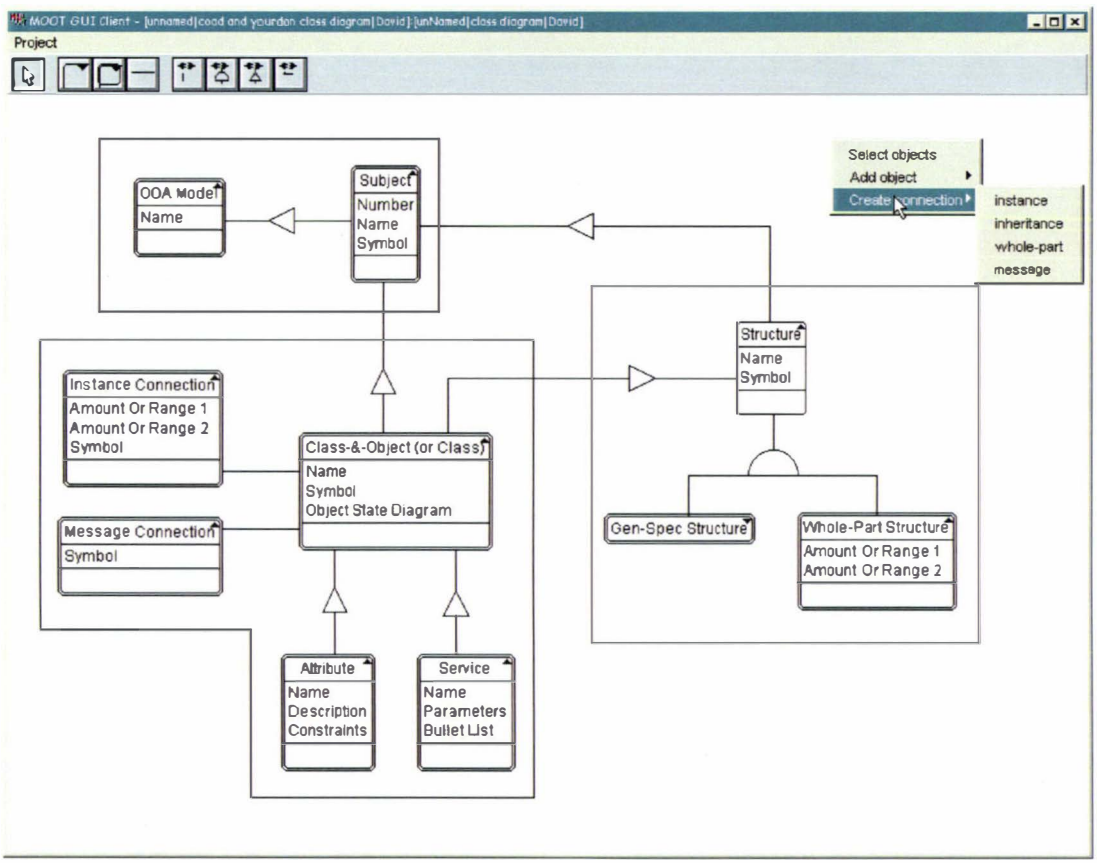


Figure 8-9 - An Object-Orientated Analysis model of Object-Orientated Analysis (Coad and Yourdon, 1991a)

8.3 Supporting Patterns

Patterns are an important concept that has gained much interest in the object-orientated literature (Gamma *et al.*, 1995; Fowler, 1997; Pree, 1994). The idea behind patterns is very simple yet extremely powerful. It provides a standard vocabulary for software engineers to use when developing systems. System developers can now talk in terms of larger components than class and object and understand phrases such as "Abstract Factory", "Adapter" and "Chain of responsibility." The advent of patterns is so important that a

technical decision has been made to support patterns with MOOT. This is the first step toward the support of Component Based Development.

> "One thing expert designers know *not* to do is solve every problem from first principles. Rather they re-use solutions that have worked for them in the past. When they find a good solution, they use it again and again. Such experience is part of what makes them experts. Consequently you'll find recurring patterns of classes and communicating objects in many object-oriented systems. These patterns solve specific design problems and make object-oriented designs more flexible, elegant, and ultimately reusable."
>
> *From Design Patterns: Elements of Reusable Object-Oriented Software*
> *(Gamma et al., 1995)*

There are several requirements that the support for patterns in MOOT had to satisfy.

1. It should be possible to instantiate a pattern on any class diagram. This must include class diagrams created with modelling languages that have not yet been defined in MOOT.
2. There should be a well-defined protocol or method for adding new patterns in the future.

The support for patterns was implemented in the GOOKB by a set of abstract super-classes, which satisfies requirement 1. The classes involved define the protocol for the instantiation of a pattern on a class diagram. The protocol between these classes satisfies requirement 2.

A new class (*Pattern*) has been defined in the GOOKB. The interfaces of some of the existing classes have been extended (*Object Model*, *Object Diagram* and *Class*). The implementation of patterns is shown in Figure 8-10.

Concrete patterns are implemented as sub-classes of the abstract super-class *Pattern* (Figure 8-10). Each pattern implements the instantiate operation[66]. The instantiation process uses an instance of *Object Model* and an instance of *Object Diagram*. A pattern object instantiates itself onto the *Object Diagram* object with the assistance of the *Object*

[66] They may also overload instantiate with an operation that takes additional arguments.

Model object. The Pattern hierarchy is an example of the Template method pattern (Gamma *et al.*, 1995), where the entire instantiation process is different for each sub-class of Pattern.

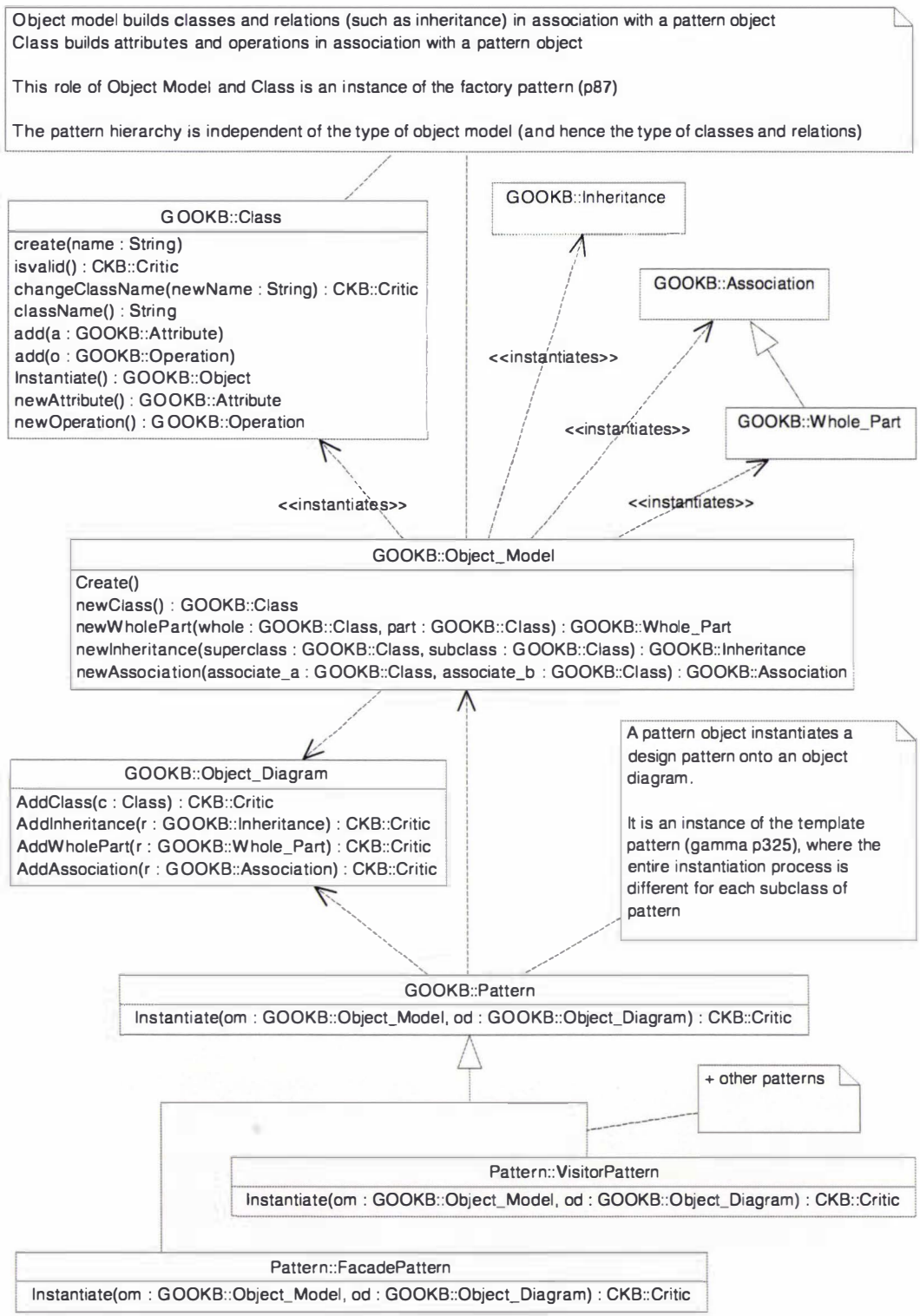


Figure 8-10 - Extending the GOOKB to support Patterns

The *Object Diagram* class in Figure 8-10 provides operations for an *Object Diagram* object to add new classes and relations to itself. The interface of *Object Model* has been extended with operations for creating new classes and relations. A pattern object delegates the responsibility for creating classes and relations to an *Object Model* object. This role of *Object Model* is an example of the Abstract factory pattern (Gamma *et al.*, 1995, p87). The interface of *Class* has been similarly extended with operations for creating new attributes and operations. This role of *Class* is also an example of the Abstract factory pattern (Gamma *et al.*, 1995). The use of the Abstract factory pattern in *Object Model* and *Class* ensures that the Pattern hierarchy is independent of the type of object model (and hence the type of classes and relations).

New types of object model that are implemented in MOOT are always defined as extensions of the *Object Model* class in the GOOKB. Extensions of *Object Model* must override the *newclass*, *newInheritance*, *newWholePart* and *newAssociation* operations to support patterns. New types of class are always defined as extensions of *Class* in the GOOKB. Extensions of *Class* must override the *newAttribute* and *newOperation* operations to support patterns.

8.4 Supporting UML

Work in progress to support the UML v1.1 meta-model (OMG, 1997c-j) as an extension of the CKB and GOOKB is documented in Figure 8-11, Figure 8-12 and Figure 8-13.

Figure 8-11 shows how the following packages from the UML specification (OMG, 1997i) have been modelled as extensions of the CKB and GOOKB:

- UML v1.1 Foundation: CORE: Backbone
- UML v1.1 Foundation: CORE: Extension Mechanisms
- UML v1.1 Foundation: CORE: Auxiliary Elements

Figure 8-12 and Figure 8-13 show how the following packages from the UML specification (OMG, 1997i) have been modelled as extensions of the CKB and GOOKB:

- UML v1.1 Behavioural Elements: Collaborations
- UML v1.1 Behavioural Elements: Common Behaviour

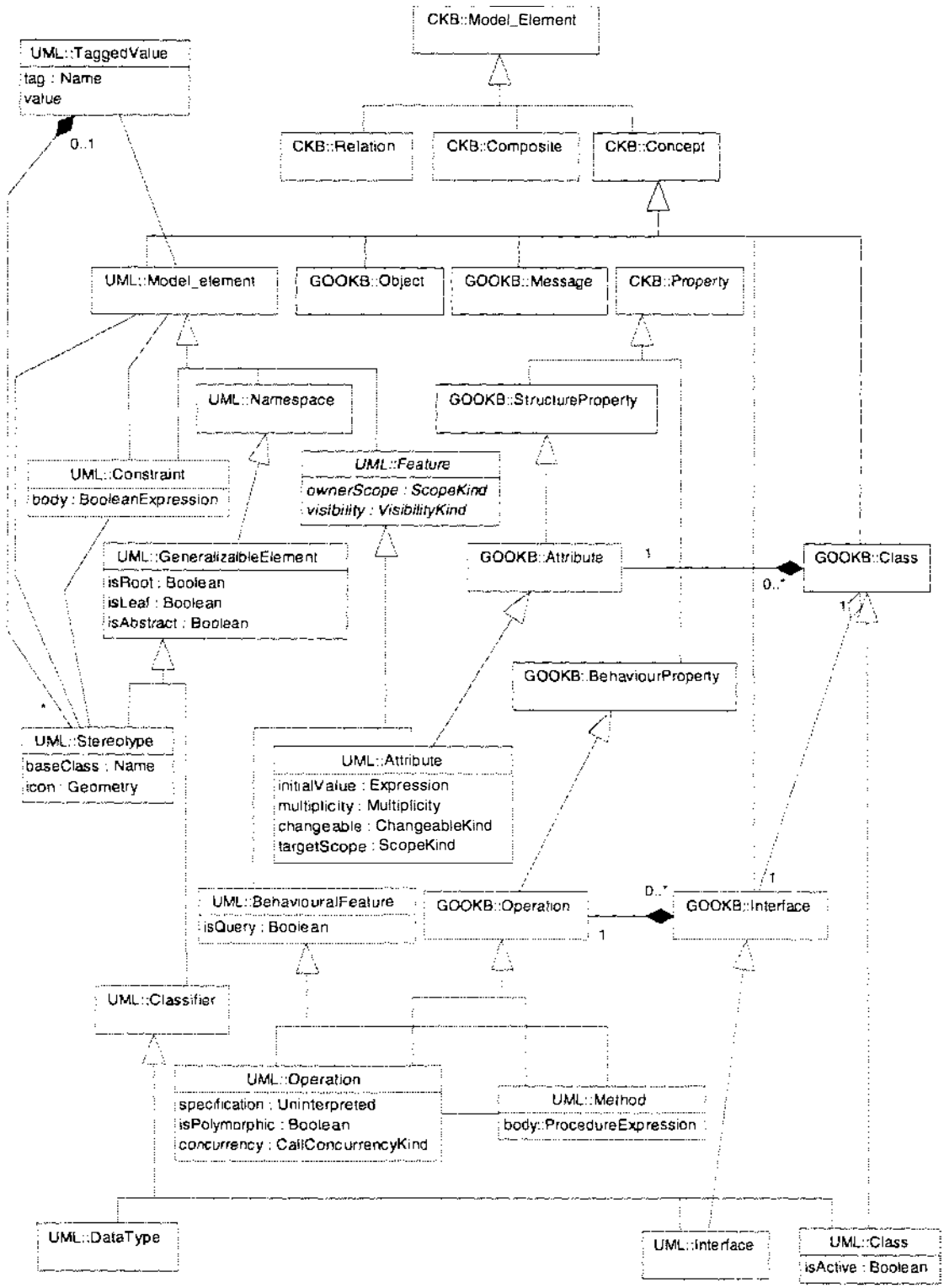


Figure 8-11 - UML v1.1 Foundation: CORE: Backbone + Foundation: CORE: Extension Mechanisms + Foundation: CORE: Auxiliary Elements

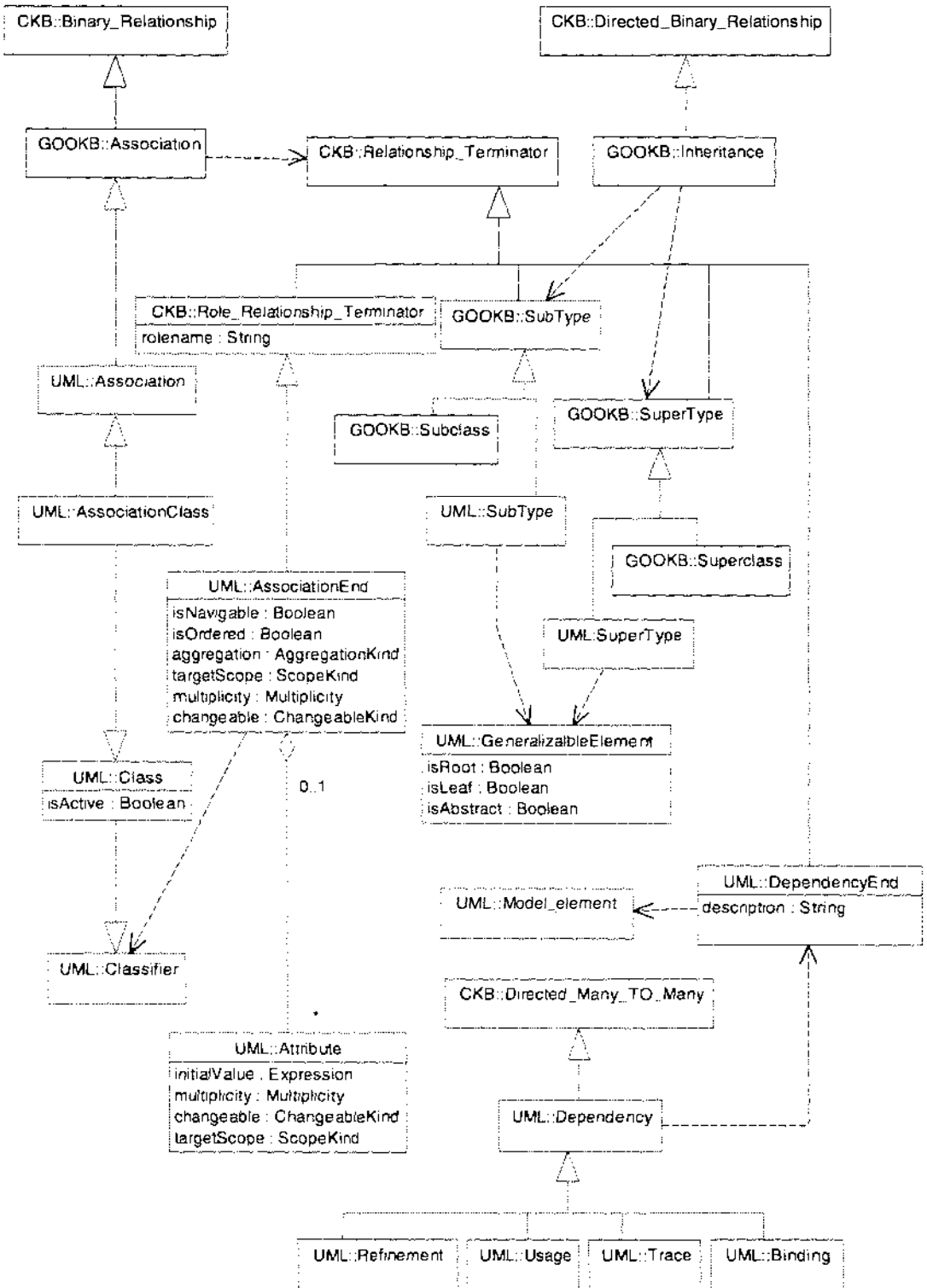


Figure 8-12 - UML v1.1 Behavioural Elements: Collaborations

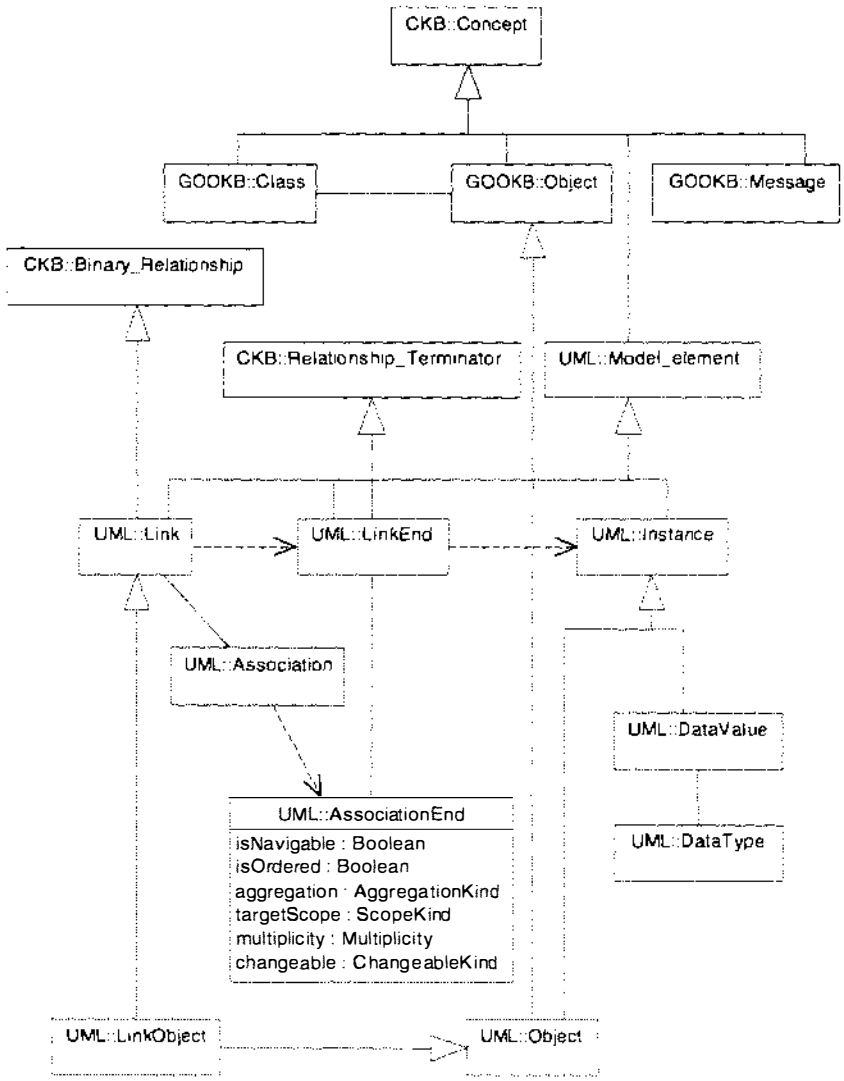


Figure 8-13 - UML v1.1 Common Behaviour: Common Behaviour

The definition of the core UML meta-model as an extension of the CKB and GOOKB validates the novelty, flexibility and extensibility of the MOOT approach as UML is an example of a new methodology that did not exist when the research commenced.

8.5 Preliminary Development of the Semantics Editor

The Semantics editor (Figure 3-10 - Proposed, top level, system architecture) is used by methodology engineers to define the semantics of methodologies in SSL. The development of the semantics editor by using MOOT is a bootstrapping approach where NDL and SSL are used to develop a tool for building SSL specifications.

Two modelling languages are proposed to support the development of SSL specifications. These are:

- SSL module structure modelling language. The purpose of this language is to define the class and module structure of an SSL specification. It supports the construction of a collection of diagrams, each of which corresponds to an SSL module.
- SSL method modelling language. The purpose of this language is to define the implementation of an operation shown in an SSL module structure model.

Figure 8-14 shows how the two modelling languages are supported in MOOT.

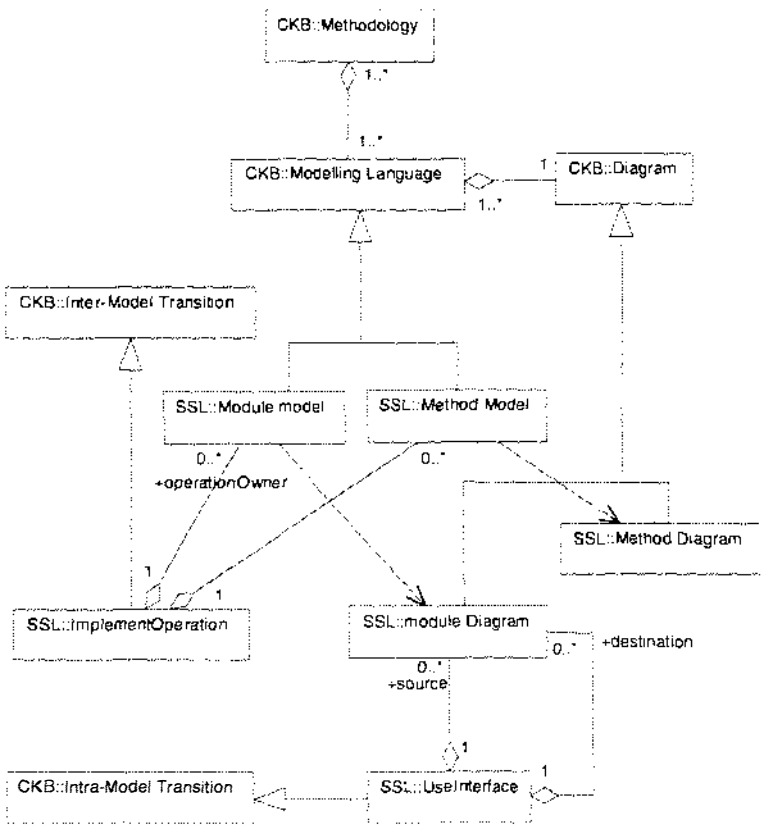


Figure 8-14 - SSL modelling languages

The two modelling languages are represented by sub-classes of *Modelling Language* (*Module Model* and *Method Model*). *ImplementOperation* is an inter-model transition between an *SSL Module Model* and an *SSL Method Model*. The implementation of an operation in a *SSL Module Model* by a method is represented by an instance of *ImplementOperation*. *UseInterface* is an intra-model transition between two SSL module structure diagrams. This transition corresponds to using a class in a module that is defined in another module.

Figure 8-15 documents work in progress, to define the semantics of SSL as an extension of the GOOKB. This extension is called the SSL Knowledge BASE (SSLKB). The SSLKB is partially designed and has not been implemented in SSL.

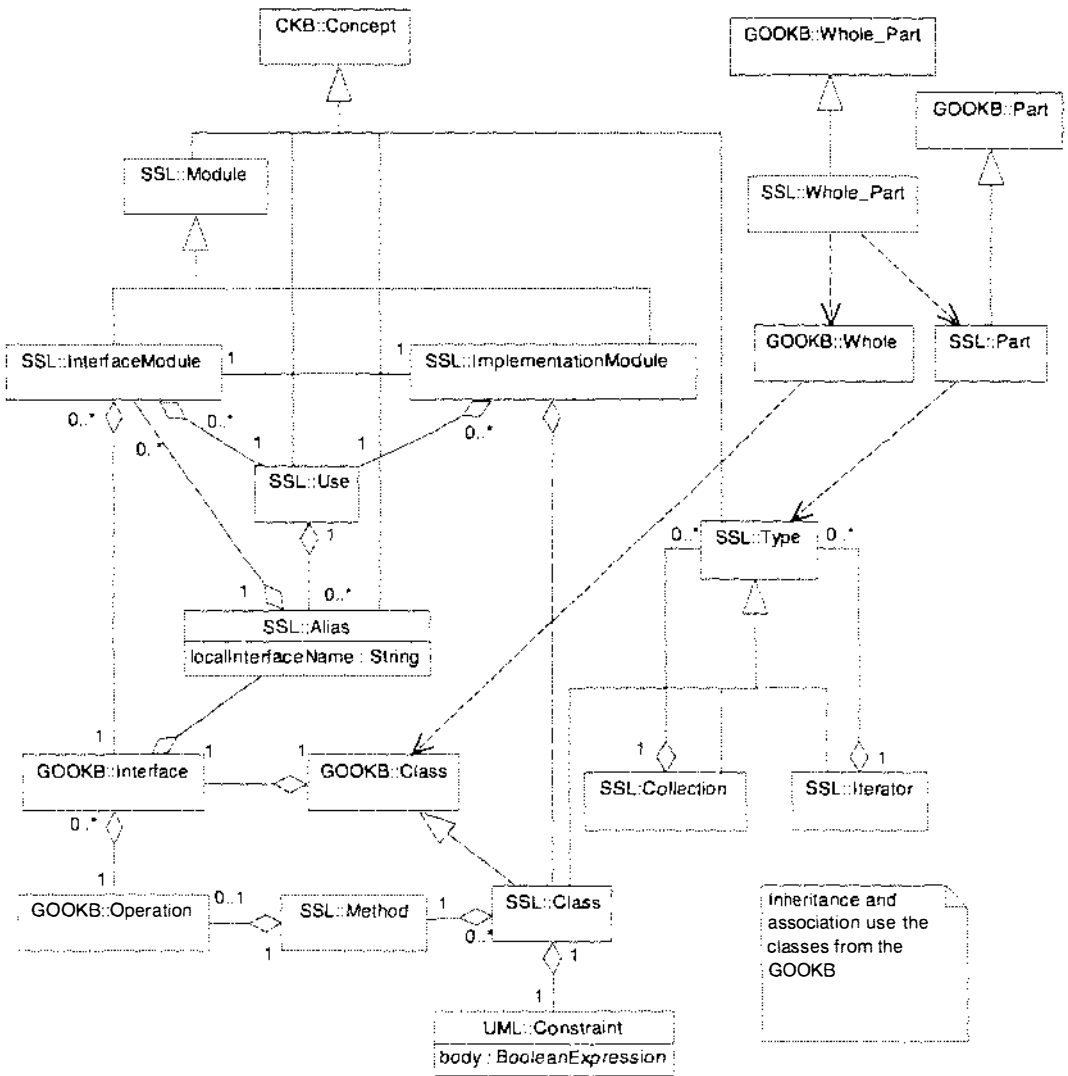


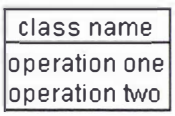
Figure 8-15 - Representing SSL as an extension of the GOOKB

Figure 8-15 represents the module structure of SSL. An *Interface Module* contains zero or more *Interfaces*. *Implementation modules* contain zero or more *SSL Classes*. Each *Class* in an *implementation module* corresponds to an *interface* in an *interface module*. The SSLKB defines an extension of *GOOKB Class* (*SSL Class*) as SSL classes also have a constraint. *SSL Class* is also a sub-class of *Type*. The other Sub-classes of *Type* include *Collection* and *Iterator*. Inheritance and association are already supported in the GOOKB. Figure 8-15 shows an extension of *GOOKB Whole_Part* and *GOOKB Part* that permits any SSL type to take the role of 'part' in a whole-part relation.

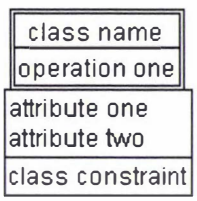
The development of NDL specifications for the two proposed modelling languages have been conducted independently from the development of the SSLKB.

8.5.1 Notation for the SSL Module Structure Modelling Language

There are two symbols in the notation of the SSL module structure modelling language.



Interface symbols represent SSL class interfaces. An interface symbol has two compartments, one for the class name and one for the operations. Each use of a class from another SSL module corresponds to an interface in an SSL module model.



The Class symbol has three compartments. The top compartment contains an interface. The last two compartments contains the attributes and a constraint.

The current implementation of the SSL module modelling language uses elements from the CKB as its semantic definition. SSL classes and interfaces are represented by an instance of *CKB concept*. Inheritance and Using relations are represented by instances of *CKB Directed Binary Relation*. A portion of the NSM table used is given in Figure 8-16.

```
a)
0 11 setlabel-S#
0 12 setlabel-S#
0 13 setlabel-S#
0 14 setlabel-S#
b)
Interface   ckb_modelelement:concept create-#
Class       ckb_modelelement:concept create-#
Inheritance ckb_modelelement:directed_binary_relationship 2
     _plain                 1
     _inheritance_end       1
     create-Vckb_modelelement:concept#Vckb_modelelement:concept##
Uses ckb_modelelement:directed_binary_relationship 2
     _plain                 1
     _arrow_end             1
     create-Vckb_modelelement:concept#Vckb_modelelement:concept##
c)
Interface     addconcept-Vckb_modelelement:concept##
Class         addconcept-Vckb_modelelement:concept##
Inheritance   addrelationship-Vckb_modelelement:relationship##
Uses          addrelationship-Vckb_modelelement:relationship##
```

Figure 8-16 - NSM table for the SSL module modelling language

Figure 8-16 (a) shows the NSM action map. The update of each field is mapped to a *setlabel* message (which takes a single string argument). Figure 8-16 (b) contains the create concept and create relation maps. Classes and interfaces are represented by an instance of *Concept* from the CKB. All relations are represented by an instance of *Directed Binary*

Relation from the CKB. Figure 8-16 (c) contains a portion of the add map. These entries specify the message that is used to add an item that corresponds to a particular NDL template to a diagram.

Figure 8-17 shows a snapshot of MOOT being used to draw an SSL module structure diagram that corresponds to the *Critic* module of the CKB (see Figure 6-11 - Critics).

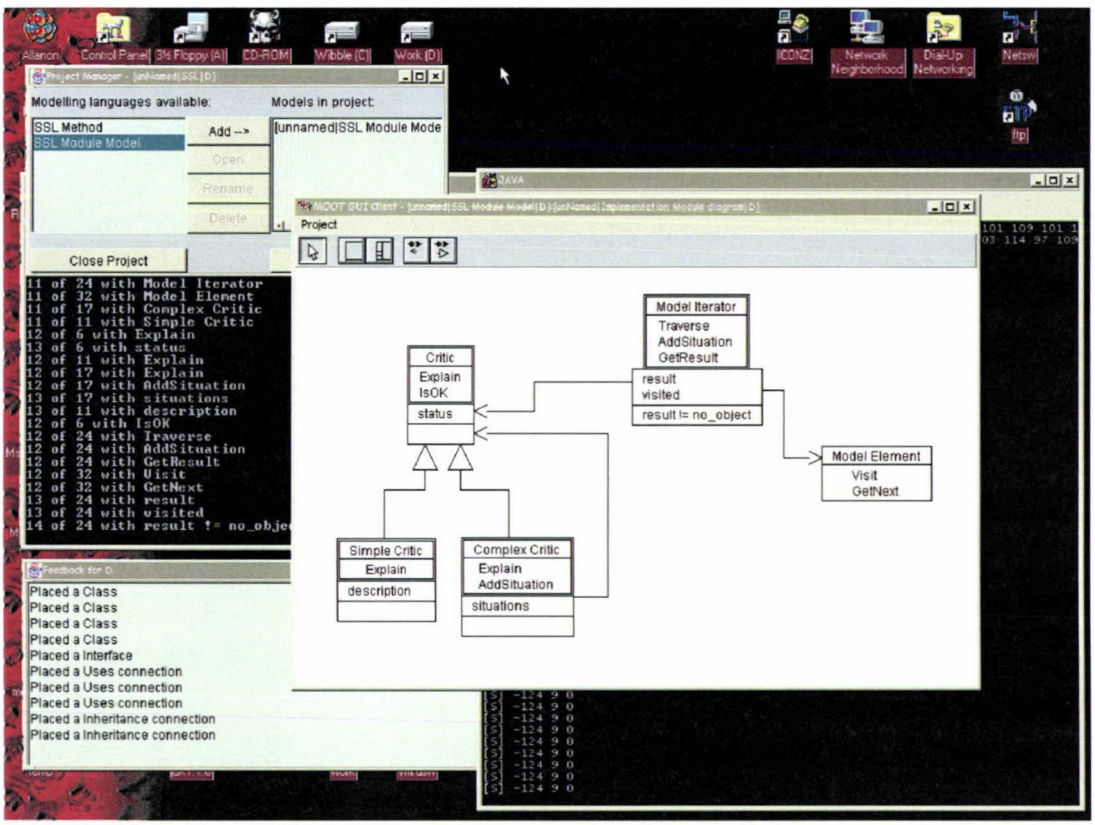
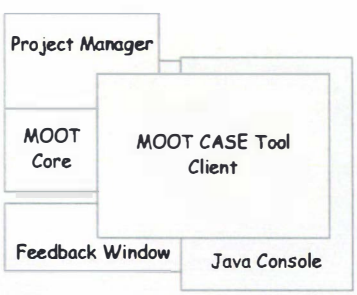


Figure 8-17 - Supporting SSL with MOOT

The example in Figure 8-17 shows the CASE tool client and the MOOT core running on the same machine. A description of each window is given below.



MOOT CASE Tool Client

The interface of the CASE Tool client consists of a drawing surface, a toolbar and a menu bar. The toolbar is generated automatically from the NDL specification for the current modelling language. This example also shows inheritance and uses connections.

MOOT Core

The output displayed in the console corresponds to the translation of actions at the CASE tool client.

Feedback Window

Explanations generated by the MOOT core and feedback related to the successful creation of symbols and connections are displayed in this window.

Project Manager

The project manager is used to manipulate the models in a project.

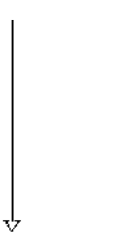
Java Console

The output displayed in the console corresponds to success response packets sent from the MOOT core in response to a request by the CASE tool client.

8.5.2 Notation for the SSL Method Modelling Language

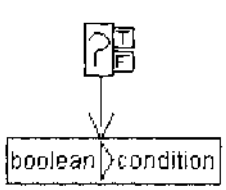
The proposed SSL method modelling language is used to define the SSL code in the body of an SSL method. The notation described here is a proposal, whose primary purpose is to demonstrate that a notation of this type can be described successfully with NDL.

The proposed notation has fourteen symbols and three connections. The symbols correspond to SSL statements, SSL operators and values. The connections represent invocation, value and part relations. The symbols and connections are discussed below.



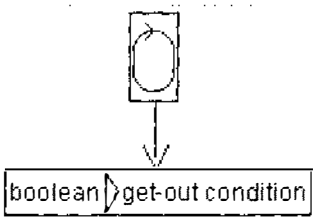
Statement block

This symbol represents a sequence of statements, which are connected to the statement block with invocation connections. The order of the invocation connections represents the order of execution of the statements.



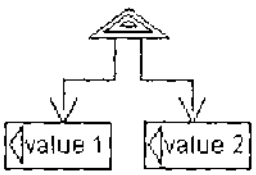
If statement

The boolean condition is attached to the bottom of the if symbol with a value connection. This example shows a variable symbol whose name is *condition* and whose type is *boolean*. Single statements can attached to the T and F parts of the if symbol.



Loop statement

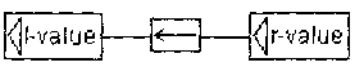
The get-out condition is attached to the bottom of the symbol with a value connection. In this example the get-out condition is a *boolean* variable called *get-out condition.*



Return statement

The return statement indicates the end of a method. The values that are returned a result of the method are attached at the bottom of the symbol with value connections.

Assignment statement

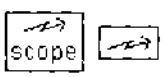


The assignment statement is used to update the value of a variable. This example can be read as 'l-value *is given the value of* r-value'.

Arithmetic operators

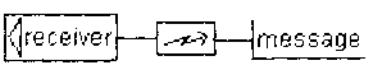
Each arithmetic operator symbol may accept multiple value connections at the top and have one or two value connections at the bottom.

Message send operators



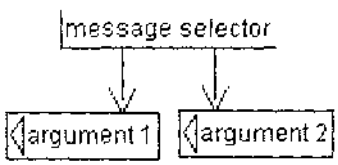
The two message-send operators are used to represent the binding of messages to an SSL object, SSL collection or SSL iterator.

Part connection

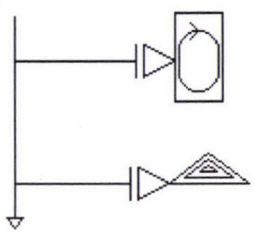


The part connection is used to associate an assignment operator with its l-value and r-value and to associate a message send operator with its message receiver and message. In this example the message *message* is being sent to an object called *receiver.*

Value connection



Value connections represent the evaluation of an expression. This example shows a message with two arguments *argument 1* and *argument 2.*



Invocation connection

The invocation connection represents the execution of a statement. In this example a loop statement and then a return statement are being invoked, from within a statement block.

Figure 8-18 shows SSL code for the *Explain* method of the *CompositeCritic* SSL Class defined in the Core Knowledge Base. A *ComplexCritic* object encapsulates a collection of other critic objects. The *Explain* method concatenates the explanations generated by the component critic objects.

```
// Explain method from CKB::CompositeCritic

string Explain()
   string explanation;
   Iterator[ Critic ] sit;
   Critic c;
{
   // if there is no problem return ok
   if( isOK() )
   {
      return "ok";
   }
   else
   {
      // otherwise build an explantion
      explanation = "";
      // situations is a collection of critic objects
      sit = situations.front();
      loop
      { // finish when we have checked everyone
         endloop when( sit.end() );
         c = sit.item();
         // update the explanation
         explanation = explanation + c.Explain();
         sit.next();
      }
      // return the result
      return explanation;
   }
}
```

Figure 8-18 - Explain method of the *ComplexCritic* class in the CKB

Figure 8-19 shows the corresponding SSL method model and shows how the explain method can be captured using the proposed notation.

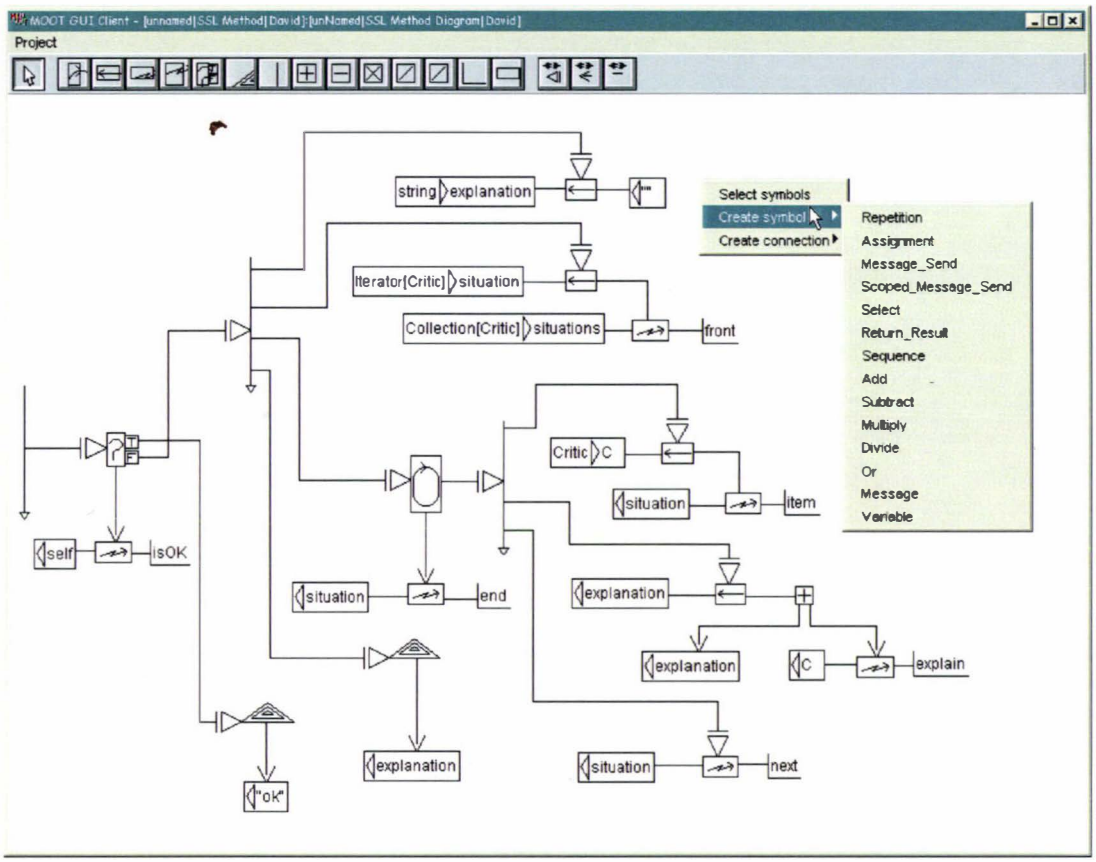


Figure 8-19 - An example SSL method model

Whilst no claim is made to suggest that MOOT should be used to implement visual programming languages, this result is interesting because the development of the notation for the SSL method modelling language (with 14 symbols, three types of connection and logical distortion) was achieved in a matter of hours.

8.6 Toward Supporting Joosten Workflow Modelling

This work started after an expression of interest, by another researcher, to use MOOT to model and implement several workflow methodologies. The aims of this work are to:

- Model several workflow methodologies.
- Derive a meta-model of workflow methodologies. This work is similar to the GOOKB in scope and intent.
- Assess the MOOT approach when used to model a non object-orientated methodology.

This research is in its preliminary stages. An NDL specification of the Joosten (Joosten, 1995) workflow methodology has been derived. An example Joosten trigger model (Joosten 1995) which has been drawn using MOOT is given in Figure 8-20.

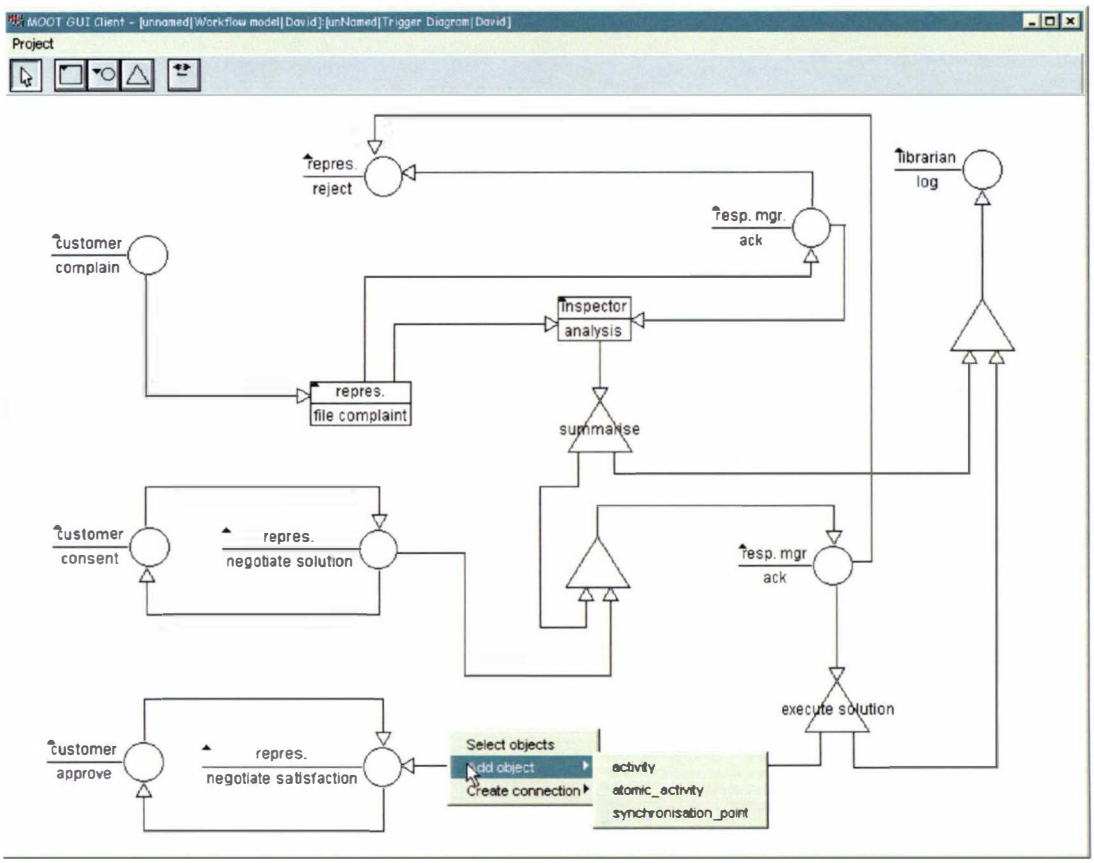


Figure 8-20 - Joosten trigger model (Joosten, 1995)

8.7 Summary

This chapter has presented a series of examples showing how the results of the research, as discussed in chapter 3 - 7, can be applied. This included:

- Describing Coad and Yourdon
- Implementing the support for patterns as part of the GOOKB
- Initial work on describing the UML meta-model as an extension of the GOOKB
- Initial work on supporting the semantics editor (as described in the description of the proposed architecture in chapter 3) using MOOT
- Initial development of the Joosten Workflow methodology

The successful modelling and implementation of patterns and UML (recent advances in object technology), which did not exist when the research commenced, shows the innovative and original nature of the approach and the new methodology representation strategy. This research is a significant move toward building adaptive systems; the ultimate future of software engineering.

Chapter 9

Conclusion and Future Work

If we knew what it was we were doing, it would not be called research, would it?

Albert Einstein

9.1 Introduction

The objectives of the research, as stated in section 1.7, are:

- Develop a novel meta-CASE tool methodology representation strategy that:
  - Uses an object-orientated meta-model.
  - Allows methodology descriptions to be re-used.
  - Minimises the coupling between methodology syntax and semantic descriptions such that methodology syntax and semantic descriptions can be re-used independently.
  - Permits software engineering projects to be re-used, even if they are built with different methodologies.
- Design and implement a prototype meta-CASE tool that realises the new methodology representation strategy via the development of:
  - Languages that support the description of syntax and semantics of a methodology.
  - The efficient execution strategy of syntax and semantic descriptions.

This thesis presents the development of a new modifiable CASE environment designed to satisfy these objectives (***M***eta ***O***bject ***O***rientated ***T***ool). The results of this research are manifest in the existence of NDL, SSL, SSL-BC, the SSL-VM, NSM tables, the CKB, the GOOKB and the MOOT prototype.

The thesis is summarised in section 9.2 and a critical evaluation of the study is presented in Section 9.3. Some of the future work that has been envisaged is outlined in section 9.4. Final concluding remarks are made in section 9.5.

9.2 Summary of the Thesis

Chapter 1 introduced and defined fundamental terms used throughout the thesis. The type of CASE tool that is the subject of the research was defined (methodology CASE tool). A classification hierarchy of methodology CASE tool categories was derived and a review of CASE tools was presented with respect to the hierarchy. The limitations of methodology CASE tools were discussed from organisational and CASE tool perspectives and the goals of the study presented.

Chapter 2 examined meta-modelling and meta-CASE technology. In particular the following limitations of meta-CASE technology were identified:

- Reliance on data models
- Separation of 'structural' and 'behavioural' elements of semantic descriptions, which decreases the cohesion of semantic descriptions
- High coupling between the syntax and semantic descriptions, primarily because of an assumed, fixed, mapping between elements of syntax and semantic descriptions
- No consideration of software process
- No consideration for re-use of methodology descriptions or software engineering projects
- No relation between supported methodologies
- Very poor usability

Chapter 3 presented the approach taken to address the limitations of methodology CASE tools and meta-CASE tools. The proposed architecture of a new modifiable CASE environment (MOOT) was presented. The methodology representation strategy supported by MOOT was outlined and a prototype of MOOT described.

Chapter 4 presented the development of NDL (Notation Definition Language). An overview of graphical notations used in software engineering methodologies was presented. The requirements and design of NDL were discussed and a prototype NDL interpreter (the basis of the MOOT CASE tool client) was presented.

Chapter 5 described the development of SSL (Semantic Specification Language). SSL implements the MOOT meta-model; integrates the description of 'structure' and 'behaviour'; supports more than completeness and consistency checking; emphasises

programming, rather than formally defining the semantics; supports re-use and provides efficient execution and platform independence.

SSL is an object-orientated language that supports a subset of the facilities of a general purpose programming language. It is a statically type checked language that provides clean separation between 'class interface' and 'class implementation'. SSL supports dynamic binding, multiple inheritance, built-in primitive types, polymorphic collection and iterator types and provides a module system.

Chapter 6 presented the development of the Core Knowledge Base (CKB) and the Generic Object Orientated Knowledge Base (GOOKB). The CKB was derived using a meta-modelling approach and implements a meta-model of methodology, which provides simple facilities for cognitive support. The GOOKB was derived by meta-modelling and implements a meta-model of concepts germane to all object-orientated methodologies.

Chapter 7 discussed the realisation of methodologies and software projects in MOOT. The derived Methodology Description Table (MDT), Notation-Semantic Mapping (NSM) tables and the communication protocol between the CASE tool client and MOOT core were presented. It was shown that the association of syntax and semantic descriptions involved: the relation between a software engineering project, in terms of its models, diagrams and documents, to the methodology used to create it; the relation between the syntax and semantic descriptions expressed with NDL and SSL; the relation between logical actions performed using the CASE tool client and semantic actions performed by the MOOT core.

Chapter 8 presented a series of examples showing how the results of the research can be applied. This included: implementing Coad and Yourdon's methodology; implementing support for patterns as part of the GOOKB; initial work on describing the UML meta-model as an extension of the GOOKB; initial work on supporting the semantics editor and initial development of Joosten's Workflow methodology.

Table 9-1 summarises the practical work and the publications completed as a result of this research.

Chapter	*Practical Work*	*Implementation Details*	*Related Publications*
3	Prototype of the MOOT Core	≈ 90 classes ≈ 8000 lines of C++	Page *et al.*, 1997, 1998 Mehandjiska *et al.*, 1997
4	Prototype NDL Interpreter	≈ 80 classes ≈ 5000 lines C++ ≈1600 lines tcl ≈ 450 lines of PCCTS grammar	Page *et al.*, 1994 Mehandjiska *et al.*, 1995b, 1996a
5	SSL Compiler	≈ 80 classes ≈ 5000 lines of C++ ≈ 800 lines of PCCTS grammar	Page *et al.*, 1997, 1998 Mehandjiska *et al.*, 1997
6	Core Knowledge Base and Generic Object Orientated Knowledge Base	≈ 45 classes ≈ 1000 lines of SSL	Page *et al.*, 1998 Mehandjiska *et al.*, 1996b, 1996c, 1997
7	Communication protocol in the CASE tool client	≈ 10 classes ≈ 800 lines of Java	Mehandjiska *et al.*, 1997 Phillips *et al.*, 1998b, 1998c

Table 9-1 - Practical work completed during the research

9.3 Discussion

The discussion summarises the novel meta-CASE tool methodology representation strategy. The overall MOOT approach is critically reviewed and the new modelling languages (NDL and SSL) are discussed in turn. Finally the two re-usable libraries of semantic methodology descriptions (the CKB and GOOKB) are considered.

9.3.1 The Novel Meta-CASE Tool Methodology Representation Strategy

The novelty of this research is the philosophy and implementation of the new methodology representation strategy for meta-CASE tools.

Novel Principles of the Methodology Representation Strategy

- A language for modelling methodology syntax. This is an advantage in contrast to existing approaches, which only provide simple support for 'pen and paper' notations.

- A single modelling language for representing methodology semantics. This is an advantage in contrast to existing approaches, which are typified by 'data model and separate constraints.'
- Independent development of syntax and semantic descriptions. This is supported by the scope of the modelling languages and late binding of methodology syntax and semantic descriptions.
- Re-usable methodology description components.
- Explicit relation between methodology descriptions.
- Facilities such as auto-correction, intelligent feedback and cognitive support.

Novelty of the Implementation

- An object-orientated meta-model used for a meta-CASE tool.
- NDL, a new language for describing methodology notations.
- SSL, a new language for describing methodology semantics.
- SSL-VM, a new virtual machine which supports efficient processing of SSL.
- CKB and GOOKB, two libraries of re-usable methodology semantic description components.

9.3.2 The MOOT Approach

The approach described in the thesis addresses issues related to CASE tools and meta-CASE tools. Positive and negative ramifications of the novel methodology representation strategy and its implementation in MOOT have been identified based on empirical results gained by using the MOOT prototype. The positive ramifications are related to: the adoption of an object-orientated meta-model, the scope and separation of NDL and SSL, and the emphasis on re-use. The negative ramifications are common to all meta-systems and are related to redundancy, efficiency and complexity.

9.3.2.1 An Object-Orientated Meta-Model

The integration of state and behaviour

Previous meta-CASE tools typically provide two or more separate languages for the semantic specification of methodologies. One is used to define 'structure' and the second

to define constraints on the structure (a form of behaviour). There are several problems with this approach: a) there are multiple languages for the same task b) the coupling of methodology semantic specifications increases and c) the cohesion of methodology semantic specifications decreases. MOOT addresses these issues by providing a single language (SSL) that integrates the description of structure and behaviour.

Inheritance and polymorphism

Using inheritance is the logical extension of the support for 'sub-typing' that the majority of meta-CASE tools provide. The integration of state and behaviour in SSL, combined with inheritance and polymorphism, fosters a 'model by derivation' approach to methodology meta-modelling in MOOT. This approach has significant advantages in comparison to some meta-CASE tools, which only support accidental re-use of previous methodology meta-modelling results.

Support for re-use

An object-orientated approach promotes re-use, as widely propounded in the literature. The benefits of an object-orientated approach, in terms of fostering and enabling re-use, applies to MOOT methodology semantic descriptions, as MOOT incorporates an object-orientated meta-model and meta-modelling is simply modelling, at a different level of abstraction.

9.3.2.2 Separate Syntax and Semantic Modelling Languages

Key benefits of the separation of the syntax and semantic modelling languages in a meta-CASE tool include:

- *Syntax and semantic descriptions can be developed in isolation*

  The new approach to meta-modelling in MOOT allows syntax and semantic descriptions to be derived separately. Methodology engineers with sound HCI skills can develop notations whilst those with sound modelling skills can derive methodology semantic descriptions. This permits the development of effective 'screen' notations to be considered. If it can be said, "to a user of a system, the interface is the system" (Apperley and Duncan, 1994), perhaps it can analogously be said, "to a user of a methodology, the notation is the methodology."

- *Increased cohesion and reduced coupling*

  This is a direct consequence of ensuring that the syntax modelling language can only be used to model syntax and the semantic modelling language can only be used to model semantics. Therefore, the cohesiveness of syntax and semantic descriptions must be the same as, or better than, that achieved with other meta-CASE tools. The coupling between the descriptions is certainly low as each may be developed independently.

- *Syntax and semantic descriptions can be plugged together*

  The MOOT approach fosters a culture of 'develop the semantics once' rather than developing similar semantic descriptions with different syntax, which in turn emphasises that 'different syntax' and 'same semantics' is not the same as 'different methodology.'

- *The modelling languages may be extended in the future without affecting each other*

  A complete separation of NDL and SSL ensures that each language may be extended independently in the future.

9.3.2.3 Viewing Methodology Descriptions as Potentially Re-usable Components

The development of the CKB and GOOKB was driven by the realisation of the homology of object-orientated methodologies. The fact that the CKB and GOOKB *can be built at all* is evidence of the potential of the MOOT approach. These two libraries have successfully been used to derive Coad and Yourdon's methodology, the semantics of SSL, the UML meta-model and support for patterns.

9.3.2.4 Redundancy

The existence of two separate languages in MOOT may lead to redundancy in the methodology descriptions. For example, the developer of a syntax description for an object-orientated methodology constrains inheritance connections to occur between classes, a feature also captured by the semantic description. However, MOOT syntax and semantic descriptions serve completely different purposes. Therefore the scope and representation of similar concepts (in syntax and semantic description) is different. The MOOT approach increases the cohesion of methodology descriptions and reduces the syntax – semantic coupling.

9.3.2.5 Efficiency

Meta-systems typically suffer with respect to efficiency in time and space because of the additional layers of representation they entail. There are three aspects of the MOOT system where efficiency should be considered: processing NDL specifications, processing SSL messages and mapping syntax and semantics with NSM tables. The time/space efficiency considerations include:

- *Processing NDL specifications*

  Empirical experience gained thus far, from using MOOT, indicates that the additional overhead in terms of time is not noticeable in the client. For example NDL is used to dynamically update symbols, as a user types text directly onto the drawing surface. An NDL template is subsequently interpreted, in between keystrokes, to resize affected symbols and connections. However, no delay noticeable by users of the CASE tool client[67] has been observed.

- *Size of NDL specifications*

  The space overhead of NDL specifications is insignificant. For example, a complete textual NDL definition of Coad and Yourdon, including support for logical distortion, is approximately 4500 bytes. The major overhead in the client is the space it takes to represent NDL templates in memory. Currently the client parses the NDL specification and builds an abstract syntax tree for each template. The overhead is, however, not large.

- *Processing SSL messages*

  Two aspects of processing SSL messages have been considered. The first is the time taken to execute the body of a method. SSL is compiled to a platform independent binary representation to address this issue. The second is the time it takes to bind a message to an SSL object. SSL is statically type checked to address this issue. In addition the SSL class run-time representation includes a method lookup table.

  This research has not been concerned with multi-user access, so the current prototype only implements very primitive multi-user facilities[68]. The impact of object

[67] For example on a low-end computer, such as an Intel Pentium 150 based machine running Windows-95.

[68] Multiple users can connect to the MOOT core and all requests for access to SSL objects automatically succeed.

level locking, therefore, cannot be qualitatively or quantitatively assessed at this time, although it is expected to be significant.

- *Size of SSL specifications*

  The space overhead of maintaining semantic specifications (SSL classes in MOOT) is no greater than that of other meta-CASE tools[69].

- *Applying NSM tables*

  Empirical evidence gained from using MOOT shows that the run-time cost of implementing late binding of syntax and semantic descriptions, with NSM tables, is not significant in comparison to processing NDL and SSL.

- *Size of NSM tables*

  The space overhead of NSM tables is insignificant in comparison to the NDL and SSL specifications that comprise a methodology description and the NDL views and SSL objects that comprise a software engineering project.

9.3.2.6 Complexity

MOOT is more complex than a methodology CASE tool and *some* existing meta-CASE tools as two languages are needed to describe a methodology (NDL and SSL). However, the contention of the MOOT approach is that the scope and separation of these languages provide significant advantages to the methodology engineer, which compensate for the complexity.

The current MOOT prototype requires a methodology engineer to write code in NDL, SSL and develop NSM tables by hand. Learning two new languages constitutes a significant learning overhead. This issue can be resolved by providing visual editors to aid the methodology engineer in the task of creating methodology specifications (see the Semantics editor, Notation editor and Methodology editor in Figure 3-10 - Proposed, top level, system architecture).

9.3.2.7 Structure of the Persistent Store

The meta-modelling approach adopted requires careful design of the MOOT repository. Methodology descriptions consist of a collection of SSL classes, NDL specifications, an

[69] Except that MOOT supports more than completeness and consistency checking.

NSM table and an entry in the Methodology Description Table. Software engineering projects consists of a collection of NDL views and SSL objects. The persistent store contains instances of the C++ classes that implement SSL class, SSL object and so on. The structure of the persistent store, whilst logically is very rich (it corresponds to the SSL class hierarchies), is physically flat (as it contains instances of approximately four C++ classes). The significance of this becomes apparent when browsing of software engineering projects is considered. The MDT provides sufficient indexing to locate the SSL objects that correspond to individual projects, models and diagrams. However, browsing a software engineering project is also concerned with browsing the *content of the models* that have been derived. A method for supporting browsing of software engineering projects in MOOT, at granularity finer than that of the diagram, has not yet been proposed.

9.3.3 The Notation Definition Language

NDL allows notations to be described and supports the 'screen notation' in contrast to the limited support for 'pen and paper' notations provided by other meta-CASE tools.

The limitations of NDL are related to supporting operations over groups of symbols and connections, facilities that are not supported by the syntax representation mechanism adopted by other meta-CASE tools. Some notations represent semantic information by the relative positions of symbols and connections (e.g. RDD). Whilst NDL can be used to describe the symbols and connections of such notations, it does not provide facilities to capture such a spatial relation. Composite symbols such as the Booch bubble (where a class bubble may appear inside another Booch bubble) and the Coad and Yourdon subject area cannot be represented. This is the purpose of the NDL composite template, which has yet to be implemented.

Each of these limitations are addressed in section 9.4 - Future Work

9.3.4 The Semantic Specification Language

Existing specification languages and virtual machines were investigated to determine their applicability to the implementation of MOOT. The main reasons for deriving a specialised language for MOOT were:

- The approach adopted by other meta-CASE tools only focuses on completeness and consistency checking. MOOT was required to support additional features such as cognitive support and auto-correction.
- The formal approach adopted by other meta-CASE tools does not allow a software engineering project to be in an inconsistent state. This is a barrier to an exploratory approach to development that software engineers naturally use.
- A new language can be readily extended and modified based on results gained form its use and future research ideas.

The SSL execution strategy was developed based on the requirement for efficient execution of SSL specifications and platform independence. The decision to translate SSL to a platform independent representation and execute it on a virtual machine was a natural one.

Existing object-orientated virtual machines were investigated (e.g. the Smalltalk virtual machine and the Java virtual machine). The following issues were noted:

- Existing virtual machines implement representations of general-purpose programming languages and therefore provide facilities that SSL does not require (such as support for input and output).
- The support required for concurrency is different to that of existing virtual machines. SSL requires only a single thread of control to be active in the SSL-VM, yet multiple instances of the SSL-VM can be active at the same time processing messages from a common pool of SSL objects. Object-level locking is therefore required and is tightly coupled with the virtual machine.
- In contrast to other virtual machines, the SSL-VM only required a small instruction set and a close correlation to SSL.

9.3.5 Core Knowledge Base and Generic Object Orientated Knowledge Base

One of the primary goals in developing the CKB and GOOKB was to demonstrate the feasibility of producing libraries of re-usable methodology semantic components for a meta-CASE tool.

The CKB is a base, from which other meta-models may be derived in the future. It is similar in scope and intent to the OMG Meta Object Facility. For example the GOOKB has been defined as an extension of the CKB (see section 9.4 – Future Work).

The focus of the GOOKB is limited to static modelling of class hierarchies and the various types of association supported by object-orientated methodologies. The GOOKB was designed to model concepts germane to *all* object-orientated methodologies. Its scope is similar to the latter COMMA project, which was for a "critical minimality that could be supported by all methods" (Henderson-Sellers and Bulthuis, 1996a). Implementing the support for behavioural modelling in MOOT is addressed in section 9.4 – Future Work.

Meta-modelling of 'software process' is a significant research task in its own right and is on-going in the MOOT project. The inclusion of the process and document classes in the CKB acknowledges the importance of these concepts, which is an improvement over existing meta-CASE tools.

9.4 Future Work

The overall goal of MOOT is to support all phases of the software development life-cycle, promote re-use and support component based software engineering methodologies. The planned future work can be classified as:

1. Extending the MOOT prototype so it completely implements the architecture proposed in chapter 3.
2. Extending the methodology representation strategy.
3. Extending the scope of MOOT.

Subsequent sections describe future work related to categories two and three.

9.4.1 The Notation Definition Language

Extension of existing NDL facilities

- Introduce a module system and improve the scope rules for NDL. Currently template names and NDL IDs are unique within an NDL specification. Therefore, portions of an NDL specification can not be easily re-used. Experience gained by

using NDL indicates that the group template mechanism is very useful and that building libraries of group templates is efficacious.

- Consider supporting format specification for text areas. Currently text fields contain strings in an 'unparsed' form. The CASE tool client transfers the content of the text fields to the MOOT core, which translates them. The communication between the CASE tool client and the MOOT core can be reduced if the CASE tool client can check the syntax of text fields.
- Introduce a general action template to improve user-defined actions. NDL supports two built-in action types, update and transition. User defined actions are currently supported by permitting a user defined action ID to be associated with an active area. These actions are propagated by a CASE tool client to the MOOT core but cannot have any arguments.

Adding new facilities to NDL

- Support parameterised symbol and connection types. For example a class symbol could be parameterised by an outside and an inside group template. A parameterised template could be instantiated to create a concrete template type.
- Consider manual re-sizing of symbols. This could be implemented by simple scaling. However this would lead to symbols that are distorted. A better solution would be to include optional 'stretch in x' and 'stretch in y' properties for the primitive template types that correspond to graphical elements.
- Support repetitive subgroups in symbols. This would allow a greater range of notations to be described and also simplify the description of others. This technique could also be used to replace the multi-line-text template segment type. Implementing the support for repetitive subgroups would provide better targeting of events and actions to sub-parts of symbols.
- Allow 'position information' to be propagated to the MOOT core. One possible technique is to apply a logical grid over a diagram with a 'snap to grid factor'. The origin of this grid would be relative to the first symbol placed in a diagram.
- Support constraints on item placement. One possible technique is to use the MetaView idea of Clusters.

- Investigate supporting animation of diagrams. This is something that is outside the initial scope of NDL.

9.4.2 The Semantic Specification Language

Extension of existing SSL facilities

- Permit SSL classes to define local methods, which have the same visibility as the attributes and are only relevant to the class implementation. This ensures the class interface does not become polluted with operations that are *only* related to the class implementation.
- Add support for parameterised types to SSL.
- Extend the use of the SSL tuple type.

Adding new facilities to SSL

- Consider implementing the CKB classes *Methodology*, *Model*, *Diagram*, *Concept* and *Relation* as built-in SSL types. This work implies an extension of the SSL-VM.
- Address optimisation of compiled SSL.
- Investigate the need to support concurrency in SSL. MOOT allows multiple instances of the SSL-VM to be active at the same time, processing messages from a common pool of SSL objects (a form of concurrency). Supporting concurrency in SSL would require more than one thread of control in the SSL-VM and perhaps in SSL objects. Two approaches are: a) provide explicit programmer support (e.g. a programmer API or programming language constructs, related to concurrency) b) automate the support for concurrency.

  The preferred method for supporting concurrency in SSL would be the second approach.

9.4.3 Notation Semantic Mapping Tables

Two avenues of future work are envisaged for NSM tables:

- NSM tables could be extended with a simple scripting language (NSM-SL). The elements on the right-hand-side of the majority of table entries would consist of a

block of NSM-SL code. This would support mapping the creation of an NDL view to the creation of one *or more* SSL objects; mapping the server side creation of an SSL object to instances of one *or more* NDL templates and mapping an NDL action to *several messages* to *several* SSL objects.

- Generalise the NSM table to permit representations other than NDL to be bound to SSL. For example a simple command-line client has been implemented and associated to an SSL semantic description via an NSM table.

9.4.4 Support for Re-use

A significant amount of research has already been conducted on adopting re-use strategies and on the problems of building, indexing and searching through, a collection of re-use assets (Yu, 1999). Future research must consider how these techniques can be applied and extended within the context of MOOT. This includes:

- Descriptions of re-usable components in the re-use pool. One possibility is to extend SSL to permit descriptions of 'meaning' to be attached to SSL objects. Another is to use a separate language for describing the components in the re-use pool.
- Assistance in selecting re-usable components, which includes intelligent searching of the re-use pool and the promotion of new items into the re-use pool.
- The management of re-usable components over their lifetime.
- Implementation of a re-use pool browser.

This work must also consider the requirements of emerging, component-based, development methodologies (D'Souza and Wills, 1998; Wills and D'Souza, 1997) and technologies such as SOM, COM, DCOM, CORBA, JavaBeans IIOP and ActiveX (Forman *et al.*, 1995; I-Kinetics, 1998; LaMonica, 1997; Montgomery, 1997; OMG, 1991, 1992, 1998; Orfali *et al.*, 1996; Siegel *et al.*, 1996; Soley, 1998).

9.4.5 *Cognitive Support*

This future work involves implementing the ARGO/UML[70] scheme for cognitive support (Robbins *et al.*, 1996, 1997, 1998), in the context of the Core Knowledge Base. This would be an extension of the *Critic* SSL module of the CKB to support:

- Building user models.
- Providing a change history. The CASE tool client currently implements this in a primitive way. It records the request-result pairs that correspond to communication with the MOOT core and displays them in a separate window for the user to view.
- Supporting auto-correction.
- Introduce support for ARGO/UML style Critics. The ARGO scheme allows critics to be active and monitor the user as they work.

The ARGO/UML approach is specific to object-orientated methodologies and focuses on the support of design. The scope of the ARGO/UML scheme must be re-considered in terms of:

- *Generality.* MOOT is a meta-CASE tool that aims to support arbitrary methodologies.
- *Scope.* MOOT is intended to support methodologies across a wide portion of the life-cycle. Ultimately this includes tasks such as requirements gathering and implementation.

9.4.6 *Meta-Modelling*

- *Core Knowledge Base and Generic Object Orientated Knowledge Base*

  The CKB and GOOKB should be compared to other meta-modelling developments, as they become stable. Further modelling of the UML[71] meta-model (see section 8.4 - Supporting UML) and the OPEN meta-model as extensions of the GOOKB is necessary. A comparison of the OMG Meta Object Facility to the CKB is also of particular interest.

---

[70] Argo/UML is a methodology dependent CASE tool (Figure 1-3 - Classification hierarchy of CASE tool categories) that has been developed as part of a research project related to support the cognitive needs of designers.

[71] The current implementation of the UML meta-model has not required any modifications to the CKB or GOOKB.

- *Meta-modelling of software engineering process*

  The result of this research will be a software process meta-model, which can be defined in SSL as a part of the Core Knowledge Base. This research is also related to the future work on cognitive support for software engineers as it deals with the suggestions and guidelines implicit in the software process.

- *Meta-modelling the behavioural modelling languages supported by object-orientated methodologies*

  This research will consider the behavioural modelling languages adopted by object-orientated methodologies. It will also consider the COMMA project, the UML meta-model and the submissions for the OMG OA&D facility.

- *Meta-modelling other approaches to software engineering*

  The original intent of the MOOT project was to solely address object-orientated methodologies. The subsequent development of the CKB, however, suggested that the MOOT approach is more widely applicable than was initially intended.

  The objective of this research is to determine if meta-models of other software engineering approaches can be implemented as extensions of the CKB. These meta-models will have the same scope and intent as the GOOKB. Examples include Workflow methodologies (preliminary work on this is described in section 8.6) and Information Engineering.

9.4.7 Validation of a Complete Implementation of MOOT

A complete implementation of the MOOT CASE architecture (proposed in chapter 3) must be validated with respect to the two types of user that MOOT supports; it must be validated as a *CASE tool* and as a *meta-CASE tool.*

An evaluation framework has been derived (appendix I) to support validation of a complete implementation of MOOT. The results of applying the evaluation framework to MOOT will be compared to evaluation results already generated for other CASE and meta-CASE tools (Choi, 1996; Gray, 1995; Phillips *et al.*, 1998a).

9.5 Conclusion

This research has demonstrated the efficacy of adopting an object-orientated approach to the development of a methodology representation strategy for meta-CASE tools. The novel methodology representation strategy reinforces fundamental object-orientated principles:

- *Encapsulation*

  Everything related to the description of a methodology's syntax is written in a single, separate, purpose built language (NDL) and *grouped together*. *Everything* related to the description of a methodology's semantics is written in a single, separate, purpose built language (SSL) and *grouped together*.

- *Information Hiding*

  The implementation of syntax and semantic descriptions of a methodology are totally hidden from each other. Semantic elements do not know, *and do not need to know*, how they are visualised. Syntax elements do not know, *and do not need to know*, what they represent.

- *Polymorphism and Late Binding*

  An NDL specification can be bound to any SSL specification via an NSM table.

- *Re-Use*

  Re-use is promoted by viewing methodology specifications as potentially re-usable components and by the development and subsequent use of the CKB and GOOKB.

The results of this research are manifest in the existence of NDL, SSL, SSL-BC, the SSL-VM, NSM tables, the CKB, the GOOKB and the MOOT prototype. Empirical results gained from applying the MOOT prototype demonstrated the flexibility, extensibility and potential of the novel methodology representation strategy. This approach permitted the implementation and modelling of UML and patterns, two recent advances of object technology that did not exist when the research commenced.

The novel strategy presented in this thesis is more than an untried theory. It has been implemented, applied and is being evaluated. Simply, it is real and it works.

Section IV

Appendices

Appendix I

Evaluation Framework

I.1 Existing Evaluation Frameworks

A CASE tool Evaluation Framework should support both qualitative and quantitative assessment. The framework should provide the structure from which a set of questions can be generated that are designed to assess the functionality, methodology support and usability of CASE tools.

Examples of evaluation frameworks that have been developed in the past include the work of Misra (1990), Mosley (1992) and Ovum (1996).

These approaches suffer from several important problems. Existing evaluation frameworks:

- Do not address all the features and characteristics of CASE tools.
- Are often out of date with respect to CASE and software engineering technology.
- Cannot be systematically modified to address new advances in CASE technology. Their structure is not conducive to simple extension or refinement.
- Cannot be easily targeted toward tools of a particular type. For example some of the evaluation criteria related to meta-CASE tools are not relevant to a methodology dependant tool.
- Are difficult to use to focus on one particular dimension of the properties of CASE tools (e.g. usability).

I.2 A New Evaluation Framework

A new evaluation framework has been developed, as a part of this research, to address the problems identified with existing evaluation frameworks (Phillips, 1998a).

The new evaluation framework:

1. addresses usability, methodology support, life-cycle support and information exchange

2. can be easily extended in the future to allow for emerging technology

3. copes with the plethora of different methodologies and tools

The new evaluation framework is based on a classification hierarchy of OO CASE tool categories (Figure 1-3 - Classification hierarchy of CASE tool categories).

Each node in the hierarchy represents a CASE tool category and has a set of associated evaluation criterion. Each node inherits evaluation criteria from parent nodes. The hierarchical structure permits the framework to be extended to support new types of CASE tool.

A classification based evaluation framework provides the necessary flexibility needed to cope with changing CASE and software engineering technology. This structure also prevents the evaluation framework from becoming unmanageable, as evaluation criteria are always associated with a node in the classification hierarchy of an appropriate level of abstraction. This structure also permits evaluation criteria to be specialised and refined in a systematic way, in less abstract CASE tool categories.

Evaluation criteria are further classified with respect to usability, methodology support, life-cycle support and information exchange. The four evaluation criteria hierarchies are orthogonal to the CASE-tool-category classification hierarchy (Figure I-1). Each evaluation criteria hierarchy is further structured into a hierarchical series of categories. Evaluation criterion is therefore classified in two ways a) based on the CASE tool category it is relevant to and b) based on the property of CASE tools it addresses.

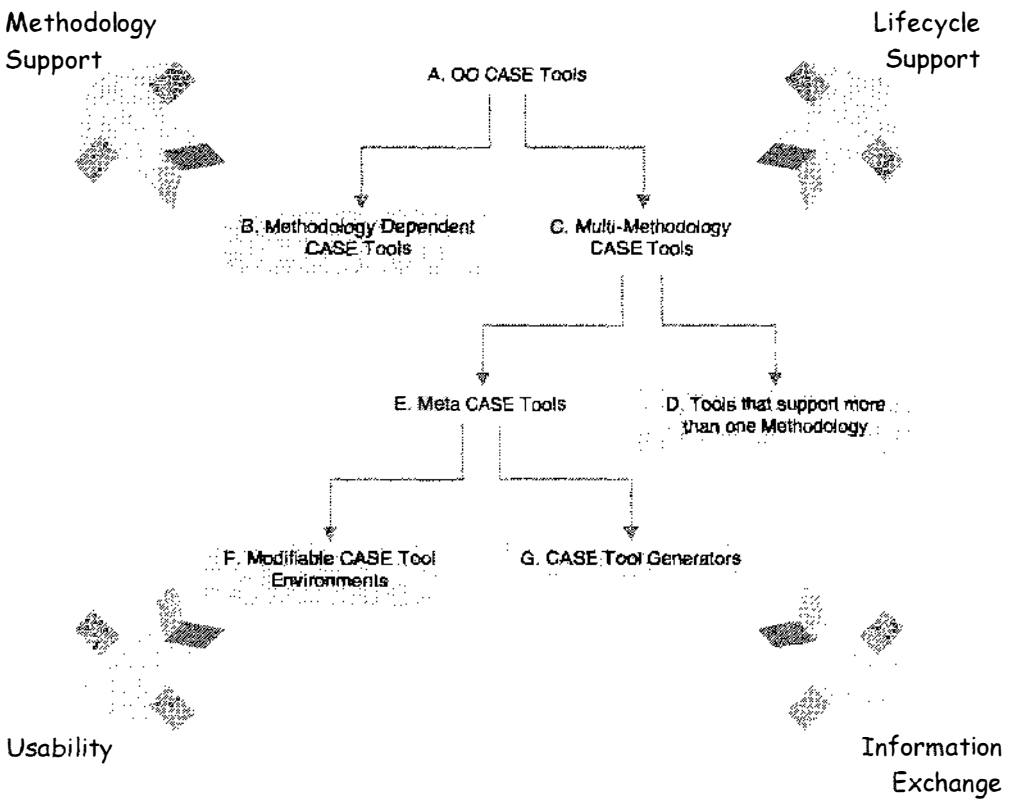


Figure I-1 - Dimensions of the evaluation framework

For further information on the evaluation framework and its application see (Choi, 1996, Gray, 1995, Phillips *et al.*, 1998a).

Appendix II

NDL Grammar

II.1 Introduction

The following grammar is an abridged version of the PCCTS grammar used by the CASE Tool Client. This grammar uses version 2.x of the PCCTS syntax.

A description of PCCTS can be found at the PCCTS web site (PCCTS, 1998).

II.2 Reserved Words

```
NOTATION                CONNECTION_TERMINATOR_TEMPLATE
CONNECTION_SYMBOL_TEMPLATE                CONNECTION_TEMPLATE
{                       }                 ;
=                       (                 )
,                       GROUP_TEMPLATE    SYMBOL_TEMPLATE
DEFAULT                 TEXT              ARITY
LINE                    ARC               LISTTEXT
GROUP                   DA                ACTIVE
AREA                    TRANSITION        TO
UPDATE                  POINT             BOUNDING
RECT                    UNCONSTRAINED     TOP
HEAD                    TAIL              WIDTH
HEIGHT                  GROUPWIDTH        GROUPHEIGHT
MAX                     MIN               SYMBOL
TERMINATORS
```

II.3 Operators

```
+                       -                 *
/
```

II.4 Grammar

```
notation :
   NOTATION IDENTIFIER
   ( group )*
   ( symbol )*
   ( connection_symbol )*
   ( connection_terminator )*
   ( connection )*
   ;
```

```
group :
  GROUP_TEMPLATE IDENTIFIER
  BEGIN
    expressions graphical_components active_areas
    bounding_region
  END
  ;

template :
  expressions graphical_components bounding_region
  ;

symbol :
  SYMBOL_TEMPLATE IDENTIFIER
  BEGIN
    template active_areas docking_areas
    ( default_text_property )?
  END
  ;

default_text_property :
  DEFAULT TEXT
  OPENBRACKET IDENTIFIER COMMA IDENTIFIER CLOSEBRACKET
  ;

flag :
  INTEGERVAL
  ;

connection_symbol :
  CONNECTION_SYMBOL_TEMPLATE IDENTIFIER
  BEGIN
    template TOP point docking_areas
  END
  ;

connection_terminator :
  CONNECTION_TERMINATOR_TEMPLATE IDENTIFIER
  BEGIN
    template HEAD point TAIL point
  END
  ;

connection :
  CONNECTION_TEMPLATE IDENTIFIER
  BEGIN
    template
    ARITY INTEGERVAL
    ( SYMBOL IDENTIFIER ) ?
    TERMINATORS IDENTIFIER ( IDENTIFIER )*
  END
  ;

expressions :
  ( IDENTIFIER EQUALS expression ENDEXPR )*
  ;
```

```
expression :
  (
      PLUS   expression expression
    | MINUS  expression expression
    | TIMES  expression expression
    | DIVIDE expression expression
    | term
  )
  ;

term :
  ( INTEGERVAL | function | IDENTIFIER )
  ;

function :
  (
      WIDTH       OPENBRACKET IDENTIFIER CLOSEBRACKET
    | HEIGHT      OPENBRACKET IDENTIFIER CLOSEBRACKET
    | GRP_WIDTH   OPENBRACKET IDENTIFIER CLOSEBRACKET
    | GRP_HEIGHT  OPENBRACKET IDENTIFIER CLOSEBRACKET
    | MAX argument_list
    | MIN argument_list
  )
  ;

argument_list :
  OPENBRACKET
    expression ( COMMA expression )*
  CLOSEBRACKET
  ;

graphical_components :
  (
      LINE      point point ( point )*
    | ARC       point point point
    | TEXT      IDENTIFIER point
    | LISTTEXT  IDENTIFIER point
    | Group     IDENTIFIER IDENTIFIER point
  )*
  ;

active_areas :
  ( ACTIVE AREA point point action )*
  ;

action :
  (
    UPDATE IDENTIFIER
    | TRANSITION TO IDENTIFIER
  )
  ;
```

```
docking_areas :
   (    POINT DA point connection_count point
          allowable_connectors
      |  LINE DA flag rect_area connection_count INTEGERVAL
          allowable_connectors
      |  ARC DA rect_area point flag connection_count
          INTEGERVAL allowable_connectors
   )*
   ;

rect_area :
   point point
   ;

connection_count :
   ( UNCONSTRAINED | INTEGERVAL )
   ;

allowable_connectors :
   OPENBRACKET ( IDENTIFIER )* CLOSEBRACKET
   ;

bounding_region :
   BOUNDING RECT point
   ;

point :
   OPENBRACKET expression COMMA expression CLOSEBRACKET
   ;
```

Appendix III

SSL Grammar

III.1 Introduction

The following grammar is an abridged version of the PCCTS grammar used by the SSL compiler. This grammar uses version 1.33 of the PCCTS syntax. A description of PCCTS can be found at the PCCTS web site (PCCTS, 1998) and in Terrance Parr's PCCTS book (Parr, 1997).

III.2 Reserved Words

```
MODULE            USES              ATTRIBUTES
OPERATIONS        CONSTRAINT        {
}                 (                 )
[                 ]                 ,
;                 INTEGER           REAL
BOOLEAN           STRING            COLLECTION
ITERATOR          CREATE            DESTROY
DEBUG_PRINT       SELF              CURRENT_MODEL
CURRENT_DIAGRAM   CURRENT_PROJECT   NO_OBJECT
IF                ELSE              LOOP
ENDLOOP           WHEN              RETURN
```

III.3 Operators

```
.                 ::                :
+                 -                 *
/                 div               mod
<                 >                 >=
<=                =                 <>
and               or                not
```

III.4 Grammar

```
moduleinterface :
   MODULE IDENTIFIER
   uses_lists
   classinterfacedefs
   ;

uses_lists :
   ( use_clause )*
   ;
```

```
use_clause :
   USES uses_item ( COMMA uses_item )* ENDSTATEMENT
   ;

uses_item :
   ( IDENTIFIER MODULESCOPE IDENTIFIER EQUALS IDENTIFIER )
   | IDENTIFIER
   ;

classname :
   ( IDENTIFIER | IDENTIFIER MODULESCOPE IDENTIFIER )
   ;

classinterfacedefs :
   ( classinterfacedef )*
   ;

classinterfacedef :
   IDENTIFIER
   { superclasslist }
   BEGIN
      ( operation )*
   END
   ;

operation :
   (
      DESTROY LPAREN RPAREN
      | ( CREATE | { operation_result } IDENTIFIER )
        parameter_list
   )
   ;

module :
   MODULE IDENTIFIER
   uses_lists
   classdefs
   ;

classdefs :
   ( classdef )*
   ;

classdef :
   IDENTIFIER
   BEGIN
      ATTRIBUTES ( attribute_list ENDSTATEMENT )*
      OPERATIONS ( method )*
      CONSTRAINT ( expression )*
   END
   ;

superclasslist :
   ISA classname
   (
      COMMA classname
   )*
   ;
```

```
attribute_list :
   type IDENTIFIER
   ( COMMA IDENTIFIER )*
   ;

type :
   INTEGER
   | REAL
   | STRING
   | BOOLEAN
   | classname
   | COLLECTION LSQBRACKET type RSQBRACKET
   | ITERATOR LSQBRACKET type RSQBRACKET
   ;

method :
   operation
   ( attribute_list ENDSTATEMENT )*
   block
   ;

statementlist :
   ( statement )*
   ;

block :
   BEGIN statementlist END
   ;

operation_result :
   type
   | LPAREN type ( COMMA type )* RPAREN
   ;

parameter_list :
   LPAREN
      { type IDENTIFIER ( COMMA type IDENTIFIER )* }
   RPAREN
   ;

statement :
   (
      send_message ENDSTATEMENT
      | destroy_message ENDSTATEMENT
      | return_statement
      | assignment
      | selection
      | iteration
      | debugstatement
   )
   ;

debugstatement :
   DEBUG_PRINT LPAREN expression RPAREN ENDSTATEMENT
   ;
```

```
lvalue :
   IDENTIFIER
   | LPAREN IDENTIFIER ( COMMA IDENTIFIER )* RPAREN
   ;

assignment :
   lvalue EQUALS expression ENDSTATEMENT
   ;

return_statement :
   RETURN ( expression ( COMMA expression )* ) ENDSTATEMENT
   ;

send_message :
   { ( IDENTIFIER | SELF | DIAGRAM    | PROJECT ) DOT }
   { LSQBRACKET classname RSQBRACKET }
   IDENTIFIER
   LPAREN { expression ( COMMA expression )* } RPAREN
   ;

create_message :
   classname DOT CREATE
   LPAREN { expression ( COMMA expression )* } RPAREN
   ;

destroy_message:
   IDENTIFIER DOT DESTROY LPAREN RPAREN
   ;

selection :
     ifstatement
     ;

iteration :
     loopstatement
     ;

ifstatement :
     IF condition block
     #pragma approx
     { ELSE block }
     ;

loopstatement :
   LOOP
   BEGIN
     statementlist
     ENDLOOP WHEN condition ENDSTATEMENT
     statementlist
   END
   ;

condition :
   LPAREN expression RPAREN
   ;
```

```
expression :
  arithmetic_expression
  { ( EQUALS|NOTEQUALS|LESS|LESSEQ|GREATER|GREATEREQ )
    arithmetic_expression
  }
  ;

arithmetic_expression :
  multiplicative_expression
  ( ( PLUS|MINUS|OR )
    multiplicative_expression
  )*
  ;

multiplicative_expression :
  factor
  ( ( TIMES|DIVIDE|DIV|MOD|AND )
    factor
  )*
  ;

factor :
  INTEGERVAL
  | BOOLEANVAL
  | STRINGVAL
  | CURRENT_MODEL
  | CURRENT_DIAGRAM
  | CURRENT_PROJECT
  | NO_OBJECT
  | SELF
  | IDENTIFIER
  | send_message
  | create_message
  | LPAREN expression RPAREN
  | NOT factor
  | MINUS factor
  ;
```

Appendix IV

SSL Examples

IV.1 The Sieve of Eratosthenes Version 1

This implementation of the Sieve of Eratosthenes[72] was written during the development of SSL to test the efficiency of object creation, object destruction and message binding. It is not intended to be an efficient implementation of the Sieve of Eratosthenes. The classes involved in this example are given in Figure IV-1.

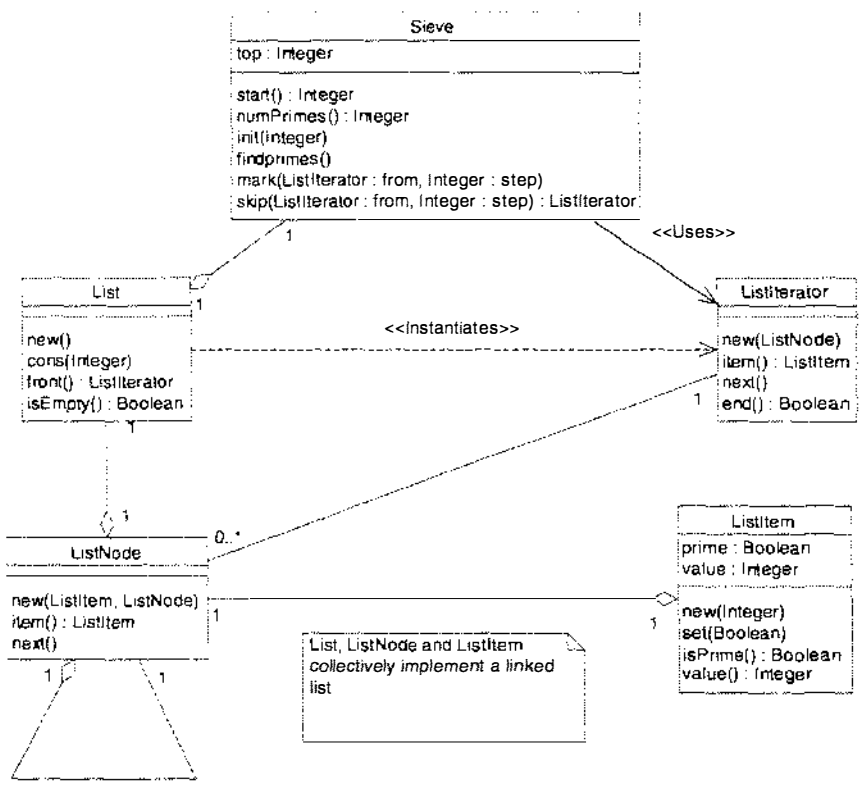


Figure IV-1 - Sieve of Eratosthenes version 1

In this example a sieve object maintains a linked list of boolean flags of a fixed size. The lined list is implemented with the *ListNode* and *ListItem* classes. The sieve object uses instances of the *ListIterator* class to perform traversals of the list.

[72] The Sieve of Eratosthenes is a well-known and extremely elegant algorithm for calculating prime numbers.

IV.1.1 Interface Module

```
module sieve

list
{
   new()
   cons( integer value )
   listIterator front()
   tail()
   boolean isEmpty()
}

listItem
{
   new( integer value )
   set( boolean isPrime )
   boolean isPrime()
   integer value()
}

listNode
{
   new( listItem i, listnode n )
   listItem item()
   listnode next()
}

listIterator
{
   new( listnode p )
   listItem item()
   next()
   boolean end()
}

sieveClass
{
   integer start()
   init( integer top )
   findprimes()
   mark( listiterator l, integer step )
   integer numPrimes()
   listiterator skip( listiterator l, integer n )
}
```

IV.1.2 Implementation Module

```
module sieve

sieveClass
{
   attributes

      list ints;
      integer top;
```

```
operations

  integer start()
    listitem i;
    listiterator li;
  {
    init( 1000 );
    findPrimes();
    return numPrimes();
  }

  init( integer t )
    integer c;
  {
    top = t;
    c = top;
    ints = list.create();
    ints.new();
    loop
    {
      ints.cons( c );
      endloop when( c = 2 );
      c = c - 1;
    }
  }

  findPrimes()
    integer step;
    integer upperlimit;
    listiterator l;
    listitem i;
  {
    step = 2;
    upperlimit = top div 2;
    loop
    {
      l = skip( ints.front(), step - 2);
      if( not l.end() )
      {
        i = l.item();
        if( i.isprime() )
        {
          debug_print( step );
          mark( l, step );
        }
      }
      step = step + 1;
      endloop when( step = upperlimit );
    }
  }
```

```
    mark( listiterator l, integer s )
      listitem i;
    {
      loop
      {
        l = skip( l, s );
        endloop when( l.end() );
        i = l.item();
        i.set( false );
      }
    }

    integer numPrimes()
      integer total;
      listiterator l;
      listitem i;
    {
      total = 0;
      l = ints.front();
      loop
      {
        endloop when( l.end() );
        i = l.item();
        if( i.isprime() )
        {
          total = total + 1;
        }
        l.next();
      }
      return total;
    }

    listiterator skip( listiterator l, integer n )
      integer c;
    {
      c = 0;
      loop
      {
        endloop when( l.end() OR (c = n) );
        l.next();
        c = c + 1;
      }
      return l;
    }
}

listNode
{
  attributes

    listItem item_;
    listNode next_;

  operations

    new( listItem i, listnode n )
    {
      item_ = i;  next_ = n;
    }
```

```
    listItem item()
    {
      return item_;
    }

    listnode next()
    {
      return next_;
    }
}

list
{
  attributes

    listnode l;

  operations

    new()
    {
      l = no_object;
    }

    cons( integer value )
      listitem i;
      listnode newl;
    {
      i = listitem.create();
      i.new( value );
      newl = listnode.create();
      newl.new( i, l );
      l = newl;
    }

    listIterator front()
      listiterator it;
    {
      it = listiterator.create();
      it.new( l );
      return it;
    }

    tail()
    {
      if( not ( l = no_object ) )
      {
        l = l.next();
      }
    }

    boolean isEmpty()
    {
      return l = no_object;
    }
}
```

```
listItem
{
  attributes

    boolean prime;
    integer val;

  operations

    new( integer value )
    {
      prime = true;
      val = value;
    }

    set( boolean isPrime )
    {
      prime = isPrime;
    }

    boolean isPrime()
    {
      return prime;
    }

    integer value()
    {
      return val;
    }
}

listIterator
{
  attributes

    listnode pos;

  operations

    new( listnode p )
    {
      pos = p;
    }

    listItem item()
    {
      return pos.item();
    }

    next()
    {
      if( not ( pos = no_object ) )
      {
        pos = pos.next();
      }
    }
```

```
    boolean end()
    {
      return pos = no_object;
    }
}
```

Sample output from executing this implementation of the Sieve of Eratosthenes algorithm with an early prototype of the SSL virtual machine is given below.

```
> mooto -c sieve:sieveclass -p 2
No message specified, using start-#

There are 1 item(s) in the stack.
Item 1 is an integer (168)

Total number of opcodes interpreted : 3367969
Total time (seconds)                : 352
Opcodes/Sec                         : 9568
A total of 2498 objects were created
A total of 617047 messages were processed
>
```

It was executed on a Sun Sparc Server 1000e. On average, 10000 SSL-VM instructions and 1800 messages were processed per second.

IV.2 The Sieve of Eratosthenes Version 2

This implementation of the Sieve of Eratosthenes was written during the development of SSL to test the SSL collection and iterator types. The classes involved in this example are given in Figure IV-2.

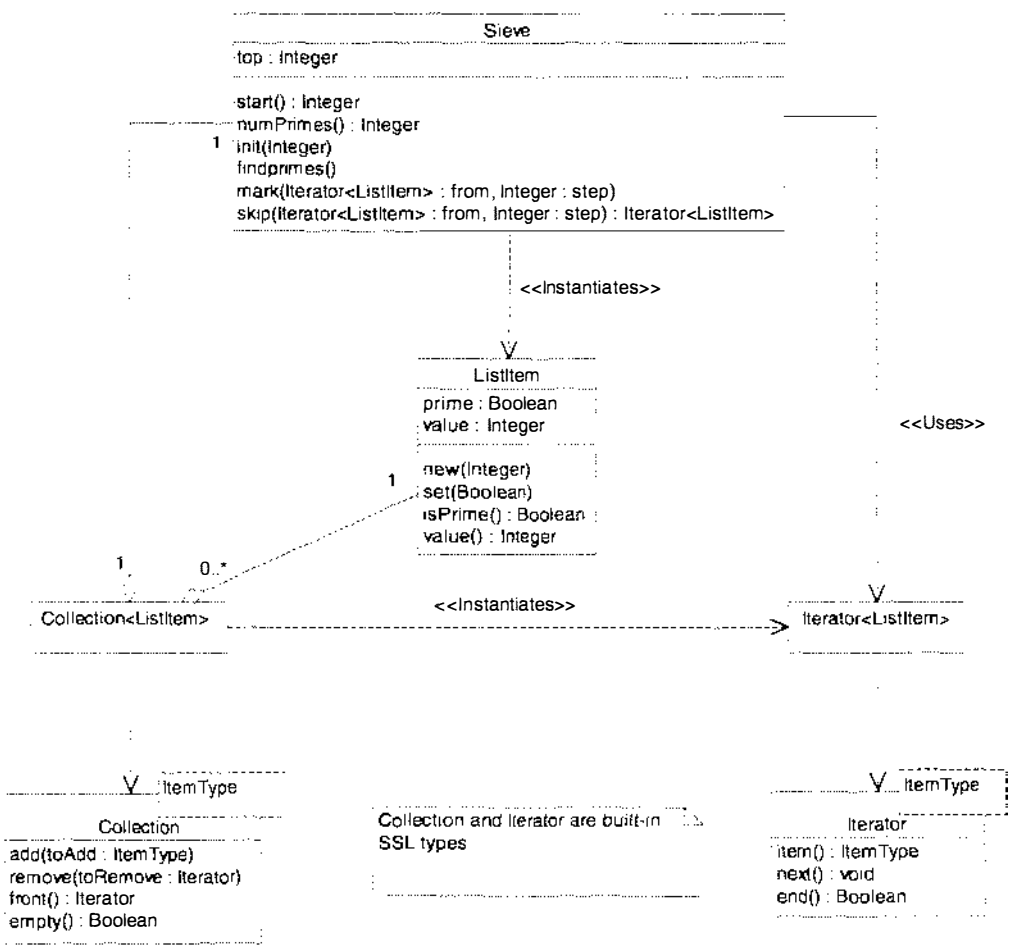


Figure IV-2 - Sieve of Eratosthenes version 2

IV.2.1 Interface Module

```
module sieve2

listItem
{
   new( integer value )
   set( boolean isPrime )
   boolean isPrime()
   integer value()
}
```

```
sieveClass
{
   integer start()
   init( integer top )
   findPrimes()
   mark( iterator[listitem] l, integer s )
   iterator[listitem] skip( iterator[listitem] l,integer n)
   integer numPrimes()
}
```

IV.2.2 Implementation Module

```
module sieve2

sieveClass
{
   attributes

      collection[listitem] ints;
      integer top;

   operations

      integer start()
      {
         init( 1000 );
         findPrimes();
         return numPrimes();
      }

      init( integer t )
         integer c;
         listitem i;
      {
         top = t;
         c = 2;
         loop
         {
            i = listitem.create();
            i.new( c );
            ints.add( i );
            endloop when( c = 1000 );
            c = c + 1;
         }
      }
```

```
findPrimes()
  integer step;
  integer upperlimit;
  iterator[listitem] l;
  listitem i;
{
  step = 2;
  upperlimit = top div 2;
  debug_print( "Working ......" );
  loop
  {
    l = skip( ints.front(), step - 2 );
    if( not l.end() )
    {
      i = l.item();
      if( i.isprime() )
      {
        debug_print( step );
        mark( l, step );
      }
    }
    step = step + 1;
    endloop when( step = upperlimit );
  }
}

mark( iterator[listitem] l, integer s )
  listitem i;
{
  loop
  {
    l = skip( l, s );
    endloop when( l.end() );
    i = l.item();
    i.set( false );
  }
}

integer numPrimes()
  integer total;
  iterator[listitem] l;
  listitem i;
{
  total = 0;
  l = ints.front();
  loop
  {
    endloop when( l.end() );
    i = l.item();
    if( i.isprime() )
    {
      total = total + 1;
    }
    l.next();
  }
  debug_print( "number of primes under 1000 is" );
  debug_print( total );
  return total;
}
```

```
    iterator[listitem]
    skip( iterator[listitem] l, integer n )
      integer c;
    {
      c = 0;
      loop
      {
        endloop when( l.end() OR (c = n) );
        l.next();
        c = c + 1;
      }
      return l;
    }
}


listItem
{
  attributes

    boolean prime;
    integer val;

  operations

    new( integer value )
    {
      prime = true;
      val = value;
    }

    set( boolean isPrime )
    {
      prime = isPrime;
    }

    boolean isPrime()
    {
      return prime;
    }

    integer value()
    {
      return val;
    }
}
```

Appendix V

SSL-VM Instruction Set

V.1 Introduction

Instructions described here have the following format:

Name Type-mode Address-mode [Operands]

Name is the instruction name. *Type-mode* corresponds to the type that an instruction operates on. *Address-mode* specifies where the instruction's arguments are.

Type-mode is one of the following:

Boolean	Int	Real	String
Collection	Iterator	ObjRef	

Address-mode is one of the following:

Imp (Implicit)	Imm (Immediate)	Ind (Indirect)

V.2 Instruction Set

Mgs	Send a message to an object
Format	Mgs ObjRef Ind aReference aMessage
	Retrieve the object reference indicated by *aReference* from the context. Send *aMessage* to the object identified by the object reference.

Cmg	Send a create message to a class
Format	Cmg ObjRef Imm classname
	Create an instance of the class with the name indicated by *classname*.

Smg	Send a scoped message				
Format	Smg	ObjRef	Imm	aReference	classname	aMessage

Retrieve the object reference indicated by *aReference* from the context. Send *aMessage* to the object identified by the object reference, as if it were an instance of *classname*.

Rtn	Return from message		
Format	Rtn	Void	Imp

Set the instruction counter (IC) to –1 (end of a message).

Psh	Push item onto the stack			
Format	Psh	Int	Imm	anInt
	Psh	Int	Ind	aReference
	Psh	Real	Imm	aReal
	Psh	Real	Ind	aReference
	Psh	Boolean	Imm	aBoolean
	Psh	Boolean	Ind	aReference
	Psh	Collection	Ind	aReference
	Psh	String	Imm	aString
	Psh	String	Ind	aReference
	Psh	ObjRef	Imm	anObjRef
	Psh	ObjRef	Ind	aReference
	Psh	Iterator	Ind	aReference

If *AddrMode* is *Ind*, get the value from context and push it into stack. If *AddrMode* is *Imm*, get the following value and push it into stack.

Pop	Pop an item from the stack			
Format	Pop	Int	Ind	aReference
	Pop	Real	Ind	aReference
	Pop	Boolean	Ind	aReference
	Pop	Collection	Ind	aReference
	Pop	String	Ind	aReference
	Pop	ObjRef	Ind	aReference
	Pop	Iterator	Ind	aReference

Reset the variable indicated by *aReference* in the context, with the value on the top of the stack. Remove the top item from the stack.

Add	Addition			
Format	Add	Int	Imp	
	Add	Real	Imp	
	Add	String	Imp	
	Add	Collection	Ind	aReference
	Add	Iterator	Ind	aReference

Int and *Real*

The top two values are popped off stack and added together. Push the result onto the stack. The two values of the stack must have the same type.

String

The top two values are popped off stack and appended. The result is pushed back onto stack.

Collection

Pop the item off the stack and add it into the collection indicated by *aReference* in context.

Iterator

Move the iterator indicated by *aReference* forwards along the list it points to.

Sub	Subtraction		
Format	Sub	Int	Imp
	Sub	Real	Imp
	Sub	Collection	Ind

Int, Real

Pop the top two items off stack. Subtract them and push the result onto stack. Both items must be of the same type.

Collection

Pop an *iterator* off the stack and remove the item that the *iterator* points to from the collection.

Mul	Multiplication		
Format	Mul	Int	Imp
	Mul	Real	Imp

Pop the top two items off stack. Multiply them and push the result onto stack. Both items must be of the same type.

Div	Division		
Format	Div	Int	Imp
	Div	Real	Imp

Pop the top two items off stack. If the second operand is zero, a maximum value is pushed onto stack. Otherwise divide them and push the result onto stack. Both items must be of the same type.

Mod	Modulus		
Format	Mod	Int	Imp

Pop the top two items off stack. If the second operand is zero, a maximum value is pushed onto stack. Otherwise apply the modulus operation and push the result onto stack.

Cnv	Convert type		
Format	Cnv	Int	Imp
	Cnv	Real	Imp
	Pop a value off stack, convert its type from *Int* to *Real* or from *Real* to *Int*, and push the result back to stack.		

Neg	Unary minus		
Format	Neg	Int	Imp
	Neg	Real	Imp
	Change the sign of the topmost value on stack.		

And	Boolean And		
Format	And	Boolean	Imp
	Pop two boolean values off stack. Push the logical conjunction of these values onto the stack.		

Or	Boolean Or		
Format	Or	Boolean	Imp
	Pop two boolean values off stack. Push the logical disjunction of these values onto the stack.		

Not	Boolean Not		
Format	Not	Boolean	Imp
	Pop topmost boolean value off stack. Push the logical negative of it onto the stack.		

Eq	Typed equal comparison		
Format	Eq	Int	Imp
	Eq	Real	Imp
	Eq	Boolean	Imp
	Eq	String	Imp

Pop two values off stack. If their value and type is equal push *true* onto the stack otherwise push *false*.

Neq	Typed not equal comparison		
Format	Neq	Int	Imp
	Neq	Real	Imp
	Neq	Boolean	Imp
	Neq	String	Imp

Pop two values off stack. If their value and type are not equal push *true* onto the stack otherwise push *false*.

Grt	Greater than		
Format	Grt	Int	Imp
	Grt	Real	Imp
	Grt	String	Imp

Pop two values off stack. If the first is greater than the second one push *true* onto the stack otherwise push *false*. Both operands must have the same type.

Lss	Less than			
Format	Lss	Int	Imp	
	Lss	Real	Imp	
	Lss	String	Imp	
	Pop two values off stack. If the first is less than the second one push *true* onto the stack otherwise push *false*. Both operands must have the same type.			

Brt	Branch if true			
Format	Brt	Boolean	Imm	anAddr
	Pop top value off stack. If it has the value *true*, set the IC to the address *anAddr*.			

Brf	Branch if false			
Format	Brf	Boolean	Imm	anAddr
	Pop top value off stack. If it has the value *false*, set the IC to the address *anAddr*.			

Fnt	Create an iterator			
Format	Fnt	Collection	Ind	aReference
	Create an iterator that points to the first item of the collection indicated by *aReference* in the context. Push the iterator onto stack.			

End	Test if the iterator is at the end of a collection			
Format	End	Iterator	Ind	aReference
	If the iterator indicated by *aReference* refers to the end of a collection, push *true* to stack. Otherwise, push *false* to stack.			

Itm	De-reference an iterator		
Format	Itm	Iterator	Ind aReference

Push the item that the iterator refers to onto the stack. If the iterator is at the end of a list, push *no_object* onto the stack.

Prj	Get the value of the project register	
Format	Prj	Void Imp

Push the value of the project register onto the stack.

Mdl	Get the value of the project register	
Format	Mdl	Void Imp

Push the value of the model register onto the stack.

Dgm	Get the value of the project register	
Format	Dgm	Void Imp

Push the value of the diagram register onto the stack.

Appendix VI

SSL Compiler

VI.1 Introduction

This appendix describes the design and implementation of the SSL compiler (SSLC).

VI.2 The SSL Compiler

The SSL compiler (SSLC) is a command line tool that accepts a collection of SSL module names as input. It initially compiles each interface module and then compiles the corresponding implementation modules. Any additional interface modules that are used by these modules are also compiled, if needed. SSLC generates SSL-BC and SSL-assembler for each class.

SSLC was developed using:

- Gnu g++ 2.7.2 for Solaris 2.5
- Microsoft Visual C++ 5.0 for Windows 95/NT
- PCCTS 1.33 (Parr, 1997; PCCTS, 1998)

PCCTS (Purdue Compiler Construction Tool Set) is a public domain tool that aids in the construction of language recognisers and translators. It consists of a parser generator (ANTLR) and a lexical analyser generator (DLG). PCCTS generates LL(k) parsers that dynamically adjust the token look-ahead depth (k). PCCTS v 1.33 generates lexical analysers and parsers in C and C++.

An abridged PCCTS grammar for SSL is given in appendix III.

The major components of the compiler are shown in Figure VI-1.

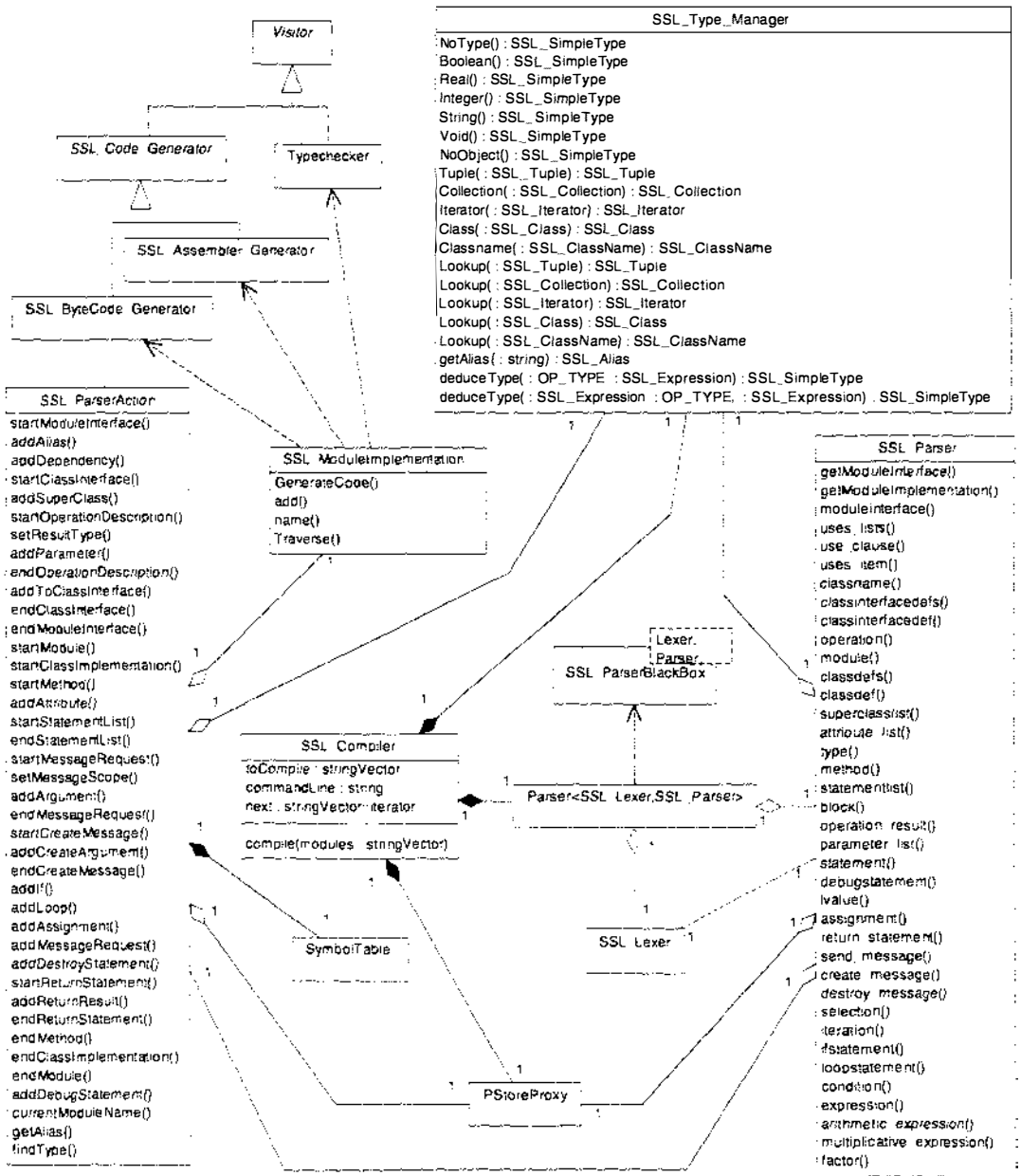


Figure VI-1 · The main components of the SSL compiler

SSL Compiler is a singleton (Gamma *et al.*, 1995) class that is responsible for parsing the compiler command line and generating a vector of SSL modules that need to be compiled. It directs the action of the parser, type checker and code generator.

SSL ParserBlackBox is a parameterised class that is responsible for binding a parser and lexer object together. The parsing sub-system of the compiler is made by instantiating the *Lexer* and *Parser* type arguments of *SSL ParserBlackBox* with *SSL Lexer* and *SSL Parser* respectively. The *SSL Lexer* class is the lexical analyser that is generated by the DLG and ANTLR tools. The *SSL Parser* class is the parser that is generated by the DLG and

ANTLR tools. The interface of *SSL_Parser* consists of a set of member functions, which correspond to the rules in the SSL grammar (appendix III).

Actions (such as creating nodes for an abstract syntax tree) can be associated with the rules in a PCCTS grammar. The embedded actions are copied into the related member functions of the PCCTS generated parser. These actions have been placed into a singleton class (*Parser Action*) so the entire parser does not need to be regenerated and recompiled each time an action is modified.

Type Manager is a singleton class that stores details relating to the SSL classes, SSL collections, SSL iterators and SSL tuples as they are recognised by the parser. It provides facilities for searching for and registering new types, checking to see if a type is defined and for deducing the type of an expression. It is used during type-checking.

The *PstoreProxy* class isolates the compiler from the persistent store. It provides facilities for retrieving and storing classes, interfaces, modules and their compiled representations.

Symbol Table is a singleton class that stores details related to attributes of SSL classes, local variables of SSL methods and message arguments.

Visitor is an abstract super-class that implements the Visitor pattern (Gamma *et al.*, 1995). *SSL Type Checker*, *SSL Bytecode Generator* and *SSL Assembler Generator* are all sub-classes of *Visitor*. The use of the Visitor pattern is discussed in more detail in section VI.4.

VI.3 Representing Types in the SSL Compiler

Figure VI-2 illustrates how types are represented in the SSL compiler. All primitive types are represented by a single instance of *SSL Simple Type* that is managed by *SSL Type Manager*. Classes, collections, iterators and tuples are represented by sub-classes of *SSL Simple Type*. The type manager is responsible for maintaining all instances of the type classes. It provides facilities for checking to see if types have been previously defined and for registering new types.

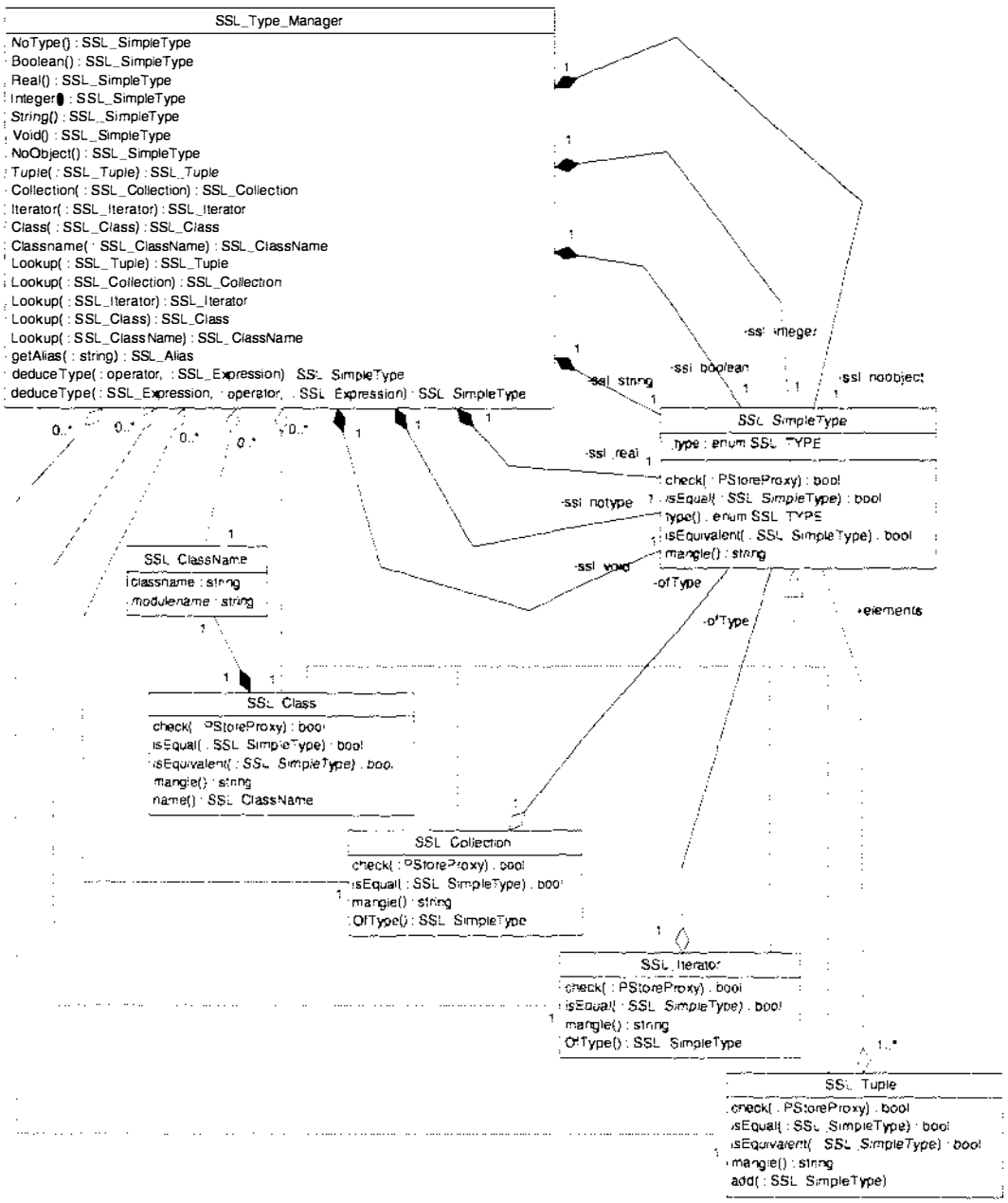


Figure VI-2 - Representing types in the SSL compiler

VI.4 Representing Statements and Expressions in the SSL Compiler

Figure VI-3 shows how statements and expressions are represented by the SSL compiler. The parser (*SSL Parser* in Figure VI-1) builds Abstract Syntax Trees (AST) for each method. The types of node in the AST are defined by the sub-classes of *AST Node* in Figure VI-3.

The interface of the abstract *Visitor* class of Figure VI-3 defines a visit operation that corresponds to each sub-class of *AST_Node*. Concrete classes of the AST node hierarchy

implement the *Accept* operation defined in *AST Node*. The sub-classes of *Visitor* implement specific operations of the compiler (such as type checking and code generation). The type checker and code generators are related by the order in which they visit nodes in an AST (i.e. the traversal algorithm is the same).

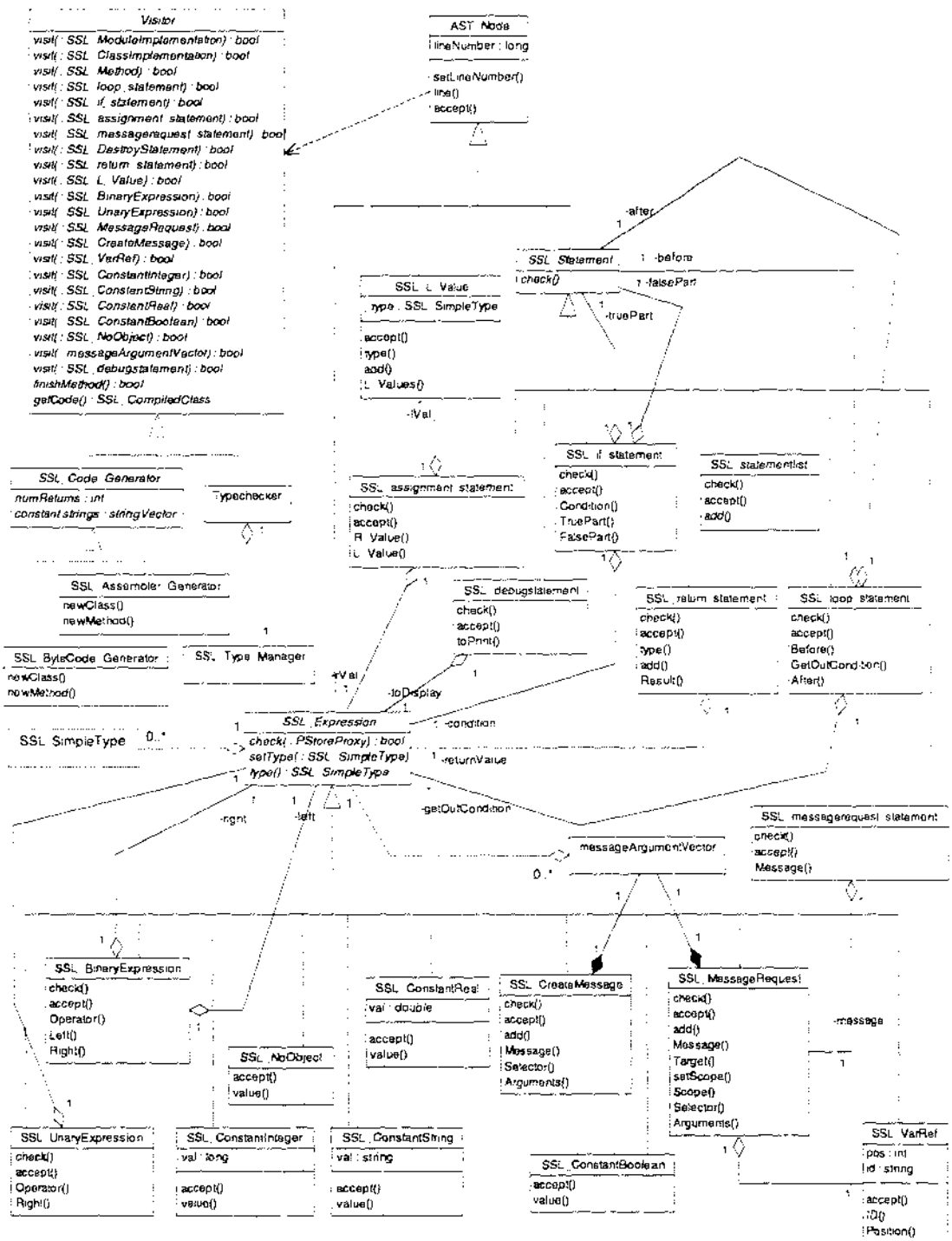


Figure VI-3 - Statements and expressions in the SSL compiler

VI.5 Representing Modules in the SSL Compiler

Figure VI-4 shows the classes involved in representing SSL modules in the compiler.

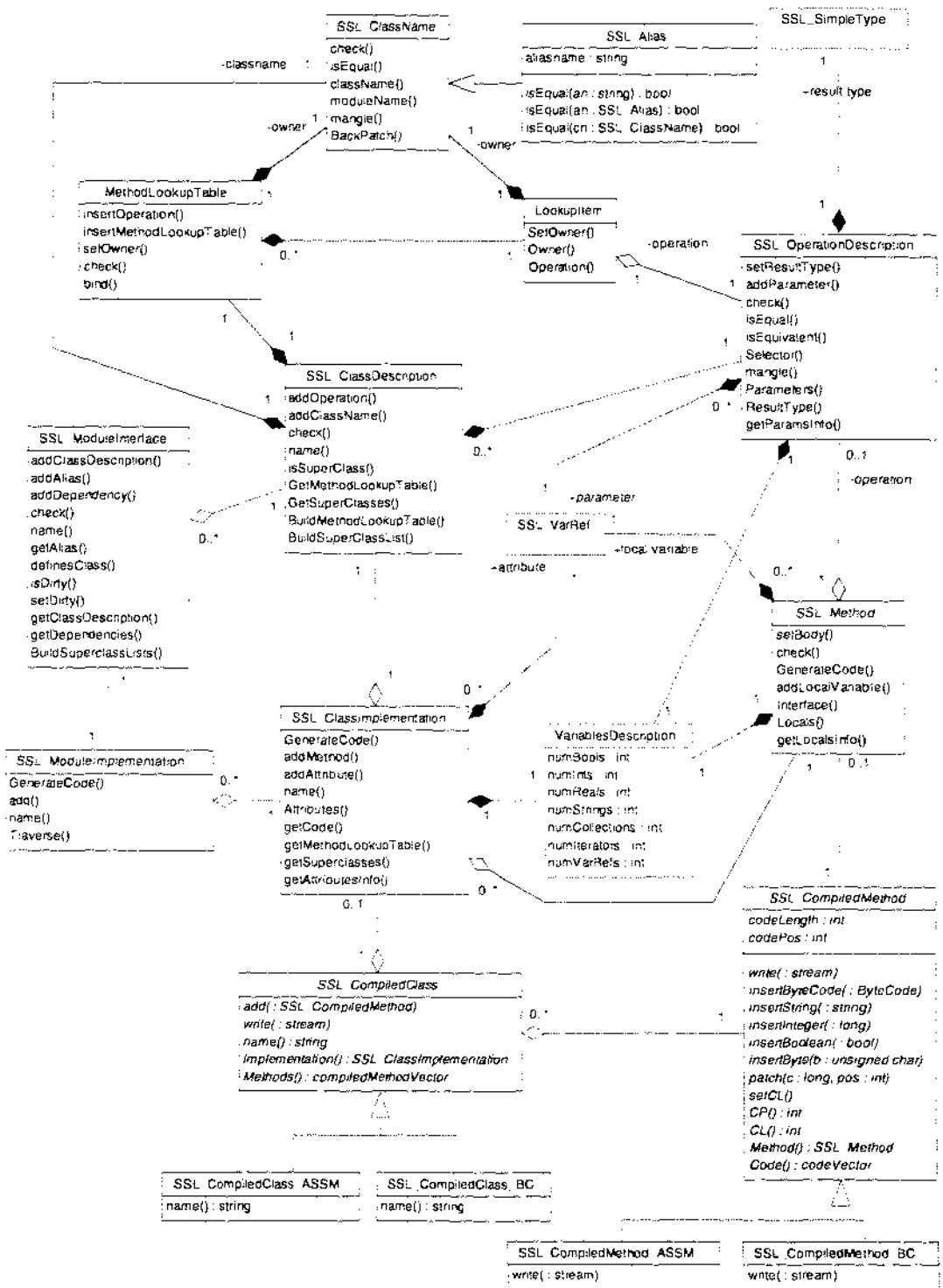


Figure VI-4 - Representing modules, classes, operations and methods in the SSL compiler

The classes in Figure VI-4 can be grouped into three categories:

- Classes that represent interfaces (*SSL Module Interface, SSL Class Description* and *SSL Operation Description*).
- Classes the represent implementations (*SSL Module Implementation*, *SSL Class Implementation* and *SSL Method*).
- Classes that represent compiled implementations (*SSL Compiled Class*, *SSL Compiled Class ASSM*, *SSL Compiled Class BC*, *SSL Compiled Method*, *SSL Compiled Method ASSM* and *SSL Compiled Method BC*).

SSL Module Interface represents an SSL interface module. It contains a collection of *SSL Class Description* instances (one for each class interface in the interface module) and is associated with an instance of *SSL Module Implementation*.

All *Class Description* objects have an associated instance of *SSL ClassName*. Class names are always fully qualified by a module name. SSL Alias extends *SSL ClassName* by overriding and overloading the *isEqual* operation. An instance of SSL Alias corresponds to a local alias introduced with a modules uses list.

The interface of an SSL class contains all of its operations and defines its super-classes. Operations are represented with an instance of *SSL Operation Description*. Each operation parameter is represented by an instance of *SSL VarRef* (a sub-class of *AST Node* in Figure VI-3).

A class's method lookup table is derived based solely on its interface. The *Method Lookup Table* class encapsulates a collection of *LookupItem* objects, each of which is an *SSL Class Name – SSL Operation Description* pair.

SSL Module Implementation (Figure VI-4) represents SSL implementation modules. It contains a collection of *SSL Class Implementation* instances, one for each *Class Description* instance in its associated *SSL Module Interface* object.

Each *SSL Class Implementation* object has a *Variables Description* object that defines the number of attributes, of each type, the SSL class has. The operations that are implemented by the SSL class are represented by a collection of *SSL Method* objects. The

body of the method is represented by an instance of *SSL Statement* (a sub-class of *AST Node* in Figure VI-3).

SSL Module Implementation, *SSL Class Implementation* and *SSL Method* all provide a *Generate Code* operation, which takes an *SSL Code Generator* object as an argument. *SSL Assembler Generator* overrides the *newClass* and *newMethod* operations to create instances of SSL *Compiled Class ASSM* and *SSL Compiled Method ASSM* respectively. *SSL ByteCode Generator* overrides the *newClass* and *newMethod* operations to create instances of SSL *Compiled Class BC* and *SSL Compiled Method BC* respectively.

SSL Compiled Class (Figure VI-4) defines the internal representation of compiled SSL classes. Its sub-classes (*SSL Compiled Class ASSM* and *SSL Compiled Class BC*) override the *name* method to produce file names for classes compiled into assembler or SSL-BC. The current implementation only uses a different format for the methods (i.e. assembler vs. SSL-BC) so the *write* method in only defined in *SSL Compiled Class*. *SSL Compiled Class ASSM* and *SSL Compiled Class BC* may override the *write* method in the future.

SSL Compiled Method defines the internal representation of compiled methods. Its sub-classes (*SSL Compiled Method ASSM* and *SSL Compiled Method BC*) override the *write* method to produce SSL assembler code and SSL BC code respectively.

Appendix VII

The SSL Virtual Machine

VII.1 Introduction

This appendix presents the design and implementation of the SSL Virtual Machine. The initial implementation of the SSL virtual machine is presented in (Griffin, 1997; Page *et al.*, 1997, 1998; Mehandjiska *et al.*, 1997).

VII.2 The SSL Virtual Machine

The SSL-VM was developed using:

- Gnu g++ 2.7.2 for Solaris 2.5 and Linux
- Microsoft Visual C++ 5.0 for Windows 95/NT

The primary design goal of the SSL-VM was that it be easy to modify, especially with respect to its instruction set. The SSL-VM design makes heavy use of patterns (especially the proxy pattern). The SSL-VM implementation makes heavy use of the C++ STL (Standard Template Library).

The major classes in the design of the SSL-VM are given Figure VII-1.

The class *Virtual Machine* has a stack (*Alpha Stack*) and three registers (*Project Register*, *Model Register* and *Diagram Register*). All message requests that occur whilst a method is executing on the SSL-VM are satisfied via the *Request Broker*, which is implemented using the singleton pattern (Gamma *et al*, 1995).

The classes *SSL Instance Manager* and *SSL Class Manager* correspond to the *SSL Object Client* and *SSL Class Client* of the Methodology Interpreter (Figure 3-10 - Proposed, top level, system architecture and Figure 3-11 - Architecture of the MOOT prototype). *SSL Class Manager* and *SSL Instance Manager* are responsible for isolating the SSL-VM from the persistent store.

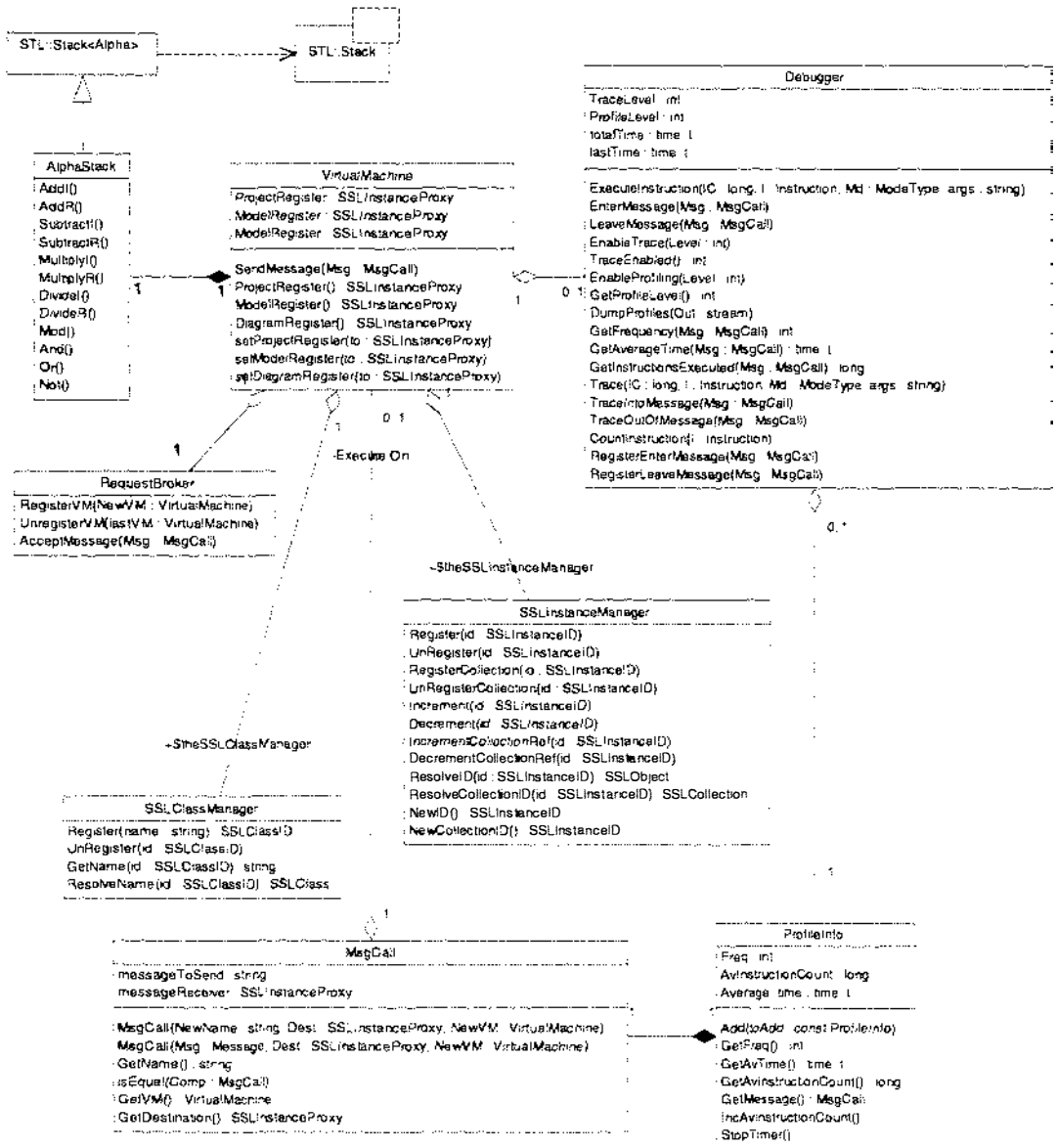


Figure VII-1 - Components of the SSL-VM

The *Debugger* class in Figure VII-1 provides facilities for tracing the messages and SSL-VM instructions that are executed by an instance of the SSL-VM. It also provides profiling of methods.

The *MsgCall* class represents a request for a particular message to be executed on the virtual machine. It encapsulates a message selector, a message receiver and a reference to the SSL-VM that the message is to be executed on. A more detailed description of binding and message execution is presented in sections VII.5 - Processing Messages and VII.6 - Binding.

VII.3 Representing SSL Types

The SSL types are implemented in the following manner:

Type	Implemented with
SSL integer	C++ long
SSL real	C++ double
SSL boolean	C++ bool
SSL String	C++ STL string
SSL Collection	C++ STL map
SSL Iterator	C++ STL pair
SSL Class	C++ class called *SSL Class*

Table VII-1 - Implementation of SSL types in the SSL-VM

Variables of the primitive types (integer, real, boolean and string) contain instances of the corresponding C++ types. Variables of SSL Collection and SSL Class contain a proxy object. Variables of an SSL iterator contain an instance of STL pair, where the first item is a collection proxy and the second is an index into the collection. The classes involved in representing SSL classes and SSL objects are discussed in section VII.4 - SSL Proxies.

VII.4 SSL Proxies

One of the goals of the design of the SSL-VM was that the persistence of SSL objects should be completely hidden. To achieve this goal, access to instances of SSL Class and SSL Collection is always via a proxy object. SSL proxies are an example of the Proxy design pattern (Gamma *et al.*, 1995).

Proxies encapsulate a unique ID and a static reference to a manager object. The manager object is responsible for resolving unique IDs into concrete objects and collections.

The SSL instance manager is responsible for managing instances of classes and collections. The SSL class manager is responsible for managing classes.

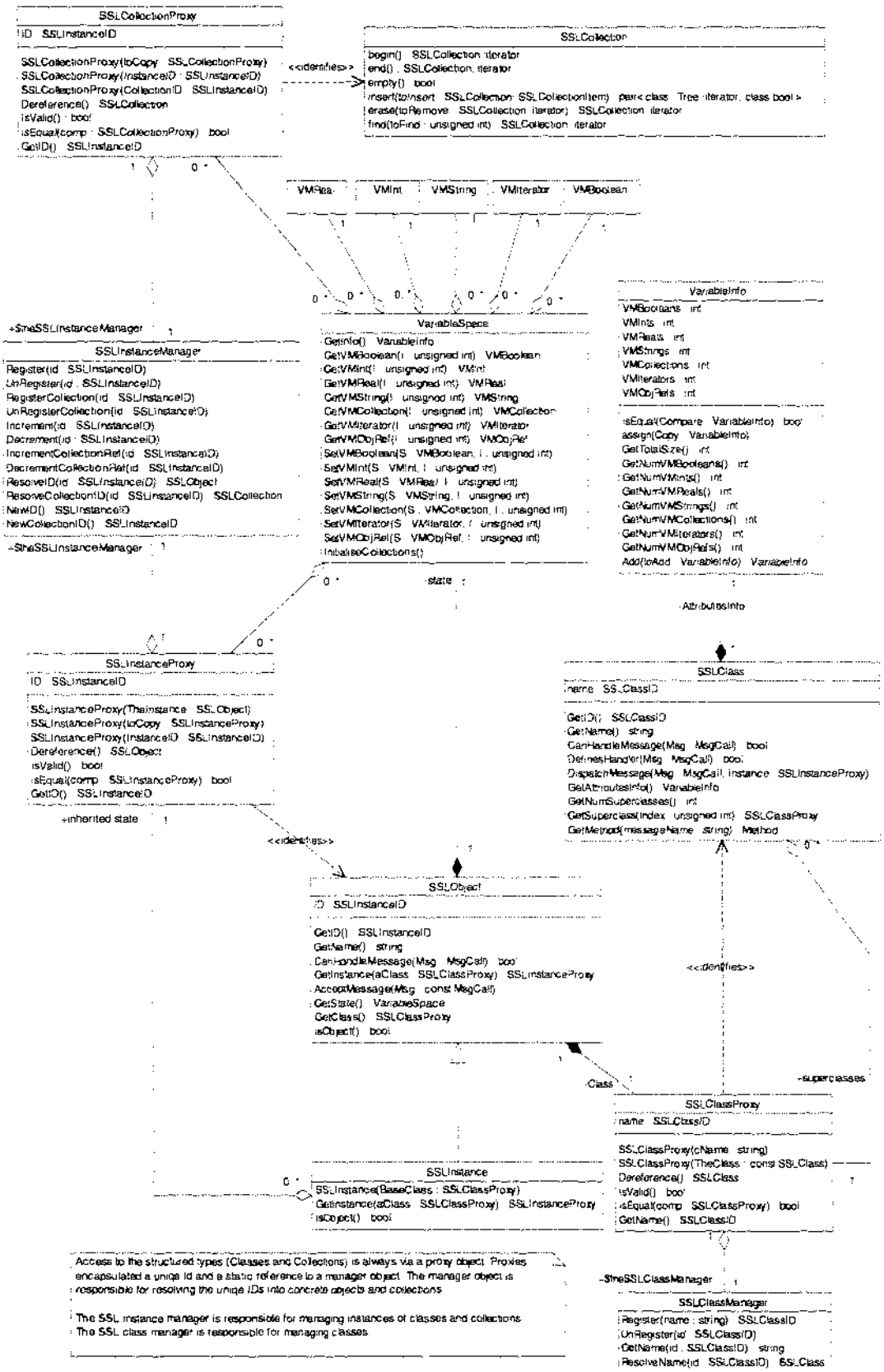


Figure VII-2 - Representing SSL objects and SSL classes

An instance of *SSL Class* encapsulates a *Variable Info* object that describes the number of attributes, of each type, that the class defines. Instances of an SSL Class encapsulate a *Variable Space* object that defines the state of the object. A *Variable Space* object contains collections of instances of the basic types and collections of proxy objects for SSL Collections and SSL Objects. The sizes of the collections are defines by a *Variable Info* object.

An instance of *SSL Class* encapsulates a collection of *SSL Class Proxy* objects that identify its direct and indirect super-classes. An instance of *SSL Object* encapsulates a collection of *SSL Instance Proxy* objects that correspond to the state described by the super-class *SSL Class Proxy* collection defined in its class.

VII.5 Processing Messages

Figure VII-3 shows the classes involved in processing a message on the SSL-VM.

A *MsgCall* object identifies a method to be executed for an object, on a particular SSL-VM.

The *Context* class encapsulates the context a method is interpreted in. It includes the attributes of *self* (the object that receives the message), the message arguments and local variables that are used within the method. The attributes are represented with an instance of *Variable Space*. Message arguments and local variables are represented with a second instance of *Variable Space*. SSL-VM instructions that change the value of an attribute, local variable or message argument act on an instance of *Context*.

Methods are executed by sending an *Interpret* message to an instance of the *Method* class. The method body consists of a sequence of bytes that corresponds to a set of SSL-VM instructions and their operands.

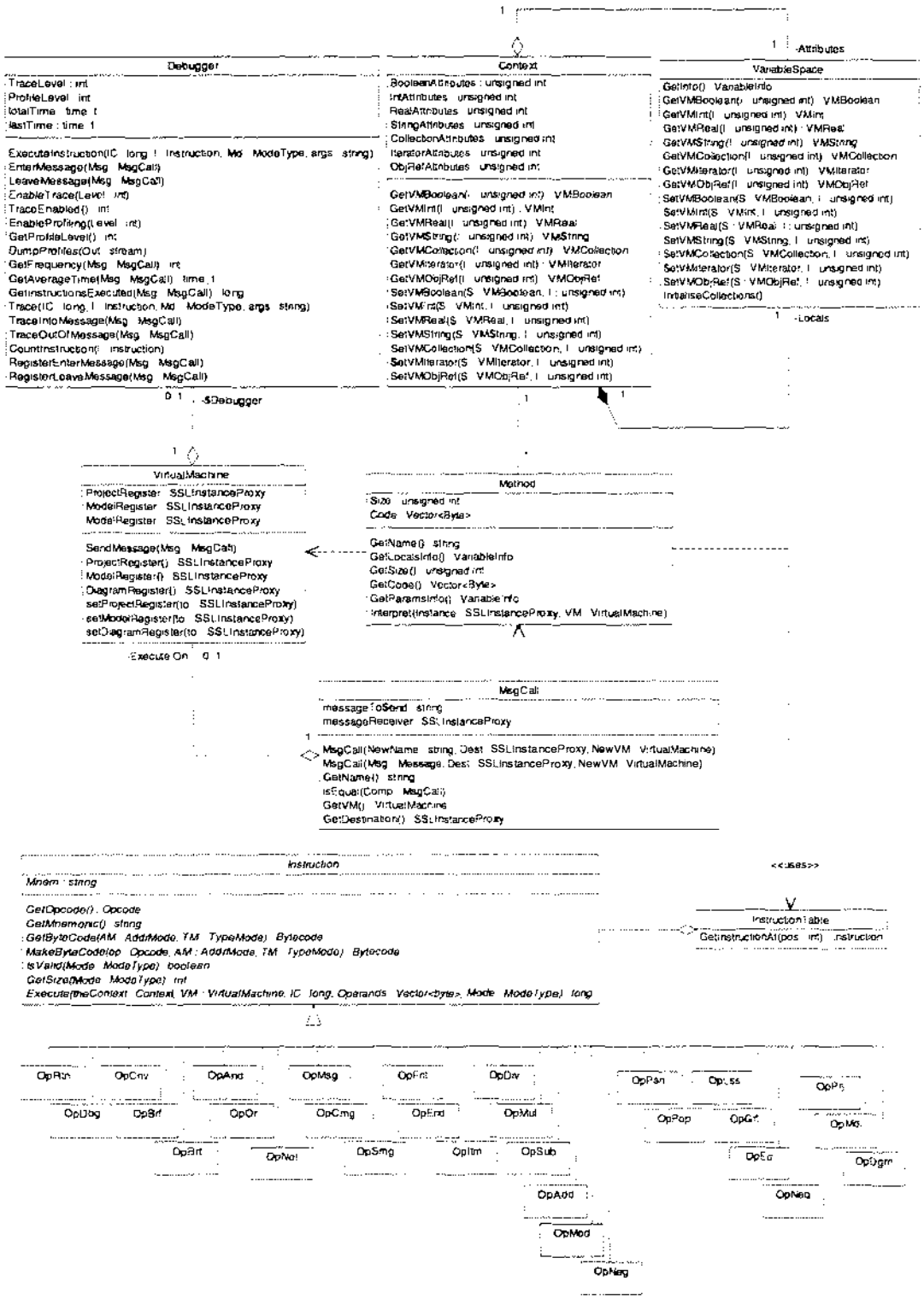


Figure VII-3 - The classes involved in processing a message on the SSL-VM

Figure VII-4 shows an abridged fragment of C++ from the Interpret method in the *Method* class.

```
Long IC = 0; // Instruction Counter
do
{
   if( IC < 0 || IC >= Size ) { /* signal an error */ }

   // Fetch
   Opcode next = GetInstruction(Code + IC);

   // Decode
   Instruction *Instr = InstructionTable()[next.instCode()];

   // Execute
   IC = Instr->Execute(
          theContext,
          VM,
          IC,
          Code + IC + sizeof(Opcode),
          next.addrMode() );
}
while( IC != ReturnFromMessage );
```

Figure VII-4 - Executing a method on the SSL-VM

Figure VII-4 shows the fetch-decode-execute cycle that is performed for each SSL-VM instruction. The first step (fetch) involves translating the byte located at the Instruction Counter (IC) into an SSL-VM opcode. The next step (decode) involves retrieving the corresponding instance of the *Instruction* class from the *InstructionTable* (*Instr*). The *Instruction* class, and its sub-classes, are instances of the Flyweight design pattern (Gamma *et al.*, 1995). The last step (execute) is performed by the Opcode object (*Instr*) itself.

VII.6 Binding

Figure VII-5 shows how a message is bound to method and executed on the SSL-VM.

The message binding process starts when the message request broker receives a request to dispatch a message to a particular object. The request will be accompanied by:

- An *SSL Instance Proxy* object
- An instance of the *MsgCall* class (see Figure VII-3)

The SSL Instance Proxy object identifies the object that is to receive the message. The *MsgCall* object identifies the method to be executed and the particular SSL-VM it is to be executed on.

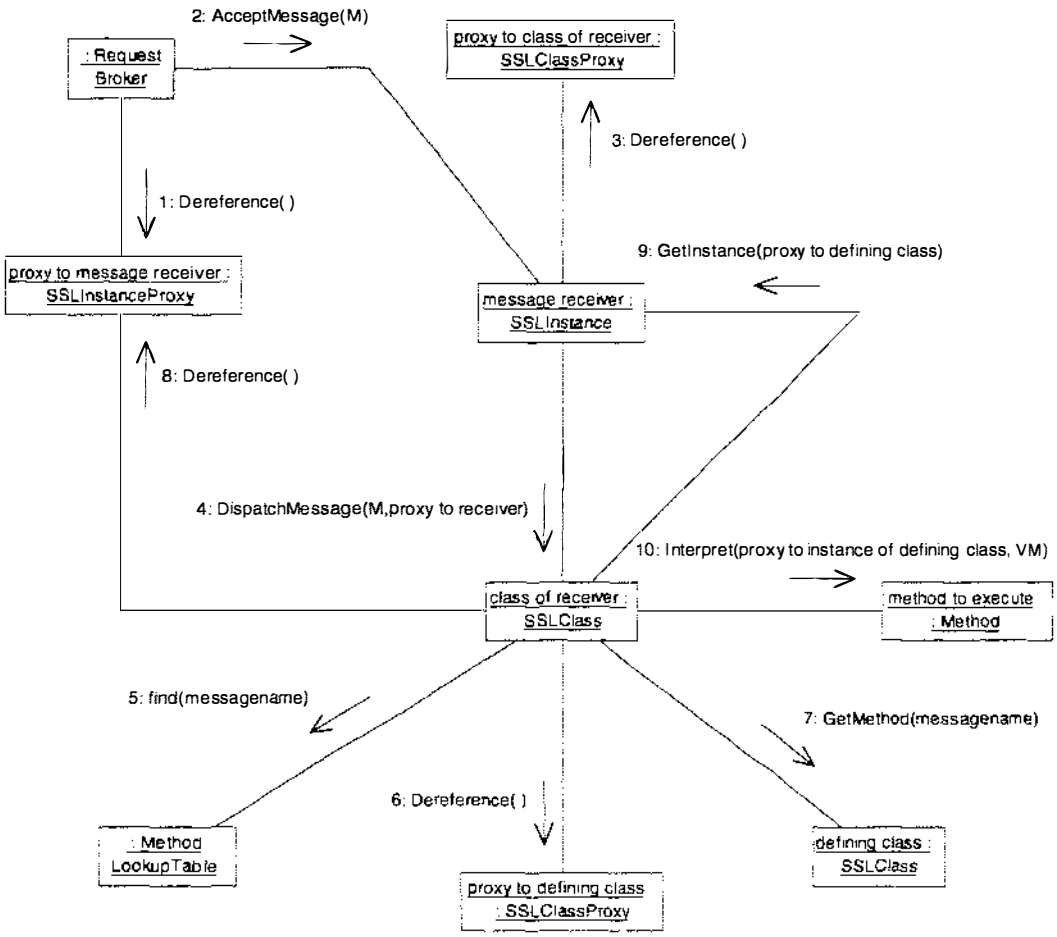


Figure VII-5 - Binding a message to a method on the SSL-VM

The following steps are performed to bind a message to a method and execute it on an SSL-VM:

1. The message request broker first de-references the proxy to the receiving object (*proxy to message receiver* in Figure VII-5). It then asks the message receiver object to accept the message.

2. The message receiver object (*message receiver* in Figure VII-5) delegates the responsibility of finding an appropriate method to its class. It must first de-reference the proxy that defines its class (*proxy to class of receiver* in Figure VII-5). The message receiver then sends a *Dispatch Message* message to its class (*class of receiver* in Figure VII-5), with the message and a proxy to itself as arguments.

3. The class of the message receiver (*class of receiver* in Figure VII-5) uses its *Method Lookup Table* to determine which class defines a method that can be bound to the message.

4. The *class of receiver* object then de-references the proxy to the method defining class (*proxy to defining class* in Figure VII-5). The method defining class (*defining class* in Figure VII-5) is then asked to provide a method that corresponds to the message (*method to execute* in Figure VII-5).

5. The *class of receiver* object then asks the *message receiver* object to return a proxy to the instance of the method defining class (*proxy to instance of defining class* in Figure VII-5). The proxy returned will either: correspond to the message receiver, or to the instance of one of the super-classes, of the message receiver's class (that is part of its inherited state).

6. Finally the *method to execute* object is asked to interpret itself (message number 10 in Figure VII-5).

VII.7 Garbage Collection

The garbage collection scheme is completely transparent to the Virtual machine. It is a simple adaptation of the reference counting garbage collection algorithm (Jones and Lins, 1996).

Figure VII-6 shows the classes involved in the reference counting garbage collection scheme used in the SSL-VM.

The classes *SSL Instance Server* and *SSL Class Server* correspond to the SSL Object Server and SSL Class server in Figure 3-10 - Proposed, top level, system architecture and Figure 3-11 - Architecture of the MOOT prototype respectively.

In the prototype implementation of the MOOT core *SSL Instance Manager* and *SSL Class Manager* simply forward requests directly on to the corresponding server object. *SSL Instance Server* maintains a map of SSL Objects (indexed by their *SSL Instance ID*) and a map of SSL Collections (also indexed by their *SSL Instance ID*). It maintains two maps of reference counts (indexed by an *SSL Instance ID*), one for the *Instance Map* and one for the *Collection Map*. When a collection or object is registered for the first time a new entry is added to the appropriate *Reference Count Map* with an initial count of 1.

Figure VII-6 shows that the *SSL Class Server* maintains a map of SSL Classes (indexed their *SSL Class ID*). The SSL Class Server also maintains complementary maps of SSL class names and SSL Class IDs.

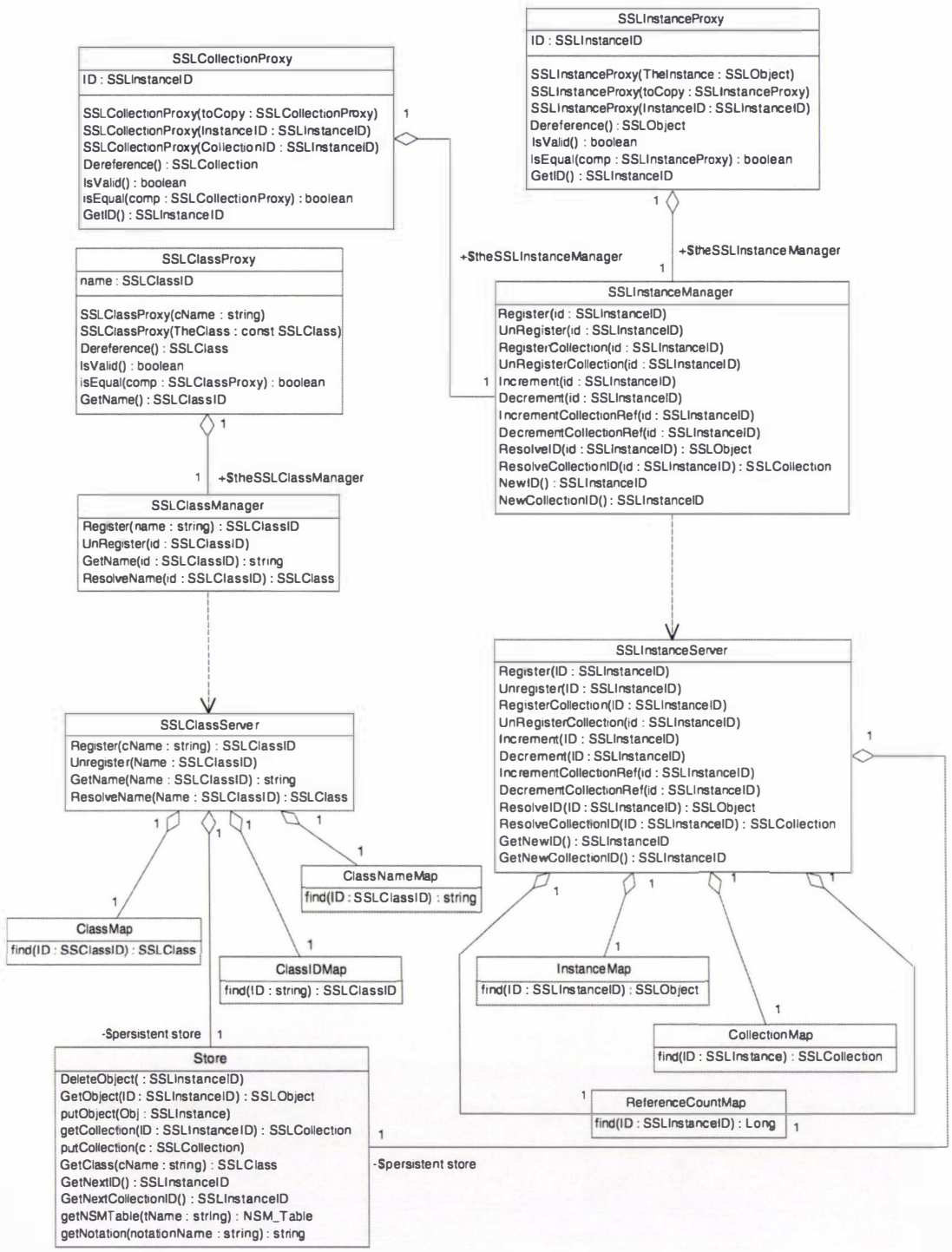


Figure VII-6 - Implementation of the reference counting garbage collection scheme

The *SSL Instance Proxy* and *SSL Collection Proxy* classes in Figure VII-6 drive the reference counting process. Proxy objects notify their manager whenever they are created, copied and deleted. Creating and copying proxies corresponds to incrementing a reference count and deleting a proxy corresponds to decrementing a reference count. Such a scheme is easily implemented as shown in Figure VII-7.

```
// a proxy with ID zero corresponds to no object
SSLInstanceProxy::SSLInstanceProxy() : ID(0) {}

// contructed from an exisiting object
SSLInstanceProxy::SSLInstanceProxy(
    const SSLObject &TheInstance ) : ID( TheInstance.GetID() )
{
    assert( IsValid() );
    Manager->Increment( ID );
}

// create from an exisiting SSL ID
SSLInstanceProxy::SSLInstanceProxy(
    const SSLInstanceID &InstanceID ) : ID( InstanceID )
{
    assert( IsValid() ); assert( InstanceID != 0 );
    Manager->Register( ID );
    Manager->Increment( ID );
}

// copy constructor
SSLInstanceProxy::SSLInstanceProxy(
    const SSLInstanceProxy& toCopy ) : ID( toCopy.ID )
{
    assert( IsValid() );
    if( ID ) Manager->Increment( ID );
}

// destructor
SSLInstanceProxy::~SSLInstanceProxy()
{
    assert( IsValid() );
    if( ID ) Manager->Decrement( ID );
}

// assignment operator
SSLInstanceProxy &SSLInstanceProxy::operator = (
    const SSLInstanceProxy &Copy )
{
    if( this != &Copy )
    {
      assert( IsValid() );
      if( ID ) Manager->Decrement( ID );
      ID = Copy.GetID();
      if( ID ) Manager->Increment( ID );
    }
    return *this;
}
```

Figure VII-7 - Implementation of the SSL Instance Proxy class

An instance of *SSLInstanceProxy* (Figure VII-7) sends *Increment* and *Decrement* messages to its manager in the constructors, destructor and the overloaded assignment operator. These C++ member functions collectively define the primitive operations on a type in C++ (i.e. duplication, instantiation, deletion and assignment).

References

Adams, S. K. (1998) Development of a Client Interface for a Methodology Independent Object-Oriented CASE Tool, Massey University Masters Thesis, Department of Computer Science, Massey University, Palmerston North, New Zealand

Alderson, A. (1991) Meta-CASE Technology, Proceedings of Software Development Environments and CASE Technology, LNCS, Springer-Verlag, New York, Vol. 509, pp81-91

Alfabet (1998) Alfabet CASE Tool homepage, http://www.alfabet.de/

Amulet (1998) Amulet home page
http://www.cs.cmu.edu/Groups/amulet/amulet-home.html

Apperley, M. and Chester, M. (1995) Tree Browsing, Working Paper 96/13, Department of Computer Science, The University of Waikato, Hamilton, New Zealand

Apperley, M. and Duncan, A. (1994) Human-Computer Interface Design in the Software Lifecycle, SRIG-ET'94, University of Otago, IEEE Computer Society Press, pp60-64

Aranow, E. (1996) Growing a Software Reuse Program, Tutorial notes for TOOLS New Zealand Workshop

Arnold, P., Bodoff, S., Coleman, D., Gilchrist, H. and Hayes, F. (1991) An Evaluation of Five Object Oriented Development Methods, Research Report, Hewlett Packard Laboratories, Bristol, United Kingdom

Artsy, Y.S. (1995) Meta-modelling the OO methods, Tools, and Interoperability Facilities, position paper for Meta-modelling in OO OOPSLA'95 Workshop

ASD (1995a) Graphical Designer Language ver 1.2.24 Advanced Software Technologies Inc

ASD (1995b) Graphical Designer Users Manual, Advanced Software Technologies Inc

ASD (1998) Advanced Software Technologies Ltd (Graphical Designer) homepage http://www.advancedsw.com/

Ayoma, M. (1998) New Age of Software Development: How Component-Based Software Engineering Changes the Way of Software Development, ICSE International Workshop on Component-Based Software Engineering, Kyoto, Japan

Barbacci, M.R and Weinstock, C.B. (1998) Mapping MetaH into ACME, Software Engineering Institute Special Report CMU/SEI-98-SR-006, Carnegie Mellon University

Behforooz, A. and Hudson, J. (1996) Software Engineering Fundamentals, Oxford University Press, Oxford

Bergner, K., Rausch, A. and Sihling, M. (1998) Componentware – The Big Picture, ICSE International Workshop on Component-Based Software Engineering, Kyoto, Japan

Beynon-Davies, P. (1989) Information Systems Development: An Introduction to Information Systems Engineering, Computer Aided Information Systems Engineering (CAISE), Macmillan Press Ltd, London, pp75-81

Blanchard, T. (1995) Meta-Models as a Foundation for Implementation of Business Rules, position paper for Meta-modelling in OO OOPSLA'95 Workshop

Boehm, B. W. (1976) Software Engineering, IEEE Transactions on Computers, C-25(12), pp1226-1241

Booch, G. (1991) Object Oriented Analysis and Design with Applications, The Benjamin Cummings Publishing Company Inc., Redwood City, California

Booch, G. (1994) Object Oriented Analysis and Design with Applications, 2nd edition, The Benjamin Cummings Publishing Company Inc., Redwood City, California

Booch, G. (1996) Object Solutions: Managing the Object-Oriented Project, Addison-Wesley, Reading, Massachusetts

Booch, G. and Rumbaugh, J. (1995) Unified Method, Version 0.8, Rational Software Corporation, (unpublished)

Booch, G., Rumbaugh, J. and Jacobson, I. (1999) The Unified Modelling Language User Guide, Addison-Wesley, Reading, Massachusetts

Brinkkemper, S., Hong, S., Bulthuis, A. and van den Goor, G. (1998) Object-Oriented Analysis and Design Methods a Comparative Review, http://wwwis.cs.utwente.nl:8080/dmrg/OODOC/oodoc/oo.html

Brough, M. (1992) Methods for CASE: a Generic Framework, Advanced Information Systems Engineering: 4th International Conference CAiSE'92, Ed. Loucopoulos, P., Springer-Verlag, Berlin, pp524-545

Brown, A. W. (1997) CASE in the 21st Century: Challenges Facing Existing CASE Vendors, Proceedings of the 8th International Workshop on Software Technology and Engineering Practice (STEP'97), IEEE Computer Society Press, London, UK

Brown, A.W. and Jaeger, K. (1998) The Future of Enterprise Application Development with Components and Patterns, August, Sterling Software, Tennyson Parkway, Plano, Texas

CDIF (1998) CDIF homepage, http://www.cdif.org/

Choi, M.D. (1996) Object-Oriented Meta CASE Tools: Information Interchange between Methodologies, Massey University Honours Report, Department of Computer Science, Massey University, Palmerston North, New Zealand

Clark, P. (1994) Methodology Independent CASE Tool - A Prototype. Massey University Masters Thesis, Department of Computer Science, Massey University, Palmerston North, New Zealand

Coad, P. and Nicola, J. (1993) Object Oriented Programming, Yourdon Press, Englewood Cliffs, New Jersey

Coad, P. and Yourdon, E. (1990) Object Oriented Analysis, Yourdon Press, Englewood Cliffs, New Jersey

Coad, P. and Yourdon, E. (1991a) Object Oriented Analysis, 2nd edition, Yourdon Press, Englewood Cliffs, New Jersey

Coad, P. and Yourdon, E. (1991b) Object Oriented Design, Yourdon Press, Englewood Cliffs, New Jersey

Coleman, D., Arnold, P., Bodoff, S., Dollin, C., Gilchrist, H., Hayes, F., and Jeremaes, P. (1993) Object-Oriented Development: The Fusion Method, Prentice-Hall

Collins (1995) Collins Shorter English Dictionary: The Authority on Current English, Harper-Collins publishers, Glasgow

COTAR (1998) Centre for Object Technology Applications and Research homepage, http://www.csse.swin.edu.au/cotar/

Coxhead, G. and Fisher, B. (1994a) An Introduction to Ipsys (Tool Fragment), Lincoln Software Limited Manchester, England

Coxhead, G. and Fisher, B. (1994b) An Introduction to Ipsys (SSADM 4+), Lincoln Software Limited Manchester, England

Coxhead, G., Ellyard, J. and Stead, S. (1994) An Introduction to Ipsys (HOOD 4.2.1), Lincoln Software Limited Manchester, England

Cribbs, J., Roe, C. and Moon, S. (1992) An Evaluation of Object-Oriented Analysis and Design Methodologies, SIGS books, New York

Crozier M., Glass, D., Hughes, J.C., Johnston, W. and McChesney, I. (1989) Critical Analysis of Tools for Computer-Aided Software Engineering, Information and Software Technology, Vol. 31 (9), pp486-496

D'Souza, D., and Wills, A. (1998) Objects, Components, and Frameworks with UML: The Catalysis Approach, http://www.trireme.com/catalysis/book/

Dasari, S., Mehandjiska, D. and Page, D. (1995) Construction of a Generic Knowledge Base for a Methodology Independent CASE Tool. Addendum to the ANNES'95 Proceedings, The Second NZ International Two-Stream Conference on Artificial Neural Networks and Expert Systems, Dunedin, pp466-473

Day, D. (1998) Behavioural Effects of Attitudes Toward Constraint in CASE: The Impact of Development Task and Project Phase, Centre for Advanced Empirical Software Engineering, The University of New South Wales, Sydney 2052, Australia

de Champeaux, D. and Faure, P. (1992) A Comparative Study of Object Oriented Analysis Methods, Journal of Object Oriented Programming, Vol. 5 (1), pp21-32

Demetrovics, J., Knuth, E. and Rado, P. (1982) Specification Meta Systems, IEEE Computer, pp29-35

Demphlous, S. and Lebastard, F. (1995) Persistence of Multiple Object Models, position paper for Meta-modelling in OO OOPSLA'95 Workshop

Deutsch L. and Schiffman, A. M. (1984) Efficient Implementation of the Smalltalk-80 System, Proceedings of the 11th Annual ACM Symposium on the Principles of Programming Languages POPL84, Salt Lake City, Utah

Douglas, B.P (1998) Real-Time UML: Developing Efficient Objects for Embedded Systems, Addison-Wesley, Reading, Massachusetts

Ebert, J., Suttenbach, R. and Uhe, I. (1996) Meta-CASE in practice: A Case for KOGGE, Technical report 22/96 University of Koblenz-Landau Institute of Software Technology

EIA (1998) Electronic Industry Association homepage, http://www.eia.org/

EIA CDIF (1994a) Extract of Interim Standard - CASE Data Interchange Format - Overview EIA/IS-106

EIA CDIF (1994b) Extract of Interim Standard - CDIF Framework for Modelling and Extensibility EIA/IS-107

EIA CDIF (1994c) Framework for Modelling and Extensibility, Interim Standard, EIA

EIA CDIF (1994d) Integrated Meta-model, Data Flow Subject Area, Interim Standard, EIA

EIA CDIF (1994e) Integrated Meta-model, Foundation Subject Area, Interim Standard, EIA

EIA CDIF (1994f) Transfer Format - General Rules for Syntaxes, Interim Standard, EIA

EIA CDIF (1994g) Transfer Format - Transfer Format Encoding - ENCODING.1, Interim Standard, EIA

EIA CDIF (1994h) Transfer Format - Transfer Format Syntax - SYNTAX.1, Interim Standard, EIA

EIA CDIF (1996) Integrated Meta-model, Common Subject Area, Interim Standard, EIA

Ernst, J. (1996) EIA/CDIF Technical Committee, Working Document – Aligning Meta-meta-models CDIF-JE-N27-V1, November 13

ExcelSoftware (1998) Homepage of the MacA&D and WinA&D CASE tools, http://www.excelsoftware.com/

Ferguson, I. (1998) The Centre for MetaCASE and Method Engineering homepage, http://osiris.sunderland.ac.uk/rif/metacase/metacase.home.html

Feylock, S. (1977) Transition Diagram-Based CAI/HELP systems, Int. J. Man-Machine Studies, Vol. 9, pp399-413

Fichman R.G. and Kemerer, C.F. (1992) Object-Oriented and Conventional Analysis and Design Methodologies: Comparison and Critique, IEEE Computer 1992, pp22-38

Findeisen, P. (1993) The Graphical Extension for the EARA Model – version 2, Department of Computer Science, University of Alberta

Findeisen, P. (1994a) The EARA Model for MetaView – A Reference, Department of Computer Science, University of Alberta

Findeisen, P. (1994b) A Complete Definition of Data Flow Diagram Environment for MetaView, Department of Computer Science, University of Alberta

Findeisen, P. (1994c) The MetaView System, Department of Computer Science, University of Alberta

Findeisen, P. (1994d) Environment Constraint Language – A Draft Proposal, Department of Computer Science, University of Alberta

Firesmith, D.G. (1993) Object-Oriented Requirements Analysis and Logical Design, Addison-Wesley, Reading, Massachusetts

Firesmith, D.G and Eykholt, E.M. (1995) Dictionary of Object Technology, SIGS Books Inc., New York

Firesmith, D. G. and Henderson-Sellers, B. (1998a) Clarifying Specialised Forms of Association in UML and OML, Journal of Object-Oriented Programming, Vol. 11 (2), pp47-50

Firesmith, D.G. and Henderson-Sellers, B. (1998b) Upgrading OML to Version 1.1 Part I. Referential Relationships, Journal of Object-Oriented Programming, Vol. 11 (3), pp48-57

Firesmith, D., Henderson-Sellers, B. and Graham, I. (1997) OPEN Modelling Language (OML) Reference Manual, SIGS Books Inc., New York, pp271

Flatscher, R. G. (1996) Modelling of Business Information Systems: An Overview of the Architecture of EIA's CASE Data Interchange Format (CDIF), Journal of the WG 5.2, Architecture of Information Systems-German Society of Informatics, SG 5.2.1, Vol. 3 (1), pp26-30

Forman, I.R., Conner, M.H., Danforth, S.H. and Raper, L.K. (1995) Release-to-Release Binary Compatibility in SOM, Tenth Annual Conference on Object-Oriented Programming Systems, Languages and Applications OOPSLA'95, Austin, Texas, USA, pp426-438

Fowler, M. (1997) Analysis Patterns: Reusable Object Models, Addison-Wesley, Reading, Massachusetts

Fowler, M. and Scott, K. (1997) UML Distilled – Applying the Standard Object Modelling Language, Addison-Wesley, Reading, Massachusetts

FreeCASE (1998) The FreeCASE project, http://www.freecase.seul.org/index.html

Froehlich, G. (1994) Process Modelling in MetaView, University of Saskatchewan Masters thesis, Department of Computer Science, University of Saskatchewan, Saskatoon, Saskatchewan, Canada

Fung, M., Henderson-Sellers, G. and Yap, L. (1997) A Comparative Evaluation of OO Methodologies from a Business Rules and Quality Perspective, The Australian Computer Journal, Vol. 29 (3), pp95-101

Gadwal, D., Lo, P. and Millar, M. (1994a) EDL/GE User's Manual, Department of Computer Science, University of Alberta

Gadwal, D., Findeisen, P., Sorenson, P.G., Tremblay, J.P. and Millar, B.L. (1994b) Generating Customisable Software Specification Environments Using MetaView, Research Report 94-2, Department of Computer Science, University of Saskatchewan

Gamma, G., Helm, R., Johnson, R. and Vlissides, J. (1995) Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, Reading, Massachusetts

Gibson, M. (1988) A Guide to Selecting CASE Tools. Datamation, pp65-66

Goldberg A. and Robson D. (1983) Smalltalk-80: The language and its Implementation, Addison-Wesley, Reading, Massachusetts

Goldberg, A. and Rubin, K.S (1995) Succeeding with Objects: Decision frameworks for Project Management, Addison-Wesley, Reading, Massachusetts

Graham, I.M. (1994) Migrating to Object Technology, Addison-Wesley, Wokingham, UK

Graham, I. and Henderson-Sellers, B. (1997) OPEN's Toolbox of Techniques, Addison-Wesley, Reading, Massachusetts

Graham, I., Henderson-Sellers, B. and Younessi, H. (1997) The OPEN Process Specification, Addison-Wesley, Reading, Massachusetts

Gray, A. (1995) GUI of Object Oriented CASE Tools, Massey University Honours Report, Department of Computer Science, Massey University, Palmerston North, New Zealand

Griffin, D. P. (1997) From the Ridiculous to the Sublime: Representation and Execution of Semantic Methodology Meta-Knowledge in an Object Oriented Meta-CASE Tool, Massey University Honours Report, Department of Computer Science, Massey University, Palmerston North, New Zealand

Haine, P. (1992) Second Generation CASE: Can it be Justified?, CASE: Current Practice – Future Prospects, Eds. Spurr, K. and Layzell, P., John Wiley and Sons Ltd, New York, USA, pp137-145

Ham, J. (1994) Template Generator for a Methodology Independent Object-Oriented CASE Tool. Massey University Honours Report, Department of Computer Science, Massey University, Palmerston North, New Zealand

Henderson-Sellers, B. (1996) The OPEN-Mentor Methodology, Object Magazine, Vol. 6 (9), pp56-59

Henderson-Sellers, B. (1997) ●PEN Relationships - Compositions and Containments, Journal of Object-Oriented Programming, Vol. 10 (7), pp51-55

Henderson-Sellers, B. (1998) OPEN Relationships - Associations, Mappings, Dependencies, and Uses, Journal of Object-Oriented Programming, Vol. 10 (9), pp49-57

Henderson-Sellers, B. and Bulthuis, A. (1996a) An overview of the COMMA project, Report on Object Analysis and Design, Vol. 2 (7), pp49-52

Henderson-Sellers, B. and Bulthuis, A. (1996b) COMMA: Sample Metamodels, Journal of Object-Oriented Programming, Vol. 9 (7), pp44-48

Henderson-Sellers, B. and Bulthuis, A. (1997) Object-Oriented Metamethods, Springer-Verlag, New York

Henderson-Sellers, B. and Edwards, J. (1994) The Working Object, Book Two of Object-Oriented Knowledge, Prentice-Hall, Object-Oriented Series, Englewood Cliffs, New Jersey

Henderson-Sellers, B. and Firesmith, D. (1997a) COMMA: Proposed core model, Journal of Object-Oriented Programming, Vol. 9 (8), pp48-53

Henderson-Sellers, B. and Firesmith, D. (1997b) Evaluating Third Generation OO Software Development Approaches submitted to Information and Software Technology, http://www.csse.swin.edu.au/cotar/OPEN/istwww.pdf

Henderson-Sellers, B. and Graham, I. M. (1996) OPEN: Toward Method Convergence?, IEEE Computer, Vol. 29(4), pp86-89

Henderson-Sellers, B., Graham, I.M., Firesmith, D., Reenskaug, T., Swatman, P. and Winder, R. (1996) The OPEN heart, TOOLS 21, Eds. Mingins, C., Duke, R. and Meyer, B., TOOLS/ISE, pp187-196

Henderson-Sellers, B., Firesmith, D.G. and Graham, I.M. (1997a) COMMA: Its Influence on OPEN, Journal of Object-Oriented Programming, Vol. 10 (1), pp47-51

Henderson-Sellers, B., Firesmith, D.G. and Graham, I.M. (1997b) Methods Unification: The OPEN Modelling Language (OML), Journal of Object-Oriented Programming, Vol. 10 (5), pp28-34

Henderson-Sellers, B., Graham, I.M. and Firesmith D.G. (1997d) Methods Unification: The OPEN Methodology, Journal of Object-Oriented Programming, Vol. 10 (2), pp41-43, 55

Henderson-Sellers, B., Graham, I. and Younessi, H. (1997e) The OPEN Process Specification, Addison-Wesley, Reading, Massachusetts

Hoffman, D. and Strooper, P. (1995) Software Design, Automated Testing and Maintenance: A Practical Approach, International Computer Press, London

Hong, S., van den Goor, G. and Brinkkemper, S. (1993) A Formal Approach to the Comparison of Object-Oriented Analysis and Design Methodologies, Hawaii International Conference on System Sciences (HICSS), IEEE Computer Society Press, Hawaii, Vol. IV, pp689-698

Huff, C., Smith, D., Stepien-Oakes, K. and Morris, E. (1992) Proceedings of the CASE Adoption Workshop, Technical Report CMU/SEI-91-TR-14 ESD-TR-91-14

Hutt A.T.F. (1994) Object Analysis and Design: Comparison of Methods, Ed. Andrew, T.F., John Wiley and Sons Inc., New York, USA

ICON (1998) ICON Computing, http://www.iconcomp.com/

I-Kinetics (1998) The Internet, Java & CORBA Resource Guide, http://www.componentware.com/wp/ijcprimer.htm

Innovative Software (1998) Object Engineering Workbench homepage, http://world.isg.de/world/

Isazadeh, H. and Lamb, D.A. (1997) CASE Environments and MetaCASE Tools, External Technical Report ISSN-0836-0227-1997-403, Department of Computing and Information Science, Queens University, Ontario

ISO (1998a) International Standards Organisation homepage, http://www.iso.cn/

ISO (1998b) ISO/IEC JTC1/SC7 homepage, http://saturne.info.uqam.ca/Labo_Recherche/Lrgl/sc7/

Jacobson, I., Christenson, M., Jonsson, P. and Overgaard, G. (1993) Object Oriented Software Engineering: A Use Case Driven Approach, Addison-Wesley, Wokingham, England

Jacobson, I., Booch, G. and Rumbaugh, J. (1996) Unified Standard Moves Closer, Object Expert, Vol. 1(6), pp64-65

Jacobson, I., Booch, G. and Rumbaugh, J. (1999) The Unified Software Development Process, Addison-Wesley, Reading, Massachusetts

Jacobson, I., Ericsson, M. and Jacobson, A. (1995) The Object Advantage: Business Process Reengineering with Object Technology, Addison-Wesley, Reading, Massachusetts

Jones, R. and Lins, R. (1996) Garbage Collection: Algorithms for Automatic Dynamic Memory Management, John Wiley & Sons Ltd, New York, USA

Joosten, S. (1995) A Method for Analysing Workflows. Tutorial for Workflow Management Support, ECSCW'95

JrCASE (1998) The CSIRO-Macquarie University Joint Research Centre for Advanced Systems Engineering, CASEMaker, http://www.jrcase.mq.edu.au/

LaMonica, M. (1997) Middleware - Managing Objects: DCOM and CORBA vie for Supremacy, InfoWorld Electric, October 20

Lang, B. (1991) CASE Support for the Software Process: Advances and Problems, ESEC'91 Eds. Lamsweerde, A. and Fugetta, A., Springer-Verlag, Berlin, pp512-515

Lano, K. and Haughton, H. (1994) Object-Oriented Specification Case Studies, Prentice-Hall, London

Leung, Y.K. and Apperley, M. D. (1993) A Taxonomy of Distortion-Oriented Techniques for Graphical Data Presentation, Eds. Salvendy, G. and Smith, M., Advances in Human Factors/Ergonomics, Vol. 19B, Elsevier, Amsterdam, pp104-110

Leung, Y.K. and Apperley, M.D. (1994) A Review and Taxonomy of Distortion-Oriented Presentation Techniques, ACM Transactions on CHI, Vol. 1(2), pp126-160

Leung, Y.K., Spence, R. and Apperley, M.D. (1995) Applying Bifocal Displays to Topological Maps, International Journal of Human-Computer Interaction, Vol. 7(1), pp79-98

Lincoln (1994) The FastPath Approach to Client/Server Development, Lincoln Software Limited Manchester, England

Lincoln (1998) Lincoln software - Toolbuilder homepage, http://www.ipsys.com/

Lindholm T. and Yellin F. (1997) The Java Virtual Machine Specification, Addison-Wesley, Reading, Massachusetts

Lo, P. (1995) Graphical Interface for CASE environment definitions in MetaView, University of Alberta Masters thesis, Department of Computer Science, University of Alberta, Canada

Loy, P.H. (1990) A Comparison of Object-Oriented and Structured Development Methods, ACM SIGSOFT Software Engineering Notes, Vol. 15 (1), pp44-48

Lyytinen. K., Kerola, P., Kaipala, J., Kelly, S., Lehto, J., Liu, H., Marttiin, P., Oinas-Kukkonen, H., Pirhonen, J., Rossi, M., Smolander, K., Tahvanainen. V. and Tolvanen, J. (1994) MetaPHOR: Meta-Modelling, Principles, Hypertext, Objects and Repositories, Tech Report TR-7, Depart of Computer Science and Information Systems, University of Jyvaskyla, Finland

Malmborg, L. (1992) Diffusion of CASE An Obstacle Race?, Scandinavian Journal of Information Systems, Vol. 4, pp105-118

Maokai, G. and Scott, L. (1998) Developing the User Interface for the MetaCASE Toolset (Concept document), ftp://ftp.mpce.mq.edu.au/pub/jrcase/metaCASE/metaui.ps.Z

MarkV (1994) ObjectMaker Version 4 User's Guide Mark V Systems, Limited

MarkV (1998) ObjectMaker homepage, http://www.markv.com/

Marlin, C. (1996) Multiple Views Based on Unparsing Canonical Representations - The MultiView Architecture, Proceedings of International Workshop on Multiple Perspectives in Software Development (Viewpoints'96), Association for Computing Machinery, New York, pp222-226

Marlin, C., Peuschel, B., McCarthy, M.J. and Harvey, J.G. (1993) MultiView-Merlin: An Experiment in Tool Integration, Proceedings 1993 Software Engineering Environments Conference, Reading, England, pp35-48

Martin, J. and Odell, J.J. (1995) Object-Oriented Methods: A Foundation, Prentice-Hall, Englewood Cliffs, New Jersey

Martin, J. and Odell, J.J. (1993) Principles of Object Oriented Analysis and Design, Prentice-Hall, Englewood Cliffs, New Jersey

Marttiin, P. (1994) Towards Flexible Process Support with a CASE Shell, Advanced Information Systems Engineering, Proceedings of the 6th international conference CaiSE'94, Springer-Verlag, pp14-27

Marttiin, P., Rossi, M., Tahvanainen, V. and Lyytinen, K. (1993) A Comparative Review of CASE Shells: A Preliminary Framework and Research Outcomes, Information and Management, Elsevier Science Publishers B.V, pp11-31

Mathiassen, L. and Sørensen, C. (1995) The Why, What, Who, Where, and How of CASE Management, Proceedings of the Information Systems Research Seminar in Scandinavia, http://iris.informatik.gu.se/conference/

McWirter, J. (1998) Escalante homepage, http://www.cs.colorado.edu/~jeffm/research/escalante/escalante.html

McWhirter, J.D. and Nutt, G.J. (1994) Escalante: An Environment for the Rapid Construction of Visual Language Applications, IEEE/CS Symposium on Visual Languages (VL'94), IEEE Computer Society Press, pp15-22

MDC (1997) Metadata Interchange Specification (MDIS) Version 1.1

MDC (1998) Meta Data Coalition homepage, http://www.he.net/~metadata/

Mehandjiska D., Page, D. and Clark P. (1994) An Intelligent Object Oriented CASE Tool, Proceedings of 1st International Conference on Object-Oriented Information Systems, Eds. Patel, D., Sun, Y. and Patel, S., Springer-Verlag, London, pp168-172

Mehandjiska, D., Apperley, M.D., Phillips, C., Page, D. and Clark, P. (1995a) A Methodology Independent Object Oriented CASE Tool, New Zealand Journal of Computing, Vol. 6 (1A), New Zealand Computer Society Conference, Wellington, pp95-105

Mehandjiska, D., Page, D. and Ham, J. (1995b) Template Generator for Methodology Independent Object Oriented CASE Tool, Proceedings of 2nd International Conference on Object-Oriented Information Systems, Eds. Murphy, J. and Stone, B., Springer-Verlag, Dublin, Ireland, pp431-440

Mehandjiska, D., Apperley, M.D., Phillips, C.H.E., Dasari, S. and Page, D. (1996a) Advancing Information Technologies Through CASE, Proceedings of the 19th Australasian Computer Science Conference (ACSC'96), Ed. Ramamohanarao, K., Melbourne, Australia, pp213-222

Mehandjiska, D., Page, D. and Choi, M.D. (1996b) Meta-Modelling and Methodology Support in Object-Oriented CASE Tools, Proceedings of 1996 International Conference on Object-Oriented Information Systems, Eds. Patel, D., Sun, Y., Patel, S., Springer-Verlag, London, pp370-386

Mehandjiska, D., Page, D. and Dasari, S. (1996c) Generic Knowledge Base for a Methodology Independent Object-Oriented CASE Tool, Proceedings of the IASTED International Conference on Artificial Intelligence, Expert Systems and Neural Networks, Ed. Hamza, M., IASTED/Acta Press, Honolulu, Hawaii, pp23-26

Mehandjiska, D., Page, D., Griffin, D. and Usherwood, L. (1997) Methodology Knowledge Representation and Interpretation for a Methodology Independent OO CASE Tool, Proceedings of IASTED International Cenference on Software Engineering (SE'97), ACTA Press, San Francisco, USA, pp243-247

MerridanMarketing (1998) MetaEdit+: Linking Software Development To Business Modelling, http://www.meridian-marketing.com/METAEDIT/index.html

MetaCase Consulting (1996a) Developing New Methods with the MetaEdit Personal Environment, White Paper. MetaCase Document No. PEWP-1.0

MetaCase Consulting (1996b) MetaEdit+: A Fully Configurable Multi-User and Multi-Tool Case and CAME Environment, White Paper

MetaCASE Consulting (1998) MetaEdit+ homepage, http://www.jsp.fi/metacase/

MetaModel.com (1998) MetaModel.com homepage, http://www.metamodel.com/

MetaView (1998) MetaView project homepage, http://web.cs.ualberta.ca/~softeng/Metaview/project.shtml

Meyer, B. (1995) Object Success: A Managers Guide to Object Orientation, its Impact in the Corporation and its Use in Reengineering the Software Process, Prentice-Hall, London

Meyer, B. (1997) Object-Oriented Software Construction – 2nd Edition, Prentice-Hall, New Jersey

Minas, M. and Viehstaedt, G. (1995) DiaGen: A Generator for Diagram Editors Providing Direct Manipulation and Execution of Diagrams, Proceedings 11th International IEEE Symposium on Visual Languages, IEEE Computer Society Press, Los Alamitos, California, pp203-210

mip GmbH (1998a) Alfabet frequently asked questions mip GmbH & Co. Berlin

mip GmbH (1998b) Alfabet Technology Overview mip GmbH & Co. Berlin

mip GmbH (1998c) Alfabet User Manual mip GmbH & Co. Berlin

mip GmbH (1998d) Introduction to the Alfabet-Technology mip GmbH & Co. Berlin

Mirriam-Webster (1998) WWWebster Dictionary, http://www.m-w.com/netdict.htm

Misra, S. K. (1990) Analysing CASE System Characteristics: Evaluative Framework, Butterworth-Heinemann Ltd, Vol. 32 (6)

Monarchi, D.E. and Puhr, G.I. (1992) A Research Typology for Object-Oriented Analysis and Design, Communications of the ACM, Vol. 35 (9), pp35-47

Montgomery, J. (1997) Distributing Components, BYTE, http://www.byte.com, April

Mosely, V. (1992) How to Assess Tools Efficiently and Quantitatively, IEEE Software, Vol. 9 (3), pp160-163

Mugridge, W.B., Hosking, J.G. and Grundy, J.C. (1998) Vixels, CreateThroughs, DragThroughs and AttachmentRegions in BuildByWire, Proceedings Australasian Computer Human Iteraction Conference, OZCHI'98, IEEE Computer Society Press, Adelaide, South Australia, pp320-327

Muller, P. (1997) Instant UML, Wox Press Ltd., Olton, Birmingham

MultiView (1998) MultiView project home page, http://see.cs.flinders.edu.au/Projects/MultiView/

Myers, B.A., McDaniela, R., Miller, R., Ferrency, A., Faulring, A., Borinson, E., Kyle, B., Mickish, A., Klimovitski, A. and Doane, P. (1997) The Amulet Environment: New Models for Effective User Interface Software Development, IEEE Transactions on Software Engineering, Vol. 23 (6), pp347-365

Nilsson, E. G. (1990) CASE Tools and Software Factories, Advanced Information Systems Engineering, CAiSE'90, Eds. Goos, G. and Hartmanis, J., Springer-Verlag, Berlin, pp42-47

Nuttals (1902) Nuttals Standard Dictionary, Frederick Warne and Company Publications, Chandos House, Bedford St, Strand, London.

Oakes, K.S., Smith, D. and Morris, E. (1992) Guide to CASE Adoption Technical Report CMU/SEI-92-TR-15 ESC-TR-92-015

Object Agency (1998) A Comparison of Object-Oriented Development Methodologies, http://www.toa.com/pub/html/mcr.html

Odell, J. (1995) Meta-Modelling, position paper for Meta-modelling in OO OOPSLA'95 Workshop

OMG (1991) The Common Object Request Broker: Architecture and Specification, OMG Document Number 91.12.1, Revision 1.1, Framingham, USA

OMG (1992) Object Management Architecture Guide, OMG TC Document 92.11.1, Revision 2.0, Framingham, USA

OMG (1997a) Meta Object Facility (MOF) Specification OMG Document ad/97-08-14

OMG (1997b) Meta Object Facility Appendices OMG Document ad/97-08-15

OMG (1997c) OA&D CORBA facility v1.1, Rational and Partners Submission to the OMG OA&D facility, ad97-08-09

OMG (1997d) Object Constraint Language Specification v1.1, Rational and Partners Submission to the OMG OA&D facility, ad97-08-08

OMG (1997e) UML Extension for Business Modelling v1.1, Rational and Partners Submission to the OMG OA&D facility, ad97-08-07

OMG (1997f) UML Extension for Objectory Process for Software Engineering v1.1, Rational and Partners Submission to the OMG OA&D facility, ad97-08-06

OMG (1997g) UML Notation Guide v1.1, Rational and Partners Submission to the OMG OA&D facility, ad97-08-05

OMG (1997h) UML Proposal Summary v1.1, Rational and Partners Submission to the OMG OA&D facility, ad97-08-02

OMG (1997i) UML Semantics and appendices v1.1, Rational and Partners Submission to the OMG OA&D facility, ad97-08-04

OMG (1997j) UML Summary v1.1, Rational and Partners Submission to the OMG OA&D facility, ad97-08-03

OPEN (1996) Proposing an Open Standard, Object Expert, 2(1), pp14-15

OPEN (1998) OPEN homepage, http://www.csse.swin.edu.au/cotar/OPEN/

Orfali, R., Harkey, D. and Edwards, J. (1996) The Essential Distributed Objects Survival Guide, John Wiley & Sons Inc., New York, USA

Ovum (1996) Ovum Evaluates: CASE Products, http://www.ovum.com

Oxford (1990) Consise Oxford Dictionary – 8th Edition, Oxford University Press, Walton Street, Oxford, England

Page, D., Clark, P. and Mehandjiska, D. (1994) An Abstract Definition of Graphical Notations for Object-Orientated Information Systems, Proceedings of 1st International Conference on Object-Oriented Information Systems (OOIS'94), Eds. Patel, D., Sun, Y. and Patel, S., Springer-Verlag, London, pp266-276

Page, D., Griffin, D., Usherwood, L. and Mehandjiska, D. (1997) Implementation of a Semantic Specification Language Interpreter for a Methodology Independent OO CASE Tool, Proceedings of IASTED International Cenference on Software Engineering (SE'97), ACTA Press, San Francisco, USA, November 2-4, pp239-242

Page, D., Mehandjiska, D. and Phillips, C.H.E. (1998) Methodology Independent OO CASE: Supporting Methodology Engineering, Proceedings of Software Engineering: Education and Practise (SE:E&P'98), IEEE Computer Society Press, Dunedin, New Zealand, pp373-380

Papahristos, S. and Gray, W. A. (1991) Federated CASE Environment, Advanced Information Systems Engineering, CAiSE'91, Eds. Goos, G. and Hartmanis, J., Springer-Verlag, Berlin, pp461-478

Parr, T.J. (1997) Language Translation Using PCCTS & C++, Automata Publishing Company

PCCTS (1998) PCCTS and ANTLR home page, http://www.antlr.org/

Phillips, C.H.E., Mehandjiska, D. and Page, D. (1998a) The Usability Component of a New Framework for the Evaluation of Object-Oriented CASE Tools, Proceedings of Software Engineering: Education and Practise (SE:E&P'98), IEEE Computer Society Press, Dunedin, New Zealand, pp131-141

Phillips, C.H.E., Adams, S., Page, D. and Mehandjiska, D. (1998b) The Design of the Client User Interface for a Meta Object-Oriented CASE Tool, Proceedings of TOOLS Pacific'98, Monash Printing Services, Victoria, pp145-157

Phillips, C.H.E., Adams, S., Page D. and Mehandjiska, D. (1998c) Design of the User Interface for a Methodology Independent OO CASE Tool, Proceedings of OZCHI'98, IEEE Computer Society Press, Los Alamitos, California, pp106-114

Platinum (1998) Pardigm+ homepage, http://www.platinum.com/products/appdev/pplus_ps.htm

Popkin Software (1998) System Architect CASE tool, http://www.popkin.com/

Pree, W. (1994) Design Patterns for Object-Oriented Development, Addison-Wesley, Reading, Massachusetts

Pressman, R.S. (1997) Software Engineering: A Practitioners Approach, 4th Edition, McGraw-Hill Companies Inc., New York

Purchase, H.C. (1998) The Effects of Graph Layout, Proceedings Australasian Computer Human Iteraction Conference, OZCHI'98, IEEE Computer Society Press, Adelaide, South Australia, pp80-86

QED (1992) Object Management Group, *The Common Object Request Broker: Architecture and Specification*, distributed by QED Publishing Group, Wesley, MA

Quantrani, T. (1997) Visual Modelling with Rational Rose and UML, Addison-Wesley, Reading, Massachusetts

Rational (1997a) UML Notation Guide, version 1.0 (unpublished)

Rational (1997b) UML Semantics Guide v1.0 (unpublished)

Rational Software (1998) Rational Software homepage, http://www.rational.com/

Read, M.C. and Marlin, C.D. (1996) Generating Direct Manipulation Program Editors within the MultiView Programming Environment, Proceedings International Workshop on Multiple Perspectives in Software Development (Viewpoints'96), Association for Computing Machinery, New York, pp232-236

Read, M.C. and Marlin, C.D. (1998) Specifying and Generating Program Editors with Novel Visual Editing Mechanisms, Tenth International Conference on Software Engineering and Knowledge Engineering, San Francisco, California, pp418-425

Readers Digest (1988) The Readers Digest Dictionary Universal Dictionary, Readers Digest Association Ltd, London

Reenskaug, T., Wold, P. and Lehne, O.A. (1996) Working with Objects. The OOram Software Engineering Manual, Manning, Greenwich, CT, USA

Robbins, J.E., Hilbert, D.M. and Redmiles, D.F. (1996) Using Critics to Analyse Evolving Architectures, Proceedings Second International Software Architecture Workshop (ISAW-2), SigSoft'96, pp90-93

Robbins, J.E., Hilbert, D.M. and Redmiles, D.F. (1997) Argo: A Design Environment for Evolving Software Architectures, Proceedings of 19th International Conference on Software Engineering, ICSE97, Springer, pp600-601

Robbins, J.E., Hilbert, D.M. and Redmiles, D.F. (1998) Software Architecture Critics in Argo Proceedings of the 1998 Conference on Intelligent User Interfaces, Adaptation and Critiquing, pp141-144

Rossi, M., Gustafsson, M., Smolander, K., Johansson, L. and Lyytinen, K. (1992) Meta-Modelling Editors as a Front End Tool for a CASE Shell, CAiSE'92, Ed. Loucopoulos, P., Springer-Verlag, Berlin

Rumbaugh, J, Blaha. M., Premerlani, W., Eddy, F. and Lorensen, W. (1991) Object Oriented Modelling and Design, Prentice-Hall, Englewood Cliffs, New Jersey

Rumbaugh, J. (1995a) OMT: The Functional Model, Journal of Object Oriented Programming, Vol. 8. (1), pp95

Rumbaugh, J. (1995b) OMT: The Object Model, Journal of Object Oriented Programming, Vol. 7. (8), pp21

Rumbaugh, J., Jacobson, I. and Booch, G. (1999) The Unified Modelling Language Reference Manual, Addison-Wesley, Reading, Massachusetts

Sahraoui, H., Missaoui, R., and Gagnon, J. (1995) Using a Meta-Modelling Approach for Building an Object-Oriented Database Modelling and Design Tool, position paper for Meta-modelling in OO OOPSLA'95 Workshop

Schach, S.R. (1993) Software Engineering – 2nd Edition, Richard D Irwin Inc. and Aksen Associates Inc., Boston

Schach, S.R. (1997) Software Engineering with Java, Richard D Irwin Inc., Boston

Schmidt, R. and Assmann, U. (1998) Concepts for Developing Component-Based Systems, ICSE International Workshop on Component-Based Software Engineering, Kyoto, Japan

Schottland, G. (1996) Successful Design Tool Selection and Deployment, Object Magazine, November

Scott, L. (1998) MetaCASE Concept Document, Macquarie University Joint Research Centre for Advanced Systems Engineering , Sydney, Australia ftp://ftp.mpce.mq.edu.au/pub/jrcase/metaCASE/concept.ps.Z

Seacord, R.C., Hissam, S.A. and Wallnau, K.C. (1998) Agora: A Search Engine for Software Components, Software Engineering Institute Technical Report CMU/SEI-98-TR-011, Carnegie Mellon University

SEI (1998) Software Engineering Institute, http://www.sei.cmu.edu/sei-home.html

Senn, J. A. (1990) Information Systems in Management – 4th Edition, Tools and Methods for Developing Information Systems, Wadsworth Publishing Co., Belmont, California, pp727-736

Sharble, R.C. and Cohen, S. S. (1993) The Object Oriented Brewery: A Comparison of Two Object Oriented Development Methods, Software Engineering Notes, Vol. 18 (2), pp60-73

Shlaer, S. and Mellor, S. J. (1988) Object Oriented Systems Analysis: Modelling the World in Data, Yourdon Press, Englewood Cliffs

Shlaer, S. and Mellor, S. J. (1991) Object Lifecycles: Modelling the World in States, Yourdon Press, Prentice-Hall, Englewood Cliffs, New Jersey

Short, K. (1997) Component Based Development And Object Modelling, Version 1.0, Sterling Software, February, Sterling Software, Tennyson Parkway, Plano, Texas

Siegel, J., Mirsky, H., Hudli, R., de Jong, P., Thomas, A., Coles, W., Baker, S. and Balick, M. (1996) Corba: Fundamentals and Programming, John Wiley & Sons Inc., New York, USA

Sigfried, S. (1996) Understanding Object Oriented Software Engineering, IEEE Computer Society Press, Los Alamitos

Smith, R. and Anderson, P. (1996) Relating Distortion to Performance in Distortion Oriented Displays, Proceedings of OzCHI'96, Hamilton, New Zealand, pp6-11

Smolander, K, Lyytinen, K., Tahvanainen, V. P. and Marttiin, P. (1991) MetaEdit: A Flexible Graphical Environment for Methodology Modelling, Proceedings of Advanced Information Systems Engineering CAiSE'91, Eds. G. Goos and J. Hartmanis, Springer-Verlag, Berlin, pp168-193

Soley, R.S. (1996) The World Wide Web and Distributed Computing: A Natural Match?, Java Developer's Journal, Vol. 1, Issue 2

Sommerville, I. (1996) Software Engineering, Addison-Wesley , Reading, Massachusetts

Sorensen, C. (1993) What Influences Regular CASE Use In Organisations? An Empirically Based Model, Scandinavian Journal of Information Systems, Vol. 5, pp25-50

Sorenson, P. G. (1988) First Generation CASE Tools: All Form but Little Substance, Research Report, Dept Computational Science, University of Saskatchewan, Saskatoon, Canada, pp1-8

Sorenson, P. G., Tremblay, J. and McAllister, A. J. (1988) The MetaView System for Many Specification Environments, IEEE Software, Vol. 5 (2), pp30-38

Sterling (1998) TeamWork CASE tool, http://www.cool.sterling.com/products/

STP (1998) Software though Pictures homepage, http://www.ide.com/Products/SMS/sms.html

Sumner, M. (1992) The Impact of Computer-Assisted Software Engineering on Systems Development, IFIP Transactions - The Impact of Computer Supported Technologies on Information Systems Development, Eds. Kendall, K.E, Lyytinen, K. and DeGross J. I., Elsevier Science Publishers, Amsterdam

SUN (1998) Sun Microsystems Java homepage, http://java.sun.com/

Taskon A/S (1997) The OOram Meta-Model: combining role models, interfaces, and classes to support system centric and program centric Modelling. Version 1.0 A proposal in response to OMG OA&D RFP-1 8

Taylor, D.A. (1998) Object Technology: A Managers Guide, 2nd Edition, Addison-Wesley, Reading, Massachusetts

Tolvanen, J. and Lyytinen, K. (1993) Flexible Method Adaptation in CASE, Scandinavian Journal of Information Systems, Vol. 5, pp51-77

UML-RTF (1998) UML Revision Task Force homepage, http://uml.shl.com/

van den Goor, G., Hong, S. and Brinkkemper, S. (1992) A Comparison of Six Object Oriented Analysis and Design Methods, Report Centre of Telematics and Information Technology, University of Twente, the Netherlands and Computer Information Systems Department, Georgia State University, Atlanta, USA

Verhoef, T.G, Hofstede, A.H.M. and Wijers, G.M. (1991) Structuring Modelling Knowledge for CASE Shells, Advanced Information Systems Engineering, Proceedings CAiSE'91, Eds. Goos, G. and Hartmanis, J., Springer-Verlag, Berlin, pp502-524

Vessey, I., Jarvenpaa, S. and Tractinsky, N. (1992). Evaluation of Vendor Products: CASE Tools as Methodology Companions, Communications of the ACM, Vol. 35(4), pp90-105

Walden, K. and Nerson, J. (1995) Seamless Object-Oriented Software Architecture : Analysis and Design of Reliable Systems, Prentice-Hall, New York

Wallnau, K.C. (1992) Issues and Techniques of CASE Integration with Configuration Management, Software Engineering Institute Technical Report CMU/SEI-92-TR-5, Carnegie Mellon University

Wallnau, K.C. and Feiler P.H. (1991) Tool Integration and Environment Architectures, Software Engineering Institute Technical Report CMU/SEI-91-TR-11, Carnegie Mellon University

Warmer, J. and Kleppe, A. (1999) The Object Constraint Language: Precise modelling in UML, Addison-Wesley , Reading, Massachusetts

Warwick, B., Mugridge, B., Hosking, J.G. and Grundy, J.C. (1996) Towards a Constructor Kit for Visual Notations, Proceedings 1996 Australasian Computer Human Iteraction Conference, OZCHI'96, IEEE Computer Society Press, Hamilton, New Zealand, pp169-176

Webster, B.F. (1995) Pitfalls of Object-Oriented Development, M&T Books, New York

Weiderman, N., Northrop, L., Smith, D., Tilley, S. and Wallnau, K. (1997) Implications of Distributed Object Technology for Reengineering, Software Engineering Institute Technical Report CMU/SEI-97-TR-005, Carnegie Mellon University

Whitten, J.L., Bentley, L.D. and Barlow M. (1994) Systems Analysis and Design Methods – Third Edition, Irwin Inc, Sydney, Australia

Wills, A. and D'Souza, D. (1997) Rigorous Component-Based Development, Trireme Object Technology & ICON Computing

Wirfs-Brock, R.J. and Johnson, R.E. (1990) Surveying Current Research in Object-Oriented Design, The Communications of ACM, Vol. 33(9), pp104-1124

Wirfs-Brock, R., Wilkerson, B. and Wiener, L. (1990) Designing Object-Oriented Software, Prentice-Hall

Younessi, H. and Henderson-Sellers, B. (1998) Cooking Up Improved Software Quality, Object Magazine, Vol. 7 (8), pp14-15

Yourdon, E. and Argila, C. (1996) CASE Studies in Object-Oriented Analysis and Design, Prentice-Hall

Yu, H. (1999) A Web-Based Interface for a Methodology Independent Object-Orientated CASE Tool, Massey University Masters Thesis (to be submitted), Department of Computer Science, Massey University, Palmerston North, New Zealand

Zarrella, P.F. (1990) CASE Tool Integration and Standardisation, Software Engineering Institute Technical Report CMU/SEI-90-TR-14, Carnegie Mellon University

Zarrella, P.F., Smith, D.B. and Morris, E.J. (1991) Issues in Tool Acquisition, Software Engineering Institute Technical Report CMU/SEI-91-TR-8, Carnegie Mellon University

Zhuang, Y. (1994) Object-Oriented Modelling in MetaView, University of Alberta Masterate thesis, Department of Computer Science, University of Alberta, Canada

Zhuang, Y. Findeisen, P. and Sorenson, P. (1995) Object-Oriented Modelling in MetaView, Department of Computer Science, University of Alberta

Errata Sheet

Spelling Errors and Clarifications

page 6, Fig 1.2	The arrow with the 'Meta' label only points from the 'modelling process' on the left to the 'modelling process' on the right
page 8, line 16	'Some of the most prominent' should read 'Some prominent'
page 14, line2	'Existing CASE tools' should read 'Several existing CASE tools'
page 29, line 19	'CASE Date' should read 'CASE Data'
page 36, line 19	'tool configuration' should read 'tool configurations'
page 48, line 7	'Primary' should read 'Primarily'
page 64, Fig 3.3	Directed arrows correspond to inheritance relations
page 75, footnote	Warwick *et al.* 1996 should read Mugridge *et al.* 1996
page 87, Fig 4.4	'hieght' (bottom left of figure) should read 'height'
page 93, line 13	'to separate' should read 'to a separate'
page 124, Fig 5.1	'persistance' (bottom right of figure) should read 'persistence'
page 128, line 17	*'currnt_model'* ahould read *'current_model'*
page 129	The memory management scheme has reference loop detection
page 137, Fig 5.6	*listiterator* and *listIterator* are the same as SSL is not case sensitive
page 210, section 8.3	*Instantiate* is overridden to achieve different behaviour for pattern instantiation (see footnote page on page 211)
page 213	Consistency between models is achieved by the meta-modelling approach supported by MOOT
page 313	Reference Warwick, B., Mugridge, B., Hosking, J.G. and Grundy, J.C. (1996) should read Mugridge, W.B., Hosking, J.G. and Grundy, J.C. (1996)

Systems Relating to Graphical Notations

Additional information about some of the systems noted in chapter 4 – Notation Definition Language, is given below.

DiaGen (Minas and Viehstaedt, 1995)

This system is used to generate a bespoke editor for a particular graphical notation. Notations are defined using hypergraphs, hypergraph grammars and layout constraints. The role of NDL in the MOOT system is similar to role of the hypergraphs, hypergraph grammars and layout constraints in the DiaGen system.

BuildByWire (Mudgridge et al., 1996, 1998)

BuildByWire generates a collection of JavaBean components that correspond to a notation, as well as an editor JavaBean component. BuildByWire is similar to the MOOT Notation Editor, which generates NDL descriptions of notations that are subsequently interpreted by the CASE tool client.

Amulet (Amulet, 1998; Myers et al., 1997)

Amulet is a User Interface development toolkit. It does not have a separate language for describing visual notations, but provides an extendable hierarchy of widget classes. The coupling between application and notation presentation logic is much higher in Amulet than between NDL and SSL.

Escalante (McWhirter, 1998; McWhirter and Nutt, 1994)

Escalante is an environment for specifying and generating applications for graph based visual languages. A language (and its notation) is described via elements of a type hierarchy that provide the base classes for the various components of graph based languages. These include structural and visual language constructs (eg. nodes and edges) and a set of structured graphics objects. Escalante is used to build bespoke visual language systems. The coupling between application and notation presentation logic is much higher in Escalante than between NDL and SSL.