



A generic model for representing software development methods.

WONG, Alan C.Y.

Available from the Sheffield Hallam University Research Archive (SHURA) at:

<http://shura.shu.ac.uk/20559/>

A Sheffield Hallam University thesis

This thesis is protected by copyright which belongs to the author.

The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the author.

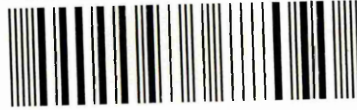
When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given.

Please visit <http://shura.shu.ac.uk/20559/> and <http://shura.shu.ac.uk/information.html> for further details about copyright and re-use permissions.

CITY CAMPUS POND STREET
SHEFFIELD S1 1WB

367370

101 522 987 5



Fines are charged at 50p per hour

24 APR 2002

4.12 PM

ProQuest Number: 10701206

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.



ProQuest 10701206

Published by ProQuest LLC (2017). Copyright of the Dissertation is held by the Author.

All rights reserved.

This work is protected against unauthorized copying under Title 17, United States Code
Microform Edition © ProQuest LLC.

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

A Generic Model for Representing Software Development Methods

Alan C.Y. WONG

A thesis submitted in partial fulfilment of the requirements of
Sheffield Hallam University
for the degree of Doctor of Philosophy

May 1996

ABSTRACT

This thesis has adopted the premise that the use of a method offers a valuable contribution to the software development process. Many methods have not been adequately defined. This thesis is based on the hypothesis that it is possible to represent software development methods using a Generic Method Representation (GMR). This GMR includes the three basic components of the method, which are the product model, the process model and the heuristic model. The elements and interrelationships of these models are investigated. In addition to a graphical representation, a method specification language (MSL) is derived, to enhance the expressive and executable power of GMR. A three-stage knowledge acquisition model, known as IFV (inspection, fabrication and verification), is also introduced to elicit method semantics from the available acquisition media. Moreover, the key benefits of meta modelling, such as method comparison, fragment dissection, method evaluation and selection (or customisation) of a method, are highlighted. An application of GMR, that is the mapping to a practical metaCASE tool model, is also illustrated comprehensively to demonstrate the applicability of the approach.

ACKNOWLEDGMENTS

I wish to thank the many individuals who have made this research possible. I want to thank System Applied Technology Ltd. for initiating the project and in giving me the opportunity to develop the ideas by working on object-oriented methods. I especially wish to thank my colleagues at Sheffield Hallam University for their support and facilities in writing this thesis. I acknowledge the important contribution of Kevin Conheeney, who participated in the original development of this research and worked with me in exploring this interesting field. I also wish to thank Frank Poole for his continuous encouragement throughout the research.

Many individuals helped in the review of the thesis, in particular I wish to thank Matthew Love, Paul Woods and Becca Doyle for their thorough reviews and perceptive comments.

Finally, I wish to thank my family for their unceasing patience and encouragement.

TABLE OF CONTENTS

	PAGE
ABSTRACT	i
ACKNOWLEDGEMENT	ii
TABLE OF CONTENTS	iii
TABLE OF FIGURES	xii
TABLE OF TABLES	xvii
1. INTRODUCTION	1.1
1.1 INTRODUCTION	1.1
1.1.1 Background	1.1
1.1.2 History of Software Engineering	1.2
1.1.3 Managing Complexity of Software Engineering	1.3
1.2 THE REAL PROBLEMS	1.4
1.2.1 Panacea Syndrome	1.4
1.2.2 Looking Under the Lamppost Syndrome	1.5
1.2.3 Three Blind Men Touching an Elephant Syndrome	1.6
1.3 THE PROPOSED SOLUTION	1.7
1.4 EPISTEMOLOGICAL HIERARCHY OF KNOWLEDGE	1.8
1.5 ROAD MAP	1.10
1.6 CONCLUSION	1.12
2. INVESTIGATION OF SOFTWARE DEVELOPMENT METHODS	2.1
2.1 INTRODUCTION	2.1
2.2 STRUCTURED METHODS	2.3
2.2.1 DeMarco Structured Analysis (DeMarco SA)	2.3
2.2.2 Jackson Structured Design (JSD)	2.4
2.2.3 Yourdon Modern Structured Analysis (Yourdon)	2.5
2.2.4 Structured Systems Analysis & Design Method	2.6

2.3 OBJECT-ORIENTED METHODS	2.7
2.3.1 Object-Oriented Structured Design (OOSD)	2.8
2.3.2 Object-Oriented System Analysis (OOSA)	2.9
2.3.3 Object-Oriented Analysis/Design (OOA/OOD)	2.10
2.3.4 Nielsen Object-Oriented Design (Nielsen OOD)	2.11
2.3.5 Object-Oriented Software Engineering (OOSE)	2.12
2.4 CHOSEN METHODS	2.13
2.4.1 Booch Object-Oriented Design (Booch OOD)	2.14
2.4.2 Hierarchical Object-Oriented Design (HOOD)	2.16
2.4.3 Object Modelling Technique (OMT)	2.18
2.4.4 Ptech	2.20
2.4.5 Codarts/DA	2.22
2.5 OTHER METHODS	2.25
2.5.1 Class, Responsibility and Collaboration (CRC)	2.25
2.5.2 ASTS Development Method 3 (ADM3)	2.26
2.5.3 The Fusion Method (Fusion)	2.27
2.5.4 The KADS Method	2.29
2.6 SUMMARY OF INVESTIGATION	2.31
2.7 CONCLUSION	2.32
 3. INVESTIGATION OF METHOD INTEGRATION, META MODELLING RESEARCH AND METACASE TOOLS	 3.1
3.1 INTRODUCTION	3.1
3.2 METHOD INTEGRATION	3.2
3.2.1 CASE Data Interchange Format (CDIF)	3.2
3.2.2 Portable Common Tool Environment (PCTE)	3.3
3.3 META MODELLING RESEARCHES	3.5
3.3.1 ALF-MASP	3.5
3.3.2 SOCRATES Project	3.9
3.3.3 MethodBase	3.12
3.4 METACASE TOOLS	3.16
3.4.1 ObjectMaker	3.16
3.4.2 MetaEdit	3.18
3.4.3 IPSYS ToolBuilder	3.21

3.5 SUMMARY OF INVESTIGATION	3.23
3.6 CONCLUSION	3.24
4. SEMANTIC KNOWLEDGE BASE	4.1
4.1 INTRODUCTION	4.1
4.2 EVOLUTION APPROACH	4.2
4.2.1 Traditional Approach	4.2
4.2.2 Prototyping Approach	4.3
4.2.3 Method Engineering Approach	4.4
4.2.3.1 Requirement Specification Prototyping	4.5
4.2.3.2 Methodology Prototyping	4.6
4.2.3.3 Software Development Prototyping	4.7
4.3 THREE LAYERED MODEL	4.7
4.4 SINGLE METHODOLOGY MODEL	4.9
4.5 KNOWLEDGE BASE REPRESENTATION	4.11
4.6 METHODOLOGY PROTOTYPING	4.12
4.6.1 Method Controller	4.13
4.6.2 Canonical Concept Dictionary	4.14
4.6.3 System References	4.15
4.7 CONCLUSION	4.15
5. PRODUCT MODEL	5.1
5.1 INTRODUCTION	5.1
5.2 AIMS & OBJECTIVES	5.2
5.3 CONCEPT MODELLING	5.3
5.3.1 Concept Types	5.3
5.3.1.1 Entity Concept	5.4
5.3.1.2 Fragment Concept	5.4
5.3.1.3 Property Concept	5.5
5.3.1.4 Link Concept	5.5
5.3.1.5 Group Concept	5.5
5.3.2 Concept Relationship Properties	5.6
5.3.2.1 Cardinality	5.6
5.3.2.2 Optionality	5.7
5.3.2.3 Directional	5.7

5.3.2.4 Role	5.7
5.3.2.5 Overlapping and Completeness Features	5.8
5.3.2.6 Constraint Rule	5.9
5.3.3 Concept Relationships	5.11
5.3.3.1 Subtyping	5.11
5.3.3.2 Composition	5.12
5.3.3.3 Linking	5.13
5.3.3.4 Grouping	5.15
5.3.3.5 Referencing	5.17
5.4 ADVANCED CONCEPT MODELLING	5.19
5.4.1 Aggregation vs Decomposition vs Referencing	5.19
5.4.2 Composition vs Referencing	5.20
5.4.3 Merging Source and Target Links	5.21
5.4.4 Overriding	5.21
5.4.5 Inclusion and Exclusion	5.22
5.5 PROCESS MODEL OF PRODUCT MODEL	5.24
5.6 ZOOM HIERARCHY	5.29
5.6.1 Zoom by Detail	5.29
5.6.2 Zoom by Feature	5.30
5.7 PRODUCT MODEL OF CONCEPT DIAGRAM	5.30
5.8 CONCLUSION	5.32
 6. PROCESS MODEL	 6.1
6.1 INTRODUCTION	6.1
6.2 META MODELLING PROCESSES	6.2
6.2.1 Method Process	6.3
6.2.2 MetaCASE Process	6.4
6.2.3 CASE Tool Process	6.5
6.3 BASIC ISSUES	6.6
6.4 THREE PRELIMINARY APPROACHES	6.7
6.4.1 Menu Driven Approach	6.7
6.4.2 Event Sequence Approach	6.8
6.4.3 Frame Based Navigation Approach	6.9
6.5 TASK FUNCTIONS	6.10

6.5.1 perform Function	6.11
6.5.2 do Function	6.11
6.5.3 draw Function	6.12
6.5.4 insert Function	6.12
6.5.5 delete Function	6.12
6.5.6 modify Function	6.12
6.5.7 adjust Function	6.13
6.5.8 retype Function	6.13
6.5.9 specify Function	6.13
6.6 TASK SEQUENCE	6.14
6.7 TASK MODELLING	6.16
6.7.1 A Task	6.16
6.7.2 Task Trigger and Concept Flow	6.17
6.7.3 Task Diagram	6.19
6.7.4 Task Decomposition	6.19
6.7.5 Task Refinement	6.20
6.7.6 Parallel Tasks	6.21
6.8 META PROCESS MODEL	6.22
6.9 META META MODEL	6.23
6.10 CONCLUSION	6.26
7. HEURISTIC MODEL	7.1
7.1 INTRODUCTION	7.1
7.2 METHOD HEURISTIC	7.2
7.2.1 Concept Heuristic	7.2
7.2.2 Task Heuristic	7.3
7.3 HEURISTIC TEXT	7.4
7.4 HEURISTIC RULE	7.5
7.5 HEURISTIC LINK	7.6
7.5.1 Composition Link	7.7
7.5.2 Reference Link	7.8
7.6 MAPPING TO THE TWO MODELS	7.9
7.7 CONCLUSION	7.10
8. METHOD REPRESENTATION	8.1

8.1 INTRODUCTION	8.1
8.2 OVERVIEW OF GMR REPRESENTATION	8.2
8.3 ADDITIONAL CONSIDERATIONS	8.3
8.3.1 Dissection Set	8.3
8.3.2 Shared Concept	8.5
8.3.3 Other Relationships	8.7
8.3.3.1 Derived Relationship	8.7
8.3.3.2 Multiple Inheritance	8.8
8.3.3.3 Delegation	8.9
8.3.4 Text Fragment	8.9
8.4 METHOD SPECIFICATION LANGUAGE	8.11
8.5 PROLOG CLAUSE FORMAT	8.14
8.6 CONCLUSION	8.16
 9. METHOD EVALUATION	 9.1
9.1 INTRODUCTION	9.1
9.2 FRAGMENT DISSECTION	9.1
9.2.1 Product Dissection	9.2
9.2.2 Process Dissection	9.5
9.3 METHOD COMPARISON	9.9
9.3.1 Numerical Comparison	9.10
9.3.2 Fragment Comparison	9.13
9.4 SELECTION OF METHOD	9.15
9.5 CONCLUSION	9.16
 10. KNOWLEDGE ACQUISITION OF METHOD MODELS	 10.1
10.1 INTRODUCTION	10.1
10.2 METHOD KNOWLEDGE ACQUISITION	10.2
10.2.1 Knowledge Acquisition Problems	10.2
10.2.2 Knowledge Acquisition Techniques	10.3
10.3 KNOWLEDGE ACQUISITION MEDIA	10.4
10.3.1 Advantages of Method Acquisition Media	10.6
10.3.2 Disadvantages of Method Acquisition Media	10.6
10.4 ELICITATION OF METHOD KNOWLEDGE	10.7
10.4.1 Method Knowledge Inspection	10.8

10.4.1.1 Cursory Reading	10.9
10.4.1.2 Chronic Reading	10.11
10.4.1.3 Conclusive Reading	10.12
10.4.2 Method Knowledge Fabrication	10.13
10.4.3 Method Knowledge Verification	10.14
10.4.3.1 Correctness	10.15
10.4.3.2 Completeness	10.15
10.4.3.3 Contradiction	10.16
10.4.3.4 Consistency	10.16
10.4.3.5 Contrast	10.16
10.5 METHOD MODEL ELICITATION	10.17
10.5.1 Product Model Elicitation	10.17
10.5.2 Process Model Elicitation	10.18
10.5.3 Heuristic Model Elicitation	10.19
10.6 CONCLUSION	10.20
11. MAPPING METHOD SEMANTICS TO METACASE TOOLS	11.1
11.1 INTRODUCTION	11.1
11.2 SIGNIFICANCE OF METACASE TECHNOLOGY	11.3
11.2.1 Incremental Equilibrium in Method Engineering	11.3
11.2.2 Method Engineering in MetaCASE	11.4
11.2.3 Two CASE Studies	11.4
11.3 CASE STUDY A - SCRATCH METHOD	11.6
11.3.1 Product Model Mapping	11.6
11.3.1.1 Basic Product Mapping	11.7
11.3.1.2 Mapping Method Specific Semantics	11.8
11.3.1.3 Extra Semantics	11.11
11.3.2 Process Model Mapping	11.14
11.3.2.1 Basic Process Mapping	11.14
11.3.2.2 Five Steps Mapping Approach	11.15
11.3.2.3 Mapping Concept Dependence to Process Routes	11.19
11.3.3 Heuristic Model Mapping	11.20
11.3.4 More Points on Mapping Semantics	11.22
11.4 CASE STUDY B - BOOCH91 METHOD	11.23

11.4.1 Product Model Matching	11.23
11.4.2 Process Model Matching	11.26
11.4.3 Heuristic Model Matching	11.29
11.5 CONCLUSION	11.30
12. CONCLUSION	12.1
12.1 INTRODUCTION	12.1
12.2 CURRENT ACHIEVEMENT	12.2
12.3 FUTURE WORK	12.4
12.4 CONCLUSION	12.5
APPENDIX A. GLOSSARY	A.1
APPENDIX B. IPSYS TOOLBUILDER	B.1
B.1 Toolset Architecture	B.1
B.2 Entity Model	B.2
B.3 Frame Model	B.7
B.4 Shape Model	B.10
APPENDIX C. THE KADS AND MIKE METHODOLOGIES	C.1
C.1 The KADS Methodology	C.1
C.2 The MIKE Methodology	C.9
APPENDIX D. CONCEPT DIAGRAMS	D.1
D.1 HOOD Concept Diagram	D.1
D.2 Booch OOD Concept Diagram	D.2
D.3 Codarts/DA Concept Diagram	D.3
D.4 OMT Concept Diagram	D.4
D.5 Ptech Concept Diagram	D.5
APPENDIX E. OMT TASK DIAGRAMS	E.1
E.1 OMT: Top Level Task Diagram and <i>objectModelling</i> Decomposition	E.1
E.2 OMT: <i>perform(identifyClass)</i> Task Diagram	E.1
E.3 OMT: <i>perform(identifyAssociation)</i> Task Diagram	E.2
E.4 OMT: <i>perform(identifyAttribute)</i> Task Diagram	E.2
E.5 OMT: <i>perform(organizeInheritance)</i> Task Diagram	E.3

E.6 OMT: <i>perform(verifyObjectModel)</i> Task Diagram	E.3
E.7 OMT: <i>do(checkClass)</i> Task Diagram	E.3
E.8 OMT: <i>do(checkAssociation)</i> Task Diagram	E.4
E.9 OMT: <i>do(checkAttribute)</i> Task Diagram	E.4
APPENDIX F. OMT MSL STATEMENTS	F.1
APPENDIX G. OMT PROLOG CLAUSES	G.1
BIBLIOGRAPHY	

TABLE OF FIGURES

	PAGE
Figure 1.1 A Generic View of Software Engineering	1.2
Figure 1.2 Epistemological Knowledge Hierarchy	1.9
Figure 1.3 Road Map of This Thesis	1.10
Figure 2.1 DeMarco Structured Analysis	2.3
Figure 2.2 JSP Structure Diagrams and Structure Text	2.4
Figure 2.3 Yourdon Modern Structured Analysis Tools	2.5
Figure 2.4 Introduction to SSADM	2.6
Figure 2.5 OOSD Notations	2.8
Figure 2.6 OOSA Models for a One-Minute Microwave Oven	2.9
Figure 2.7 Coad/Yourdon OOA Notations	2.10
Figure 2.8 Nielsen OOD Notations	2.11
Figure 2.9 Jacobson OOSE Notations	2.12
Figure 2.10 Booch OOD Models and Process	2.14
Figure 2.11 Booch Class Diagram and Object Diagram Notations	2.15
Figure 2.12 HOOD Notations	2.17
Figure 2.13 OMT Model Notations	2.19
Figure 2.14 Ptech Notations	2.21
Figure 2.15 Codarts/DA Notations	2.24
Figure 2.16 Codarts/DA Cruise Control Device I/O	2.24
Figure 2.17 CRC: ATM Graphs	2.25
Figure 2.18 ADM3 Icons for Interaction Diagram	2.27
Figure 2.19 Influences on the Fusion Method	2.28
Figure 2.20 Class Description	2.28
Figure 3.1 CDIF Standards	3.2
Figure 3.2 PCTE Standards	3.3
Figure 3.3 MASP Specification	3.6
Figure 3.4 ALF-MASP Approach	3.7
Figure 3.5 SOCRATES Three Levels of Abstraction	3.9
Figure 3.6 SOCRATES Concept Structure	3.10
Figure 3.7 SOCRATES Task Structure	3.11

Figure 3.8 MethodBase Architecture	3.13
Figure 3.9 MethodBase Description of OOA	3.14
Figure 3.10 MetaEdit Approach	3.18
Figure 3.11 MetaEdit OPRR Modelling	3.19
Figure 3.12 OPRR Specification of DFD	3.20
Figure 4.1 Traditional Software Development Approach	4.3
Figure 4.2 Prototyping Approach	4.4
Figure 4.3 Three Stages Prototyping Approach	4.5
Figure 4.4 Three Levels of Software Development	4.8
Figure 4.5 Semantic Knowledge Base	4.9
Figure 4.6 Proposed Methodology Prototyping	4.12
Figure 5.1 OMT: concepts in a <i>stateDiagram</i>	5.4
Figure 5.2 OMT: <i>splittingControl</i> Grouping Cardinalities	5.7
Figure 5.3 Bi- and Uni- Directional Relationships in <i>process</i>	5.7
Figure 5.4 Examples of Overlapping and Completeness Features	5.8
Figure 5.5 A Simple <i>dataFlowDiagram</i> Product Model	5.9
Figure 5.6 HOOD: objectType Subtyping	5.10
Figure 5.7 OMT: <i>state</i> and <i>transition</i> Subtyping	5.11
Figure 5.8 OMT: <i>state-action</i> Compositions in <i>stateDiagram</i>	5.12
Figure 5.9 Composition Cardinalities	5.13
Figure 5.10 A Simple <i>stateTransitionDiagram</i> Example	5.14
Figure 5.11 OMT: <i>transition</i> Links in <i>stateDiagram</i>	5.14
Figure 5.12 OMT: <i>nestedStateDiagram</i> Grouping	5.15
Figure 5.13 OMT: <i>splittingControl</i> and <i>mergingControl</i> Grouping	5.16
Figure 5.14 Ptech: <i>product</i> and <i>activity</i> Decompositions	5.16
Figure 5.15 OMT: Referencing between Three Fragments	5.17
Figure 5.16 OMT: Bidirectional Referencing Relationships	5.17
Figure 5.17 Concept Diagram Notations	5.18
Figure 5.18 Aggregation vs Decomposition vs Referencing	5.19
Figure 5.19 Composition vs Referencing	5.20
Figure 5.20 Merged Link between <i>interState</i> and <i>transition</i>	5.21
Figure 5.21 OMT: <i>instantiation</i> Concept Overrides the Superconcept Relationship	5.21
Figure 5.22 Four Cases of Possible Confusions	5.22
Figure 5.23 Examples of Inclusion and Exclusion	5.23

Figure 5.24 Example of Cut Operations	5.23
Figure 5.25 IPSYS ToolBuilder Notation for Resolving Cut Operation	5.24
Figure 5.26 Step One: Fragment Concepts	5.25
Figure 5.27 OMT: Notation for <i>state</i> and <i>transition</i> Properties	5.26
Figure 5.28 Step Four: Properties of Existing Concepts	5.26
Figure 5.29 Step Five: Complex Grouping	5.27
Figure 5.30 Final Step: a Complete Product Model of <i>stateDiagram</i>	5.28
Figure 5.31 Codarts/DA: Overview Mode	5.29
Figure 5.32 Product Model of Concept Diagram	5.31
Figure 6.1 Meta Modelling Processes	6.2
Figure 6.2 IPSYS ToolBuilder: An Example of MetaCASE Process	6.4
Figure 6.3 Booch OOD: the Configurable Process	6.5
Figure 6.4 Menu Driven Approach	6.7
Figure 6.5 Event Sequence Approach	6.8
Figure 6.6 Frame Based Navigation Approach	6.9
Figure 6.7 Behaviour of Task Functions	6.10
Figure 6.8 Process Model: Task	6.16
Figure 6.9 Ptech: Trigger Rule	6.17
Figure 6.10 Process Model: Task Trigger and Concept Flow	6.18
Figure 6.11 OMT: <i>dynamicModel</i> Task Diagram	6.19
Figure 6.12 OMT: <i>draw(stateDiagram)</i> Task Diagram	6.20
Figure 6.13 OMT: <i>do(verifyAssociation)</i> Task Diagram	6.20
Figure 6.14 OMT: Parallel Tasks in Identifying Elements	6.21
Figure 6.15 Step 2: OMT <i>dynamicModel</i> Task Sequence	6.22
Figure 6.16 Task Diagram of Product Model	6.24
Figure 6.17 Concept Diagram of Process Model	6.25
Figure 6.18 Task Diagram of Process Model	6.25
Figure 7.1 OMT: Heuristic Network	7.6
Figure 7.2 Codarts/DA: <i>objectStructuringCriteria</i> Composition Link	7.7
Figure 7.3 Codarts/DA: <i>taskCohesionCriteria</i> and <i>taskInversionCriteria</i>	7.7
Figure 7.4 Codarts/DA: Reference Link of <i>functionalCohesion</i>	7.8
Figure 7.5 Codarts/DA: <i>taskStructuringCriteria</i> Reference Link	7.8
Figure 7.6 OMT: Mapping Heuristics to Concepts in Product Model	7.9
Figure 7.7 Codarts/DA: Mapping Heuristics to Tasks in Process Model	7.9

Figure 8.1 Method Development by GMR	8.2
Figure 8.2 Codarts/DA: <i>dataFlow</i> and <i>dataTransformation</i> Concepts	8.3
Figure 8.3 Codarts/DA: Dissection Sets	8.4
Figure 8.4 HOOD: <i>constrainedOperation</i> Shared Concept	8.5
Figure 8.5 Common Concepts between Fragments	8.6
Figure 8.6 OMT: Derived Relationship	8.7
Figure 8.7 Multiple Inheritance	8.8
Figure 8.8 Actor Delegation	8.9
Figure 8.9 HOOD: Relationships Between Graphical and Textual Fragments	8.10
Figure 8.10 BNF Definition of MSL	8.12-8.13
Figure 9.1 Method Product Dissection	9.3
Figure 9.2 OMT: Product Fragment Dissection	9.4
Figure 9.3 Ptech: Product Fragment Dissection	9.4
Figure 9.4 Simple Process Dissections	9.6
Figure 9.5 Codarts/DA: Process Model	9.8
Figure 9.6 Booch OOD: Process Model	9.9
Figure 10.1 General Approach for Knowledge Acquisition	10.4
Figure 10.2 Specific Approach for Method Knowledge Acquisition	10.6
Figure 10.3 Top Level Process Model for Method Elicitation	10.7
Figure 10.4 Method Knowledge Inspection	10.8
Figure 10.5 Method Knowledge Fabrication	10.13
Figure 10.6 Method Knowledge Verification	10.14
Figure 11.1 Two Ways of Modelling Method in MetaCASE Technology	11.2
Figure 11.2 Reflective Equilibrium in Inductive Inference	11.3
Figure 11.3 Incremental Equilibrium in Method Engineering	11.3
Figure 11.4 Semantic Gap in Method Engineering	11.4
Figure 11.5 Work-Flows of The Two CASE Studies	11.5
Figure 11.6 Concept Diagram for Scratch	11.6
Figure 11.7 ToolBuilder Entity Model Diagram for Sratch	11.9
Figure 11.8 ToolBuilder Entity Model of the <i>startState</i> to <i>stopState</i> Constraint	11.11
Figure 11.9 Task Diagram for Sratch Method	11.15
Figure 11.10 Creation Paths of Scratch Method Entities	11.16
Figure 11.11 ToolBuilder Navigation Model for Scratch	11.17
Figure 11.12 Sratch: <i>identifyObject</i> Task	11.22

Figure 11.13 Frame Based ToolBuilder Entity Model of Booch91	11.24
Figure 11.14 Booch91 Navigation Model in ToolBuilder	11.26
Figure B.1 ToolBuilder Windowing System	B.1
Figure B.2 ToolBuilder: Three Basic Models in ToolBuilder	B.2
Figure B.3 ToolBuilder: Entity Types and Inheritance	B.4
Figure B.4 ToolBuilder: Reference Relationships	B.5
Figure B.5 ToolBuilder: Derived Relationships	B.6
Figure B.6 ToolBuilder: Examples of Subsections	B.8
Figure B.7 ToolBuilder: Graphics Primitives	B.10
Figure B.8 ToolBuilder: Possible Shape Sets and LinkStyles	B.10
Figure B.9 Frame Based ToolBuilder MetaCASE Tool	B.11
Figure C.1 KADS Architecture Overview	C.2
Figure C.2 KADS Graphical Representation of a Domain Description	C.3
Figure C.3 KADS Inference Structure	C.4
Figure C.4 KADS Task Structure	C.6
Figure C.5 MIKE Hierarchical Process Model	C.10
Figure C.6 MIKE: Example of Context and Model Connections	C.11
Figure D.1 HOOD Concept Diagram	D.1
Figure D.2 Booch OOD Concept Diagram	D.2
Figure D.3 Codarts/DA Concept Diagram	D.3
Figure D.4 OMT Concept Diagram	D.4
Figure D.5 Ptech Concept Diagram	D.5
Figure E.1 OMT: Top Level Task Diagram and <i>objectModelling</i> Decomposition	E.1
Figure E.2 OMT: <i>perform(identifyClass)</i> Task Diagram	E.1
Figure E.3 OMT: <i>perform(identifyAssociation)</i> Task Diagram	E.2
Figure E.4 OMT: <i>perform(identifyAttribute)</i> Task Diagram	E.2
Figure E.5 OMT: <i>perform(organizeInheritance)</i> Task Diagram	E.3
Figure E.6 OMT: <i>perform(verifyObjectModel)</i> Task Diagram	E.3
Figure E.7 OMT: <i>do(checkClass)</i> Task Diagram	E.3
Figure E.8 OMT: <i>do(checkAssociation)</i> Task Diagram	E.4
Figure E.9 OMT: <i>do(checkAttribute)</i> Task Diagram	E.4

TABLE OF TABLES

	PAGE
Table 2.1 Basic Supported Features of the Five Chosen SDMs	2.13
Table 2.2 ADM3 Applicability of Models and the Associated Diagrams	2.26
Table 3.1 Cross Reference of Semantics amongst Meta-Modelling Techniques	3.25
Table 4.1 Examples of the Three Layered Models	4.8
Table 5.1 Uniform Roles for Concept Relationships	5.8
Table 5.2 HOOD: <i>objectDescriptionSkeleton</i> layout based on <i>objectType</i>	5.10
Table 6.1 Booch OOD: Method Process	6.3
Table 6.2 OMT: Analysis Phase Sequence	6.14
Table 6.3 Step 1: OMT <i>dynamicModel</i> Tasks and Operations	6.22
Table 6.4 Step 3: OMT <i>dynamicModel</i> Concept Tokens	6.23
Table 6.5 A Map for Meta Meta Model	6.24
Table 6.6 Task Sequence of Product Model	6.24
Table 6.7 Task Sequence of Process Model	6.25
Table 7.1 Mapping Heuristic Links to Product and Process Models	7.10
Table 8.1 BNF Grammar Rule	8.11
Table 9.1 Numerical Comparison of the Five Chosen Methods	9.11
Table 9.2 Fragment Comparison based on State Modelling	9.14
Table 11.1 Mapping Product Model to ToolBuilder Semantics	11.8
Table 11.2 Mapping Cardinality Constraints	11.9
Table 11.3 Mapping Process Model to ToolBuilder Semantics	11.14
Table 11.4 Mapping Method Semantic Dependence to Tool Process Routes	11.19
Table 11.5 Mapping Heuristic Model to ToolBuilder Semantics	11.22
Table 11.6 Matching Functionality	11.28
Table 11.7 Matching Dependence	11.29
Table B.1 ToolBuilder: Summary of Derived Relationships	B.6
Table C.1 KADS Typology of Knowledge Sources	C.5
Table C.2 KADS Knowledge Identification	C.8
Table C.3 KADS Knowledge Modelling	C.8

1. INTRODUCTION

A number of groups have conducted research into systems development. Some projects aim to develop new (requirement/analysis/design) methods of modelling, whereas others aim to develop new software tools. A common short-coming of these approaches is that they are usually specific to certain problem domains and/or work environments. They fail to unify the diversification amongst themselves. An interesting research challenge is to seek a generic model to represent these modelling techniques so that they can be chosen and/or fabricated into a customised tool to suit the requirement of various applications. This introductory chapter outlines and explains the significance of this approach. A few anticipated advantages are listed, together with the constraints and assumptions of this research project.

1.1 INTRODUCTION

To give an understanding of the ideas and thoughts behind the hypothesis, we will present the background of this work. This section introduces how the research originated, and is followed by a brief history of software engineering (hereinafter abbreviated as SE). It then discusses how complexity can be managed in SE.

1.1.1 BACKGROUND

The original aim of this research was to investigate and/or identify an appropriate in-house object-oriented design method for a medium size software company, in order to improve the efficiency and workability of its software development [Wong 93]. The main products of the company are expert systems and computer-based training packages which make extensive use of graphical user interfaces and message passing techniques. An object-oriented design method seems an ideal choice for such applications, so a large number of object-oriented methods (see chapter 2) are investigated for this purpose.

The direction of the research gradually shifted for the following reasons. At the beginning of the research the company had only limited experience of object-oriented design methods. There was a risk that the method investigation stage would discover an existing design method that suited the current requirements of the company, and no further research would be needed. However, the application domain may easily change in a few years time. A rigid method is not a satisfactory solution to the problem (described as panacea syndrome in section 1.2). A better approach is to employ an integrated system of methods, which allows various viewpoints on the same design, and/or allows techniques of different methods to contribute in

different stages of a single system design. Moreover, the available models and techniques are themselves evolving and advancing day by day. The current technology may very soon be outdated. A system that allows new 'ingredients' to be mixed (or integrated) with the existing ones at any time, and can accommodate the rapidly changing environment of SE effectively, is required.

Nevertheless, this does not alter the primary objective of the research, but it tackles the problem in a broader (or more general) way. That is, instead of finding a single method suitable for the current requirement of the company, it presents a generic model to represent 'all' methods such that the appropriate techniques can be selected and used.

1.1.2 HISTORY OF SOFTWARE ENGINEERING

The classic paradigms of SE, such as waterfall life cycle, prototyping and fourth generation techniques etc., are described in the various literature of the field, i.e. [Davis 83], [Pressman 87] [Sommerville 89]. The software development process contains three generic phases as shown in figure 1.1, regardless of the SE paradigm chosen, the application area, the project size and its complexity. The **definition phase** focuses on *what*: what information is to be processed, what performance is desired, what design constraints exist etc. Thus the key requirements of the system are identified. The **development phase** focuses on *how*: how data structures and the software architecture are to be designed, how procedural details are to be performed etc. The **implementation phase** focuses on the development of the final software: coding, testing and general maintenance. The approach to each step in a phase varies from paradigm to paradigm, but well-defined methods can be adopted in different phases.

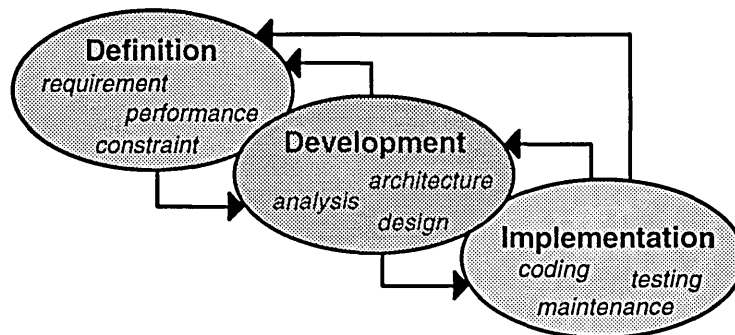


Figure 1.1 A Generic View of Software Engineering

The main interest of this research is in the methods available for the development of generic views, since this lays the crucial discipline required in software planning and development. The subject is also a key strategic problem that occurred in the company.

The history of SE¹ as a whole can be considered as a software project itself performed in phases. The definition phase started in the late 1960s and continued for some years. There were no methods and standards or agreed ways in doing things. At the same time, a Babel of programming languages were developed. This was followed by the development phase, which covered most of the seventies. Researchers concentrated on methods for software analysis and design. These methods were mostly intended to suit a part of the software life cycle or were sometimes restricted to specific areas of application [Wynekoop 93]. Then the implementation phase began in the late 70s and is still ongoing. Massive numbers of software tools implemented the methods and promoted their practical use in industrial environments. Nowadays, many tools are commercially available as products.

One of the purposes in SE is to 'model chaos (or real-world situations) into formality'. In other words, the objective of SE is to manage real-world complexity by software techniques and engineering disciplines. There is, however, another layer of complexity within SE.

1.1.3 MANAGING COMPLEXITY OF SOFTWARE ENGINEERING

The benefit of methods and tools is clarified in the literature [PACT 85] [Gillies 94]. Their richness of variety results in our requiring certain 'meta' techniques to help select, compare and evaluate them. The following list of generic method types may be exhausting, but it is not exhaustive: business analysis, systems analysis and design, application design, application development, systems integration, project planning and version control [Madsen 95]. Similarly the tools implemented with these methods also possess various generic types, such as transformation tools, interpreters, simulators and integrated programming support environments (IPSE) [Tontsch 90]. Even considering software development methods alone, there are distinctions to suit different needs: structured methods, real-time methods, object-oriented methods, etc. Hence the mapping of available methods (or tools) to the required analysis techniques of the problem domain is an important role in 'modern SE'.

Method (or tool) integration promises to synthesise portable components into combined systems (see chapter three). This approach allows multiple viewpoints on a specific application, but it does not properly handle the analytical problem stated above. Instead it merely defers the decision back to the software developer. The embedded information (or semantics) gathered in the methods are not efficiently addressed.

Hence there are three areas of systems development needing research [Freeman 92]. The first one is the **gathering of information about the problem domain**. This is a generic activity

¹ Although the original idea on software project of SE comes from [Tontsch 90] and [Madsen 95], some adjustments are made to suit the description of generic view of SE and managing complexity.

that includes analysis of requirements or needs, preparation of reusable components, writing of specifications, design, testing, etc. The second general area is **analysis of design decisions**, which are methods of analysing a proposed system to determine characteristics of interest. They embody consistency of successive design decisions; system performance and resources needed to take the next step of development. The third area, naturally enough, is that of **representations**.

This research concentrates on the last area by pursuing a standard or generic model to represent software development methods. In order to stress the goals of this hypothesis, it is necessary to note three current syndromes in SE.

1.2 THE REAL PROBLEMS

This section describes problems faced by most software engineers in the industry, especially those who are working in medium to large sized software systems in a collaborating environment. The problems are illustrated as syndromes of software development.

To give a better picture of each syndrome, a real situation from the collaborative company (SAT) is demonstrated below. The project involved is known as IFA (intelligent financial advisor). IFA is a software system comprised of three separate components: a spreadsheet style data model, a human-computer interface (HCI) windowing system and an expert advisor. Each component is designed and developed by a dedicated software engineer. The prime technique employed by SAT is a programming model called MVC (model-view-controller), which is based on modelling the data, presentation and control aspects of a system. Since the main concern of this research is a method in the development phase, MVC is not considered appropriate for this purpose in the first place. It is just a technique to describe program models, although it is an ad-hoc approach learned by SAT.

1.2.1 PANACEA SYNDROME

There are two reasons why software engineers often find themselves using the wrong language, method or tool for a description. They are closely related but distinct [Jackson 92]:

- *The first is that we are still immature enough to claim that each new medicine will cure all diseases. To a Prolog interpreter, everything in the world looks like a set of Horn clauses; to a systems analyst with a relational database management system, everything in the world looks like a relation in third-normal form.*
- *The second reason is that we have some highly developed tools and techniques for handling each of our models in isolation, but few or none for combining descriptions made from different models. Therefore we take what seems to be the*

easy way out: we choose one modelling formalism, for which we have the skill or the tools, and we make all our descriptions in that model.

The outcome is to make the best of a bad job, and then to claim the chosen formalism is all that is needed. MVC is a good technique for identifying various aspects in the implementation phase. For instance, in the IFA project, the spreadsheet describes the data model, the HCI module shows the presentation view and the expert advisor forms the control mechanism. However, when the software engineer attempted to develop the expert advisor in isolation there was no sufficient technique to describe the internal structure of inferencing and he had to replace it with the conventional 'hacking' method.

For this syndrome, a multiple viewpoints approach is necessary to widen the descriptive power upon the specific problem domain. A proverb says that 'no person can break a bunch of arrows in one go, but it can be done by a group of people each splitting a few arrows'.

1.2.2 LOOKING UNDER THE LAMPPOST SYNDROME

This is a thought pattern best illustrated by the following story [Gilb 88]:

... a drunk person is found searching for his lost wallet under a lamppost at a street corner. When asked where he lost the wallet, he says, "Oh, down the street there by the alley. But I'm looking for it up here because the light is so much better!"

The typical SE and management solutions are found primarily by looking under the lampposts, such as algorithms, formal specification or specific software acquisition methods. The lamps illuminating areas such as software maintainability, portability and user-friendliness have been relatively faint. This syndrome suffers in overemphasis on certain areas where the developer is strong and underestimating the grey or weak areas. It differs from the panacea syndrome by contrasting the shortcoming of personal and social awareness.

Most people suffer from the syndrome to some extent, but it can be a particular concern in software development models. Each member of a cooperative team has strong lamps that illuminate his own well-understood parts, however it is also important to use the fainter lamps to provide some illumination to the less-understood but equally critical parts of the system.

For example, the software engineer of the expert part in the IFA project can neither neglect the interface specification to the spreadsheet module, nor ignore the HCI requirement from which the advisory information is displayed. There must be an agreement to cover the shaded areas between the bright areas. Due to lack of common representation in the shaded areas, a substantial part of the advisor module had to be redone to allow the interaction.

1.2.3 THREE BLIND MEN TOUCHING AN ELEPHANT SYNDROME

This syndrome is borrowed from a famous Chinese idiom with a similar meaning. The blind wise men came to three different opinions about the ‘reality’ they were dealing with after touching different parts of an elephant [Yourdon 89]:

- *One blind man touched the sharp end of one of the elephant's long tusks. “Aha”, he said, “what we have here is a bull. I can feel its horns.”*
- *The second blind man touched the bristly hide of the elephant. “Without a doubt”, he said, “this is a ... what? A porcupine? Yes, indeed - a porcupine!”*
- *The third blind man felt one of the elephant's thick legs and said, “this must be a tree that we're dealing with.”*

Yourdon uses this story to illustrate what he called ‘balancing the models’ in his modern structured design (see section 2.2.3). This is essentially a direct mapping between different aspects of a system, so that they maintain a consistent interpretation of the reality. In a larger scope of integration we can consider the coherence between different models from various methods specifying a unique system. The syndrome may be observed as a number of developers viewing the reality from different perspectives. Thus, it also promotes the idea of multiple viewpoints on a system specification.

It is obvious that different initial viewpoints will lead to people employing different modelling formalisms to interpret the system. In the IFA project the software engineers viewed the common data repository differently. The HCI developer regarded it as a set of information to be displayed; the expert advisor regarded it as a formulated cell for calculating data elements for ratio analysis, whereas the spreadsheet developer regarded it as a storage item in the relational database. Since there was no common agreement on the formalism they communicated through a complex interface for data integration. The overhead was vast. In addition, this complexity - albeit induced complexity - will prevent any future system modifications.

To conclude, distinct problem domains and/or work environments demand different software development techniques. Since there is no single perfect method for system description, it is necessary to permit method integration amongst the techniques that are currently available, taking account of strengths and weaknesses of the techniques employed. The coherence across techniques is an additional and significant issue when developing a whole picture of the required system. The ultimate solution (common goal) is to formulate a generic representation of these software development methods which allows a full integration of techniques and manages the semantics simultaneously.

1.3 THE PROPOSED SOLUTION

The body of this thesis develops a generic model for representing software development methods. The model is named **GMR**, which stands for *generic method representation*. Various components and aspects of GMR are described in the succeeding chapters, however it is important to stress the hypothesis and the goals of the research at this point. The advantages, constraints and assumptions are also listed in the following subsections.

THE HYPOTHESIS

Meta modelling (that is generic representation of methods) enables methods to be used unambiguously, since they are defined by an abstract model. Moreover, it permits comparison between methods and the ability to assess the suitability of the semantics of a method to a problem domain.

THE GOALS

1. By investigating various methods and meta modelling systems, identify the components and techniques to represent software development methods.
2. For each component, determine the internal characteristics to provide a concise and precise representation of that component. The interrelationships between components must also be addressed in the representation.
3. The representation should comprise both textual and graphical forms. The former should be a specification language of the method, whereas the latter should capture different components by a finite set of diagrammatic notations.
4. Try out the representation in as many proficient methods as possible, so that it covers all modelling aspects.
5. Develop a knowledge acquisition model for sketching methods in the representation; certain appropriate verification techniques must also be investigated.

THE ADVANTAGES

1. Provide a generic standard for representing methods. Avoid specific tool modelling mechanisms but rather bridge the semantic gap with metaCASE tools.
2. Support the model with acquisition techniques of method knowledge.
3. Allow method integration in terms of semantic components, so that dissection of portable parts can be easily achieved. In addition, provide better channels for method comparison, selection and evaluation.

THE CONSTRAINTS

1. GMR is constructed from the current modelling formalism available.
2. Although there are a number of metaCASE tools in the market, the only tool accessible for this research is the IPSYS ToolBuilder.
3. Due to the limitation of time (three working years), a full proof of items in Advantage 3 is impossible. However, an informal attempt is shown to an appropriate level.

THE ASSUMPTIONS

1. *GMR only serves class-based software development methods* (see chapter two for a detailed definition). Although it may be adapted in a larger scope of method taxonomy, we make no claim about GMR's utility outside this scope.
2. There are hundreds of software development methods in the world. Since it is impossible to verify all methods within the time limitation, it is necessary to assume that *if GMR works for a finite set of complex methods, it works for all methods*.
3. As the fact of resource limitation in constraint 2, there is an assumption that *if GMR maps into a competent metaCASE tool, it can map to other tools*.

1.4 EPISTEMOLOGICAL HIERARCHY OF KNOWLEDGE

Before ending this introductory chapter it may interest the reader to look at the nature of GMR in a knowledge hierarchy. [Hirschheim 92] shows the nature of human knowledge and inquiry broken down into four fundamental sets of beliefs:

ontological (beliefs about the nature of the world around us); epistemological (beliefs about how knowledge is acquired); methodological (beliefs about the appropriate mechanisms for acquiring knowledge); and beliefs about human nature (i.e. whether humans respond in a deterministic or nondeterministic way).

Figure 1.2a depicts an epistemological hierarchy of a system modelling a world [Gaines 87]. The foundational role in knowledge acquisition is evident in the hierarchical representation of distinctions in the modelling system. The levels of hierarchy itself are the results of distinctions made so that no additional primitives are introduced (see [Klir 76] for the meaning of terminology). Note that the upper levels of modelling are totally dependent on the system of distinctions used to express experience through the source system.

Obviously GMR fulfills a role of the methodological belief, but it is also an epistemological belief as a meta model to acquire knowledge. Therefore a method knowledge hierarchy in software development can be viewed as different levels of system modelling. Figure 1.2b

illustrates the direct mapping of Klir's modelling hierarchy to the software method modelling hierarchy. Each level presents the differences between those distinctions in the lower level, that is a higher abstraction of the respective semantic model. The formal representation of each model has an associated modelling (or specification) language. All languages of this software method modelling hierarchy are basically in textual form, though graphical representation is always possible and often useful. Moreover, each model should be capable of sketching both static and dynamic behaviours of the respective level.

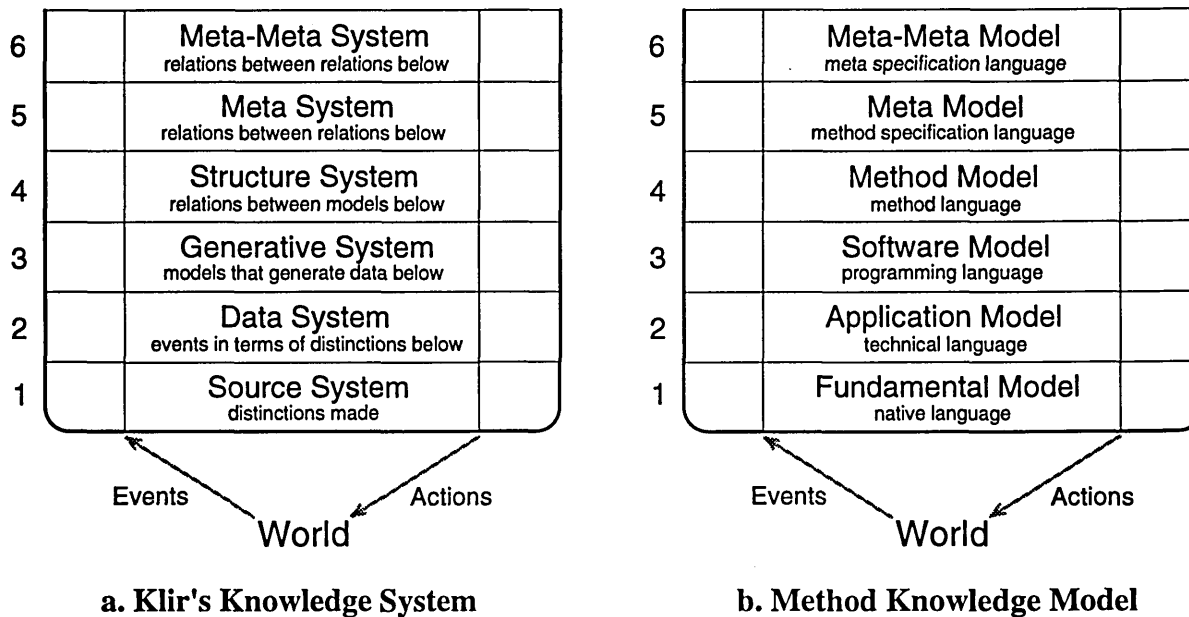


Figure 1.2 Epistemological Knowledge Hierarchy

The fundamental model provides descriptive terms of the system's domain in the world. This is normally given in simple native language. The application model provides formal descriptions in these terms by a technical language appropriate to the expertise. The software model provides a regeneration of these descriptions in terms of executable statements by a programming language. The rational construction bridges the two models below (levels 1 and 2) to the models above (levels 4 through 6). The method model provides a theoretical framework in assisting the development of software models. The language incorporated is called method language. The meta model provides a descriptive form of the method known as method specification language. The meta-meta model provides an attempt to further classify the description in meta model specification language.

Throughout, this research concentrates on the discussion of the world in the knowledge environment, as defined in [Popper 68], from level 4 to level 6 in the modelling hierarchy. The software model (level 3) is also discussed, since it provides a direct illustration of the method model and the meta model (see chapter four for details).

1.5 ROAD MAP

Since the GMR has evolved from the investigation of current approaches and related technologies it is essential to give a brief report on the significant points of the literature reviewed in order to show the progression of observations and implications. In addition, apart from the main body of the GMR approach, there are a few associated areas that must be considered to fully explain the meta model. To convey these complex matters logically this thesis is structured as depicted in figure 1.3. The solid arrows denote the main route through the report, whereas the v-shaped arrows show the supporting materials. Rectangles represent chapters and circles denote appendices. There are altogether twelve chapters (1-12) and seven appendices (A-G) in this thesis.

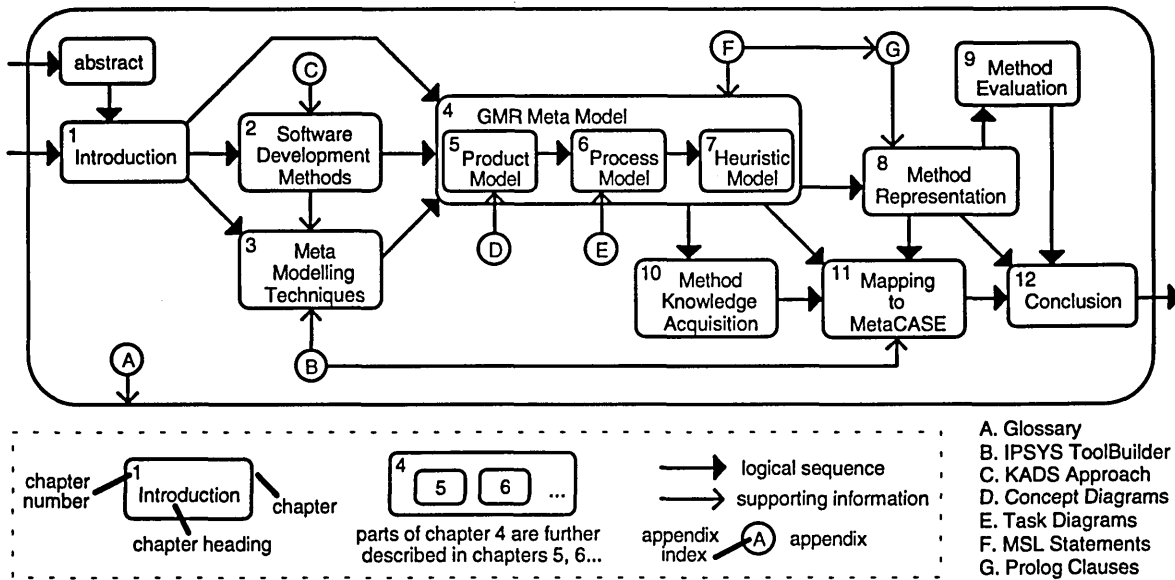


Figure 1.3 Road Map of This Thesis

The abstract of each chapter is as follows:

1. *Introduction* - This chapter identifies certain problems and complexities in software engineering, and proposes an evolutionary approach towards system development. Then it focuses the research on the goals and advantages of the approach. A road map of this thesis is given.
2. *Software Development Methods* - An investigation of software development methods is reported with a special emphasis on meta modelling and knowledge engineering. The description is divided into four categories: structured methods, object-oriented methods, main interests and other methods. This chapter also refines the scope of the research by defining different types of 'software development methods'.

3. *Meta Modelling Techniques* - The current meta modelling techniques are observed from three categories, namely method integration, meta modelling researches and metaCASE tools. The drawbacks of each approach are noted, and their significant points about method representation are gathered. Chapters two and three contribute to the basis of meta modelling in GMR.
4. *GMR Meta Model* - This chapter presents an overview of the GMR meta model. Detailed descriptions of individual components (i.e. product, process and heuristic models) are given in the three succeeding chapters. The semantic knowledge base for GMR is also considered, with examples.
5. *Product Model* - This chapter demonstrates a formal and rigorous representation of concepts in software development methods. This model consists of concepts, relationships and their properties, which can be denoted individually in a concept diagram. The activities involved in produce modelling are also illustrated.
6. *Process Model* - This chapter presents a generic process model, which is loosely defined in order to provide flexibility and freedom for developer creativity. The structure of a task and task functions are identified. Task sequence is introduced to document the process model in a tabular form, which can easily map to a task diagram.
7. *Heuristic Model* - This chapter discusses the two types of method heuristics. The textual structure and graphical presentation are also described.
8. *Method Representation* - Some representation topics are presented in further detail. A method specification language (MSL) is introduced as a formal declaration of GMR.
9. *Method Evaluation* - The anticipated advantages of GMR for fragment dissection, method comparison and selection of method (see later for the definitions) are ascertained in this chapter.
10. *Method Knowledge Acquisition* - Although there are some rudimentary difficulties in knowledge acquisition of methods, a variety of techniques can help to enrich the quality of expertise transfer. The method acquisition media are introduced and an IFV model is shown to guide the knowledge elicitation.
11. *Mapping to MetaCASE Tool* - A model is ineffectual unless it can be implemented into a tool, so it is important to show that GMR can be mapped into practical metaCASE tools. Although the GMR does not depend on any particular tool, IPSYS ToolBuilder is chosen for this illustration. Two case studies are given to demonstrate the mapping.
12. *Conclusion* - This chapter summarises the work done to prove the hypothesis of a generic model for representing methods. A number of speculative future works are also discussed in detail.

Seven appendices are provided. They are mainly supportive information for this thesis, and are briefly described as follows:

- A. *Glossary* - defines the main terminology used in this research;
- B. *IPSYS ToolBuilder* - gives a detailed description of a general metaCASE tool that is referred to throughout the thesis;
- C. *KADS Approach* - outlines this knowledge-based engineering approach to advocate the modelling techniques and knowledge acquisition of methods;
- D. *Concepts Diagrams* - depict the GMR product models of five selected methods, namely Booch OOD, Codarts/DA, HOOD, OMT and Ptech (see section 2.4 for details), to illustrate the discussion in chapter 5;
- E. *Task Diagrams* - shows the GMR process model of OMT;
- F. *MSL Statements* - presents a complete representation of a sample software development method (i.e. OMT) by using the method specification language (MSL), which is developed by the GMR approach;
- G. *Prolog Clauses* - shows the compiled Prolog form of the MSL statements illustrated in appendix F.

1.6 CONCLUSION

By describing the background experiences and problems faced in software engineering, this chapter focuses the research on the hypothesis. The ultimate goal is to model a generic representation of software development methods known as GMR. Certain constraints and essential assumptions are listed, with the advantages from the representation. The relation of GMR towards the epistemological and methodological beliefs are addressed. It also presents a road map of the thesis.

2. INVESTIGATION OF SOFTWARE DEVELOPMENT METHODS

In this chapter, a large number of software development methods (SDMs) are reviewed with no bias on any particular programming paradigm. The investigation emphasises the prospects of meta modelling, where method components are resolved. Also, the significance of meta modelling in method comparison and tool integration is highlighted, together with extensive comments about each analysis and design methods.

2.1 INTRODUCTION

Nowadays most software methods are concerned with either structured or object-oriented paradigms. One must appreciate how the semantics of these analysis and design methods can be unified, thus allowing components from any stage in the development life cycle and from different environments and methods to be shared via a common meta model [Carmichael 94]. This meta model must be allowed to represent the semantics of various methods from different programming paradigms. [Masini 91] investigates three types of object-oriented languages; each of them has a different emphasis in software development viewpoints:

- **Class based languages** consider objects from a structural point of view. An object is a data type defining a model of the structure of its physical representatives and a set of operations applicable to this structure. Languages of this type, such as C++ [Stroustrup 86], are mainly for system engineering with traditional software life cycle. A large number of methods are available for these languages, such as those described in [Graham 94].
- **Frame based languages** consider objects from a conceptual point of view. An object is a unit of knowledge representing the prototype of a concept. These languages are used to develop executable knowledge systems, for instance KRL, KL-ONE [Rich 91] and conceptual graphs [Sowa 84]. Since these applications have a very specific problem domain, there is no particular development method associated with them. However, the technology of KADS [Schreiber 93] [Tansley 93] is emerging, which provides some general techniques for developing knowledge-based systems from frame based languages.
- **Actor based languages** consider objects from an active point of view. An object is an autonomous and active entity reproduced by copying and it delegates work by sending messages to another actor [Tello 89]. This type of language is still under-developed and most examples are only used for research purposes, so no attempts of formalised method for these languages have been found.

Different problem domains and/or work environments demand different sets of development languages and techniques. Although there are various languages or tools to suit the specific needs, not all of them are accompanied by a proper development method. For instance, LPA Prolog has an extension to the Prolog++ language [Moss 94], which tries to amalgamate frame-based programming with object-orientation. The consequence is an extra set of techniques inserted to form a hyper-language so no rigorous method is encountered.

Hence, the meta model is only dedicated for software development paradigms with distinct methods, which have concise and precise notions of semantics and apparent design strategies or specifications. The model mainly concerns class based system development (though KADS is also discussed). Due to the limitation of time and resources, the suitability of other approaches is not considered within the scope of this research.

The review of software development methods is not just a literature survey of current available methods, but it shall also look into ways of comparing methods, of structuring concepts, of identifying tools and formalising design guidance etc. In addition, it shall stress the products and activities of methods in order to distinguish various component types as well as to represent them as semantics in an efficient and effective way. The following points summarise the particular interests of the investigation (not in any order of preference):

- meta modelling viewpoints, including components and techniques;
- method paradigms, such as structured paradigms or object-oriented paradigms
- development phase(s) in software life cycle, i.e. requirements, analysis and/or design;
- structural, dynamic and behavioural (i.e. object, state and function) aspects of the method;
- textual and graphical notations;
- costs and benefits, i.e. comparison and evaluation of the method;
- application domain of programming language or tool dependence;
- support with automated CASE tool(s);
- relationship or integration with other methods, such as CRC and Fusion method;
- other development features, such as real-time, concurrency and distributed.

This chapter gives a condensed description of eighteen structured and object-oriented methods with special focus on method representation. Five of them are chosen for detailed meta modelling experimentation, namely Booch OOD, Codarts/DA, HOOD, OMT and Ptech. The justifications of these selections are also provided. Therefore, the method descriptions are classified into four categories: structured methods, object-oriented methods, chosen methods and other methods. Some notable points are recalled in the summary of investigation, which is followed by a brief conclusion.

2.2 STRUCTURED METHODS

From the 1960s to the late 70s, there was an urgent software crisis [Cox 87], due to the increasing requirements on software quality and quantity in vast problem domains. The computer industry has been revolutionised by a number of new philosophies and techniques [Yourdon 79]. One of the most popular of these techniques, *structured programming*, has led to order-of-magnitude improvements in the productivity, reliability and maintenance costs associated with computer systems. The methods embodied by these techniques have emerged and evolved, and some of them are still in use or affecting the present software (and/or method) development. This section looks at four historically prominent structured methods.

2.2.1 DEMARCO STRUCTURED ANALYSIS (DEMARCO SA)

DeMarco SA was perhaps the most influential of the early structured analysis method [DeMarco 79]. It promotes a structured description based on functional decomposition and process specification. The structured analysis tools include *data flow diagram* (DFD), *data dictionary*, *data structured diagram* (DSD), *structured English* and *decision table* (or tree). Individual components in a DFD are described further in the data dictionary, and a process in the DFD can be extended to a lower level DFD (such as figure 2.1a). This parent-child relationship effectively outlines a top-down refinement approach. In addition, each processes denoted in the bottom level DFD must be defined in the process specification by structured English, flow graphs or decision tables (such as figure 2.1b).

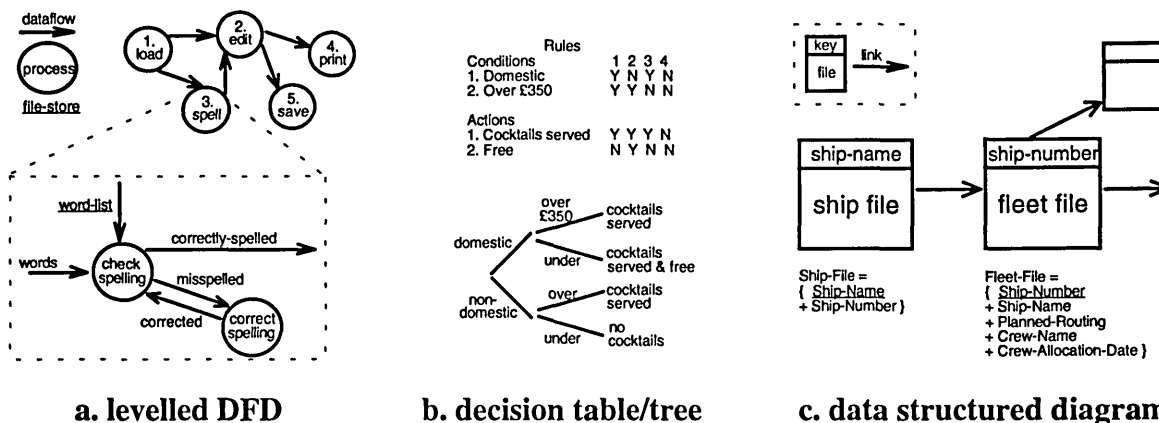


Figure 2.1 DeMarco Structured Analysis

A data structured diagram (DSD) depicts the current environment in such a way that the user can understand and verify it. DeMarco SA emphasises the functional specification rather than the data declaration. The DSD (such as is illustrated in figure 2.1c) is not much different than a database file which is referenced by a key field. In addition, the dynamic aspect of processes are not denoted, DeMarco SA is only appropriate in a sequential processing application.

2.2.2 JACKSON STRUCTURED DESIGN (JSD)

Jackson Structured Programming (JSP) is a program design method for sequential processes [Jackson 75]. It is therefore specific for sequential languages such as PL/I, Cobol, Fortran, Pascal or assembly languages. JSP program design describes input and output data streams and allocates them with proper operations in the program structure. Figure 2.2 shows the JSP notations of three basic components in structured diagrams and structured text:

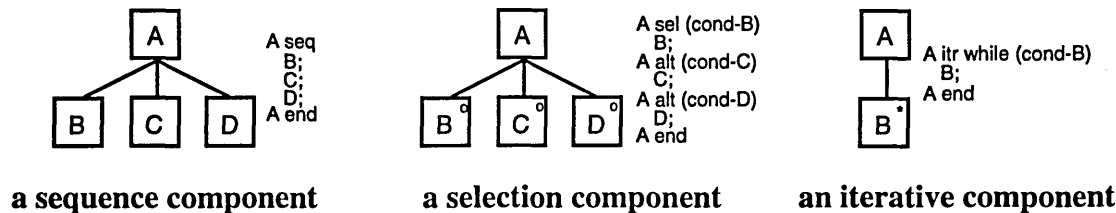


Figure 2.2 JSP Structure Diagrams and Structure Text

Jackson Structured Design (JSD) [Jackson 83] has grown out of JSP. It is a method for specifying and implementing computer systems with a strong time dimension. Again the functional specification describes the decomposition, detailing and refinement of processes. Apart from DSD and structure text, JSD uses an *entity structure diagram* (ESD) which employs the JSP notations to express the classical constructs of structured programming (figure 2.2). In addition, the *system specification diagram* (SSD) is used for arranging processes and data streams; and the *system implementation diagram* (SID) for process dismembering¹. The major differences with the DeMarco SA is that JSD provides six clear steps for software development as follow:

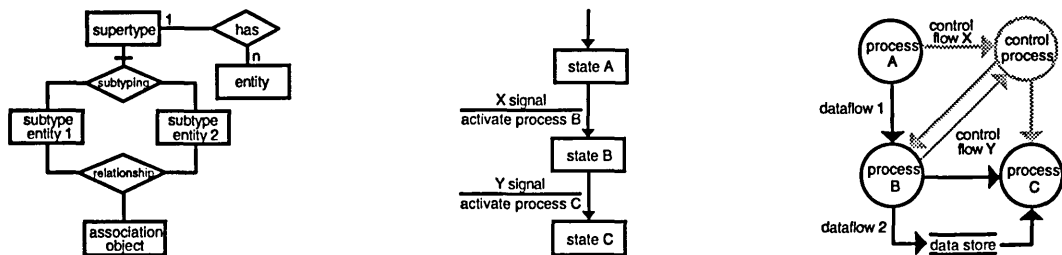
- *entity action step* - define real world area of interest by entities and actions;
- *entity structure step* - arrange actions by each entity in their orderings with time (by ESD);
- *initial model step* - describe connections with real world in terms of entities and action;
- *function step* - specify functions to produce the outputs of the system (by SSD);
- *system timing step* - consider process scheduling which affect the system function outputs;
- *implementation step* - consider software and hardware provided for running the system.

Each step has detailed criteria. The important distinction in JSD is not between analysis and programming, but between specification (the first five steps) and implementation (the last step). JSD also stresses that structured design is an iterative process, though the aim is to minimise the number of cycles. Therefore, JSD is not a top-down design.

¹ System dismembering is a transformation in which the text or state-vector of a process, or a data stream, is broken into a number of parts for convenience and efficiency in execution.

2.2.3 YOURDON MODERN STRUCTURED ANALYSIS (YOURDON)

Yourdon modern structured analysis (Yourdon) is another well-recognised method in software engineering. Aside from the traditionally structured tools, such as *DFD*, *data dictionary* and *process specification*, Yourdon borrows the *entity-relationship diagram* (ERD) [Chen 76] to describe the stored data layout of a system at a high level of abstraction (figure 2.3a). In addition, the time-dependent behaviour of a system is described in a *state transition diagram* (STD in figure 2.3b), which is a more effective way to describe event sequences in a complex entity than the entity structure diagram in JSD. Yourdon also handles real-time issues by introducing the control processes and control flows in DFD (figure 2.3c). One distinction of Yourdon is that it gives detailed techniques for *balancing the models*, that is for managing the various dependencies between different tools. For instance, each control process in DFD must be further specified in a STD. The control process in figure 2.3c is depicted as the STD in figure 2.3b where each state refers to the corresponding process in the DFD.



a. entity-relationship diagram b. state-transition diagram c. data flow diagram

Figure 2.3 Yourdon Modern Structured Analysis Tools

Although Yourdon does not explicitly denote the analysis steps, the development process can be worked out from the four analysis models in the order presented below. For each model, Yourdon presents both the classical approach and its approach. In this way, the method description has a bigger contrast with the general practice.

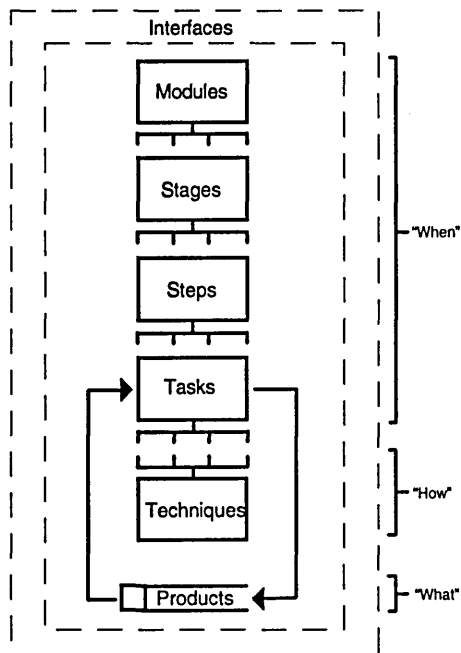
- *essential model* - what the system must do in order to satisfy the user's requirement;
- *environmental model* - define the interfaces between the system and the environment;
- *behavioural model* - what internal behaviour is required to deal with external environment;
- *user implementation model* - describe automation boundary, human interface and formats.

Furthermore, Yourdon looks into the program evaluation review technique (*PERT chart*) and project management (*Gantt chart*). The potential automated tools, such as document control, software metrics etc. are also discussed. Yourdon is obviously a more comprehensive method than the two previously described methods, but some general software features, such as concurrency, information hiding, task structuring are not represented by the method.

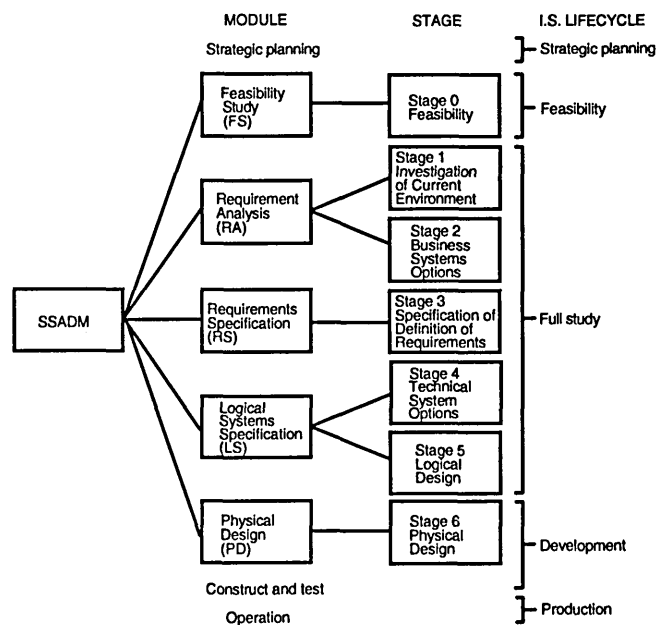
2.2.4 STRUCTURED SYSTEMS ANALYSIS & DESIGN METHOD

This method is normally abbreviated as SSADM [Downs 92], which is a method for developing computer-based information systems. It originated with the UK government's Central Computer and Telecommunications Agency (CCTA), and is widely used in many development projects. The method consists of activities and products (figure 2.4a). The activities include 'when' and 'how' something should be done, whereas the products describe 'what' is delivered. The activity structure of SSADM is presented as a five module hierarchy (figure 2.4b). Each module, as shown below, is broken down into stages, steps and tasks.

- *feasibility study* (FS) - includes non-SSADM work, such as financial cost/benefit analysis, social evaluation or writing a feasibility report;
- *requirement analysis* (RA) - has two stages: it describes the *investigation of current environment* and establishes a range of *business system options*;
- *requirement specification* (RS) - reworks the descriptions of the current environment and business system option, produced in RA;
- *logical system specification* (LS) - has two stages: it determines and helps management to select the *technical system options*; and defines dialogues, updates and enquires in a non-procedural *logical design*;
- *physical design* (PD) - takes the LS and combines it with information about the target hardware, software and organisation setting.



a. activities and products



b. module hierarchy

Figure 2.4 Introduction to SSADM

The individual stage and step descriptions can be found in [Downs 92]. However, it is interesting to note that SSADM defines each step in great detail by its *input(s)*, *output(s)*, *tasks* and *techniques*. The inputs and outputs provide the links to other steps, as well as giving the products associated with the step. SSADM comprises an extensive number of products for various tasks, such as *DFD*, *logical data structure* (LDS), *entity life history* (ELH), *effect correspondence diagram* (ECD), *business system option* (BSO), *technical system option* (TSO) etc. This is illustrated by the feasibility study step 010 below:

Input(s)

- Project Initiation Document (from Project Procedures)

Tasks

10. Working from any documents which initiated the study, create an outline description of the existing system and record known requirements
20. Establish the scope of the Feasibility Study, and agree with the Project Board
30. Tune SSADM to meet the needs of the feasibility study and agree with the Project Board

Techniques

- Data flow modelling
- Logical data modelling
- Requirements definition

New or modified output(s)

- Context Diagram (to 020)
- Current Physical Level-1 DFD (to 020)
- Overview LDS (to 020)
- Requirements Catalogue (to 020)
- Agreed study method (to Project Procedures)

From the meta modelling viewpoint, these are significant pieces of information for describing the execution of products (concepts) in terms of tasks and techniques. The inputs and outputs are virtually the requirements and consequences of the step. Since SSADM is a structured method, the meta-structure of the method is also revealed in a top-down hierarchical form. That means it can only handle sequential processes so it does not promote iterative or parallel steps. This is considered as a drawback of the representation. As with other structured methods, SSADM does not stress data abstraction or information hiding. However, the latest version of SSADM (and JSD) has taken in some object-oriented (OO) features to solve this inadequacy and to accommodate the corresponding programming paradigm. SSADM is accompanied with automated tools such as *LBMS SSADM* and *Automate-Plus*, so it is not just a 'paper-model'.

2.3 OBJECT-ORIENTED METHODS

The rise of object-orientation started in the late 70s with the emphasis on object-oriented programming (OOP) [Cox 87]. This was followed by the object-oriented design (OOD), and recently object-oriented analysis (OOA) has become the major area of interest. OOP was first thought as an AI feature [Luger 89] [Tello 89]. However, the promises of offering high reliability, productivity and maintainability through the implicit object techniques such as abstraction, encapsulation, inheritance and polymorphism [Meyer 88] has drawn the interest of

a lot of software developers [Blair 91]. Object-oriented technology has entered the mainstream of industrial applications [Taylor 92] and research interests [Khoshafian 90]. In fact, 'object-oriented' has become an extremely overloaded term and very few commercial systems live up to the pure concept of object-orientation. Nevertheless, object-oriented methods have a major role in the technology. The main concern is not so much whether a method is object-oriented or not, but *how* it is object-oriented and in *what* way it delivers the associated benefits [Graham 91].

2.3.1 OBJECT-ORIENTED STRUCTURED DESIGN (OOSD)

Object-Oriented Structured Design (OOSD) is a method intermediate between analysis and design [Wasserman 90]. It is a notation for architectural design which combines structured methods with object-orientation, as promoted by [Ward 89] and [Champeaux 91]. OOSD is influenced by Yourdon's structure charts, such as data flow, parameter passing and exception handling (figure 2.5a), but it also adopts object-oriented concepts such as encapsulation, instantiation and inheritance (figure 2.5b). In addition, concurrent or asynchronous processes are catered for by using monitors which are shown as parallelograms (figure 2.9c). These are important semantics for a requirements analysis as well as for a design method.

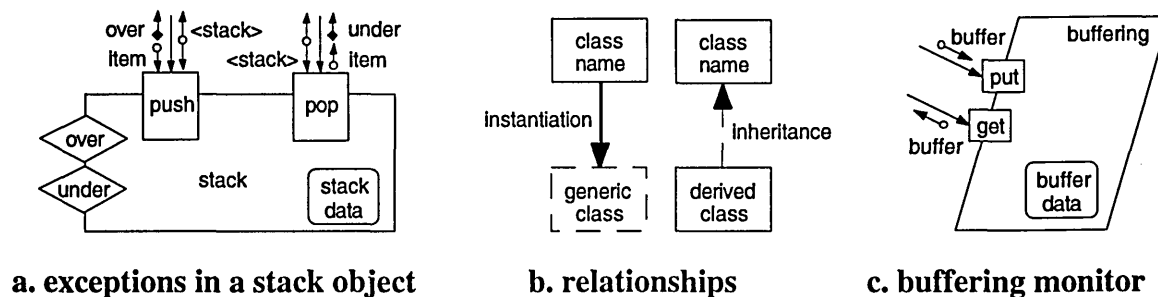


Figure 2.5 OOSD Notations

OOSD also has a formal grammar, which is a program design language ideal for improving comprehensibility of the notations in non-graphical form. OOSD also introduces design rules to guide the software development. They are possible for producing an automated CASE tool which provides consistency checking and code generation. In addition, OOSD is not tied to any programming language and is one of the more advanced hybrid, low-level OOD notations. OOSD has a ready acceptance by analysts who are familiar with structured design and its suitability for real-time systems because of the monitor concepts. However, OOSD is only a set of design notations, which even lacks an effective semantic data modelling to describe real world objects. Although it discusses object behaviour and design rules, the method gives neither design steps nor detail guidance. OOSD is more suitable for architectural or logical design than for physical design.

2.3.2 OBJECT-ORIENTED SYSTEMS ANALYSIS (OOSA)²

Shlaer/Mellor's OOSA [Shlaer 91] is a method for identifying the significant entities in a real-world problem domain and for understanding and explaining how they interact with one another. The entity modelling is descended from the Ward/Mellor real-time notation [Ward 85], hence OOSA users tend to be developer who migrated from the Ward/Mellor approach. The method is best described in three models, which OOSA refers to as three steps:

- *information model* - focus on abstracting the conceptual entities in the problem by objects, attributes and relationships - advanced *entity-relationship diagram* (figure 2.6a);
- *state model* - formalise lifecycles of objects and relationships from information model over time, in other words, express dynamic behaviour in *state transition diagram* (figure 2.6b);
- *process model* - depict actions in state model as a fundamental and reusable process by an enhanced form of the traditional DeMarco data flow diagram - *action DFD* (figure 2.6c).

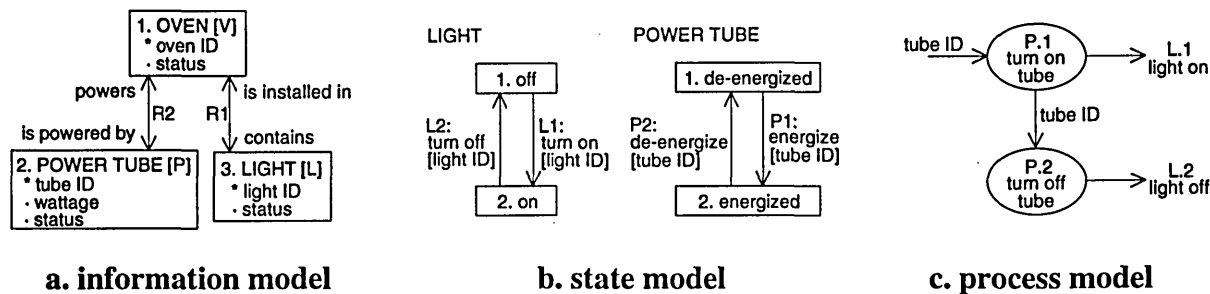


Figure 2.6 OOSA Models for a One-Minute Microwave Oven

The early version of OOSA [Shlaer 88] cannot really be regarded as object-oriented due to the absence of inheritance. Entity subtyping is only introduced in a later book [Shlaer 91]. OOSA considers object identities by sets of attributes and keys (the *status* and *IDs* in figure 2.6a), then applies normalisation rules to the objects. Thus objects are regarded as relational tables rather than abstract data types.

From the meta modelling viewpoint, OOSA lays stress on strong cohesion between models. The labels and IDs provide the reference links amongst the models. For instance, in the *microwave oven* example of figure 2.6, the *tube ID* in the information model is used in the data flows of the process model, whereas the actions *P1* and *P2* in the state model refer to the processes *P.1* and *P.2* in the process model. Furthermore OOSA is partially supported by the *teamwork* CASE tool and commonly used in real-time applications, though the method does not have a rich description on either design steps or heuristic guidance.

² OOSA also supports object-oriented design by a language-independent notation known as OODLE, which includes class diagram, class structure chart, dependency diagram and inheritance diagram [Shlaer 91].

2.3.3 OBJECT-ORIENTED ANALYSIS/DESIGN (OOA/OOD)

Coad/Yourdon's object-oriented analysis (OOA) [Coad90] is also derived from the Yourdon entity-relationship model. It is a reasonably complete, practical method and supporting notation, suitable for commercial projects, though the distinguishing between class and object was not been made until the second edition [Coad 91a]. OOA proceeds in five stages:

- *subject* - decompose the problem domain into manageable subjects and describe them by different levels of DFDs;
- *object* - define real world abstractions as objects by using data analysis, which create a stable framework for analysis and specification;
- *structure* - identify classification and composition hierarchical structures, then represent them as *generalisation-specialisation* and *whole-part relationships* respectively;
- *attribute* - define attributes of each object by conventional data analysis, the instance connection is modelled by mapping object responsibilities (similar to that of CRC);
- *service* - equip each object with services (known as 'methods') by considering the behaviours of the object, the *message connections* between objects are also identified.

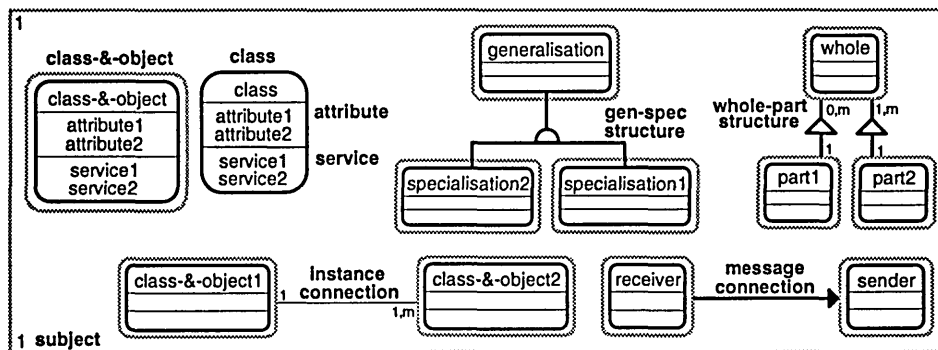


Figure 2.7 Coad/Yourdon OOA Notations

OOA introduces a less clumsy notation than Shlaer/Mellor's OOSA (refer to figure 2.7). In addition, OOA documents each class-&-object in a *specification template* and individual service is further specified by an *object state diagram* and/or a *service chart*. Coad/Yourdon's object-oriented design (OOD) adds four design components to OOA [Coad 91b]. These components allow design-specific issues to be included in the OOA diagrams:

- *problem domain component* - refine the products of OOA in the problem domain;
- *human interaction component* - design interaction, such as format of windows and reports;
- *task management component* - handle different types of tasks and their communications;
- *data management component* - provide infrastructure of storage and retrieval of object.

The main critique of OOA/OOD is that it does not really handle dynamics and the connection of services can only display thread of execution one at a time [Graham 94]. Nevertheless, the *OOATool* produced by Object International supports the denotation of the method.

From the meta modelling viewpoint, OOA/OOD does not describe design steps explicitly, although the *strategies* recorded at the end of the literature present the techniques of the nine structured activities (five activities from the OOA and four activities from the OOD). Each strategy defines the condition of the activity by the 'when to' statement(s) and the action as 'how to...' or 'what to...' statement(s). This is demonstrated by the analysis strategy of *identifying subjects* shown below. In fact, this information gives the significant guidance of the method and must be denoted in the method representation.

ANALYSIS STRATEGY - identifying subjects

Subject. A subject is a ...

How to select: Promote the name ...

How to refine: Refine subjects by using ...

How to construct: On the subject layer, draw each subject as ...

When to add: Add subjects once an overall map ...

2.3.4 NIELSEN OBJECT-ORIENTED DESIGN (NIELSEN OOD)

Nielsen OOD introduced a design method for real-time systems [Nielsen 88] and then it was expanded to a distributed design method [Nielsen 91]. The object-oriented flavour was only inserted in a later literature [Nielsen 92]. Nielsen OOD focuses more on specifying process distribution and message passing than concurrency or task structuring.

Unlike other methods described so far, Nielsen OOD is very much an Ada based method, the major aim of process abstraction is to map *process structure charts* (as shown in figure 2.8a) into *Ada Task graphs* for further transformation into Ada packages. Although Nielsen OOD method includes data abstraction to take in object-orientation, the definition of an object (as demonstrated in figure 2.8b) is only an encapsulated data structure and no inheritance is encountered.

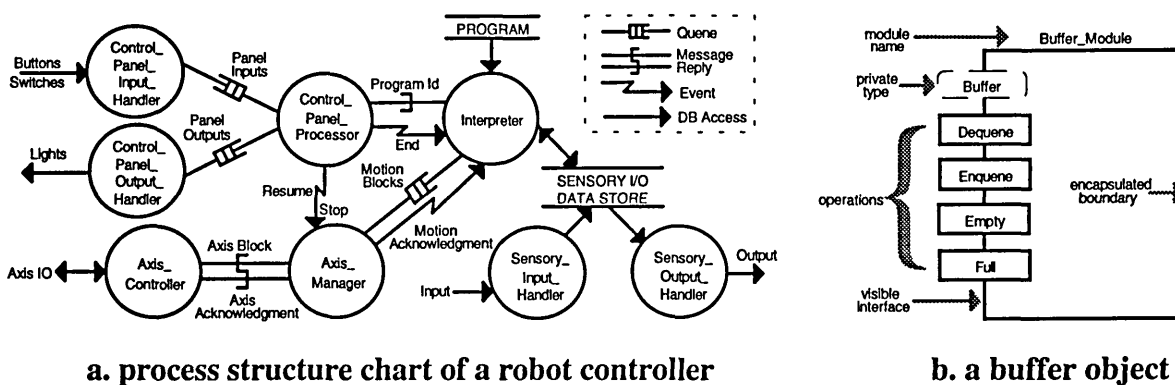


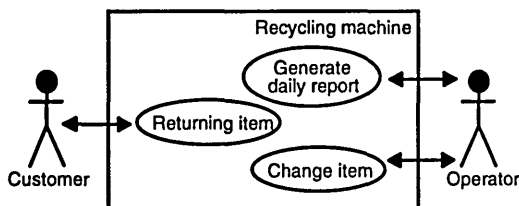
Figure 2.8 Nielsen OOD Notations

The design heuristics of Nielsen OOD are plentiful and they receive different emphasis in different literature. Although the development steps are summarised with the according representations (diagram and/or design language), there is no proper structure to arrange this guidance back to the corresponding design steps as in Coad's OOA/OOD. Later on in this chapter, another distributed real-time method, Codarts/DA, is described which uses a similar set of notations, but it associates design steps and heuristics in a more structured way.

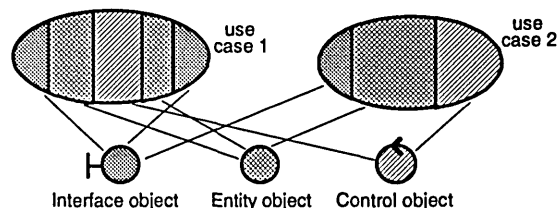
2.3.5 OBJECT-ORIENTED SOFTWARE ENGINEERING (OOSE)

Object-oriented software engineering (OOSE) is a subset of Jacobson's object-oriented method, Objectory, which is supported by his *OrySE* CASE tool. OOSE suggests dividing system development into three activities: analysis, construction and testing [Jacobson 92]. Each of these activities develop models: requirements and analysis models in analysis; design and implementation models in construction; and test model in testing. The OOSE underlying enterprise analogy is an *architecture* that is based solely upon the customised constructs below:

- *tool* - support all activities of the enterprise, i.e. architecture, method and process;
- *method* - make explicit procedures to be followed in applying architecture to projects;
- *process* - provide for scaling-up the method to a larger interacting activities and parties;



a. use case model for a recycling machine



b. mapping use cases to domain objects

Figure 2.9 Jacobson OOSE Notations

Many ideas of OOSE are similar to other OO methods, except the concept of use case. Use cases are descriptions of how users interact with a system, such as the *recycling machine* use case model shown in figure 2.9a. The domain objects, such as *interface object*, *entity object* and *control object*, can also be shared amongst different use cases as depicted in figure 2.9b.

Nevertheless, a list of methods are described with OOSE in the literature [Jacobson 92], including OOSD, OOSA, OOA/OOD, Booch OOD, HOOD, OMT and CRC (the last four methods are described later in this chapter). The concepts of these methods are mapped to OOSE and the activities are compared. These method evaluations give a better picture of the method, though a generic representation of all methods should make the job much easier.

2.4 CHOSEN METHODS

Since it is impossible to cover all software development methods in such detail within the limits of time, five methods are chosen deliberately for the study of meta modelling in depth. However, criteria can be used for selecting methods so that the outcomes are made generic to other methods. The criteria are as follows:

- The semantics must be encapsulated in a ‘distinct’ method, which provides a set of concise and precise presentable conceptual ideas (see section 2.1). Contrary, the techniques for developing actor-based systems are only given as strategies rather than conceptual models.
- The group of methods must not incline towards a particular programming paradigm or software development phase, so that the study is not biased toward certain sets of semantics. In our selections, they include structured method, object-based as well as object-oriented methods, and also cover both analysis and design development phases.
- The chosen methods should denote a wide range of software modelling techniques, so that they reflect a large spectrum of modelling features and method viewpoints.

The five selected methods are Booch OOD, Codarts/DA, HOOD, OMT and Ptech. Table 2.1 shows the basic features of software development with these methods. A ‘●’ denotes the feature is fully supported by the method, a ‘◐’ describes a partial support, whereas a ‘○’ means little support is given. If the cell is blank, there is no support from the method. These features are categorised into three groups in this order: development paradigms or phases, method viewpoints and meta model components³. Most of them are self explanatory.

Category	Feature	Booch OOD	Codarts/DA	HOOD	OMT	Ptech
Development Paradigms	Object-Oriented	●	○	◐	●	●
	Structured		◐			
	Analysis	○	●	○	●	●
	Design	●	●	●	◐	◐
Method Viewpoints	Object Model	●	○	◐	●	●
	State Model	●	●		●	◐
	Function Model	◐	◐	○	●	◐
	Real-Time	◐	●	●		
	Concurrency	◐	●	●	●	●
	Distributed	◐	●	●		
Meta Model Components	Product - Concepts	●	●	●	●	●
	Process - Activities	○	●	◐	●	◐
	Heuristic - Guidance	◐	●	◐	●	◐
	Graphical Fragment	●	●	●	●	●
	Textual Fragment	●	●	●	○	◐

Table 2.1 Basic Supported Features of the Five Chosen SDMs

³ The components of a meta model are made clearer later in this thesis. At the moment, they can just be considered as a set of concepts, activities and guidance provided by the individual method.

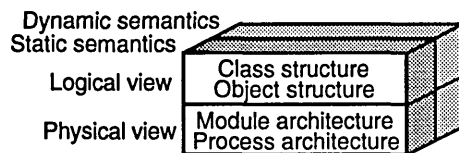
The five chosen methods⁴ are described comprehensively in the following subsections. Each method is given a general overview (see the accompanying references for detailed descriptions of the method), followed by consideration of their significance from the meta modelling viewpoints. The discussion emphasises the justification for choosing individual methods.

2.4.1 BOOCH OBJECT-ORIENTED DESIGN (BOOCH OOD)

The basic concept of Booch OOD [Booch 86] is a prime source of object-oriented methods and Booch himself is one of the pioneers in the technology [Graham 91]. The Booch method is based on objects as the unit of modularity of system design and indicates their services by relationships and message handling. In the second version the method is revised, with emphasis on object-oriented analysis and extended notations for dynamics [Booch 94]. However, the Booch OOD discussed in this thesis is the earlier [Booch 91] version. The method is supported with an automated CASE tool - *ROSE*⁵, which allows code generation to programming languages such as Ada and C++.

OVERVIEW

Booch OOD suggests breaking down the complexity of the real world by abstraction, encapsulation and inheritance [Booch 91]. The logical view of a system is shown by the class and object structures, whereas the physical view is denoted in the module and process architecture (figure 2.10a). Each model has a corresponding diagram and template(s). In addition, the dynamic semantics are also described in the method. The overall software design process is comprised of four high level activities (figure 2.10b).



a. the models

- identify the classes and objects at a given level of abstraction
- identify the semantics of these classes and objects
- identify the relationships among these classes and objects
- implement these classes and objects

b. the process

Figure 2.10 Booch OOD Models and Process

[Booch 91] also suggests that any technique could be used in the OOA phase, but a form of layering is recommended, that is organising related classes into *categories* (figure 2.11a). The *class* and *object diagrams* depict the logical static design with the well-known 'cloud' shape

⁴ These methods were chosen because they had immediate relevance to the collaborative company.

⁵ ROSE is a trademark for Rational, which is Booch's consulting company.

icon to show the *classes* (dotted), *class utilities* (shaded) and *objects* (solid line). Booch OOD has a rich set of relationships: the *use*, *instantiation*, *inheritance*, *metaclass* and *undefined* class relationships are indicated by different kinds of arrows (figure 2.11b), whereas the object relationships are shown as *inside systems* and *outside systems* with *visibility symbols* and *synchronisation symbols* for the *messages* (figure 2.11c).

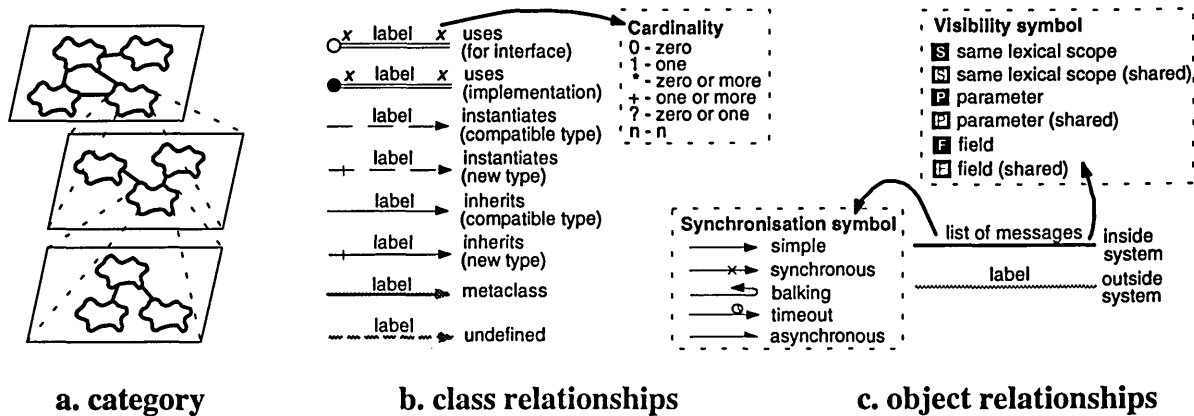


Figure 2.11 Booch Class Diagram and Object Diagram Notations

The physical design gives detailed access or processes of the classes and objects. Booch OOD distinguishes between classes with modules. Each module corresponds to a program segment and it is denoted in a *module diagram*, whereas the *process diagram* shows the communication relationships between physical devices and processors. The dynamics of a system is the key issue in OOD and Booch addresses them by the *state transition diagram* and the *timing diagram*. The former depicts the dynamics of classes, and the latter denotes the instance level dynamics [Booch 91]. However, they may only be manageable for reasonably small systems, and become impractical in most commercial systems. Furthermore, each class, operation, object, message, module, process etc. is associated with a *structured text template*, which documents the semantic details. For instance, the *class template* is shown below - the operation declaration points to a list of *operation templates*.

Name:	identifier
Documentation:	text
Visibility:	exported / private / imported
Cardinality:	0 / 1 / n
Hierarchy:	
Superclasses:	list of class names
Metaclass:	class name
Generic parameters:	list of parameters
Interface Implementation (Public/Protected/Private):	
Uses:	list of class names
Fields:	list of field declarations
Operations:	list of operation declarations
Finite state machine:	state transition diagram
Concurrency:	sequential / blocking / active
Space complexity:	text
Persistence:	persistent / transitory

SIGNIFICANCE

The Booch concept itself is extremely rich, since it covers both logical and physical design of a system, i.e. from specification model to implementation model. Although there is no direct description about real-time, concurrency and distributive features, they are partially supported by the *class declaration* and *message synchronisation*. The main weakness of Booch OOD is that the dynamics are tagged, the notation is not sufficient to describe how complex activities trigger state changes and critical object life-times in a large system [Graham 91]. Also, Booch tries to define the control structure of a system by the *use relationships* as achieved by HOOD (described next), but it leads to complex diagrammatic representation.

From a meta modelling viewpoint, Booch has comprehensive documentation of concepts, and also stresses the importance of relating graphical specification to structured text. However, the description of the design process is very brief and it is particularly important to underline this in method representation. Furthermore, Booch OOD is deliberately chosen to reinforce the comparison with OMT (section 2.4.3), which is another leading object-oriented method.

2.4.2 HIERARCHICAL OBJECT-ORIENTED DESIGN (HOOD)

Hierarchical OOD (HOOD) [Robinson 92] is an architectural design method, which is very much directed at Ada development and is developed for the European Space Agency. HOOD has resulted from merging experience on the Booch OOD and Abstract Machine.

OVERVIEW

HOOD enforces structuring of objects by three principles [HOOD 91]:

- *abstraction, information hiding and encapsulating principles*: An object is defined by its services and the internal structure is hidden to the user (see figure 2.12a). Each service is described in an *operation control structure* (OPCS), whereas the behaviour of the object is given in the *object control structure* (OBCS).
- *hierarchy principles*: There are two types of object hierarchies. In the *parent-child hierarchy*, a parent decomposes into child objects and its functionality is provided by the children through *implemented-by links*. In the *seniority hierarchy*, a senior object uses the operations of junior objects through *used-by links* (figure 2.12b).
- *control structuring principles*: Operations of objects are activated through control flows corresponding to the execution of logical processors on an underlying target machine. These flows may operate simultaneously in an object by synchronous or asynchronous or timeout requests in an *active object* (figure 2.12c). However, a *passive object* shall not use a *constrained operation* of an active object and it has no OBCS.

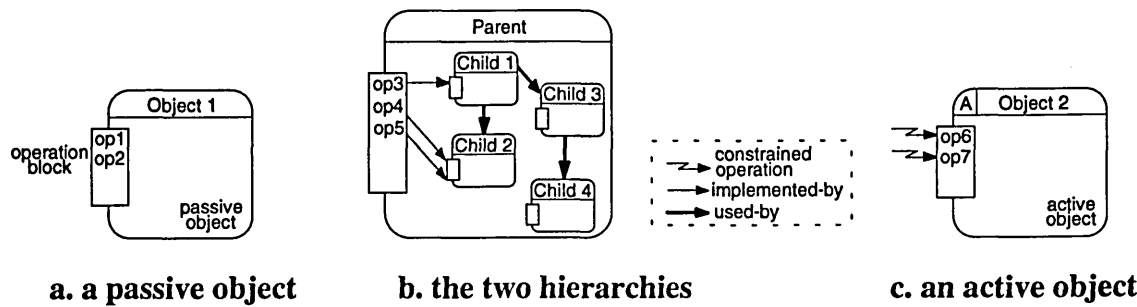


Figure 2.12 HOOD Notations

Apart from the *HOOD diagrams* illustrated above, HOOD also supports a program definition language like structured text, known as an *object description skeleton* (ODS). ODS provides the transition from the formal but incomplete HOOD diagram to the Ada source code, and it allows consistency and completeness checks. The basic set of sections in ODS is shown below and it must be noted that the content is dependent upon the *object type* (see section 5.3.2.6).

```

OBJECT Object_name IS Object_type
    [Class_formal parameters]
    [PRAGMA TARGET_LANGUAGE language]
DESCRIPTION object_description_text
IMPLEMENTATION_OR_SYNCHRONISATION_CONSTRAINTS constraints_text
PROVIDED_INTERFACE provided_interface_definitions
OBJECT_CONTROL_STRUCTURE obcs_synchronisation
REQUIRED_INTERFACE required_interface_definitions
DATAFLOWS dataflow_definitions
EXCEPTION_FLOWS exception_flow_definitions
INTERNALS internal_definitions
OBJECT_CONTROL_STRUCTURE obcs_implementation
OPERATION_CONTROL_STRUCTURES opcs_definitions
END_OBJECT Object_name

```

SIGNIFICANCE

HOOD is the only method that models control explicitly, by using the ‘used-by’ relationships and OBCS. This is a very structured way of documenting the control thread in a design method. HOOD also introduces design rules to reinforce the modelling constraints. Unlike the Booch OOD, the hierarchical structure minimises object interdependencies and produces highly cohesive components with low coupling. However, HOOD’s graphical formalism is both incomplete and inconsistent, it does not express which operations are used by which object, and which data are encapsulated by objects. The notation depicts subprograms as objects and control structure is also shown as a subprogram which passes control to a task.

HOOD is only an *object-based method*, since it does not support polymorphism, inheritance and genericity, though they are important features to provide maintainability and reusability [Meyer 88]. Besides, HOOD is a tailor-made method for the Ada language, which means the method concepts have a direct influence in the implementation phase of software development.

Similar to Booch OOD, HOOD employs four basic design phases from *problem definition* to *formalisation of solution*. HOOD does not denote state model or function model as in most methods, although the HOOD diagram expresses certain operational dependences. Moreover, HOOD's concept structure (in the ODS) is based on the *object type*, which is a speciality that must be expressed in method representation. The interrelationship between the two highly integrated products, *HOOD diagram* and *object description skeleton* is also the reason that HOOD is chosen to show the adaptability of meta modelling.

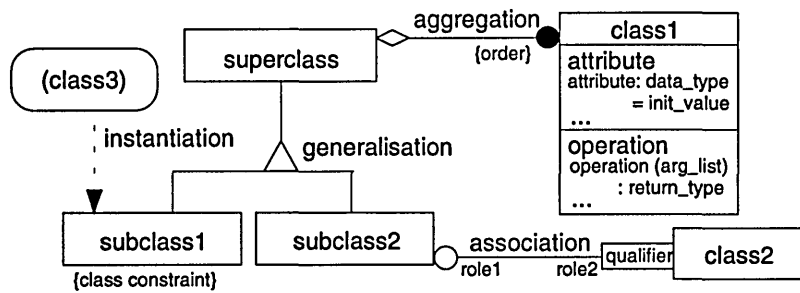
2.4.3 OBJECT MODELLING TECHNIQUE (OMT)

Object modelling technique (OMT) [Rumbaugh 91] from General Electric (GE) is another well-known OO method. Although most people refer to it as an OOA method, OMT actually embodies a software design phase by giving detailed design heuristics. The method is widely used in commercial projects and it becomes a standard method for research experiments, such as [Rossi 95] and [Plihon 95]. In addition, a number of CASE tools support the OMT notation. The popularity of OMT makes it a necessity to be included in this research.

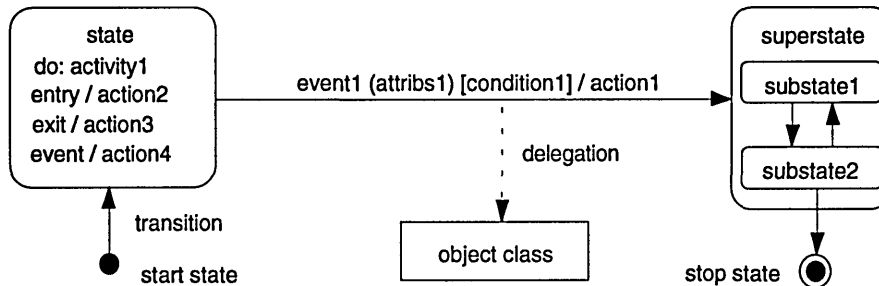
OVERVIEW

The core of OMT is based on three different software viewpoints, which are expressed as distinct models and evolved throughout the system development. The three models are:

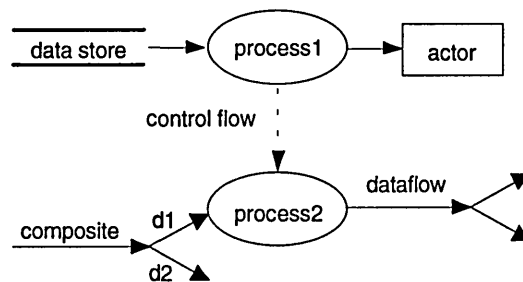
- *Object Model* (OM) describes the static structure of the objects in a system and their relationships by *object diagrams*. It consists of similar notations as Coad/Yourdon OOA with a *data dictionary*. Moreover, OM indicates attribute types and operation parameters of a class. Constraints may be inserted to classes and operations, whereas association is enriched by role names and qualifier (figure 2.13a). Some new concepts are also added, such as *discriminator* in generalisation, *link attributes* and *ternary relationships*.
- *Dynamic Model* (DM) describes the control aspects of a system by *state diagrams*. OMT advances Booch state transition diagram by introducing extensive features to both *state* and *event*. A state performs a durable 'do' *activity* which is surrounded by instantaneous *entry* and *exit actions*, whereas an event may occur with *attributes* and cause *actions* that may be guarded by *conditions* (figure 2.13b). In addition, an event may delegate action to other objects with *associated attribute*, and a *transition* without activity is also represented as an *automatic transition*.
- *Functional Model* (FM) describes the data value transformations within a system by *data flow diagrams*. This is borrowed from the structured methods, but FM enhances the notations by suggesting *control flow* between *processes* and by *depicting duplication, composition* and *decomposition* of *data flows* explicitly (figure 2.13c).



a. object modelling notations



b. state modelling notations



c. functional modelling notations

Figure 2.13 OMT Model Notations

The method consists of building the models of an application domain and then adding implementation details to it during the design stages. The programming is a relatively minor and mechanical part of the development cycle. The four major stages are:

- **analysis:** build the three models from a problem statement (existing requirement specification) to show its important properties in a real-world situation;
- **system design:** organise the target system into subsystems based on both the analysis structure and the proposed architecture such as global resources and control structures;
- **object design:** add details to the design model in accordance with the strategy established during system design, such as the data structures and algorithms needed to implement;
- **implementation:** translate object design into a particular programming language, database, or hardware implementation.

OMT suggests an iterative incremental approach of software development, each stage being structured into a number of recursive steps. Moreover, OMT is particularly rich in heuristics for concepts and activities. For example, the following criteria are given to help obtain the right classes in the object modelling. This is important information provided by the method and must be formally presented in the meta model.

- *Redundant classes.* If two classes express the same information, ...
- *Irrelevant classes.* If a class has little or nothing ...
- *Vague classes.* A class should ...
- *Attributes.* Names that primarily describe individual objects ...
- *Operations.* If a name describes an operation ...
- *Roles.* The name of a class should reflect its intrinsic nature ...
- *Implementation Constructs.* Constructs extraneous to the real world should be ...

The method also provides some heuristics for mapping OMT design into implementation, such as software systems in OO languages, non-OO languages and relational databases. However, this is outside the scope of this research.

SIGNIFICANCE

Compared with other OO methods, OMT has a relatively complicated and detailed notation with strong roots in traditional structured methods [Graham 94]. The method is relatively language-independent, though it is often associated with C++. Object relationships in OM are extremely rich and each concept is documented comprehensively in the data dictionary. DM is a useful tool to denote concurrency in three different ways: aggregation of state diagrams; aggregation of states and concurrent behaviour of a state. FM is normally used as a top level DFD or context diagrams in practice. The attributes and operations discovered in the DM and the FM are referred back to the OM.

OMT is a well structured method with clear definitions on the product and detail descriptions of each activity. These techniques are enhanced with extensive criteria. Although OMT does not describe real-time or distributed features, the good balance of various modelling aspects makes it an ideal illustration for method representation. Furthermore, the concise and precise definition of method semantics allows a perfect example in distinguishing various component types in the meta model.

2.4.4 PTECH

Ptech, from Associative Design Technology [Martin 92] [Martin 95], is a proprietary set of methods and tools covering both analysis and design. The method is based on the metaphor of process engineering as the production of systems by assembling reusable components. Hence Ptech is process-oriented rather than strictly object-oriented [Graham 94]. The method's CASE tool can also generate code for object-oriented languages, such as C++.

OVERVIEW

Ptech combines the process-driven view with the abstraction features of more data-centred, object-oriented design and some ideas from set theory and artificial intelligence. It describes the object structure and the object behaviour separately by object schemas and event schemas. The object flow diagram serves to provide an overview of the system.

- *Object schema* represents the structure knowledge as the static conceptual configurations that are applied to objects. This is by means of depicting object-relationship, generalisation hierarchy and compose-of structures in a single diagram. Thus an object schema shows the object type, association, classification and composition (figure 2.14a). In Ptech, the attribute type of an object defines its state. When the attribute value changes, the object shifts from one object type to another object type.
- *Event schema* describes how the structure applies to objects over time. It must be expressed in terms of object schema, because events change the state of given types of objects. An event schema includes state transitions, event types, trigger rules, control conditions and operations (figure 2.14b). Ptech identifies transitions by event types such as object classification, declassification, reclassification, creation, termination etc, from which the corresponding operations are determined. Each trigger rule is embedded with underlying object type(s) defined in object schema and represented as functions.
- *Object flow diagram* gives a high-level functional view of a process landscape [Martin 95]. It shows the key enterprise activities linked by the products that activities produce and exchange. The links are expressed as object flows, which basically denote the production or consumption of a product and show the precedence. In addition, each product may decompose to an object schema, whereas an activity may extend to an event schema.

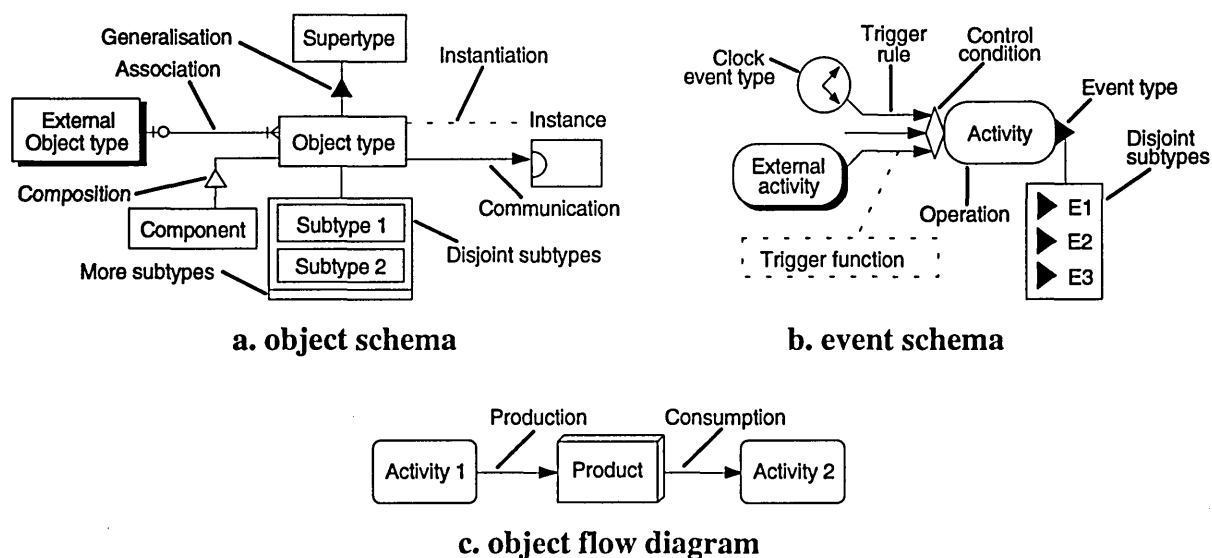


Figure 2.14 Ptech Notations

In the method processing terms, Ptech consists of four main activities:

- *Object structure analysis* (OSA) involves building object schemas to define the kinds of objects and the way in which they associate.
- *Object behaviour analysis* (OBA) uses the event schema to model what happens to the objects over time. OSA and OBA are closely related; they are not done separately but develop together to form integrated models and designs.
- *Object structure design* (OSD) adds implementation dependent aspects to the results obtained in OSA, such as data structure and operation specification.
- *Object behaviour design* (OBD) adds the translated object design into a particular programming language, database, or hardware implementation.

SIGNIFICANCE

Ptech is the only object-oriented method that is based on set theory. Its emphasis is also heavy on software processes (event operations) modelling based on domain concepts (object types). An object is defined as a collection of states represented by attribute type, whereas an event causes the underlying object of a trigger to change state. Each relation in the object schema and the control condition in the event schema are described as textual functions, which are significant semantics that must also be described in the meta model. Although most object-oriented methods use state models to describe the internal dynamics of objects, Ptech event schemas are found to give a better view of global dynamics [Graham 94]. This is because an event schema combines both purposes of state transition diagram and data flow diagram. Ptech is also rich in design concepts, processes and heuristics. Although the close relationship between the two schemas requires a CASE tool for maintenance, it is another good illustration for showing how the dependency between concepts is handled efficiently in meta modelling.

2.4.5 CODARTS/DA

Codarts stands for COncurrent Design Approach for Real-Time Systems [Gomaa 93]. It is a development method that originated from *Darts*. Both *Darts* and *Codarts* have been greatly influenced by the DeMarco SA (section 2.2.1) and the Real-Time Structured Analysis (RTSA) [Ward 85]. *Codarts* provides two major extensions to *Darts*. Firstly, an alternative approach, *Cobra*, is used to address the limitations of RTSA. **Cobra** (Concurrent Object-Based Real-Time Analysis) emphasises the decomposition of a system into subsystems that provide a set of services to support objects and functions. Secondly, the design of distributed applications is supported, so the method is normally known as *Codarts* for Distributed Applications or **Codarts/DA**. It is a general purpose method that is not oriented towards a particular language, though *Adarts* from the same family is based on Ada development.

OVERVIEW

Codarts/DA notations (figure 2.15) are fairly close to that of Nielsen OOD, but the emphasis is very different. The former emphasises a progressive transformation of real-time design with a set of structured criteria, whereas the latter is based on data and process abstractions. The seven design steps of Codarts/DA are shown below [Gomaa 93]:

1. *Develop environmental and behavioural model of system*: Cobra is used for analysing and modelling the problem domain. It provides guidelines for developing the environmental model based on the system context diagram (figure 2.15a) and subsystem structuring criteria for system decomposition. Objects and functions within a subsystem are determined by the supported criteria and they finally interact with each other using event sequencing scenarios by a behavioural approach (figures 2.15b & 2.15c).
2. *Structure the system into distributed subsystems*: This is an optional step taken for distributed systems. Codarts provides criteria for structuring and configuring a distributed application into subsystems, which communicate by means of messages.
3. *Structure the system (or subsystem) into concurrent tasks*: Determine the concurrent tasks by applying the task structuring criteria, and resolving intertask communication and synchronisation interfaces. This is also applied to each subsystem of a distributed design.
4. *Structure the system into information hiding modules*: Determine the information hiding modules in the system by applying the module structuring criteria. A module aggregation hierarchy is created in which the information hiding modules are categorised.
5. *Integrate the task and module views*: The task view and module view are integrated to produce a software architecture, and shown by task architecture diagrams (figure 2.15d).
6. *Define component interface specification*: The component interface specifications are defined for tasks and modules. These specifications represent the externally visible view of each component.
7. *Develop the software*: This stage develops detailed design, coding and testing of the software with the identification of system subsets to be used for each increment.

The progressive development of Codarts/DA is illustrated by the asynchronous device I/O example shown in figure 2.16. The *cruise control lever* device reads inputs and sends them as *cruise control requests* to the control object (step 1: figure 2.16a). From a task structuring view (step 3: figure 2.16b), the object *cruise control lever* is structured as an asynchronous *monitor cruise control input task* and the *cruise control task* receives the queued messages. From a module structuring view (step 4: figure 2.16c), the I/O device is structured as a *cruise control lever DIM* (device interface module). Finally, from the combined views (step 5: figure 2.16d), the *cruise control lever DIM* is placed inside the *monitor cruise control input task*.

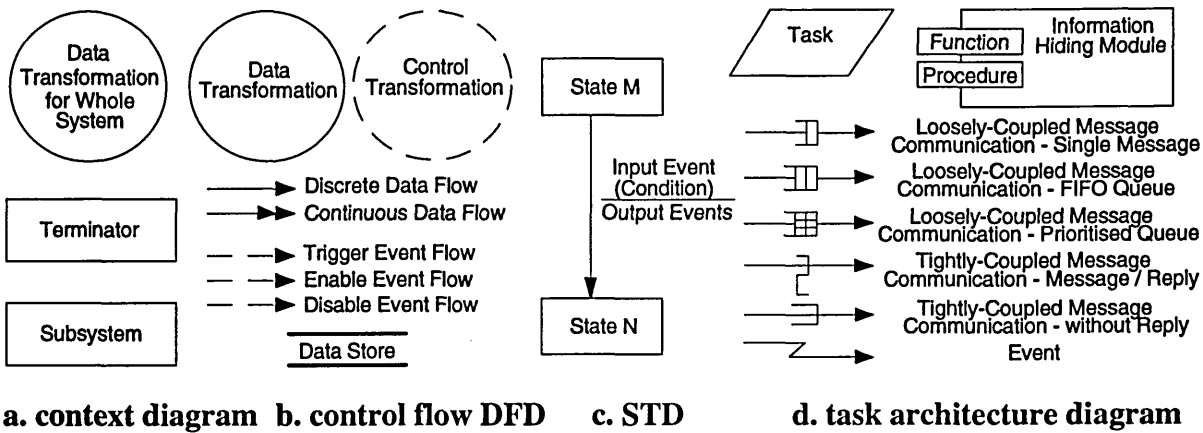


Figure 2.15 Codarts/DA Notations

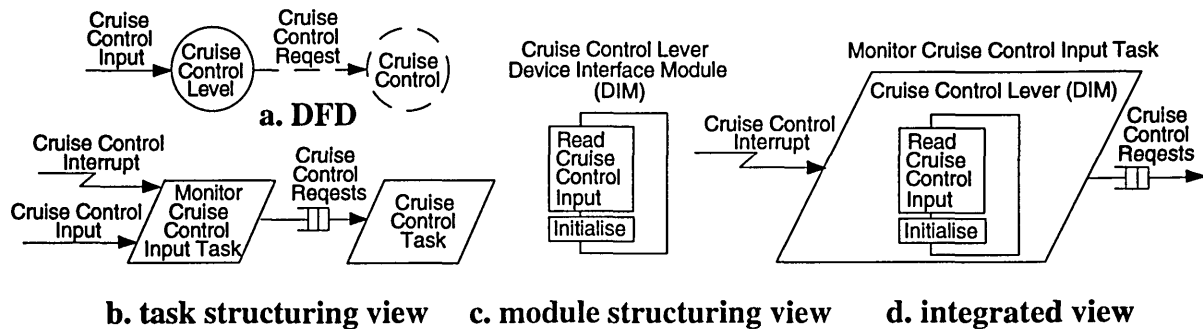


Figure 2.16 Codarts/DA Cruise Control Device I/O

SIGNIFICANCE

Codarts/DA is a structural design method that keeps concurrent objects and functions in mind. The method is not considered as an object-oriented method, since both abstraction and inheritance features are not encountered. Although it suggests ‘flattening’ the inheritance hierarchy manually, the information hiding modules are software modules rather than tangible objects. However, Codarts/DA is a real-time, concurrent, distributed method with well-structured techniques, these features are normally subsidiary or even omitted in object-oriented methods.

The most significant point of Codarts/DA is the comprehensive description on design process and heuristics. It is specially chosen to express the heuristic richness of the development method. Codarts/DA places a lot of emphasis on describing various *structuring criteria*, such as criteria for subsystem, object, function, task and module structuring. The task structuring criteria are extensively presented to guide the development of task architecture in a real-time distributed application. The detail design sequence is also well-documented with essential illustrations. Finally, it must be mentioned that the emphasis of this research is on the Codarts/DA design phase and not on the Cobra analytical phase.

2.5 OTHER METHODS

The methods described so far are normally regarded as structured methods or object-oriented methods. The rapid development of these types of methods may be probably due to the requirements of the corresponding programming paradigms and applications. This next section looks at some methods that do not classify (or are difficult to classify) in groups.

2.5.1 CLASS, RESPONSIBILITY AND COLLABORATION (CRC)

CRC is a responsibility-driven design method [Wirfs-Brock 90], which is based on a set of class and subsystem cards. Each class is described with subtyping relationships as well as its responsibilities and collaborations to other classes in the problem domain (see the card below for a *withdrawal transaction* class of an *ATM* example). The *hierarchy graph*, *venn diagram* and *collaborations graph* are used to depict these relationships (see figure 2.17).

This method is particularly good at documenting and teaching object-oriented design. It is a simplistic but very practical method. Although CRC has a bare set of notations and does not cover all aspects of a software model, it is a quick and easy way to document a specification, especially for requirements engineering. Even Booch OOD suggests employing CRC cards to denote class relationships in the early analytical stage [Booch 91].

Class: Withdrawal Transaction (Concrete)
Superclasses: Transaction
Subclasses: none
Hierarchy Graphs: page 4
Collaborations Graphs: page 8
Description: A request by a bank customer to withdraw funds from an account.
Contracts
 8. Execute a financial transaction
 This contract is inherited from Transaction.
Private Responsibilities
Prompt for the amount: uses User Interface Subsystem (4)
Withdraw funds: uses Account (1), User Interface Subsystem (9)

Contract 8: Execute a financial transaction
Server: Transaction
Client: ATM
Description: This contract supports executing financial transactions.

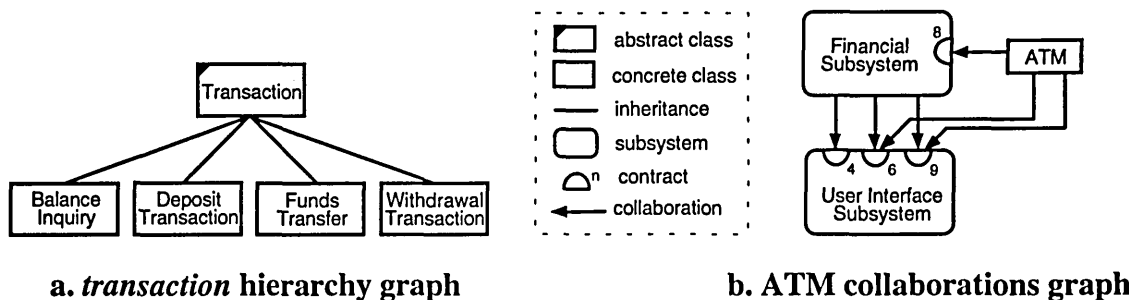


Figure 2.17 CRC: ATM Graphs

2.5.2 ASTS DEVELOPMENT METHOD 3 (ADM3)

Firesmith's ADM3 is an integrated object-oriented method [Firesmith 93], which originated from an Ada-oriented method known as ASTS. The ADM3 method covers a large spectrum of software development (from system requirement to analysis and then logical design). It borrows ideas from semantic networks and has an emphasis on real-time systems. The method denotes static architecture and dynamic behaviour by six incorporated models, where each model is associated with a set of diagrams, as shown in table 2.2. The six models are briefly described below:

- *assembly model* - a static view of the entire assembly in terms of terminators, subassemblies and the relationships among them;
- *object model* - a static view of the architecture of a subassembly by the existence, abstraction and visibility of its component objects and terminators;
- *class model* - a static view of inheritance architecture by the existence and abstraction of its component classes and the 'has subclass' and 'has instance' relationships;
- *state model* - a dynamic view of the objects and classes by their states, transitions, modifier operations and exceptions;
- *control model* - a dynamic view of subassembly, its major threads of control, objects and classes by their attributes, operations and control flows;
- *timing model* - a temporal view of the subassembly by the timing of messages passed within and between objects.

DIAGRAM APPLICABILITY		AREA OF CONCERNS			
		STATIC ARCHITECTURE		DYNAMIC BEHAVIOUR	
S C O P E	ASSEMBLY	assembly model	context diagram assembly diagram	control model timing model	oo control flow diagram timing diagram
	SUBASSEMBLY	object model class model	general semantic net interaction diagram composition diagram classification diagram	control model timing model	oo control flow diagram timing diagram
	AGGREGATION HIERARCHY	object model	composition diagram		
	INHERITANCE HIERARCHY	class model	classification diagram		
	THREAD OR SCENARIO	object model	interaction diagram	control model timing model	oo control flow diagram timing diagram
	CLASS	object model class model	general semantic net interaction diagram classification diagram	state model control model timing model	state transition diagram state operation table oo control flow diagram timing diagram
	OBJECT	object model class model	general semantic net interaction diagram classification diagram	state model control model timing model	state transition diagram state operation table oo control flow diagram timing diagram

Table 2.2 ADM3 Applicability of Models and the Associated Diagrams

ADM3 allows the developer to specify the diagram notations by the *ASTS diagramming language* (ADL), and it also introduces *Object-Oriented Specification and Design Language* (OOSDL), to document the design with specification of individual assembly, class, object, operation and state etc. (refer to Booch templates). The OOSDL format is rather like a C++ header file, for instance the specification of a *SET_OF_TRAFFIC_SIGNALS* object as below [Firesmith 93]:

```

object SET_OF_TRAFFIC_SIGNALS
  parent subassembly INTERSECTION;
specification
  message CHANGE_PRIMARY raise LIGHT_FAILED, POWER_FAILED is synchronous;
  message FLASH_PRIMARY raise LIGHT_FAILED, POWER_FAILED is synchronous;
  ...
  exception LIGHT_FAILED;
  exception POWER_FAILED;
end;

```

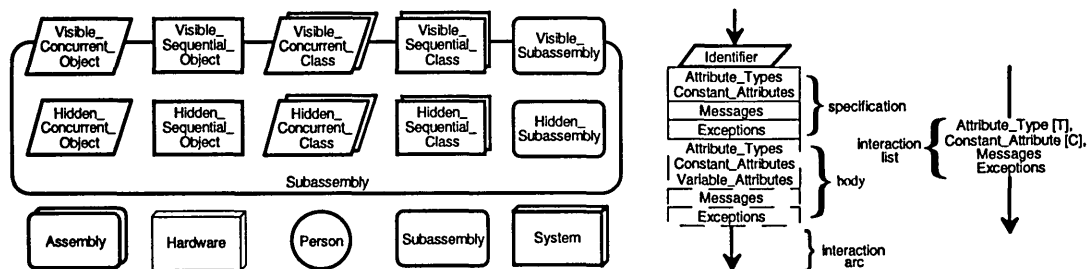


Figure 2.18 ADM3 Icons for Interaction Diagram

ADM3 has an overwhelming set of notations, such as the icons used for the *interaction diagram* shown in figure 2.18. Thus, it was decided not to use it for the detailed experimentation in this research. Moreover, ADM3 is considered as a complex ternary method reminiscent in some respects of both Booch OOD and OMT. In other words, the major semantics of ADM3 have been concealed by the two chosen methods. Since ADM3 is an *integrated* object-oriented method, it is described in this section rather than in section 2.3.

2.5.3 THE FUSION METHOD (FUSION)

The Fusion method is another integrated OO method [Coleman 94], which claims to combine the best aspects of several methods. The principle influences on Fusion are depicted in figure 2.19. The software development phases are described individually as follows:

The *analysis phase* is inspired by the OMT analysis models. Unlike other analysis methods, Fusion objects have neither interface nor dynamic behaviour in this phase, since it claims that object interfaces are likely to be invalidated by the later global decisions regarding the overall system and its behaviour. Fusion also does not use state diagrams during analysis, because they exhibit dynamics and the descriptions are cumbersome. The major influences are:

- *Object model*: the object model in OMT with relatively minor notational differences;
- *Operational model*: OMT functional model with pre- and post- condition specifications from formal methods;
- *Life-cycle model*: the regular expression of life cycles from JSD (section 2.2.2).

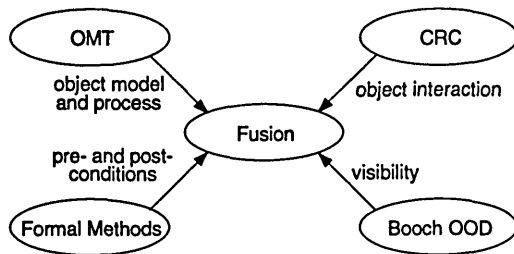


Figure 2.19 Influences on the Fusion Method

```

class Gun
  attribute constant clutch: Clutch
  attribute constant pump: Pump
  attribute trigger: TriggerStatus
  attribute status: GunStatus
  method depress_trigger()
  method release_trigger()
  method enable()
  method disable()
  method is_enabled(): Boolean
endclass

```

Figure 2.20 Class Description

The *design phase* is based on the CRC and Booch OOD methods, where a systematic design process is added. Fusion provides class descriptions by collating information from the object model and the following design models:

- *Object interaction graphs*: the communication information scattered across CRC cards;
- *Visibility graphs*: the visibility information between objects from Booch object diagrams;
- *Class descriptions and inheritance graphs*: a template notation to document the syntax of classes (as the example in figure 2.20) and graphs for recording inheritance.

In the *implementation phase*, Fusion claims that most applications can be directly resolved from the class descriptions. For instance, the class description of a *Gun* object shown in figure 2.20 (above) can be easily mapped to a C++ header file. The developer can fill in the details of each operation with supports from various graphs in the design phase. Moreover, Fusion suggests recording extra class semantics by a language-independent notation, such as extended state machine with pre- and post- conditions.

Fusion aims to meet the needs of software development which are not met by any existing methods. It requires a strong commitment to being systematic and rigorous, but this may not fit the technical requirements of an object-oriented development [Coleman 94]. One major drawback of Fusion is that the method needs to be modified for use on distributed, real-time object-oriented systems. Basically, Fusion picks the proficient tools provided from the influential methods, and mixes their semantics together by resolving the context dependence, that is how the semantics of the tools relate to one another. This advocates an important technique for meta modelling adopted in this research (see next chapter). Nevertheless, since there are no distinct features with respect to the five selected methods, Fusion is not included for detailed experimentation.

2.5.4 THE KADS METHOD

The methods described so far are mainly software development methods, although there are large sectors of development methods not addressed. This subsection addresses one of the growing areas for methodologies, that is the knowledge-based system analysis and design methods. The KADS and MIKE approaches shown in appendix C are the classic examples. KADS is described here since it is closely related to our method representation (chapter eight) and knowledge acquisition (chapter ten). This area of technology has gradually become an important ‘software’ development method⁶.

KADS [Schreiber 93] is a principled approach towards knowledge-based system (KBS) development. The approach can be characterised by two main principles that underlie the process of building KBSs: firstly the principle of multiple models; and secondly the principle of knowledge-level modelling as a way to describe problem-solving expertise in an implementation-independent way. The KADS development process is comprised of seven models, where the crucial activity of knowledge engineering is based on constructing the *model of expertise*. The KADS approach also distinguishes four knowledge categories according to their epistemological distinctions. These different types of knowledge are known as the *domain knowledge*, the *inference knowledge*, the *task knowledge* and the *strategic knowledge* (refer to appendix C for more details).

In addition, KADS supports both the graphical representation and the textual representation of various layers of knowledge. The diagrammatic notations are depicted in their respective structures (see the figures in appendix C). These graphical representations are accompanied by textual representation, such as the *domain description language* (DDL) for specifying the domain knowledge. The DDL statements can be compiled into a set of Prolog predicates, which permits further experimentation or execution on the knowledge.

KADS is a complex method. Both products and activities are well-structured with many techniques and much guidance. KADS is still a developing method, so it has not been chosen for further method experimentation. However, it is still of interest to discuss the KADS approach itself as a ‘meta modelling method’. The following points are found to be significant from the meta modelling perspective:

- The KADS approach is intended for knowledge engineering in general, but the main focus of this research falls on the method knowledge modelling only. A few differences are encountered in this change of domain scope, such as the categorisation of knowledge and the knowledge acquisition process itself. These distinctions are elaborated briefly in the following points and details can be found in the description of the meta model later.

⁶ Ultimately, a knowledge-based system or an expert system is a piece of software so the corresponding development method can also be considered as a software development method.

- The domain layer directly reflects the concept structure required in a meta model. *Concepts, properties* and *relationships* are important constructs in modelling products of a method. Apart from adjustment of terminology, such as a *structure* may refer to a *fragment* (a group of correlated concepts), the relationships between concepts are loosely defined. The properties of relationships, such as *roles* and *cardinalities* must also be defined specifically.
- The inference layer gives significant ideas about modelling activities. The *knowledge source* denotes operations in the process and the meta-classes describes the pre- and post-conditions of the operation. However, it is found that the process of a method should lie between the inference layer and task layer, which respectively define the functional and control viewpoints of knowledge engineering. Therefore there is a direct connection from concepts as products to tasks as processes. Furthermore, the task decomposition (hierarchical task structure) in the KADS approach is not flexible enough to denote the creativity of software engineering that include iterative, recursive development processes.
- The strategic layer concerns the problem analysis in method evaluation. This is not the emphasis of this research, but is discussed as speculated work in chapter twelve.
- The knowledge acquisition of KADS involves three main activities: *eliciting* the knowledge in a formal (usually verbal) form, *interpreting* the elicited data using some conceptual framework, and *formalising* the conceptualisations in such a way that the program can use the knowledge. The activities require the direct involvement of domain expert(s), such as in structured interviews. In method engineering, this is found to be impractical, since it is difficult to identify ‘method experts’ apart from the ‘method founders’ themselves. In addition, interviewing software developers seems to be an ineffective way of method knowledge transfer due to the inclination towards their problem domain and/or work environment. Chapter ten addresses this issue by introducing a different set of method acquisition media and elicitation techniques.

From the description above, KADS is a KBS ‘development method’ rather than a set of conceptual ideas or techniques (refer to section 2.1). It is comprised of extensive method concepts, design activities and heuristic guidance. KADS also supports comprehensive graphical and textual notations, with the potential of developing automated CASE tools to manage and manipulate the compiled Prolog clauses. Besides, some database management systems (DBMS) have also started to provide development methods: the *InfoDesigner* tool is available for supporting *Microsoft Access* database development [Perschke 93]. Nevertheless, the main concern of this research is on software development methods rather than methods in general. Most other development methods are either poorly documented or too immature to discuss systematically, so they are left as future works for a wider scope of method investigation. Hereafter the term ‘method’ refers to ‘software development method’.

2.6 SUMMARY OF INVESTIGATION

From the investigation of software development methods, the following points are noted:

- All methods incline towards a certain **aspect of software development**, such as functional decomposition by levelled-DFDs (such as DeMarco SA, JSD) and abstract data types in most object-oriented methods (Booch OOD and OMT). Some methods focus upon message passing (OOA and OMT), some emphasise task structuring (Codarts/DA) and some process engineering (Ptech). There are even methods based upon a particular programming language (HOOD and Adarts for Ada development). In terms of modelling, some methods denote the structural view (CRC, OOA), some indicate the behavioural view (OMT, HOOD) and some only the functional view (JSD). No single method is perfect for all software purposes or even ideal for a particular application. Therefore, the best solution is to provide method integration and support multiple viewpoints on the problem domain. In addition, the advantages of a generic model for method evaluation and comparison are promoted (OOSE). These are the main aims of method engineering.
- Each method has its own set of **terminology**. A single term may differ in various methods and different terms may imply the same thing in a single method. For instance, *object* and *object-oriented* have multiple meanings across methods (Booch OOD, HOOD and Ptech), whereas *operation*, *function*, *procedure*, *service*, *process*, *method* may be used interchangeably in a method (Booch OOD). This problem also occurs in the meta level, for example *heuristics* may be referred to as *criteria* (Codarts/DA), *guidance* (OMT) or *design rules* (HOOD), whereas *notation* may be used to describe a symbol (OMT), a method (OOSA) or a fragment (Fusion). This problem is addressed for the method level in chapter eight (method representation); and for meta level this thesis provides a glossary (appendix A) with definitions of the terms used.
- The presentation techniques in the methods provide significant points about meta modelling, for instance using DDL-like language in structuring concepts (KADS); using pre- and post- conditions in structuring tasks (SSADM); and using definitive clauses in structuring guidance (OOA). These techniques give important strategies in representing semantics of a method; they should be incorporated in the generic model.
- During the method description the three main components in the meta model, namely product, process and heuristic, can be induced. **Product** describes the method concepts or *what* the notions are; **process** describes the method tasks or *when* to apply the notions, whereas **heuristic** describes the method guidance or *how* to deal with the notions. Each method can be identified by these three components, although some methods are strong in one aspect and weak in the others. The three components combine together to form **method semantics** as a whole. The ideas of these components will become clearer as each of them are described separately, later in this thesis (chapters five to seven).

- The investigation also helps to identify the scope of this research by defining ‘software development method’ (section 2.5.4) and the chosen methods (section 2.4). Moreover, it addresses the existence and importance of meta modelling (section 2.5.2 and 2.5.3). This point is discussed further in next chapter - investigation of method integration, meta modelling techniques and metaCASE tools.

2.7 CONCLUSION

This chapter investigated eighteen software development methods by classifying them into four categories: structured methods, object-oriented methods, chosen methods and other methods. The key notions of each method are described and comments are made from the meta modelling viewpoint. An overview and significant points are especially presented for five chosen methods. Some notable points are shown as a summary of the investigation. The chapter also identifies the focal point of this research.

3. INVESTIGATION OF METHOD INTEGRATION, META MODELLING RESEARCH AND METACASE TOOLS

A number of meta modelling techniques are available in both academic research and industrial products. Since the emphasis and the requisite of the problem domain is different in each one, there are various shortcomings to each approach. However, some significant experience and/or representation requirements can be drawn from these attempts. This chapter investigates the meta modelling techniques based on three categories: method integration, meta modelling research and metaCASE tools.

3.1 INTRODUCTION

Meta knowledge of the software development method is not a new topic to either the commercial market or the research field [Wen-yin 92] [Wijers 92]. There are many contributions from organisations and institutes all over the world [Kronlöf 93] [Carmichael 94]. However, this emerging science should be considered as a combination of technologies from software engineering, design method, advanced CASE tool and artificial intelligence. Some authors even refer to it as method engineering [Harmsen 94] [Nilsen 92]. Most people agree that *'each CASE tool is designed with a model in mind'*. A metaCASE is a tool to construct a CASE tool, therefore it must be designed with a meta-model. By looking into the internal formalism of metaCASE tools available, it will help us to recognise the representation of design method semantics and the concepts of meta-modelling embedded in the system, which includes data modelling, method processes, design constraints etc.

This chapter investigates meta modelling techniques in three categories. Firstly, the two major data interchange standards in method integration are presented in section 3.2, namely **CDIF** and **PCTE**. Secondly, recent meta-modelling research is studied in order to capture their basis and modelling perspectives. Section 3.3 looks at three proficient approaches in the subject, namely **ALF-MASP**, **SOCRATES Project** and **MethodBase**. Lastly, three prevalent metaCASE tools, namely **ObjectMaker**, **MetaEdit** and **IPSYS ToolBuilder** are reviewed in section 3.4. The costs and benefits of these meta modelling techniques are drawn out to develop the requirements of a generic representation. The final model is aimed to be both method and tool independent. Some significant points in the investigation are presented in section 3.5, which is followed by the conclusion of this chapter.

3.2 METHOD INTEGRATION

There are two bases for method integration: the first depends on data interchange between portable components [Daley 93] [Scheffström 93]; and the second relies on a common method representation [Potts 89] [Carmichael 94] (see section 3.3). This section concentrates on the first basis by reviewing two well known data interchange approaches, i.e. CDIF and PCTE.

3.2.1 CASE DATA INTERCHANGE FORMAT (CDIF)

The objective of the CDIF is to provide a set of standards that will enable CASE tools to interchange information in a standard format [Imber 91]. It adopts a common four-layer architecture for repositories and similar applications. Each layer in the framework is used to define the layer below it, as illustrated by the examples in figure 3.1a.

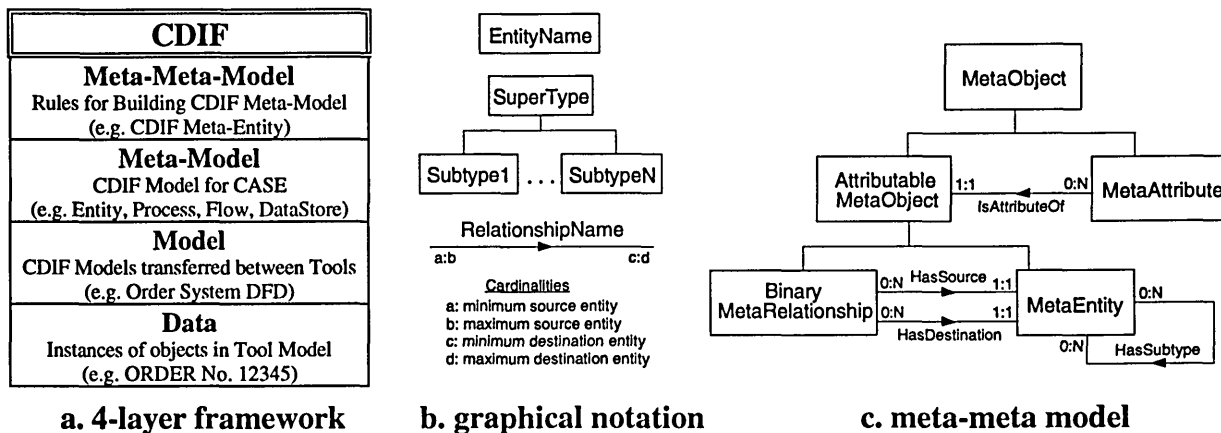


Figure 3.1 CDIF Standards

The **meta-meta model** is composed of an entity-relationship-attribute (ERA) model, which is enhanced with subtyping and relationship cardinalities, as shown in figure 3.1b. Figure 3.1c summarises the meta-meta model by depicting the ERA components in the meta-model. A **meta-model** splits into two parts: the *semantic model* and the *presentation model*. The latter describes the presentation of information to the user, whereas the former denotes the underlying meaning of information in four subject areas:

- *core* - the aspects required for any objects transferred by CDIF standard;
- *data flow modelling* - the semantics represented by data flow diagramming;
- *entity-relationship modelling* - the main forms of ERA modelling in information systems;
- *data inventory* - the definitions of attributes of any object and the underlying data type.

These formalisms determine the **model** that is transferred between tools, whereas the **data** layer describes the instances of objects in the model.

SIGNIFICANCE

CDIF stresses data interchange through standard procedure calls, software bus or broadcast messages [Schefström 93]. It provides standards for exchange information and facilities to import/export between different language platforms. No actual data repository or interface services are supported by CDIF [Thompson 93].

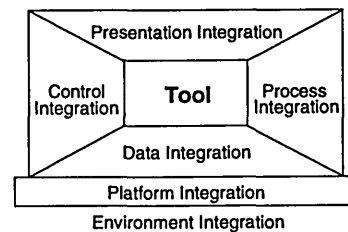
From the specification of the meta model above, CDIF does not present any software process or heuristic guidance. It provides meta modelling of the object, but even that is very limited.

3.2.2 PORTABLE COMMON TOOL ENVIRONMENT (PCTE)

PCTE claims to support integration in four dimensions [ECMA 90] [Simon 93] as shown in figure 3.2a, which is based on the framework services amongst the dimensions as depicted in figure 3.2b. Different tools can be incorporated into this model to identify the various services to support the integration [Daley 94]. PCTE considers the platform and environment integrations as external constraints which are not directly managed by the standard.

Integration	Description
Presentation	appearance of the software environment to the user
Data	data sharing and exchange amongst tools
Control	enhancement of communication between various tools used to support a software process in terms of their invocation
Process	description and enaction of software process models in terms of tools invocation

a. 4-dimensions



b. integration framework

Figure 3.2 PCTE Standards

The advanced PCTE+ Object Management System (OMS) satisfies the requirements of tool integration by introducing the following ideas into the SE database [Oquendo 91]:

- concepts of objects, attributes and links:** a specific data model that is derived from the ERA data model and the object-oriented paradigm;
 - object* - a uniquely identified entity that is characterised by a name, a set of supertypes, a set of attribute types, a kind of contents and a set of link types;
 - attribute* - a specific property of an object or a link, that is characterised by a name, a value type, an initial value and a duplication property;
 - link* - a directed binary relationship that is characterised by a name, a cardinality (one or many), a category (existence, composition, reference, designation or implication), a set of possible origin and destination object types, a set of attribute types (possibly empty) and an optional stability property, exclusiveness property and duplication property.

- *concepts of composite entities and object's contents*: provide facilities for representing and manipulating data down to the level of fine granularity;
- *concept versions of composite entities*: provide facilities for storing and accessing multiple versions of single and composite entities while retaining their version history;
- *concept of type specialisation*: provide facilities for defining new types by extending existing (base) ones while maintaining the compatibility with these base types;
- *concept of schema definition sets (SDS) and working schemas (WS)*: provide facilities for defining database views for tools (SDS - a set of related type definitions which constitute a subset of the database schema, WS - the union of all type definitions in the SDSs);

In addition, PCTE+ also supports the requirements from the SE perspectives:

- *mechanisms for concurrency and integrity control*: by providing lock mechanisms for ensuring the consistency of concurrent data access operations, and the concept of activity which includes the one of transaction;
- *mechanisms for discretionary access control*: by means of the concept of security defined as the prevention of unauthorised disclosure, amendment or deletion of information;
- *support for multiple programming languages*: by defining the database concepts in a language-independent way and by specifying the interface of operations in different programming languages;
- *support for distributed architecture and database*: by providing a data model whose design takes into account the constraints of a distributed hardware architecture, and a transparent distribution of the database over the network of workstations and mainframes.

SIGNIFICANCE

PCTE emphasises the interchanging of portable components (or tools) as outcomes, so neither design process or heuristics are recorded in the OMS declaration. It also lacks a pictorial view of the SDS or the WS. However, PCTE is employed as an information exchange standard for some meta modelling tools, such as [Oquendo 93] and [Saeki 94a].

The major shortcoming of method integration by data interchange is that the approach only handles information transfer between tools (method components) but their internal semantics are not properly addressed. Although both CDIF and PCTE outline an elementary description of 'product' as object, there is no conceptual significance to the representation. Moreover, this description is mainly for exchange tool specification rather than a method definition. In this research, the focus is on the other extreme of method integration. That is, to establish a generic model for method representation and to integrate method concepts at a semantic level. The next section presents meta modelling research endeavours to support this approach.

3.3 META MODELLING RESEARCHES

In the academic world, there are numerous research projects in the areas of software engineering [Finkelstein 92], conceptual modelling [Grosz 91] [Siau 92] and CASE tools [Gulla 91]. Some of these studies suggest a requisite for and the significance of meta modelling [Dewal 92], but only a few attempts to tackle the solution explicitly and most of these related projects are fairly recent works. This section discusses some leading meta-modelling research projects, and particularly focuses on their method representation aspects.

The three projects described are the **ALF-MASP** metaCASE Environment (from France), the **SOCRATES** Project (from Netherlands) and the **MethodBase** (from Japan). Although all three denote method product and process explicitly, the fundamental models are different.

3.3.1 ALF-MASP

The ALF metaCASE environment is developed in the framework of the ALF ESPRIT project. It claims that tool integration is a key issue in CASE environments for supporting the entire software development process [Oquendo 90]. PCTE (section 3.2.2) supports the data and presentation integrations and lacks mechanisms for the control and process integrations. The ALF project extends the PCTE functionality by:

- *Control integration* - providing means of intertool invocation, communication and synchronisation;
- *Process integration* - introducing software process interpreters (described later) and software tools between them.

A major enhancement is called the *event-reaction mechanism* (ERM), which provides the functionalities to express and enforce the two integrations of the software tools on top of PCTE. ERM is represented by a 3-tuple as (E, R, C), where:

- *E* - an event type that describes situations which raise events of this type;
- *R* - a reaction type that describes reactions taken when events of type *E* are raised;
- *C* - a control specification for the triggering of reactions of type *R*.

ALF has a formalism for software process modelling, called the **MASP** (meta-Model for Assisted Software Processes). The ALF project claims that MASP provides the mechanisms for describing a generic process model that can be incrementally and repeatedly instantiated in order to produce project-specific software process models. The MASP description is enactable and can be used to provide a better understanding of software processes and assist the communication amongst developers. Figure 3.3 presents a complete MASP specification and the composed six models are described briefly afterwards [Oquendo 93].

```

masp =
  MASP specification
  description
  END MASP ';'
specification =
  identifier HAS TYPE operator_type
  ':' (domain_and_range;)
description =
  object_model_definition ';'
  (expression_model_definition ';')
  operator_model_definition ';'
  (ordering_model_definition ';')
  (rule_model_definition ';')
  (characteristic_model_definition ';')
domain_and_range = '(' [ parameter { ';' parameter } ] ')'
parameter = par_access par_list ':' par_type
par_access = IN | OUT | INOUT | READ | READWRITE
par_list = parameter_name { ';' parameter_name }
par_type = object_type_name |
  relationship_type_name | attribute_type_name

object_model_definition =
  OBJECT MODEL IS
  list_of_schema_definition_sets ';'
  END OBJECT MODEL
list_of_schema_definition_sets = sds { ';' sds }
sds = sds_name | sds_definition | sds_extension
sds_definition =
  NEW SDS sds_name IS type_list ';'
  END sds_name
sds_extension =
  EXTEND SDS sds_name WITH type_list ';'
  END sds_name
type_list = type { ';' type }
type = object_type_definition | object_type_extension |
  relationship_type_extension |
  attribute_type_extension | type_importation
object_type_definition = ot_name ';' SUBTYPE OF
  object_type_names [ WITH
  [ATTRIBUTE attribute_type_d_list ';']
  [LINK relationship_type_d_list ';']
  END ot_name ]

expression_model_definition =
  EXPRESSION MODEL IS
  list_of_expressions ';'
  END EXPRESSION MODEL
list_of_expressions = expression { ';' expression }
expression = event_definition |
  logical_exp_definition | expression_definition
expression_definition = ex_name ';'
  ON event_d { ';' event_d }
  EVALUATE logical_expression_d
event_d = event_name | event_description
event_definition = event_name ':' EVENT
  event_description
event_description = user_defined_event_situation |
  read_event_situation | update_event_situation |
  create_event_situation | delete_event_situation |
  move_event_situation | convert_event_situation |
  expression_event_situation |
  invoke_operator_event_situation |
  exit_operator_event_situation | time_event_situation

operator_model_definition =
  OPERATOR MODEL IS
  list_of_operator_types
  END OPERATOR MODEL
list_of_operator_types =
  operator_type { ';' operator_type }
  operator_type = operator_type_definition |
  operator_type_importation
operator_type_definition =
  op_name ':' [domain_and_range]
  PRECONDITION ':' logical_expression_d
  POSTCONDITION ':' logical_expression_d
  KIND ':' 'I' | INTERACTIVE | NON INTERACTIVE
logical_expression_d =
  log_exp_name | log_exp_description
logical_exp_definition =
  log_exp_name ':' log_exp_description
log_exp_description = ... "it is a logical expression
  built using the logical connectors AND, OR,
  NEGATION and IMPLICATION, where variables
  may be typed and universally on existentially
  quantified..."

ordering_model_definition =
  ORDERING MODEL IS
  list_of_ordering ';'
  END ORDERING MODEL
list_of_ordering = ordering { ';' ordering }
ordering = [ or_name ';' ] path_expression
connection = FOR ALL variable ';' object_type_name
  [ IN (variable | constant) ';' object_type_name ]
path_expression =
  [connection DO] operator_exp |
  [connection DO] br_path_exp |
  [connection DO] bi_path_exp
br_path_exp =
  '(' path_expression ')' | '(' path_expression ')' |
  '(' path_expression ')' | '(' path_expression ')' |
  '[' path_expression ']' counter
bi_path_exp =
  path_expression '||' path_expression |
  path_expression '||' path_expression |
  path_expression ';' path_expression

rule_model_definition =
  RULE MODEL IS
  list_of_rules ';'
  END RULE MODEL
list_of_rules = rule { ';' rule }
rule = [ru_name ':' ] IF expression_d
  THEN operator_name '(' [parameter_list] ')'
expression_d = event_d | logical_expression_d |
  expression_name | expression_description
parameter_list = par { ';' par }
par = variable | constant

characteristic_model_definition =
  CHARACTERISTIC MODEL IS
  list_of_characteristic ';'
  END CHARACTERISTIC MODEL
list_of_characteristic = characteristic { ';' characteristic }
characteristic = [ch_name ':' ] logical_expression_d |
  expression_name | expression_definition

```

Figure 3.3 MASP Specification

- *Object model* - a set of object types definitions and imports which are structures in terms of PCTE+ OMS Schema Definition Sets (refer back to section 3.2.2);
- *Operator model* - a set of operator type declarations in terms of pre- and post- conditions (backward and forward reasoning), domain and range (input and output parameters);
- *Expression model* - a set of logical conditions to describe particular states of software processes based on the first-order predicate calculus;
- *Ordering model* - a set of orderings to express restrictions on the execution of operators, such as sequential, alternative or concurrent between two specified operators;
- *Rule model* - a set of rules to define the possible automatic reactions to specific situations arising during the software process;
- *Characteristic model* - a set of characteristics to describe the constraints on the software processes states that should be enforced during their enactment.

When a MASP is instantiated, which gives an **instantiated MASP** (IMASP), enough information is gathered to enact software processes. The **enaction** is an interpretation of the MASP using the IMASP as a knowledge base in a context local to an **assisted software process** (ASP). The overall approach for MASP is to interleave instantiation and enaction as depicted in figure 3.4a. This approach provides the possibility of considering the part of the development that has been executed before instantiating a further part [Oquendo 92].

Figure 3.4b sketches the architecture of the ALF metaCASE environment system, which provides the set of tools and services for generating a process-centred software environment. The main component is the *MASP interpreter*. This is a production system that monitors the software development and provides assistance whenever necessary. The MASP interpreter can be thought of as a process with dual tasks. Firstly, it performs the forward and backward reasoning upon the activity of other processes under its control. Secondly, it interprets many ASPs that are connected to the IMASP.

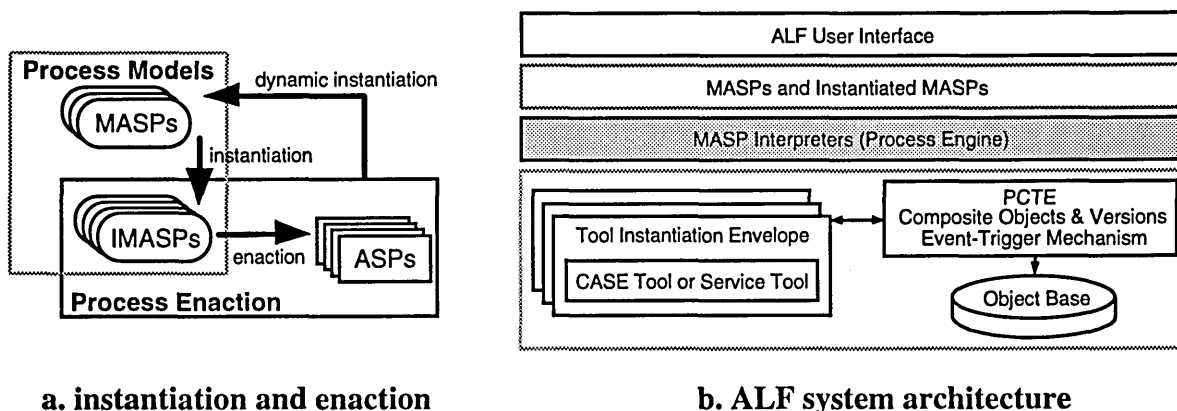


Figure 3.4 ALF-MASP Approach

SIGNIFICANCE

The ALF metaCASE system can be considered as a bond of tool integration and method engineering. It builds a generic software process model on top of the PCTE environment, so ALF is regarded as comprising ‘partial’ meta modelling techniques. Since ALF is a process-centred environment, it does not explicitly provide the modelling of products or a full method representation. It achieves the description of products by borrowing the PCTE+ OMS object base to act as a repository of models of processes or MASPs. The object model, organised through object-orientation, enhances the SDS with new mechanisms including structured and multi-valued attributes, triggers and semantic constraints.

Moreover, ALF presents a complete definition of the meta model software processes by the MASP specification. The six models (i.e. object, operator, expression, ordering, rule and characteristic models) are formally declared in a distinct structured language. Although the first two models only describe the structural-side¹ briefly, the behavioural-side² is documented comprehensively by the last three models. The expression model is just a means for declaring terms that are in the other models. The relaxed pre- and post- conditions in the operator model enable the description of the semantics of complex activities by MASPs, rather than treat them as the basis for backward and forward reasoning. This mechanism extends PCTE with high level functionalities for control and process integration of tools [Oquendo 90].

In addition, it may be advisable to simplify the overall approach by combining some models. For instance, the characteristic model definition is actually the IF-part of the rule model definition (without the THEN-part). Therefore, theoretically, the rule model is a superset of the characteristic model, and the characteristic model can be embedded. Furthermore, ALF-MASP may improve with a graphical representation of MASP interactions. The details of MASP dependence (in both rule model and characteristic model) are very difficult to observe, but a pictorial view of the MASPs can illustrate this unambiguously. The process sequencing (in the ordering model) can also be visualised in such diagrams.

The MASP is referred to as the ‘assisted’ software processes and the MASP interpreter is claimed to provide assistance and guidance to the software developer. It is stated that the interpreter is implemented using the *ALF-Rete Expert System Generator* with production rules to incorporate the event-reaction mechanism (*if condition then action*). The interpreter also provides data sharing between the procedural and the heuristic parts of an application [Oquendo 92]. However, no explicit discussion on the heuristic part is found in any available literature, and this sector of meta modelling is an add-on to the ALF system.

¹ In the PCTE+ OMS terminology, this is known as presentation and data integrations. A full description of object and operation models may be available in the PCTE environment itself.

² ALF names this the control model, it actually refers to the control and process integrations in PCTE+ OMS.

3.3.2 SOCRATES PROJECT

The SOCRATES project is based on the early SERC knowledge representation researches, such as process modelling [Wijers 90] and conceptual task modelling (CTM) [Brinkkemper 90]. The project is extended to specify task structure semantics through process algebra [Hofstede 93a] and to discuss expressiveness of conceptual data modelling [Hofstede 93b]. This section focuses on the representation of information modelling knowledge described in [Wijers 92].

SOCRATES claims that CASE tools should include knowledge of both the tasks within an information modelling process and the models resulting from the modelling process. Three levels of abstraction are suggested, namely the application, the method and the axiomatic levels, as depicted in figure 3.5a. The last (top) level is a set of axioms for composing the software method at lower levels, and that is where the main meta-modelling techniques lay. Two types of metaCASE users are distinguished: a meta-analyst defines a meta model representing relevant method knowledge and the analyst uses the metaCASE tool to support their modelling process [Hofstede 92]. This support is provided by automatically interpreting a meta-model from the meta-model base, as shown in figure 3.5b.

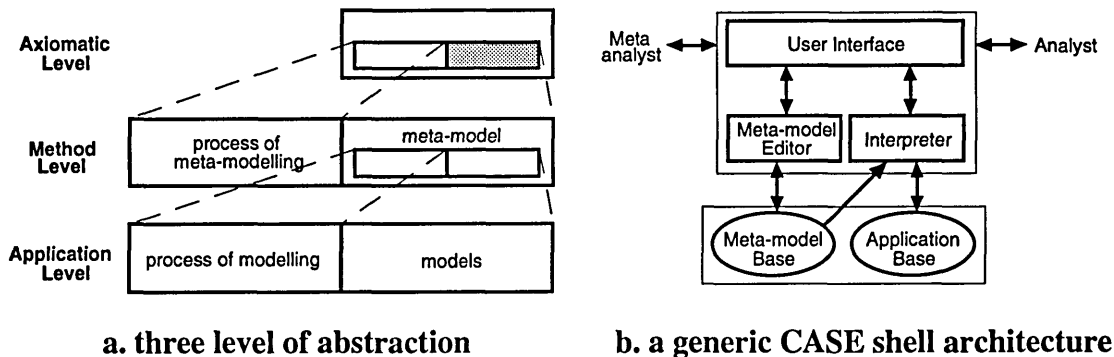


Figure 3.5 SOCRATES Three Levels of Abstraction

Unlike ALF-MASP, SOCRATES envisions information modelling knowledge as encompassing *a way of modelling* (product³) and *a way of working* (process) [Wijers 90]. The respective knowledge is formally represented by *concept structure* and *task structure*. A concept structure is represented by a NIAM schema, where the main notions are shown in figure 3.6a. The structure identifies specialisation and association. Roles are defined in each association, and each role is given a specific number. An association can be transformed into a concept through objectification. Figure 3.6b illustrates the corresponding concept structure of the JSD entity structure step (refer to section 2.2.2). In addition, all the concept definitions, properties and verification rules of the model are documented as first order

³ Confusingly, SOCRATES refers to a meta modelling product as 'model' (see figure 3.5a).

predicate logic. A concept structure is a 5-tuple $C = (C, A, R, S, Q)$, where C is a set of concepts, A is a set of associations and different relationships (association, specialisation and objectification) are formally represented as:

$R: A \times N \setminus \{0\} \rightarrow C$ is the (partial) function denoting the concept playing the n -th role,
 $S \subseteq C \times C$ is the specialisation relation defining the subtyping network,
 $Q: A \leftrightarrow C$ is the partial objectification function (a bijection).

One role property in JSD declares ‘each association has at least two roles’:

$\forall a \in A \exists c_1 \in C \exists c_2 \in C [\text{role}(a, 1) = c_1 \wedge \text{role}(a, 2) = c_2]$

and a verification rule defines ‘each not-a-leaf is either an iteration, a selection or a sequence’:

$\forall n [\text{not_a_leaf}(n) \Rightarrow \text{iteration}(n) \vee \text{selection}(n) \vee \text{sequence}(n)]$

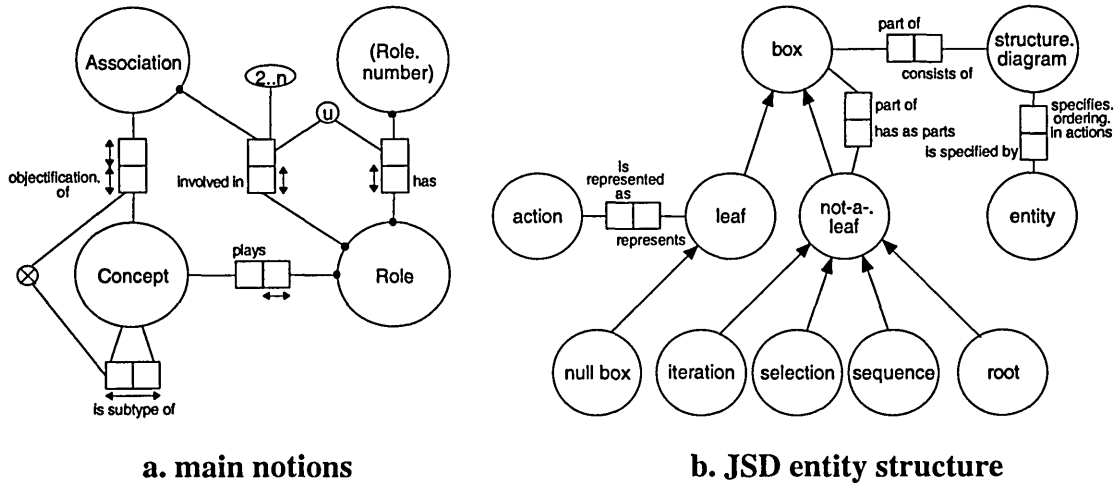


Figure 3.6 SOCRATES Concept Structure

Allied to the concept structure is the task structure where the main notions are depicted in figure 3.7a. A task is something that has to be performed in order to achieve a certain goal and it can be defined recursively in terms of subtasks. Therefore a task can be decomposed into a hierarchy of subtasks until a desired level of detail has been reached. Decisions indicate choices between subtasks and coordinate the sequence of tasks. Figure 3.7b shows the interrelationship between task and decision, whereas figure 3.7c illustrates the task structure of the five JSP specification steps. SOCRATES defines a Predicate/Transition net (PrT-net) to represent the dynamic interpretation of task structures. It is a Petri-net whose places correspond to predicates with variable extensions, and transitions represent classes of elementary changes of predicate extensions. Basic task procedures, task views, priori & posteriori verification rules, information places and decision rules are introduced to provide the links to the concept model (see [Verhoef 91] and [Wijers 92] for details).

Again, first order predicate logic is used to specify the task structure. A task structure is a 7-tuple $T = (T, D, L, N, F, G, I)$, where T is a non-empty set of tasks, D is a set of decisions, L is a set of labels. The set $T \cup D$ will be represented by O as the set of task object:

$N: O \rightarrow L$ is the function assigning labels to task objects,
 $F \subseteq O \times O$ is the relation defining the triggers,
 $G: O \leftrightarrow L_t$ is the partial function defining the decomposition with $L_t \equiv \{l \in L \mid \exists t \in T[N(t)=l]\}$,
 $I \subseteq G$ is the partial function defining the initial items.

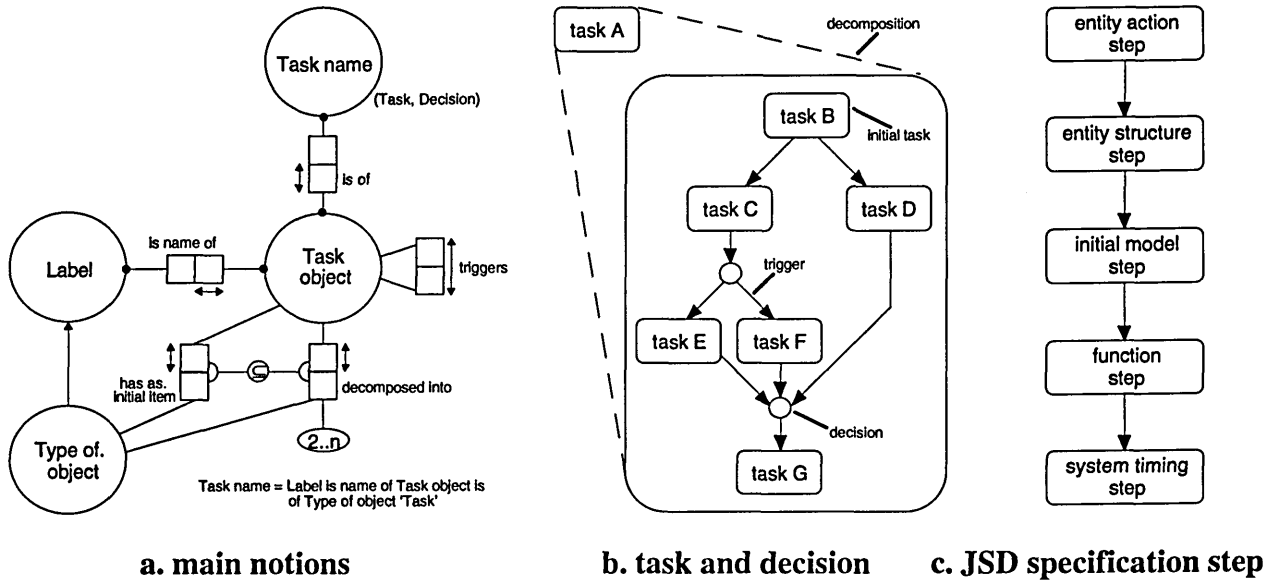


Figure 3.7 SOCRATES Task Structure

Each decision is formalised as a rule. A *decision rule* is comprised of conditions and outcomes. A simple condition is an informal statement in natural language and the decision defines various outcomes with different certainties. SOCRATES claims that certainty factors (cf) constitute the heuristics. The syntax for the decision rule is as follows:

```

<rule> ::= If <conditions> Then <outcomes> '!'
<conditions> ::= <simple_condition>
                | '(' <conditions> 'or' <conditions> {'or' <conditions>}'
                | '(' <conditions> 'and' <conditions> {'and' <conditions>}'
                | 'not' '(' <conditions> ')'
<outcomes> ::= <outcome> [with cf <cf>]
              { and <outcome> [with cf <cf>] }
  
```

SIGNIFICANCE

The main emphasis of SOCRATES is to compose a conceptual model of information modelling knowledge. Association is too unimpeded as a meta model construct, and roles are relaxed as labels or attributes in concept relationships. They can be defined more rigidly in a generic model to reduce unnecessary complexity. Also, SOCRATES does not define all key notions of a method as concepts. For instance, the meta concept structure (shown in figure 3.6a) describes the 'specialisation' concept as an association and the 'objectification' concept is hidden in a role name. The overall task architecture is also comparatively complex and detailed in meta modelling, which makes the individual components of the task structure less

reusable. The size of the data repository may increase dramatically according to the number of decisions. Hence, the task structure is a homogeneous representation of information modelling and it is designed without method integration or dissection in mind.

First order predicate logic is a very formal but efficient way to define a conceptual model, since the verification rules and consistency checks can be formulated easily (as illustrated earlier). The main drawback is that mathematical logic is less 'readable' and 'executable'. It is useful, however, to adopt a hypertext dictionary or a help system for critical concepts or tasks to reduce the human-logic barrier. This becomes essential for methods which have the same terminology but different meanings (section 2.6). Also, it is advisable to choose a formalism that is readily available for execution. For instance, the DDL statements in the KADS approach (section 2.5.4) are transformable to Prolog clauses, which is a logic programming language.

It is also realised that the general heuristics appear in a form of both design rule and textual guidance rather than simply decision rules. This argument is also reflected by the rule model and characteristic model defined in the ALF-MASP (section 3.3.1). A meta model should support the common formalism. In addition, the main heuristic information lies in the context reasoning and not on the certainty factors. Most decisions are inevitably based on individual circumstances and may not be looked upon as straight deduction rules as in an expert system.

The semantics of the SOCRATES meta model is defined by formally relating the application level and the method level, by indicating how the contents of information place instances and the application model which is an instance of the concept structure, are changed by the execution of an instance. SOCRATES has a clear integration between the structures as the main notions are based upon the same formalism.

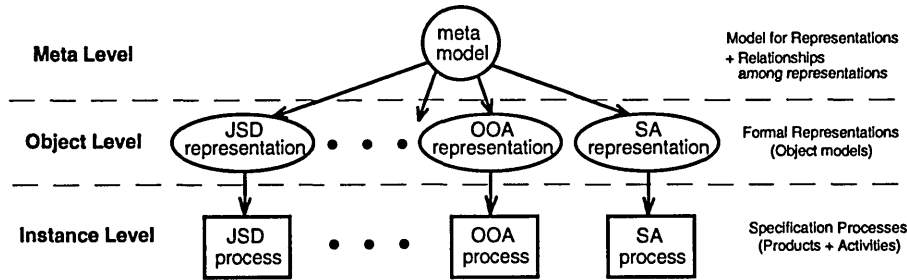
3.3.3 METHODBASE

MethodBase is a tool for developing specifications using multiple methods. The requirements of MethodBase are to document products and activities throughout the development and to support semantic relationships between design methods. Figure 3.8a shows the structural relationships among a meta model, formal representations of designs (object models), and actual specification processes. Thus MethodBase represents the semantic relationships in the meta model. The three basic characteristics of a MethodBase system are summarised as follows [Wen-yin 92]:

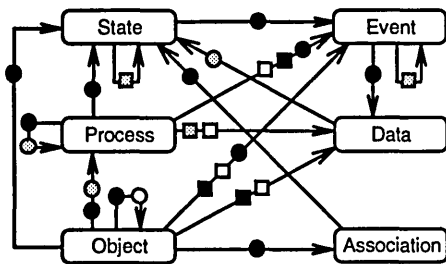
- guide and navigate the specification development according to the chosen design method;
- transform design method specification to another one at any development stage;
- choose suitable design methods appropriate for the problem domain and environment.

MethodBase not only denotes method concepts in a *product part*, but also classifies six types of concepts (meta-concepts) under three categories as follows [Saeki 93a]:

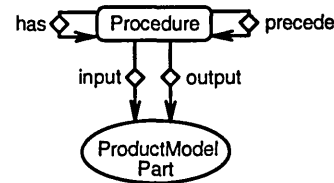
- *state* and *event* capture behavioural properties of a problem domain;
- *data* and *process* describe what functions are being performed.
- *object* and *association* represent physical components and the interactions among them;



a. logical structure



b. product part of meta model



c. procedural part of meta model

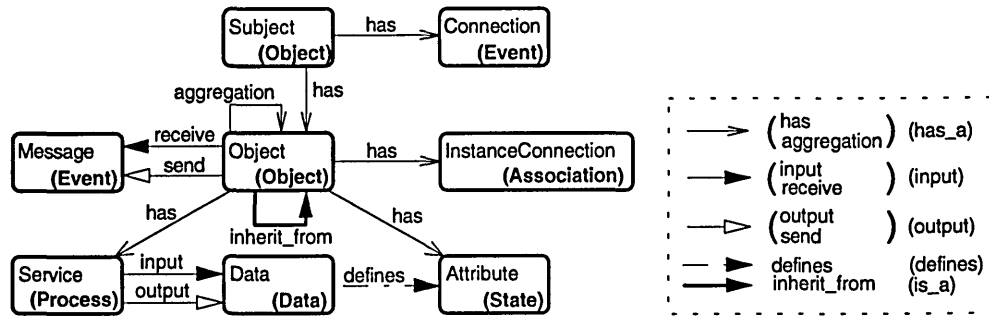
Figure 3.8 MethodBase Architecture

There are also six meta-relationships between these meta-concepts, as described below. Figure 3.8b shows the meta-concepts and their possible interrelationships. It is the product part of the meta model itself. (*next_to* relationship is normally used in the procedural part that is discussed later)

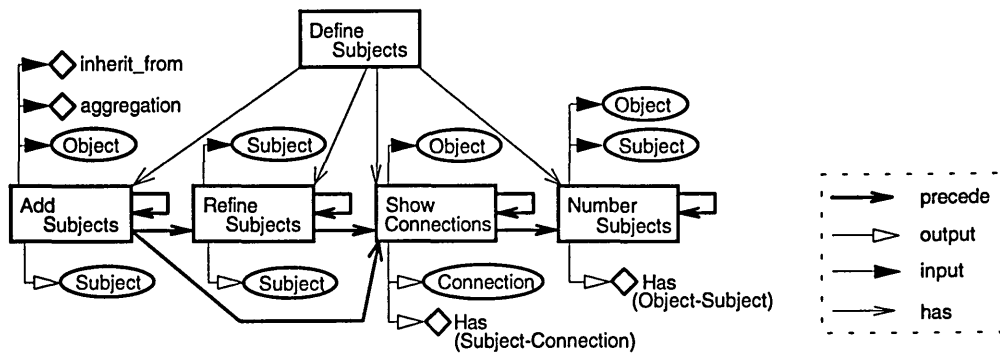
- *has_a* represents the ownership among instances of concept;
- *is_a* represents the generalisation/specialisation structure of concepts;
- *input/output* defines the source or destination of a concept to the other concepts;
- *defines* denotes the equivalence relationship between the defined and defining concepts;
- *next_to* describes the successor relationships among the ordered concepts.

MethodBase describes the process of a method in a so-called *procedural part*. The procedural part of the meta model consists of a procedure concept and the four types of relationships as depicted in figure 3.8c. A *procedure* could be directly applied to the concepts in the product part. The *precede* (or *next_to*) relationship presents the restrictions on the

sequence of procedures, whereas the *has* (or *has_a*) relationship allows decomposition of procedures into a lower level. The *input* and *output* relationships show the constituents of the procedure and their relationships to the product part as inputs and outputs respectively.



a. product part of OOA



b. procedural part of OOA

Figure 3.9 MethodBase Description of OOA

Illustrations of the product part and procedure part are given by the Coad's OOA description (section 2.3.3). Figure 3.9a shows a product part of the OOA, whereas figure 3.9b depicts the decomposition of procedure *define subjects* into four subprocedures. In addition, a process specification language (PSL) is used to document the meta model in a textual form as follows:

```
METHOD OOA %Coad's object-oriented analysis
OBJECT      Object ;
HAS_A       Aggregation : Object ;
HAS_A       Has : Service, Attribute, InstanceConnection ;
IS_A        Inherit_from : Object ;
INPUT       Receive : Message ;
OUTPUT      Send : Message ;
OBJECT      Subject ;
HAS_A       Has : Object, Connection ;
EVENT       Message ;
PROCESS     Service ;
INPUT       input : Data ;
OUTPUT      output : Data ;
STATE       Attribute ;
DATA        Data;
```

```

        DEFINES Defines Attribute
    ACTIVITY   Identify_objects ;
        INPUT : Object ;
        OUTPUT : Objects ;
        PRECEDE : Identify_Objects, Identify_Structures ;
    ACTIVITY ...
ENDMETHOD

```

The recent work of MethodBase involves formalising the representation by a formal language, Object-Z, which allows specifications of hierarchical structures and integration of method constraints etc. [Saeki 94b]. The meta model is extended by building a tool for cooperative specification processes [Saeki 93b]. Distributed individual tasks in a collaborative team is managed by two parts: various catalogued specifications and design methods stored in MethodBase; and a structured electronic mail system for communication among the members. The PCTE object base is used to hold various information together [Saeki 94a].

SIGNIFICANCE

MethodBase is the only meta modelling method that directly addresses the three types of relationships in multi-view specifications, which are: the *whole-to-whole* relationship of two products (or fragments) from different methods; the *part-to-part* relationship of two different constituents (or concepts) of two products and lastly *part-to-whole* refinement of a constituent of a product to another product. MethodBase claims to manage the hierarchical structures that are formed by these specifications, thus it fully supports the method integration.

The most outstanding point of MethodBase is that it employs the same graphical and textual representations for both product and procedure (or process) parts. This uniformity occurs throughout the meta model. However, it is found that MethodBase has no consistency in terms of notations, such as the distinctions between *concept* and *product*, and between *activity* and *procedure*. The graphical notations of meta-relationships in product and procedural parts (as shown in figure 3.8b and 3.8c) are inconsistent. The interchangeable arrow types between the two parts (in figure 3.9a and 3.9b) are also confusing.

In addition, the product part is incomplete, several substantial semantics are not encountered in the model, such as cardinality, directional aspect of relationships, parallelism and option/mandatory and part/whole features of concepts are not described at all. Thus, MethodBase is too immature to be stated as a conceptual representation of any software development method. Moreover, the idea of categorising concepts is good, but the classification according to their modelling viewpoints seems unessential to overall meta modelling. For instance, an *attribute* is defined under the *state* concept type (figure 3.9a), but as discussed in section 2.3.3, OOA does not actually describe dynamics. There are no direct links between the *state* type and the *event* type concepts (i.e. *message*). It may be better to classify by the nature of concepts (such as fragment and entity) so as to underline the multi-view specifications described above.

The input/output relationships are an effective way to relate procedural conditions back to the product part, although the multiple appearance of a single concept in a procedural part seems clumsy (such as *object* and *subject* in figure 3.9b).

Moreover, the description of process heuristics is totally missed out in the meta model. This is considered as a significant drawback in most meta modelling techniques, since development guidance is one of the crucial sets of information provided by a method. Without heuristics, a meta model is simply a set of notation definitions. MethodBase also stresses the importance of textual specification. The expressiveness is improved from PSL statements to Object Z declarations, though the execution ability remains a difficulty in such a formal language.

3.4 METACASE TOOLS

The use of CASE tool has general acceptance in terms of software development [Spurr 92] [Stobart 91] [Wijers 90]. The most appreciated aspects are the quality of diagrams, correctness and consistency with regard to applied methods and techniques. The major dissatisfactions lie within the poor interfaces with other software products and the limited possibilities to adapt a tool to our own standards. Work has been done in configuring the environment and the transparency of CASE tools [Gulla 91] [Brinkkemper 93]. However, the general solution towards integrated development methods falls within the remit of metaCASE technology [Brough 93] [Slooten 93] [Sorenson 88]. A metaCASE tool is dedicated to capture the modelling semantics of a particular method and to generate the desired CASE tool, from which the software is produced. This section presents three well-known tools for these purposes, namely ObjectMaker (from US), MetaEdit (from Finland) and IPSYS ToolBuilder (from UK). Their significance in meta modelling is emphasised in the description.

3.4.1 OBJECTMAKER

ObjectMaker [MarkV 93] is a product of MarkV in US. ObjectMaker supports the analysis and design phases of the software development process. Its meta modelling is based on the specification of *notations* in a method, which is basically a set of *diagrams* such as the *Object Class Diagram*, the *Service Chart* and the *State Transition Diagram* in Coad/Yourdon OOA (section 2.3.3). Each method is recorded in two main file types to describe its behaviour and functionality, they are known as the *menu* file and the *rule* file. A menu file denotes all possible operations in the method as menu (or menu bar) options. For instance, the *icon* menu of the *Object Class Diagram* in OOA is shown below:

```
menu_of_icons ::= menu(
    gray, disable, item(Nodes:,,NULL,),
    separator,
    item(Subject,, RECTANGLE(flags=>(thick,dash)), mth S),
```

```

        item(Class,,ROUND_REC(flags=>thick), mth C),
        item(Class and Object,, IMPORT=oac.bde, mth O),
        separator,
        gray, disable, item(Arcs:,,NULL,),
        separator,
        item(Generalisation,, ARC(head=>circle_half), mth G),
        item(Yoke,, YOKE, mth Y),
        item(Specialisation,, ARC(head=>arrow_none), mth L),
        separator,
        item(Whole/Part,, ARC(head=>v_flush_out), mth P),
        item(Instance Connection,, ARC(head=>arrow_none), mth I),
        item(Message Connection,,ARC(head=>arrow,flags=>thick), mth D),
        separator,
        item(Bend,, BEND, mth B),
    );

```

A rule file presents the constraints (MarkV refers to these as semantics) associated in the menu, such as the various *class_and_object* types in OOA are defined as follow:

```

get_oac_type1()           ::= class_and_object(object);
get_oac_type1(*error*)    ::= class_and_object(object);
get_oac_type1(round_rec,solid) ::= class_and_object(class);
get_oac_type1(rectangle,*) ::= class_and_object(object);
get_oac_type1(*)          ::= error;

```

Moreover, ObjectMaker includes a range of conventional and object-oriented methods, such as Adarts, DeMarco SA, Ward/Mellor, Yourdon, Booch OOD, OMT, OOA, Firesmith ADM3, CRC, Shlaer/Mellor OOSA etc. In addition, it provides code generation to Ada, C/C++ and COBOL sources. Data repository of the methods (diagrams) is also provided.

SIGNIFICANCE

Strictly speaking, ObjectMaker is not a metaCASE tool but a CASE tool which supports user customisation. Although MarkV claims that the supplement tool (MethodMaker) enables the user to configure a specific method, the capability is very limited and inflexible. For instance, there is neither definition of textual specification nor user-defined graphical presentation such as icons and bitmaps. The actual metaCASE mechanism remains in the kernel of ObjectMaker, which is inaccessible by the user. Thus, MarkV agrees to produce any demanded method upon request.

ObjectMaker is described in this subsection, since there are some distinct features between 'metaCASE tool' and meta modelling. Firstly, the menu file describes all functionality in a notation, such as create a subdiagram or change persistence of an object. Ideally, they should be distinguished as behaviours of an individual concept. Secondly, the rule file defines the diagrammatic integrity in first order clauses, which is fairly close to that of SOCRATES (section 3.3.2). However, the rules are notational rather than semantical. Thirdly, the common data repository promotes the idea of multi-viewpoints in a problem domain, though there are no actual semantic relationships described between the notations.

ObjectMaker is a closed system, the data repository is done internally in the CASE tool. All the methods are pre-defined and modelled with common services inside the tool kernel. Since there is no way to interface with the meta-model embedded in the tool, no further investigation can be carried out.

3.4.2 METAEDIT

MetaEdit is a CASE-shell, which claims to support configured methods by a generic tool environment [Smolander 91]. It is a user-definable tool that can even be used to model meta models. However, it is impossible to construct a suitable meta-model without some guidelines such as ‘meta-methods’ [Saeki 93a]. MetaEdit identifies two dimensions in method specification: *type-instance* and *conceptual-representational* as depicted in figure 3.10a. The first dimension distinguishes between the types in the meta model and their instances made up in the modelling target. The second dimension observes the difference between concepts and their representations.

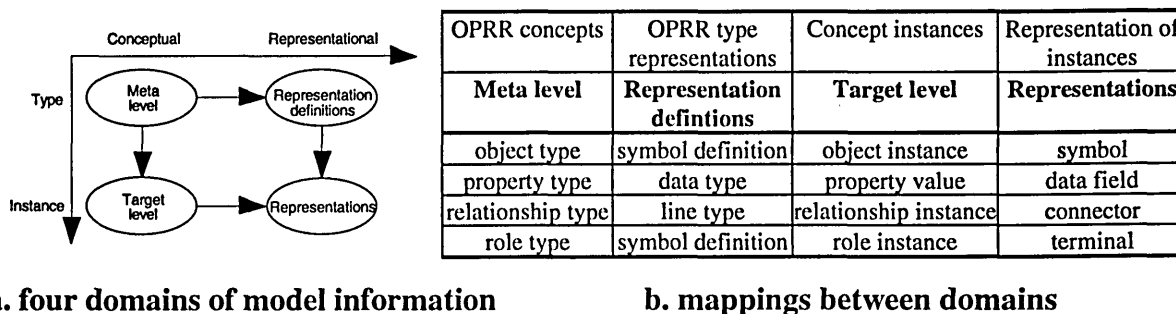


Figure 3.10 MetaEdit Approach

MetaEdit is based on the Object-Property-Role-Relationship (OPRR) data model, that can be considered as an extended version of the entity-relationship-attribute (ERA) model [Smolander 92]. These four OPRR meta-types are mapped to their representations and how they are instantiated in figure 3.10b. In the following description, the single letters shown in brackets represent the respective instance sets in the *conceptual structure* (see later).

- *Object* (O) is a ‘thing’ existing on its own, which is basically an intrinsic method concept.
- *Property* (P) is the characteristic associated with other meta-types.
- *Relationship* (R) is an association between two or more objects;
- *Role* (X) is the name of the link between an object and its connection with a relationship.

Figure 3.11a depicts the meta-metamodel of MetaEdit represented in the ERA modelling technique (see [Smolander 91] for details), whereas figure 3.11b presents a partial OPRR model of Booch OOD’s class diagram (refer to section 2.4.1). Apart from the four sets of

meta types, the meta model also describes two mappings: the mapping (r) defines the relationship types with the powerset of the roles and objects that it refers to; and the mapping (p) associates the property types with the other types (or non-property types). In addition, data type constraint, integrity constraint and connectivity constraint are also introduced to keep the valid instantiation of a meta-model.

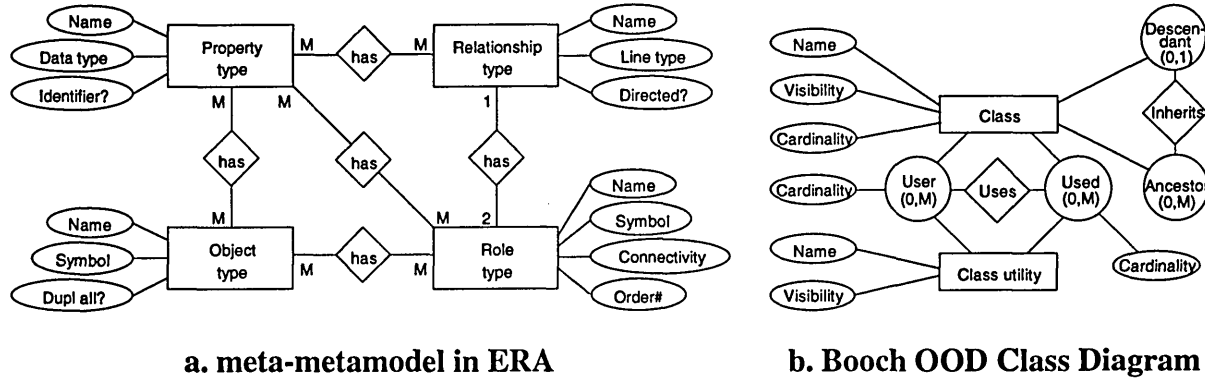


Figure 3.11 MetaEdit OPRR Modelling

MetaEdit provides a textual specification for OPRR, which is known as conceptual structure. This data structure is based on sets and mappings defined earlier and it is required in the implementation of an OPRR based CASE environment. The following conceptual structure represents the class diagram technique shown in figure 3.11b.

```
O = { Class, ClassUtility }
P = { Name, Visibility, Cardinality }
R = { Descendant, Ancestor, User, Used }
X = { Inherits, Uses }
r = { <Inherits, <<Descendant, {Class}>>, <Ancestor, {Class}>>>,
      <Uses, <<User, {Class, ClassUtility}>>, <Used, {Class, ClassUtility}>>> }
p = { <Class, {Name, Visibility, Cardinality}>, <ClassUtility, {Name, Visibility}>,
      <Descendant, { }>, <Ancestor, { }>, <User, {Cardinality}>, <Used, {Cardinality}>,
      <Inherits, { }>, <Uses, { }> }
```

SIGNIFICANCE

The developer of OPRR recognised two weak points of the model. Firstly, OPRR has only a flat structure and it provides no way to model naturally multi-dimensional structures or complex objects. This is because an OPRR model is an enhancement of traditional two-dimensional ERA models, as shown in figure 3.11a. The role is defined to put the emphasis on the association between object and relationship. This relatively simple structure in an OPRR model provides the benefit of ease of development of SQL-like query or even customised code generation. However, the cost is rigidity and unreusable modules. The solution to this is to employ object-oriented techniques, such as subtyping and composition, in the model. These will extend the meta model dimensionally.

Secondly, MetaEdit does not contain any concepts for defining the connections of multiple connected methods. This will be a major obstacle in representing rich-notation methods, such as Booch OOD and OMT. In order to reduce this inflexibility, the meta model should allow every method concept to be represented by a meta concept. In other words, each method concept has a representation, and it can compose with other method concepts. The terminal concepts are the property type as in OPRR.

The other problem with meta modelling in OPRR is illustrated by the specification of a DFD in figure 3.12 [Smolander 91]. The three object types are *Process*, *Store* and *External*. In order to avoid data flows between Store and External, two types of flows are identified: *FFP* (flows from Process only) and *Flows* (flows from Store or External). The former can flow to any object types, whereas the latter can only flow to a process. This is considered as a crutch in conceptual modelling, since the integrity should be defined specifically as a constraint rather than inducing pseudo-concepts in the representation. The result is not only to appease the implementation requirement, but also to cause unnecessary complexity to the meta model.

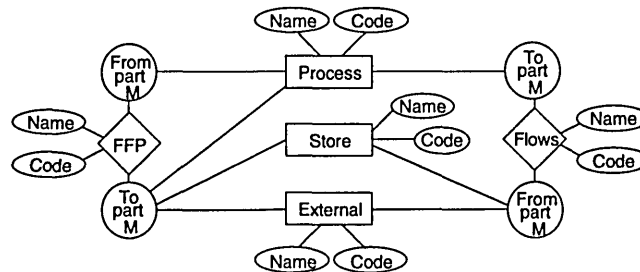


Figure 3.12 OPRR specification of DFD

The main textual specification of MetaEdit is the 'set-based' conceptual structure described earlier. In addition, MetaEdit provides the FREE-object Definition in OPRR Metalanguage and Query Language [Rossi 92], such as the *FreeNode* and *ID options* definitions shown below. The ultimate aim of a metaCASE tool is to generate a user-defined tool with a specified method, so these languages are much directed towards the need of implementing a CASE tool rather than meta modelling. The modelling techniques underline certain important ideas about the meta model, but the graphical and textual representations of the method are far from being a generic model.

```
symbol "FreeNode"
{ shapes("Rectangle"); scale 0.4;
  labels{"Free type" at (10 60 190 140) centred; } }
property type "ID options"
{ datatype list("NA","OPTIONAL","MANDATORY","none");
  number of values 1; }
```

Like most metaCASE tools, MetaEdit puts less emphasis on modelling method heuristics such as a concept dictionary. The recent related work on MetaEdit has proposed a triangular shaped meta model, to bridge the gap between system product model and system process model [Marttiin 94]. An information systems development (ISD) level consists of IS models, development group and ISD process. They are specified by meta-datamodel, activity model and agent model respectively in the method engineering (ME) level. However, this does not eliminate the discontinuity between the concept and task structures but defers the burden to the meta-level of method engineering.

3.4.3 IPSYS TOOLBUILDER

IPSYS ToolBuilder is an integrated system that combines the efforts of six universities and three research projects in the UK [Alderson 91]. Different parts of the system communicate through a common data repository. The meta-model of ToolBuilder is mainly based on three models, which are *entity model*, *frame model* and *shape model*. (see appendix B)

The entity model is an extended entity relationship diagram and a set of entity type definitions of the described method. Each entity represents a method *concept* and it can have a number of *attributes* to describe the internal state of the concept. Four types of relationships are provided, namely *subtyping*, *composition*, *reference* and *derived relationships*. The first two are borrowed from object-orientation. A reference relationship can be considered as an association between concepts in different frames, whereas a derived relationship is a link that depends on all the above relationships, including the derived relationship itself.

The frame model describes all user-defined frames which appear in run-time. Each frame is based on an entity type in the entity model. There are three types of frames. A *structured text frame* is a textual specification which can be used as a concept template, a description or a catalogue; a *diagram frame* is a graphical specification to present the method model pictorially; and a *root frame* is basically a structured text frame that initialises the system when it starts. Each structured text frame has a number of subsections to define the structure of that frame, and the frames can communicate through sharing objects [IPSYS 92].

In addition, the shape model provides facilities to define the graphical presentation, and each individual notation is stored as a *shape set* or as a *linkstyle set*. Since ToolBuilder is fairly well known in both academia and industry, it is adopted to map our meta model to the metaCASE tool semantics. A detail description of the ToolBuilder is given in appendix B and the full comparison with our meta model is shown in chapter eleven.

SIGNIFICANCE

ToolBuilder is a comprehensive integrated metaCASE tool. Despite the massive number of structured text frames needed to declare a radical method, the basic meta model is definite and explicit. The following points are noted:

- The entity model is a fairly simple and clear documentation of method concepts. Unlike ObjectMaker, each concept is associated with its own semantics directly, such as attributes, relationships and their related features. The shortcoming is that there is no straight distinction of a frame as a presentation of an entity or a tangible concept that must be denoted in the entity model. For instance, an *object diagram* can be referred to as a presentation or as a concept according to the denotation of the user. Some notions are debatable, such as *persistence* of an *object* can either be described as concept or attribute.
- ToolBuilder allows implementation 'directives' to be added in composition and/or reference relationships. These directives include *set of*, *sequence of*, *owner of*, *source*, *target*, *name* and *reverse name* etc. However, it may be advisable to describe individual conceptual aspects explicitly. The first two deal with cardinalities of source and target entities in a relationship, whereas the next three are used to denote the role of associated entities in composition relationship. The last two naming directives are used as references to the specific directional relationship. These directives are declared in this form because of the implementation requirement.
- It is relatively easy to formulate relationships in ToolBuilder, although some of them are mainly for navigational purposes rather than conceptual exposition. This is because ToolBuilder requires each navigation path in (frame/object/field) operation to be denoted by a single relationship between entities. Some reference relationships and the three types of derived relationships (path/aggregation/user-defined) are designed for this purpose. For a meta model, precise and concise representation is important. Unnecessary concepts and relationships, such as implementation constructs, should be eliminated or avoided.
- On the other hand, the navigation paths present an implicit model of the CASE tool processes, since the only accessible means of an entity is through one of the paths from the current entity.
- The *precondition* and *trigger* in each operation are defined as the guard condition and service supported to the underlying entity. These are essential modelling mechanisms in denoting processes, such the pre- and post- conditions described in a MASP specification (section 3.3.1).
- In ToolBuilder, each frame virtually represents a new state of the model with an entry action, which is basically displaying the frame and constructing a different set of operations available in the new frame. There is no explicit representation of method

process in the meta model. This is, in fact, a great drawback of most metaCASE tools in term of method representation.

- Method heuristics are not formally described in ToolBuilder and many other CASE tools. This makes the tools a notational description rather than a complete documentation of the method. A good method should have a well-balanced description of concepts, processes and heuristics. Hence, a good meta model should have a comprehensive specification on these three aspects.
- The user-defined graphical presentation and adaptability of external environment (such as embedding C code) are the main advantageous features of ToolBuilder. The open method design is more favourable than the closed method design as in ObjectMaker (section 3.4.1).

3.5 SUMMARY OF INVESTIGATION

From the investigation, the following points are noted:

- Data interchange standards support tool integration as well as method integration. Most of them provide a simple object model (rather than semantic model) for information exchange. The main emphasis is to specify transferrable components between tools or methods, and semantics are not properly described.
- Current meta modelling research have inclinations towards certain modelling requirements or narrow domain perspectives. For instance, ALF-MASP focuses on software process integration, and concept modelling is omitted, whereas SOCRATES provides conceptual task modelling, but no common method representation. MethodBase is the closest attempt at a coherent meta model. However, MethodBase is not detailed and specific enough in both product and procedural parts to cover the complete semantics. Our meta model looks into these shortcomings.
- Each metaCASE tool has a conceivable meta model, since it affirms tool generation with the underlined method. Process declaration in metaCASE tools is implicit and heuristics presentation is not normally supported, though they can be asserted by various implementation means that are specific to the tool. However, this is the main obstacle of method representation in metaCASE. A tool seeks an amalgamation of conceptual model and implementation construct rather than a generic meta model. This problem is addressed further by illustrating the mapping of our model to ToolBuilder in chapter 11.
- Moreover, substantial meta modelling techniques are drawn from the investigation. These precious experiences enlighten the determination of our generic model, which include:
 1. explicit process (MASP) specification given in ALF-MASP (section 3.3.1);

2. task structuring and decision rules described in SOCRATES (section 3.3.2);
 3. meta-concepts, relationships and their categorisations in MethodBase (section 3.3.3);
 4. various constraints for meta-model validation illustrated in MetaEdit (section 3.4.2);
 5. different relationship types and directives presented in ToolBuilder (section 3.4.3).
- Similar to software development methods (described in chapter two), meta modelling approaches also have a problem in the use of terminology. Different terms are employed to address the same (or similar) technique or semantic. For instance, ALF-MASP denotes *product* by *model*, whereas ObjectMaker refers to *diagram* as *notation*. Table 3.1 on the next page presents a cross reference of semantics amongst the approaches described in this chapter. The first column shows the terminology used in our generic model, which should be made clearer in the following chapters (four to seven). The glossary in appendix A also includes definitions of the common meta modelling terms applied in this thesis.

3.6 CONCLUSION

This investigation looks at various technologies available for meta modelling. Data interchange approaches for method integration (CDIF and PCTE) are too specific for standardising information exchange amongst portable tools and they do not express semantics properly. Most meta modelling research (ALF-MASP, SOCRATES and MethodBase) incline towards a certain modelling requirement, rather than seeking a generic representation of methods. In general, all metaCASE tools (ObjectMaker, MetaEdit and ToolBuilder) tend to embody implementation concepts in the method. Although these attempts are not satisfactory meta modelling formalisms, significant experience and techniques can be drawn from the investigation to enable derivation of our own generic meta modelling formalism.

Our Model	CDIF ⁴	PCTE ⁴	ALF-MASP ⁴	SOCRATES	MethodBase	Object Maker	MetaEdit	Tool Builder
product model	meta-model	object model	-	concept structure	product part	semantic definition	OPRR	entity model
concept	object	object	-	concept	concept	shape ⁵	object	entity
relationship	relationship	link	-	association	relationship	arc ⁵	relationship	relationship
property	attribute	attribute	-	-	-	? ⁸	property	attribute
process model	-	-	procedural part	task structure	procedural part	menu definition	? ⁸	(frame model) ⁶
task	-	-	MASP	task	procedure	(option) ⁶	(action) ⁶	(operation) ⁶
trigger	-	-	ERC mechanism	trigger	relationship	? ⁸	? ⁸	(menu option) ⁶
decomposition	-	-	ASP hierarchy	decomposition	has-relationship	(option) ⁶	? ⁸	(subsection) ⁶
heuristic model	-	-	heuristic part ⁷	decision rules	-	-	? ⁸	(help text) ⁶
MSL statement	? ⁸	SDS	MASP specification	first order predicate calculus	PSL & Object-Z notation	macro language	conceptual structure	structured-text frame

Table 3.1 Cross Reference of Semantics amongst Meta-Modelling Techniques

⁴ Both CDIF and PCTE does not denote process model, whereas ALF-MASP borrows the product model from PCTE.

⁵ ObjectMaker declares concepts and relationships by their representations which are shapes and arcs respectively.

⁶ Most of these semantics in metaCASE tools are described implicitly, i.e. they are tool features to implement the required mechanism.

⁷ Heuristic part is addressed in ALF-MASP, although the representation is not mentioned in the literature.

⁸ '?' represents that the information is not clear in the available literature.

4. SEMANTIC KNOWLEDGE BASE

This chapter is concerned with the problems of capturing the semantics of methods and thus defining a generic knowledge base for such methods. The work has been based on the investigations mentioned in chapter two and three. The basic methodological components have been identified as defining a generic structure or meta-model. The inherent semantics are represented by Prolog clauses. The potential benefits of this work are the definition of a standard meta-model, the transferability between existing methods and the automation of the generation of CASE tool support for new and evolving 'customised' methods, better suited to the requirement of any application.

4.1 INTRODUCTION

The number of different methods with similar semantics but different notation is now overwhelming. The rapid growth of programming paradigms (i.e. structured and object-oriented), programming techniques, programming languages and CASE tools have greatly contributed to the development of new methods. More importantly, this increasing number of methods does not contribute to a solution of the problem. Yet the expectations from the users such as short development time, high maintainability and reliability, require a suitable but flexible method. Although there are attempts to combine hybrid methods to solve particular applications, there is no direct attempt to generalise the ideas into standards. To achieve this standardisation, one cannot just consider the notational method representation, but one must also document the semantic components of the method. The primary concern of this chapter is to demonstrate a systemic, engineered approach to the documentation of the semantic structure of a software development method. It should therefore be possible for a method to evolve, as the project expectations similarly evolve.

Developers may plagiarise semantic concepts from existing methods, but use notational representations more familiar to their project application. This is often warranted where modifications to the semantics of a method become necessary. Ideally, we require a toolbox of pre-fabricated semantic components, which can be really integrated with new semantic components.

Documenting software development methods into a standard form, such as **generic method representation** (GMR), can help to analyze and compare different methods. The repository for such representation is known as a **semantic knowledge base** (SKB). The results are not simply useful as a reference, but provide the meta model for software generation for any method. Ultimately, in the form of a knowledge base, a subset of the tasks in the engineered

development of any methods can be automated. The knowledge base is stored in an executable specification, namely Prolog. This permits the inclusion of rules to check design completeness and system consistency, as well as help texts to guide the developer through the design steps. The knowledge base must allow incremental evolution, so that new semantics (concepts) and notations (representation) can be inserted in the future.

This chapter is organised as follows. Section 4.2 proposes an evolution approach to software development. The core of the chapter deals with the presentation of the semantic knowledge base. A single methodology model is shown in section 4.3 and section 4.4 defines the three layered model in meta modelling. Section 4.5 introduces the knowledge base representation with illustrations. Then the methodology prototyping is revisited in section 4.6. Conclusions are drawn in section 4.7.

4.2 EVOLUTION APPROACH

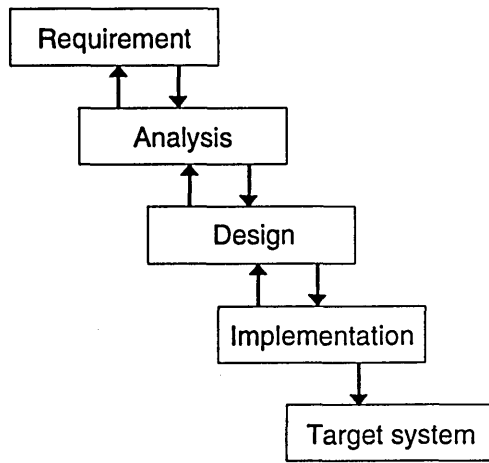
Within the field of software engineering there are a number of approaches to software development. These approaches differentiate and describe various phases in constructing software. One may argue that there is no straightforward boundary¹ between software development phases, since they are overlapping processes rather than totally disjoint. Some methods even amalgamate two phases into one. Nevertheless, the software development process is always highly iterative. Although the formal definition of the phase boundary is vague, the ordering and containment of each phase are significant.

To illustrate the evolution approach towards a software development, the traditional and the prototyping approaches of software development are described in the next two subsections.

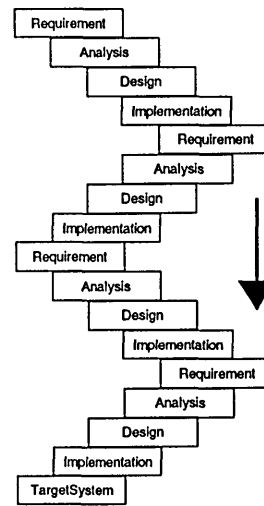
4.2.1 TRADITIONAL APPROACH

The traditional approach to develop software system is through a waterfall life cycle, as shown in figure 4.1a. Different phases of the cycle are the milestones of the development process, and each phase provides the necessary information for the forthcoming phases. For instance, the requirement phase produces a specification document which is passed onto the analysis phase for further investigation. The concept model developed in analysis phase is evolved in the design phase, where the result is ready for coding in the implementation phase. This cascading development process is ideal for simple and/or structural software development. However most systems are more complicated than a straight forward sequential process.

¹ The phase boundary does not mean 'when to start' and/or 'when to stop' a software development phase only. It also intends to describe the domain of an individual phase.



a. waterfall life cycle



b. spiral model

Figure 4.1 Traditional Software Development Approach

The other conventional approach is known as the spiral model, which is an enhancement of the waterfall life cycle [Boehm 86]. Software development is considered as a stepwise iterative life cycle, that is each cycle goes through every development phase and produces part of the final product, as illustrated in figure 4.1b. The spiral stops when the complete system is formed. This technique aims to break down the software complexity by incremental construction of a system. The fundamental difficulty of this approach is the coherence requirement of different parts. The separation must be relatively distinct and the interface must be defined. In addition, each cycle should have a substantial progress towards the final product so that there will not be too many or too few iterations.

4.2.2 PROTOTYPING APPROACH

The prototyping approach depends on evolution of software. No assumption of perfect knowledge of the application is made before implementation in prototyping. Thus any changes in user requirements during development can be accommodated. The main difference between prototyping and the other approaches is that iteration or feedback can happen at any point in prototyping. There is also an implication that the prototype itself will serve as the formal statement of requirement. The two major prototyping strategies are throw-away and evolutionary. Throw-away prototypes (figure 4.2a) are those in which the prototype is redeveloped or translated into another form before the target system is delivered, whereas evolutionary prototypes (figure 4.2b) are prototypes that themselves become delivered target systems. Due to the reusability advantages of evolutionary prototyping, it is more popular in the commercial world.

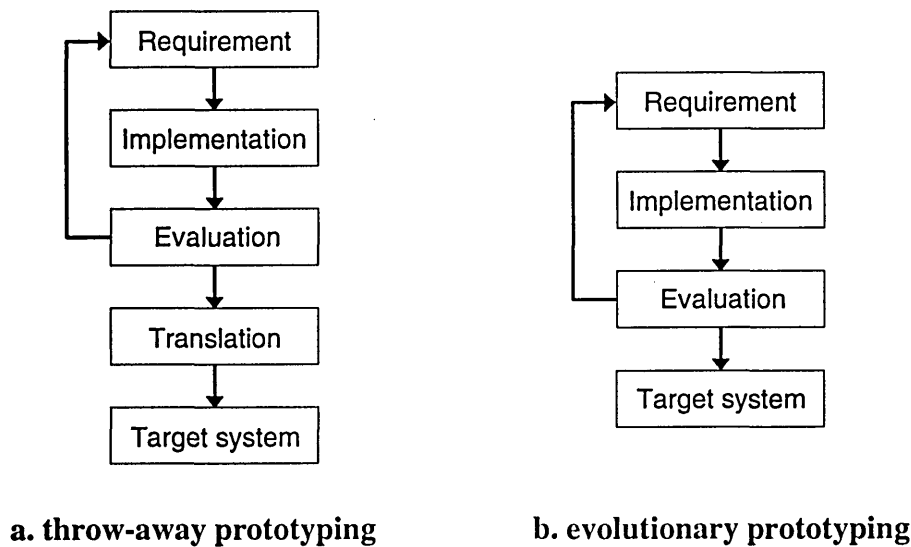


Figure 4.2 Prototyping Approach

The prototyping approach provides for the uncertainties of the real world and the best way to leverage reuse of existing software components in the development of new prototypes and applications. It also reduces long-term system expenses by narrowing the focus of maintenance effort to the component level. Therefore most object-oriented developers now appreciate that it is difficult to code good classes of objects without a thorough understanding of formal object-oriented design principles [Connell 95]. Object-oriented methods insist that object-oriented specifications will not be accurate unless a lot of iterative prototyping is done concurrently during requirements definition. This is primarily due to the fact that object-oriented programming is based on a model of the ‘natural’ world, rather than the totally artificial reality constructed by a traditional procedural language [Mullin 90].

4.2.3 METHOD ENGINEERING APPROACH

Many software development methods have been introduced in the last decade. Examples of these are object-oriented analysis and design methods, and business process reengineering methods [Rossi 95]. This rapid growth in number conforms with the software quality requirements and the increasing complexity of techniques involved. The classical ‘pencil and paper’ approach is no longer a satisfactory solution. The support of CASE tools is an interim result of the crisis, since they provide facilities to develop ‘large-scale’ software as well as to maintain system consistency. However, a CASE tool is always embedded with a fixed set of concepts (or a single methodology), which is found to be inflexible. In addition, there is no support of multiple viewpoints in software development. Therefore, method integration and method engineering have become ad hoc research topics. Meta modelling techniques and metaCASE tools are emerging to fill the gap of insufficiency.

With metaCASE technology, the capability of prototyping is also extended. It advocates an evolutionary approach to the software development cycle. The traditional waterfall life cycle and/or spiral model is elevated to include the method engineering phase, so that a tailor-made method can be produced for the required specification. The proposed three stages prototyping approach is summarised in figure 4.3. The three stages are known as **requirement specification prototyping**, **methodology prototyping** and **software development prototyping**. Since the new software development cycle is different from the approaches mentioned previously, it is referred to as the **method engineering approach**. The following subsections describe the different phases of this approach.

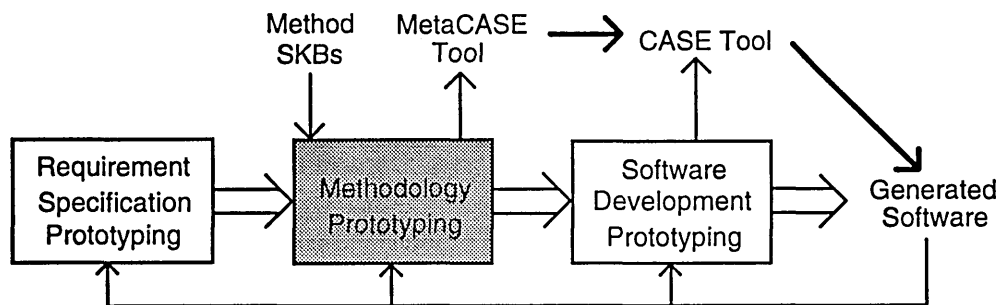


Figure 4.3 Three Stages Prototyping Approach

4.2.3.1 REQUIREMENT SPECIFICATION PROTOTYPING

The aim of requirement specification prototyping (RSP) is to provide sufficient knowledge about the problem domain and/or work environment for the next phase (i.e. methodology prototyping). The client forms a wish-list of the target system. The requirement engineer then models the requisite semantics according to the methods available in the semantic knowledge base². This process is incremental and iterative. It cycles around the client, the requirement model and the engineer until the satisfactory result is obtained.

Moreover, RSP adopts the evolutionary approach of rapid prototyping. That is, various phases of the software development are used to supply different information for the methodology prototyping. For instance, the analysis phase needs more inquiring concepts, whereas the design phase requires more resolving concepts. Different tools may be used to obtain multiple viewpoints on the system. The prototype can be specified by any requirement engineering techniques or tools available.

² A context free dictionary is provided in the system to lookup method concepts. These concepts must be stored in canonical form that is a common representation throughout the methodologies in the semantic knowledge base. (see later for details)

4.2.3.2 METHODOLOGY PROTOTYPING

Methodology prototyping (MP) is the phase for method engineering. The objective of this phase is to manipulate the required semantics and to provide a suitable method or CASE tool for the software development prototyping. The canonical concepts specified by the requirement specification prototyping are fed into an **expert server**, in which the concepts are matched with method semantics in the knowledge base. The semantics will then map to the information required by a metaCASE tool for the production of an appropriate CASE tool. Such a system is known as an expert server, because:

- It acts as an intelligent expert, which has the capability to search, match and inference the input canonical concepts to the method concepts in the knowledge base. Forward chaining, backward chaining and backtracking mechanisms are provided. The search algorithm, such as depth-first, breadth-first or best-fit search should be available. Similarly, the matching mechanism that includes parameters, such as priorities, weight factors etc., allows the user to optionally select.
- It acts as a knowledge base server to manage and propagate method semantics in the semantic knowledge base (SKB). Since the number of concepts in a method is large and it increases with the number of software development methods in the SKB, the speed of retrieving information from the data repository is vital. An effective server ensures adequate performance in projecting, selecting and joining concepts.

Methodology prototyping is the most important of the three stages. It links the 'specification stage' to the 'implementation stage' by transforming semantics from requirements to software development. A method prototype is created to bridge the semantic gap between the two stages. From a metaCASE point of view, a method prototype is a CASE tool with the desired method underneath the system.

The main focus of this research is to investigate and to formulate a generic model for representing semantics in a software development method. This is a crucial part of the evolutionary approach, especially in the methodology prototyping phase. A canonical form is invented to integrate various methods in the SKBs. The ability to share information within or amongst method fragments depends greatly on the unified model in the common data repository.

Efforts are also made to reduce the number of notations to a minimum, whilst still representing the complete semantics and various aspects of a method. In other words, the representation must be as precise and concise as possible. Moreover, the representation must be designed with the aim of mapping the method semantics to metaCASE semantics, which is the output channel of methodology prototyping.

4.2.2.3 SOFTWARE DEVELOPMENT PROTOTYPING

Software development prototyping (SDP) is a CASE tool based software development phase. As shown in figure 4.3, the SDP takes the CASE tool constructed by the methodology prototyping stage as input. It proceeds with normal CASE practice, that is to model a software system by the semantics provided by the underneath method. If the code generation facility is provided in the CASE tool, the target software is produced, otherwise the result is a paper model of the specified system.

The work environment depends on the CASE tool produced or in other words it relies on the tool generation ability of the metaCASE tool. Hence the final product varies from system to system and therefore the description is of no interest.

Our generic model captures both the conceptual base of a method as well as the practical base. That is, the model describes the structural and behavioural semantics of a system. The structural semantics show what the concepts are, whereas the behavioural semantics identify how the concepts are constructed. A capable metaCASE tool allows mapping of both types of semantics into the generated CASE tool. In addition the heuristic guidance of the software method is also an important component in the software development prototyping. Section 4.4 provides a detailed discussion about these constituents of a single development method.

4.3 THREE LAYERED MODEL

So far different models of meta modelling technology have been distinguished. This section presents a formal description of these three layers of models in method engineering. In order to reduce unnecessary confusion, hereinafter they are referred to as the **meta model**, the **method model** and the **software model**.

The software model is in the bottom layer and each layer higher up helps to model the layer below. For example, meta model is an approach to model a method, whereas the method model in turn is an approach to model a software.

A software model is a model that describes a real world situation, for instance an ATM *stateTransitionDiagram* denotes the possible *states* and *events* that happen in ATM transactions. A method model contains all concepts required to construct or demonstrate the software model. For instance, a model that describes the relationships between *state*, *event* and *transition* in a *stateTransitionDiagram* is a method model. Lastly, a model which describes the method constituents and their interrelationships is a meta model.

In other words, a software model is an instance of a method model, and a method model is an instance of a meta model. Hence, the three layered model forms an instance hierarchy. Table 4.1 summarises these models with examples.

Model	Examples	
Meta Model	MethodBase product - product part process - procedure part	IPSYS ToolBuilder product - entity diagram process - frame navigation
Method Model	OMT product - objectModel process - dynamicModel functionalModel	Ptech product - objectSchema process - eventSchema
Software Model	ATM objectDiagram ATM stateTransitionDiagram ATM dataFlowDiagram	objectSchema & eventSchema for a manufacturing application

Table 4.1 Examples of the Three Layered Models

As shown in the table, each model has two main constituents to describe the features in that particular layer. They are the product(s) and the process(es) of the corresponding model. A product describes the structural aspects of the system whereas a process describes the behavioural aspects of the system. It must be noted that a meta model can be either a pure model to represent a method or can be a metaCASE modelling approach, such as MethodBase and IPSYS ToolBuilder in the above meta model example.

In general, the action of constructing a software model using a method model is known as *software engineering*; similarly the action of creating a method model using a meta model is known as *method engineering*. Because of the complexity these models have to deal with, a software tool is normally required. The tool for software engineering is called a CASE tool, whereas the tool for method engineering is called a metaCASE tool. This is summarised in the figure 4.4.

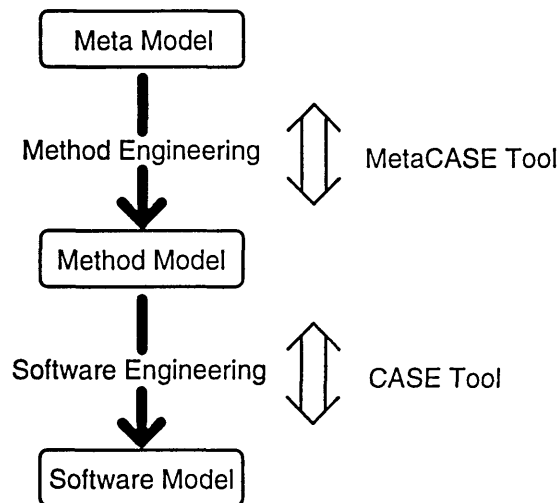


Figure 4.4 Three Levels of Software Development

4.4 SINGLE METHODOLOGY MODEL

In meta modelling technology, the formal presentation of a method takes an important role. Many definitions of a method have been proposed (such as [Lyytinen 89] [Prakash 94] [Wynekoop 93]) and most of them converge to the idea that a method is based on a group of concepts and consists of a number of tasks which should be executed in a given order. For instance, [Seligmann 89] proposes a framework for information systems development methods which comprises the *way of thinking* (the philosophy), the *way of modelling* (the models to be constructed), the *way of organising* (subdivided into the way of working, i.e. how to perform the development and the way of control i.e. how to manage the development) and the *way of supporting* (the description techniques and the corresponding tools).

In this section we are concerned with the formal description of a single methodology model. The result has emerged from the literature review on software development methods and meta-modelling techniques that were presented in the previous two chapters. Our proposal consists of a meta model which is, in fact, a generic method representation (GMR). The meta model identifies the constituents of a method and represents them in a semantic knowledge base (SKB). The single methodology model is summarised in figure 4.5.

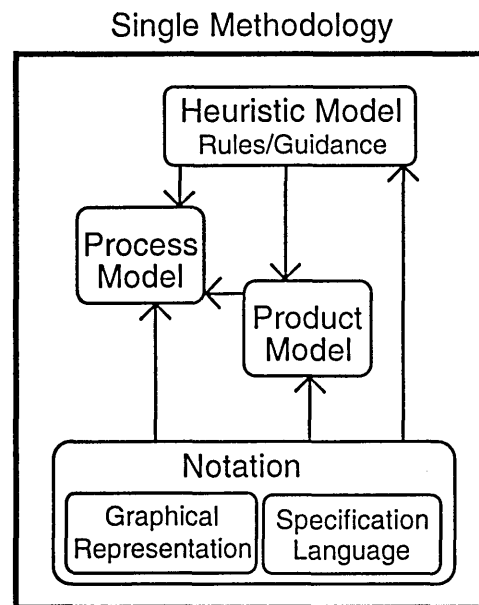


Figure 4.5 Semantic Knowledge Base

A method is composed of three constituents: the product model, the process model and the heuristic model. The product model describes *what* the concepts in the method are; the process model outlines *when* the tasks are applied, and the heuristic model documents *how* the developer can deal with these concepts and tasks. The product model supplies the concept information to the process model for task execution.

The product model encapsulates the *way of modelling*, which allows the construction of system concepts in a structured and detailed manner. Each idea or technique in a method is denoted as a concept in the model. Therefore these concepts present the basic elements for the functionality of the overall system, i.e. the inputs and outputs of each software development process. In metaCASE terminology, the product model represents the data structure of the entire method. Hence, the product model captures the way that a developer models a system.

The process model describes the *way of organising* the development. It includes how to perform software development and how to manage system **tasks**. Each task denotes a step in the development process and it is carried out by an activity known as **task function**. This task function incorporates the concepts defined in the product model. The ultimate goal is to construct a systematic way to develop a software product. The process model consists of a group of tasks arranged in some predetermined order of execution. This sequence may not be absolutely linear, but it is usually incremental and iterative. Individual tasks could even be optional steps as described in the analogous method. The major aim of the process model is to obtain a way to organise software development.

Both the product and process models are given graphical and textual notations. The graphical notation leads to an easy understanding of the methodological products and processes. It enhances the comprehensibility of the method and presents clearly the inter-relationships of concepts and tasks. In addition, both models are supported by corresponding heuristic information to guide the developer in the software construction activities.

The method model is designed to capture formal and heuristics knowledge able to support the development activities. The heuristic information can be presented as rules or guidance texts. A heuristic rule gives the arguments and the choices to assist in selection, whereas a heuristic guidance provides the definition and description of individual concepts and tasks. The heuristic model may assist the user in various ways. It serves as preliminary information to introduce different techniques in the method. It provides the specific assistance that the developer required in the current activity. It can also give context-sensitive help when a mistake is discovered in the current step. Therefore, the heuristic model describes the *way of supporting*.

All three constituents are documented by a formal description language known as **method specification language** (MSL). The semantic knowledge base is designed to support high flexibility and maintainability. Each separate component must achieve the requirement of high cohesion and low coupling so that they are autonomous entities to be shared amongst methods. The formal description is compiled to Prolog clauses to improve the ability of enquiry and ease of execution. The detailed structure of these three models will be shown in the next three subsequent chapters.

4.5 KNOWLEDGE BASE REPRESENTATION

Different constituents of a method comprise different aspects of semantics. As shown in the previous section, constituents are closely related to each other to achieve the aim of software development assistance. The semantics arising in even a single method is large³ and the overall figure handled by a multiple methodologies system is enormous. Therefore the representation of the method semantics is a key issue in the knowledge base. To produce an effective method engineering or integration system the structure of semantics is significant. Hence the following points are considered in modelling the knowledge base:

- The representation must be kept as generic as possible so that it does not bias towards any specific method or development paradigm. It is important to keep the uniformity of the semantic model to permit standard ways of comparing and evaluating methods.
- Since the amount of semantics managed is massive, speed is a crucial factor when constructing a workable environment. An effective data retrieving (such as caching) and searching (such as indexing) mechanism may be implemented to increase the performance.
- Like any other meta models, both textual and graphical representation of the method semantics should conform to a small set of notations.
- To support the expert capability of the system, the knowledge base must be stored in an executable manner. Data manipulation, such as dynamic binding, can be carried out directly on the semantic knowledge. This also allows heuristic rules including checks for design completeness and system consistency.
- In addition, the common model allows incremental evolution of the knowledge base such that new semantics or notations can be inserted in the future.
- Due to the time and space constraints of the knowledge base, it is desirable to store the semantics efficiently. The representation must keep as small in size as possible.
- The knowledge base representation should also provide some simple ways to dissect reusable components of a method. These components allow semantic sharing for method integration. The technique must be explicit to the data structure rather than depend on propagating functions.

To support the above points we have chosen to store the knowledge base as horn clauses in Prolog. This permits rapid prototyping and an efficient data repository. LPA Prolog was identified as a suitable language for the purposes of this research. [Steel 94]

³ The number of crucial semantics in a single methodology is around the order of a few hundred Prolog clauses.

4.6 METHODOLOGY PROTOTYPING

The components of a single methodology as described in section 4.4 are only the passive components of methodology prototyping. The whole system will require a program to drive the semantic knowledge. Such a system consists of four main parts as shown in figure 4.6: a method controller, a concept dictionary, a reference system and a context memory. The requirement specification prototyping examines the problem domain and environment, and generates a list of requisite concepts. These concepts are then given to the method prototyping, and they are matched against the concepts in the canonical concept dictionary. The method controller searches the semantic information in the multiple methodologies semantic knowledge base. Matching concepts or fragments may be used to construct an appropriate method for that particular application. Such method semantics are input to a competent metaCASE tool to form a suitable CASE tool.

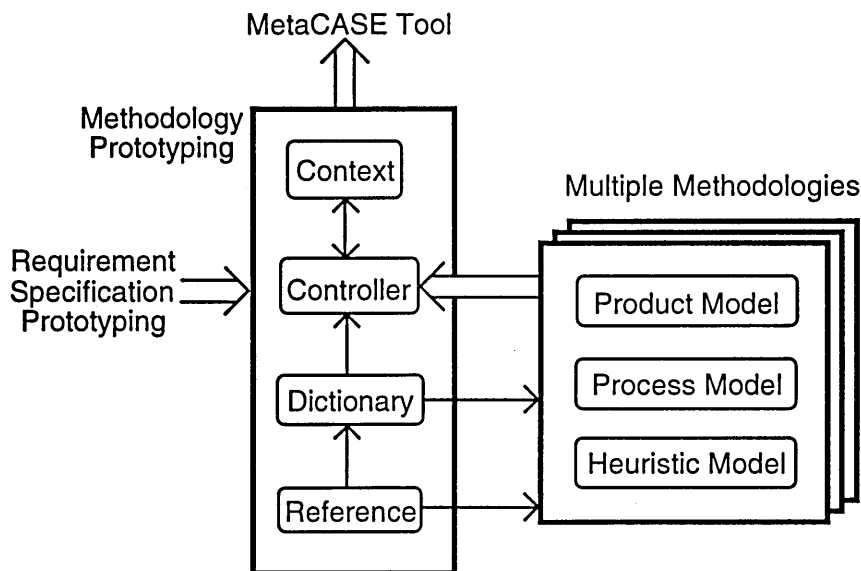


Figure 4.6 Proposed Methodology Prototyping

The method controller is the driving program of the methodology prototyping. It communicates directly with various parts of the semantic knowledge base. The product model is used for matching purposes. The process model shows the capability of creating sharable fragments. The heuristic model provides guidance for the above two models. The context memory is the working memory. It should be closely related to the controller to store information such as instantiated values, backtracking tree and the newly composed semantic model. Moreover, the canonical concept dictionary and the system references are the internal data of the method controller. They give collected information about the semantic knowledge base. These components for methodology prototyping are discussed in the following subsections.

4.6.1 METHOD CONTROLLER

The method controller is an executable program, with a structure dependent on the complexity of other parts of the system, such as the semantic knowledge base. The static information of methodology prototyping is stored separately in the concept dictionary, which is described in the next section. The internal structure of the controller is complex. It is impossible here to describe the full details of the controller, but we do list the basic features that it must contain.

Inference Engine - An intelligent guidance system that leads the user through the process model or even chooses method fragments according to the need of their chosen application domain. The engine indicates which step or rule that the system is considering and provides useful help prompts. Some heuristics in the semantic knowledge bases are deduced from work experience. A developer may refer to them as guidance, but the system will allow the developer to override the rules in certain circumstances.

Explanatory Program - The controller shall always be able to answer questions such as why it is doing this or how it has reached such a conclusion. This is a basic requirement for an expert system. During software development there are many heuristic decisions that the developer has to make. These are not just based on the problem/application domain requirements, but also upon the software requirements, such as time-space constraints, software libraries available and budgets etc. An explanatory program allows the developer to look at the heuristics underneath the system. These considerations also apply to the user defined sub-rulesets below.

User Defined Ruleset - The user must be allowed to create their own ruleset for their particular environment or application. This ruleset must integrate with the pre-defined system ruleset and adopt the same format. It may also require a consistency checking program to detect any contradictory or redundant rules.

Human Computer Interface - A complex system such as this must employ a user friendly interface. WIMP technology is used, which also provides menu selections, pop-up messages, hypertext helps. Hot keys, such as exit, help, info, forward, backward etc, must be supported at any point in the system.

Text and Graphical Editors - Since the design representation involves both text and graphical notations, the controller must provide facilities to edit text and graphics. A facility for cross checking these two representations and any attached context sensitive help must be provided. Therefore these must be closely related to the inference engine and any explanations which may be presented to the user.

4.6.2 CANONICAL CONCEPT DICTIONARY

A canonical concept dictionary is a crucial component in methodology prototyping. It not only provides the canonical concepts for semantic matching, but also provides the concept description in the semantic knowledge base. The concept dictionary can be used to provide context-sensitive help information.

From a method engineer's point of view, it is an index book for selecting concepts in software development methods. The dictionary is a set of canonical representation of concepts, which are linked to the corresponding semantics in the knowledge base. If a method fragment matches most of the requirement concepts it can be dissected from the method and used to develop the current application.

From a software developer's point of view, he/she can either highlight the appropriate object on screen or get the description through the *help* button. Each key word or key phrase is given a token key, which is used to search through or to select from the dictionary database. The following Prolog clause shows the definition record of token key object, its contents include a *description*, an *examples* field, a *seeAlso* field and a *sources* field. The *seeAlso* field allows the user to look at other related records and the *sources* field points to the (physical) source of the original citation.

```
dict(object,[
    desc('An object is a concept, abstraction, or thing with crisp boundaries and
        meaning for the problem at hand. Objects promote understanding of the real
        world as well as provide a practical basis for computer implementation.'),
    examples([Joe Smith, Simplex company, process 7648 and the top window]),
    seeAlso([class, identity, objectClass, objectModel, objectDiagram, type]),
    sources([ref(rumbaugh91)])
]).
```

One can imagine the *desc* and *seeAlso* fields to be analogous to the *text* and *link* fields in the heuristic knowledge base (see chapter seven). However, the concept dictionary is used to relate the terminology of a method, and no heuristic rules should be included.

Apart from providing the definition of the terminology of a single, exclusive method, the concept dictionary serves another significant purpose. That is to filter out any contradictions or similarities of terms between methods. This is an important issue in supporting the communication through tool fragments.

The canonical concept dictionary has a literary bias. It is not the central repository. This is provided by the knowledge base itself, of which the dictionary is a part. Perhaps the terminology is confusing but it has been chosen from the viewpoint of the method user/developer, rather than that of the software/database engineer.

4.6.3 SYSTEM REFERENCES

The system also provides a facility to refer all information to texts, for example books or papers. The example below shows the reference record of [Gomaa 93]. It includes the fields: *author*, *year*, *title* and *others*. The *others* field provides sufficient information to locate the material. In this simple example, a crude index has been used. In the software tool system to be developed, a user defined custom index which can include bookmarks, as well as commercial library indexing standards (ISBN and ACM indexes) is supported.

```
reference(ref(gomaa93), [  
    author(['Gomaa H.']),  
    year([1993]),  
    title(['Software Design Methods for Concurrent and Real-Time Systems']),  
    others(['Addison-Wesley'])  
]).
```

4.7 CONCLUSION

This chapter introduces an evolutionary approach to software engineering based on methodology prototyping. Various components for such prototyping approach are identified. The core discussion deals with the semantic knowledge base of the software methods. It is the crucial part of this software development approach. The three constituents, namely the product model, the process model and the heuristic model, are introduced. The structures of the three models are discussed in the subsequent chapters.

5. PRODUCT MODEL

Every year, software development methods (SDMs) are created by industrial developers or by academic researchers. Most of these methods borrow ideas or concepts from one or more well-proven methods. Some introduce new application (or domain) specific concepts. Although there is no single method that can be used for all situations, a meta framework can be used to develop methods to fit particular contexts.

This chapter proposes a common meta model to represent concepts in analysis or design methods. We consider a method to be a group of concepts interrelated by a finite set of primitive concept relationships. Both the concepts and the relationships have a list of concept properties to comprehend the meta modelling. This model is known as the product model as it represents the static, structural aspect of a method.

5.1 INTRODUCTION

We have looked at eighteen software development methods with regard to their method representations (see chapter two). Five well-defined SDMs are selectively chosen for detail in our method survey, they are OMT [Rumbaugh 91], Ptech [Martin 92], Codarts/DA [Gomaa 93], Booch OOD [Booch 91] and HOOD [Robinson 92]. On the other hand, a number of meta modelling research and metaCASE tool techniques have been investigated (see chapter three). Their aims and emphasis do not satisfy a generic model. This research explores this generic representation of the SDM.

From the investigations, we have identified the three distinct levels of models in the software development method, namely the meta model, the method model and the software model (see chapter four). Each higher level helps to model the level below. We have also discovered the three main components of a SDM: the **product model**, the **process model** and the **heuristic model**. The structures of these models and the relationships between them have been studied in depth. This generic method representation (GMR) has been tested and rectified by applying it to these chosen methods.

This chapter focuses on the structural aspect of SDM, that is the product model. It is represented graphically by concept diagram and textually by method specification language (MSL as described in chapter eight). The next chapter describes the behavioural aspect in the process model and the heuristic model is covered in chapter seven. Moreover, the concept diagrams of the five chosen methods are attached as appendix D.

In the next section we define the aims and objectives of the product model. Section 5.3 describes concept modelling, and presents different types of meta concepts, concept

relationship properties and meta relationships. Section 5.4 introduces some advanced concept modelling techniques to enhance as well as simplify the overall concept structure. Section 5.5 and section 5.6 discusses the process model of the product model and the zoom hierarchy of the product model. Section 5.7 presents a product model of the concept diagram. The last section gives a brief conclusion of this chapter.

5.2 AIMS & OBJECTIVES

This chapter aims to propose a common meta product model of SDMs. The following objectives are taken as basic principles:

1. It must support the semantics required by various SDMs, such that it does not incline to any particular discipline of software engineering. In doing this it is necessary to separate issues of notation from issues of semantics, and to recognise semantic equivalencies between different method concepts.
2. Every logical and physical idea is a concept itself (including its relationships and properties) and the concepts are interrelated by primitive relationships, i.e. subtyping, composition, grouping, linking and referencing (see section 5.3.3 for details).
3. A minimal set of notations shall be used to depict the complete set of method concepts and concept relationships of a SDM. In other words, the representation must be concise as well as precise.
4. Both abstract and concrete concepts are permitted; a concrete concept may be instantiated, whereas an abstract one can never be instantiated.
5. The model should represent the hidden (implicit) concepts of a SDM. These implicit concepts are necessary to comprehend the internal relationship of the concept model.
6. The concept model may compile into a frame based structure that can eventually translate into Prolog predicates and be used as an executable knowledge base.
7. It allows the user to draw the model by free-hand, with no box shading or bold arrows, etc. If it is developed in a computer graphical package, the final model shall be composed into an A4 size printer paper.

As it is intended to produce an equivalent Prolog knowledge base from the model, a basic naming convention is used throughout this thesis¹. The first letter of every word except the first is capitalised, and there are no spaces between words.

¹ Since all concepts, relationships and properties are formalised as atoms of a structure in Prolog, an atom must be a name starting with a lower-case letter, and containing only letters, digits and underscores.

5.3 CONCEPT MODELLING

The idea of the product model emanates from the IPSYS ToolBuilder data model, which is effectively an entity relationship model. It represents the notational concepts of SDMs as entities, and the relationships between the entities are mainly for navigational purposes. Apart from the subtyping and composition relationships, the associative relationships between concepts are not formally represented, though these relationships describe important semantics in SDMs. IPSYS ToolBuilder can avoid defining these semantics because their navigational meanings are of lesser importance, also they are probably harder to incorporate in ToolBuilder's process model.

However, as our aim is to discover a generic model for representing SDMs, these significant hidden concepts must be identified and included in the product model. We also distinguish different types of method concepts according to their natural traits in the method model.

As mentioned earlier, the product model represents SDM in a diagrammatical form as well as textual form. The graphical representation yields a diagram called the **concept diagram** and the approach to formalise the model is known as **concept modelling**. Object-oriented techniques are incorporated with the model so that constituents of methods have higher cohesion and lower coupling. Each concept is encapsulated with its own properties and inherited features from their supertypes. The concept ownership is also investigated so as to minimise the coupling factor between fragments (see later). All these modelling techniques benefit reusability and fragment sharing. We shall first identify the concept types and the concept relationship properties, then discuss each concept relationship in detail. Other characteristics and features of the product model are described in later sections.

5.3.1 CONCEPT TYPES

MethodBase identifies different types of concepts according to the conceptual aspect, whereas our approach is based on the concept's natural distinctions. Two categories of concept types are defined: **meta-object** concepts and **meta-association** concepts. A meta-object concept is an intrinsic method concept that is not used to relate concepts as meta-association does. A meta-object concept can be a simple entity concept, a property concept or a fragment concept. In contrast, a meta-association concept is a method concept used to interrelate meta-object concepts or even method concepts with the same type. It can be either a link concept or a group concept. Each meta-association concept can also have its own properties. Both meta-object and meta-association concepts are represented in a concept diagram, that is they are both denoted as a structure in the product model. A concept is shown as a round-angle box in the diagram. An abstract concept is shown as a box with an additional diagonal line at the top left hand corner. Each concept is labelled by its name, as shown in figures 5.1b and 5.1c.

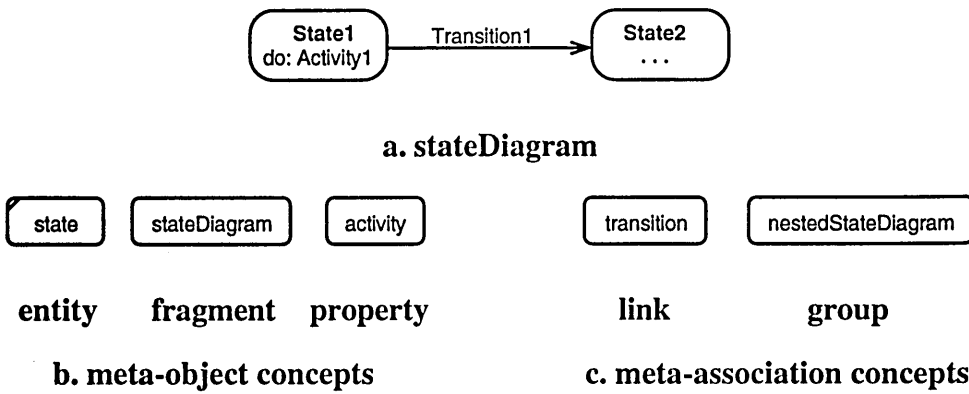


Figure 5.1 OMT: concepts in a *stateDiagram*

Figure 5.1a illustrates the meta-object concepts and meta-association concepts in a simple OMT state diagram. The *stateDiagram* has only two *states* (named *State1* and *State2*), one *transition* (*Transition1*) and one *activity* (*Activity1* in *State1*). The following five subsections introduce the five concept types.

5.3.1.1 ENTITY CONCEPT

An entity concept is a basic concept. It is normally owned by a fragment concept (diagram or text). It can be described in a method, either explicitly or implicitly. For instance, in OMT, a *stateDiagram* owns a number of *states*. A *state* concept is only an abstraction of *startState*, *interState* or *stopState*, therefore it is denoted as an abstract entity concept in figure 5.1b. An entity concept can relate to another entity concept by a link concept and to another meta-object type concept through a group concept. The *stateDiagram* concept is a special type of entity concept known as a fragment concept.

5.3.1.2 FRAGMENT CONCEPT

A fragment concept is a high level entity concept, which is either a diagram fragment, a text fragment or an aggregate of other concepts. Fragment concepts must be owned by other fragment concepts and itself may own a number of entity type concepts.

A **diagram fragment** is an explicit concept, normally known as a graphical tool fragment. For instance, the *objectDiagram*, the *stateDiagram* and the *dataFlowDiagram* are the diagram concepts of the OMT methodology. An abstract diagram fragment can also be subtyped to concrete diagram fragments; for instance in Codarts/DA, the *systemContextDiagram* concept is the subtype concept of the supertype fragment *contextDiagram*.

A **text fragment** is a structure text or pseudo code fragment, and is normally used for detail description purposes or code generation. For instance, a *dataDictionary* in OMT and an *objectDescriptionSkeleton* in HOOD are text fragments.

An **aggregate concept** is a concept that cannot be classified as either diagram or text fragment, rather it is an aggregate of a number of entity concepts. For instance, all SDMs have an ultimate aggregate called by the method name, such as *objectModelingTechnique* in OMT. This concept comprises a number of diagram fragments and text fragments. An aggregate concept is normally an implicit concept in the meta model.

5.3.1.3 PROPERTY CONCEPT

A property concept is an attribute of an entity concept or of a meta-association concept. In other words, a property concept must be owned by one of these concepts and itself does not own any concepts. For instance, in a *stateDiagram*, an *activity* concept is a property concept of *state*, whereas in an *objectDiagram*, *role* is a property concept of *association* and *qualifier* is a property concept of *qualifiedAssociation*. A property concept is a terminal entity concept, which means it can neither own (composition relationship) or relate (grouping or linking) to other concepts. However, a property concept can be subtyped and can refer to other concepts. For instance, the *association* concept has attributes *sourceMultiplicity* and *targetMultiplicity*, which are subtypes of the abstract property concept *multiplicity*.

5.3.1.4 LINK CONCEPT

A link concept is an association concept. It relates two entity concept instances together, one as its source and the other as its target. The ownership of a link concept is shared amongst the source and target concepts, but they must eventually be owned by the same fragment concept. For example, *transition* is a link concept in *stateDiagram* as shown in figure 5.1c. The source part can be a *startState* or an *interState* and the target part can be an *interState* or a *stopState*. All these *state* instances belong to the same *stateDiagram* instance (figure 5.11a).

5.3.1.5 GROUP CONCEPT

A group concept is another type of association concept. It is similar to the link concept except that it is not used to relate the source and the target concepts, but rather the host and the element concepts in a grouping relationship. Group concepts are implicit and they are method concepts that normally are without notation. For instance, *nestedStateDiagram* is a group concept with *state* as the host and *stateDiagram* as the element. Group concepts can also relate concepts in different fragment, see later for details.

5.3.2 CONCEPT RELATIONSHIP PROPERTIES

The SOCRATES project describes concept relationships by specialisation and association. Objectification is an extra feature when the association can be encapsulated as a concept itself. However, there are some useful primitive relationships that have a clear description and occur frequently in concept modelling. We define these relationships as subtyping, composition, grouping, linking and referencing. Before we look at them in detail, it is useful to identify the properties of a concept relationship.

Some meta modelling techniques suggest a rich description of concept relationships. For instance, the OPRR model in MetaEdit identifies property types and role types, whereas the concept structure in SOCRATES defines roles and role numbers in association. However, the other approaches simply ignore their existence, such as the product part in MethodBase. The main reason is the different approaches have different perspectives. MetaEdit is a metaCASE tool, and properties such as roles and cardinalities are important to construct a target tool; SOCRATES represents an information modelling knowledge to capture conceptual details. However, MethodBase needs to demonstrate meta modelling for method integration, whereas the detailed concept properties have less significance.

Nevertheless, in order to give a true and complete representation of SDMs the concept relationship properties are investigated. We look at seven properties, namely cardinality, optionality, directional, role, constraint rule, overlapping and completeness features.

5.3.2.1 CARDINALITY

There are two ways to describe cardinality of relationships. The first way is adopted by most entity relationship models, and is normally known as multiplicity (OOSA, OMT uses this term interchangeably with 'cardinality'). In this method, there are only three types of cardinality : 1-to-1, 1-to-many and many-to-many relationships. Sometimes (such as OOSA), conditions can be added onto the relationship. The second method is to include both minimum and maximum cardinalities on both sides (such as cardinality constraint in Ptech). It allows relationships such as 0-to-1, 0-to-many, 1-to-2 etc.

In our model, we adopt the second method since it contains the optional and mandatory features (see next section). Hence, the cardinality of a concept relationship is a 4-tuple of (*minimumSourceCardinality*, *maximumSourceCardinality*, *minimumTargetCardinality*, *maximumTargetCardinality*). For instance, the cardinalities of grouping relationship *splittingControl* are (0,1,1,1) on the host side and (0,1,2,n) on the element side, as shown in figure 5.2. In cardinality, n stands for a many (>1) relationship.

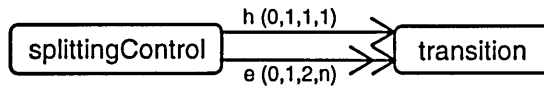


Figure 5.2 OMT: *splittingControl* Grouping Cardinalities

5.3.2.2 OPTIONALITY

In our product model, optionality is given by the *minimumCardinality* of each side of a relationship. If the cardinality is zero, the required concept is optional; otherwise the concept is mandatory. In the grouping relationship shown in figure 5.2, *splittingControl* is an optional group concept, whereas *transition* is mandatory in both host and element sides.

5.3.2.3 DIRECTIONAL

All concept relationships in our model are described in binary form. For identification purposes we present concept types in a relationship as source and target concepts. A bidirectional relationship is a relationship that is recognised by both concepts, whereas a unidirectional relationship is only recognised by the source concept. Subtyping, composition, grouping and linking are all bidirectional relationships, but the referencing relationship can be either bidirectional or unidirectional depending on the nature of the concept relationship.

Figure 5.3 illustrates the bidirectional linking relationship between *process* and *controlFlow*, and the unidirectional referencing relationship from *process* to *operation* in OMT.

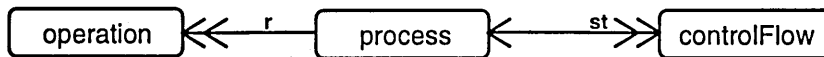


Figure 5.3 Bi- and Uni- Directional Relationships in *process*

5.3.2.4 ROLE

Some meta models, such as OPRR, allow user-defined role names. However, in a fixed simplified number of relationships, we choose to use a set of uniform roles. A role is a formal name given to a concept in a relationship. Each relationship connects two concepts, each concept is given a role for identification.

As mentioned in the previous section, all relationships are directional, so the role types can be known as *sourceRole* and *targetRole*. The uniform role names of relationships are shown in table 5.1.

disjoint/complete and *OI* stands for overlapping/incomplete. Moreover, the Martin/Odell Ptech partition notation (see section 2.4.4) also depicts the completeness feature. In figure 5.4b, the *human* subtyping shows a complete partition and the *employee* subtyping represents an incomplete partition. However, at meta level the distinction is not that significant, and most relationships are in the disjoint/complete category.

These features are not directly denoted in the concept diagram for the following three reasons. Firstly, a complete representation is required for our product model to documenting concept relationships, therefore no incomplete partition is allowed. Secondly, overlapping concepts are very rare in method modelling, and then they can be described by subtyping of multiple supertypes. Finally, if it is really necessary to describe these features, the particular relationships can be denoted as constraint rules in the concept heuristic model (see chapter seven for details). The next section briefly describes the constraint rule and method representation is discussed in chapter eight.

5.3.2.6 CONSTRAINT RULE

The relationship properties mentioned above should be able to document most conditions in a concept relationship. However, there are some special constraints which are very difficult to denote. We shall identify two examples to illustrate these description problems. The first one is a typical example in a *dataFlowDiagram* (also refer to figure 3.12) and the second one is a special complex pattern of *objectDescriptionSkeleton* in HOOD (see section 2.4.2).

In a *dataFlowDiagram*, as shown in figure 5.5, *dataFlow* is used to represent *data* passing amongst *process*, *dataStore* and *actor* (sometimes known as *source* or *sink*). However, the model should detect and avoid *dataFlow* connecting directly from *dataStore* to *actor* or vice versa. This is because *dataflow* only occurs through the *data* transformation in *process*, and by definition a static *data* transfer between *dataStore* and *actor* is not allowed. This can be considered as a meta relationship between the source and target links of a *dataFlow*. There is no easy way to document this constraint in the model.

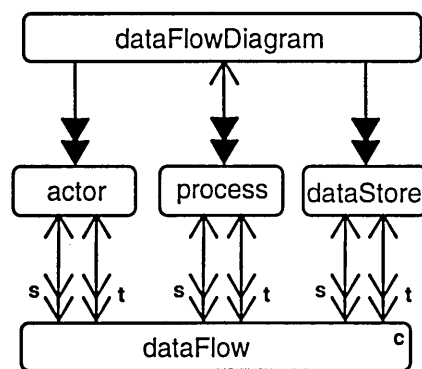


Figure 5.5 A Simple *dataFlowDiagram* Product Model

We shall now look at a more complicated example. In HOOD, *objectDescriptionSkeleton* describes each *object* by an *objectType*, where *objectType* must be one of *passive*, *active*, *environmentPassive*, *environmentActive*, *classPassive*, *classActive*, *instanceOf*, *opControl* or *virtualNode* (this list of concepts is in fact a complete composition partition of the concept *objectType*).

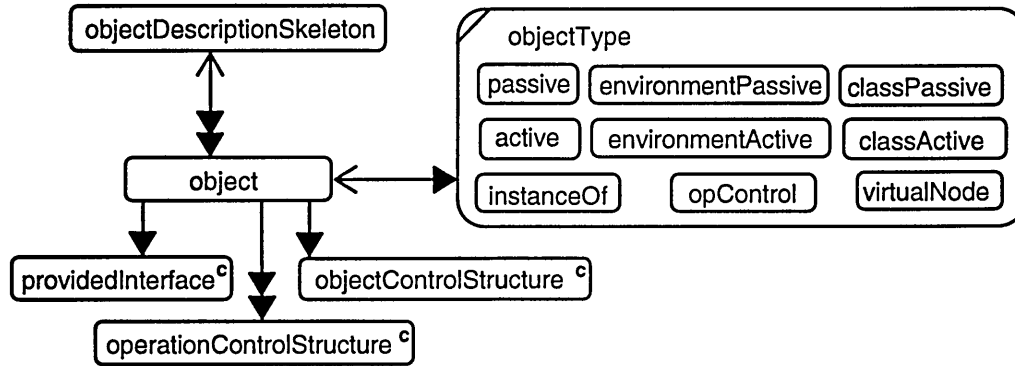


Figure 5.6 HOOD: *objectType* Subtyping

The *objectType* affects the layout of *objectDescriptionSkeleton*, since certain parts of the *objectDescriptionSkeleton* are optional. The following table shows the optional sections included according to *objectType*.

<i>objectType</i>	description
<i>passive</i>	no <i>objectControlStructure</i>
<i>active</i>	contains <i>objectControlStructure</i>
<i>environment</i>	<i>providedInterface</i> only
<i>class</i>	contains <i>formalParameters</i>
<i>instanceOf</i>	refers to <i>class</i> and has <i>parameters</i>
<i>opControl</i>	no <i>providedInterface</i> , <i>internals</i> contain one <i>operationControlStructure</i>
<i>virtualNode</i>	allocation to physical node always a parent so no <i>operationControlStructure</i>

Table 5.2 HOOD: *objectDescriptionSkeleton* layout based on *objectType*

There is no simple way to get round these representation difficulties, other than using a formal description of the internal constraints. In our product model we document them as concept rules in the heuristic model. The concept relationship or fragment is valid only if all of the rules attached with it are satisfied, i.e. a well-formed product model should necessarily meet the conditions (see section 5.5 for details). We place a letter ‘c’ at the top-right corner of the concept box to indicate that a constraint rule is bound to that concept. For instance, in figure 5.5, *dataflow* is a constrained concept, whereas three conditioned concepts are shown in figure 5.6.

5.3.3 CONCEPT RELATIONSHIPS

As with the product model at the method level, there is no uniformity in addressing meta level relationships. Each meta modelling technique has a different emphasis in its product model, such as a bias towards tool construction or towards a given formal method representation. The question is ‘*how specific or detailed should the relationship type be?*’ There are basically three approaches in addressing concept relationships. The first one is to allow no specific relationship types: all relationships are associations with different roles. For instance, the OPRR model in MetaEdit allows this flexibility. The second approach is a semi-free view, that is to allow a few basic relationship types with other types denoted as associations. For instance, the SOCRATES concept structure only allows specialisation and association, whereas the IPSYS ToolBuilder data model addresses subtyping, composition and reference relationship as a general association link. The third approach is to generalise all possible relationships in a framework and allow no more associations. We choose to adopt this third approach as we believe there is only a simple concise set of relationships in meta modelling. Moreover, it forces a uniformity in the method model. We have generalised the relationships among the concepts into five types: subtyping, composition, grouping, linking and referencing. This approach is also consistent with the MethodBase’s meta relationships.

5.3.3.1 SUBTYPING

A concept is said to be a subconcept of another concept if it inherits the properties of that concept. The latter is then called a superconcept. Subtyping is a bidirectional relationship between a superconcept and a subconcept. An abstract concept is a superconcept that does not have any instance. For example, in OMT, the *state* concept is an abstract superconcept of *startState*, *interState* and *stopState* whereas *automaticTransition* is a subconcept of *transition*. Subtyping is depicted by putting the subconcept box(es) inside the superconcept box, as shown in figure 5.7. It is intentional not to denote subtyping by an arrow. Firstly, it is the only relationship without cardinality; and secondly it gives a better ownership presentation of inherited features. A subtyping hierarchy can be induced from these interrelated subtyping relationships, since no cyclic subtyping is allowed. In concept modelling, subtyping is deduced from concept generalisation, but a specialised concept can override these features which are higher up in the hierarchy (see section 5.4.2).

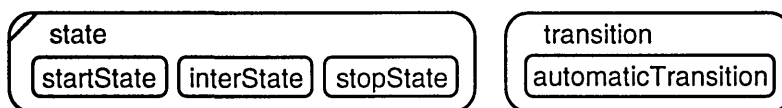


Figure 5.7 OMT: *state* and *transition* Subtyping

A subconcept with multiple superconcepts is possible. However in our survey of SDMs, no examples were found. The main purposes of subtyping in the meta model is to generalise common features, and this should be kept as simple as possible. The source and target roles are known as superconcept and subconcept respectively according to their nature. Subtyping is always a one-to-one relationship in meta level, although there may be a number of instances in a subtype concept. The cardinality is not as significant as other relationships.

IPSYS ToolBuilder supports subtyping relationships. Moreover, it allows various data models to be stored as modules. An entity in one module can be a subtype of entity in another module, hence another layer of multiple inheritance is permissible (see appendix B.2.4.1). However, this relationship can only be shown in the structured text and not in the entity diagram. In our meta model, we do not encourage subtyping across concept diagrams, since it loses the semantic dependence. However, we provide different zoom displays which effectively give the same result and do not lose the dependence (see section 5.6 for details). MethodBase's product part refers to subtyping as *is_a* relationship, but the fundamental generalisation/specialisation structure amongst concepts is still the same.

5.3.3.2 COMPOSITION

Composition is a bidirectional whole-component relationship. Sometimes it is known as a containment relationship. A concept contains another concept if the concept is a component of the other, or the concept exists within another concept. Composition is denoted by an arrow from source to target with the solid arrow head(s) at the target side. The source and target roles of composition relationship are known as whole and component respectively. This whole-component relationship of concepts forms another type of hierarchy in a concept model. For instance, in OMT, a *stateDiagram* must have one or more *state*, and each *state* may have an *entryAction*, an *exitAction* and a number of *internalActions* as shown in figure 5.8. Since all the composition relationships flow out from the abstract *state* concept, these features apply to all *state* subtypes, namely *startState*, *interState* and *stopState*.

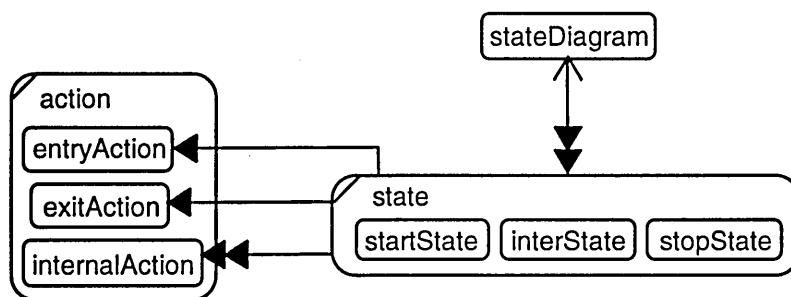


Figure 5.8 OMT: *state-action* Compositions in *stateDiagram*

As with subtyping, composition is always bidirectional. Therefore, a concept can navigate through other concepts to a root concept or vice versa. The root concept is an aggregate concept known as *method*, which comprises of a number of fragment concepts.

By definition, a component must belong to one and only one owner, but not necessarily the same concept type. In other words, the maximum source cardinality is always one, whereas minimum source cardinality can be either zero or one (depicted as question mark in the figure). The optional v-shape arrow head at the source side and the solid arrow head(s) at the target side, are used to depict the minimum and maximum target cardinalities respectively. The four possible combinations of composition cardinalities are shown in figure 5.9.

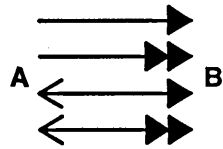



notation	description	cardinality	example	
	each A may have zero or one B	(?,1,0,1)	A	B
	each A may have zero or many B	(?,1,0,n)	state	internalAction
	each A must have one B	(?,1,1,1)	scenario	eventTrace
	each A must have one or more B	(?,1,1,n)	stateDiagram	state

Figure 5.9 Composition Cardinalities

In general, the minimum source and target cardinalities show the optionality of the corresponding concepts. For instance, the first example in figure 5.9 shows that each *state* must have zero or one *activity*, so *activity* is an optional composition for *state*. On the other hand, the third example shows that each *scenario* must have one *eventTrace*, hence this composition is manatory. MethodBase also denotes composition as a *has_a* relationship, though cardinality and optionality are not represented in its meta model.

5.3.3.3 LINKING

Subtyping and composition are basic relationships which form hierarchical structures. From this section onward, we identify three types of association in the product model. The first one is known as a linking relationship. Unlike subtyping and composition, a linking relationship is denoted by a link concept and normally it is physically shown in a fragment concept with a notation. A link relates two entity concepts, so it includes two parts: a source part connects a link to a source entity and a target part connects it to a target entity. Figure 5.10a shows the method model of a simple *stateTransitionDiagram*: *transition* is a link concept and both the source and target entities are *states*. Figure 5.10b demonstrates a possible software model for this *stateTransitionDiagram*. Due to the natural characteristics of a link, the cardinality tuple for both source and target parts is in the form of (0,1,0,1) or (0,n,0,1), (0,1,1,1) or (0,n,1,1), depending on the number of links and whether the link has a choice of different types of concepts or not.

In the simple *stateTransitionDiagram* example, each *state* may be involved in any number of *transitions*, whereas each *transition* must have exactly one source *state* and one target *state* since *state* is the only entity concept. Therefore, a linking relationship is depicted by a two v-shape arrow head on the link concept and one v-shape arrow head on the entity concept (many-to-1 relationship). The small letter 's' or 't' on the link concept is used to denote the source or target part of a link respectively.

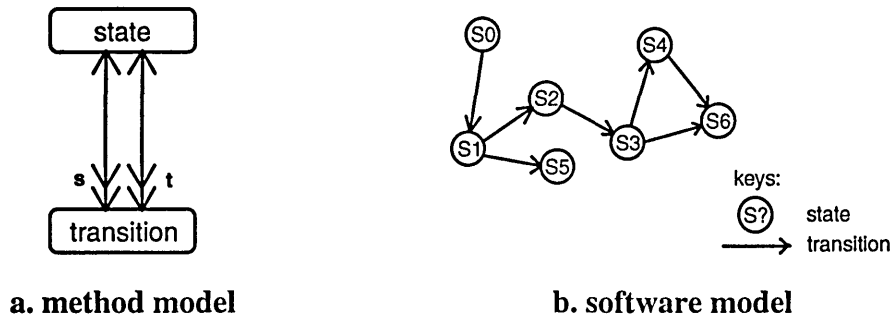


Figure 5.10 A Simple *stateTransitionDiagram* Example

Now let us look at a more complex model - the OMT *stateDiagram*. A *state* must be one of the *startState*, *interState* or *stopState*. A *startState* is only involved in the source part and a *stopState* only in the target part, whereas an *interState* is involved in both parts as shown in figure 5.11a. Therefore, there are four possible paths for *transition*: *startState* to *interState* (e.g. *transition* L1 in figure 5.11b), *interState* to *interState* (*transition* L2), *interState* to *stopState* (*transition* L3), and *startState* to *stopState*. The last path is abnormal, and it is forbidden by the constraint rule (see section 5.3.2.6 for details). Moreover, the link concept *transition* has a specialised type *automaticTransition*. An *automaticTransition* is a *transition* with no *event* (e.g. L1). This effectively doubles the number of possible paths. We defer the discussion to the exclusion section (see later for details).

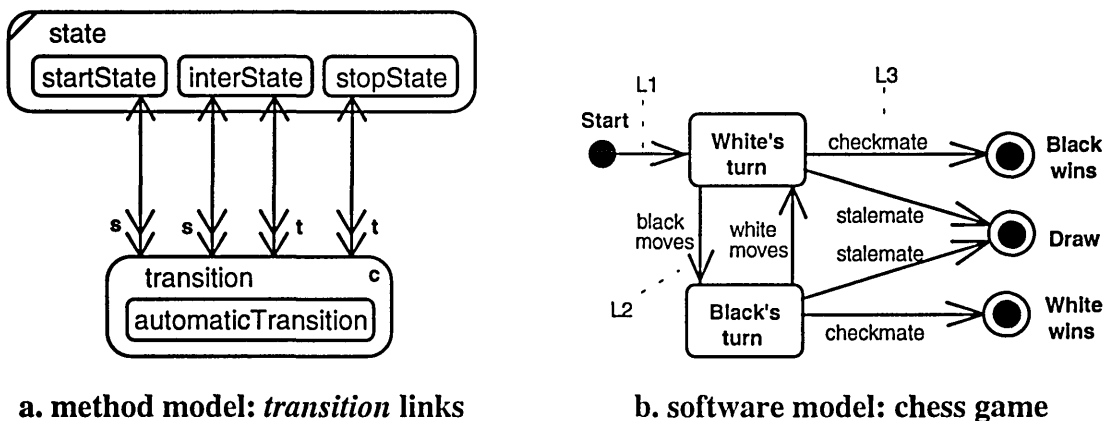


Figure 5.11 OMT: *transition* Links in *stateDiagram*

IPSYS ToolBuilder defines linking as composition. When the component part of a composition is the link type, the relationship can refine to a source, a target or both. This complex definition is demanded because the IPSYS ToolBuilder kernel requires that all non-root concepts are owned by another concept. And since the source and target concepts are the nearest neighbouring concepts, ToolBuilder forces them to hold the ownership. However, we believe that a link should be recognised as an objectified concept between two entity concepts. The ownership of a link should be shared amongst them and eventually lead back to the same fragment. In fact, we have implemented a simple fragment to illustrate this confusion. ToolBuilder have agreed that there is a problem in the link declaration, and they hope to fix it in later versions.

This linking relationship is also seen in the MethodBase product part, though they name the source linking as input relationship and target linking as output relationship. Again, MethodBase has emphasised the method representation, and relationship properties such as cardinalities and roles are omitted.

5.3.3.4 GROUPING

The second type of association in our product model is known as grouping. Similar to linking, grouping is used to relate concepts by an objectified concept. In this relationship, the concept is called a **group**. A grouping is always comprised of three components: a host concept, an element concept and the group concept itself. Figure 5.12 illustrates a grouping in OMT. The *nestedStateDiagram* is the group concept, *state* is the host concept and *stateDiagram* is the element concept. The small letters 'h' and 'e' are placed at the group concept end to denote the host and element roles of concepts in the association respectively.

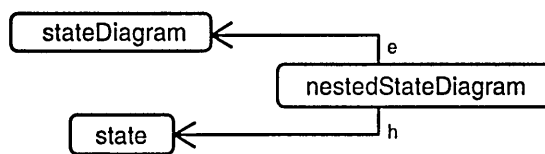
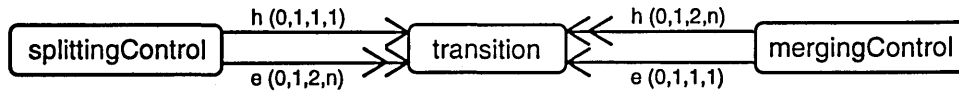


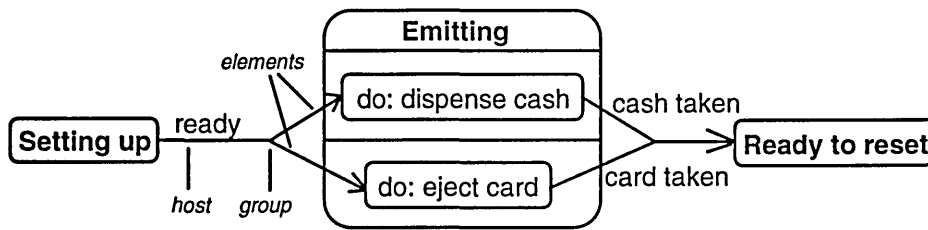
Figure 5.12 OMT: *nestedStateDiagram* Grouping

Unlike composition and subtyping, all grouping relationships are optional. In other words, the host or element does not depend on the group. For instance, a *dynamicModel* comprises of a number of *stateDiagrams* and a *stateDiagram* comprises many *states*. A *state* cannot stand alone without a *stateDiagram*. Nevertheless, *nestedStateDiagram* is a navigation concept that routes the host concept *state* to a lower level element concept *stateDiagram*, and either *state* or *stateDiagram* can occur in their own right. Grouping is normally an idea or a derived relationship between method concepts so, unlike linking, it does not have a physical notation.

A grouping has also two parts: the host part connects the group to the host, whereas element part connect the group to the element. Since a group must have at least one host and one element, the cardinality tuple for both host and element parts is in the form (0,1,x,y). Figure 5.13a shows the method model of OMT *splittingControl* and *mergingControl* in *transition*. A *splittingControl* must include of one and only one host *transition* (1-1) and at least two element *transitions* (2-many). The corresponding cardinality tuples are shown in the figure. Figure 5.13b illustrates the three components in a ATM dispenser software model.



a. method model: *splittingControl* and *mergingControl*



b. software model: emitting activity

Figure 5.13 OMT: *splittingControl* and *mergingControl* Grouping

Group concept can also relate concepts in different fragments. Figure 5.14 shows two decompositions in Ptech. Both *product* and *activity* are concepts in the *objectFlowDiagram* fragment. The *productDecomposition* grouping shows the detail of a *product* in an *objectSchema* fragment, whereas the *activityDecomposition* grouping gives the detail of *activity* in an *eventSchema* fragment. In these two examples, grouping is used for navigation paths, which provide a zooming effect from an entity concept to another type of fragment.

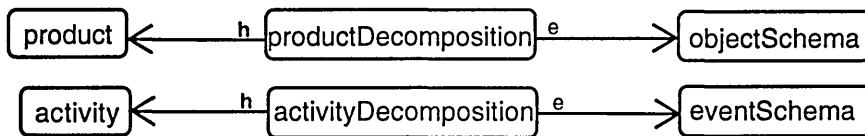


Figure 5.14 Ptech: *product* and *activity* Decompositions

IPSYS ToolBuilder does not address grouping relationships. However, navigation of nested fragments or entity decompositions can be supported by a composition relationship. An object operation is defined in the host entity. Again, we find that this is confusing. MethodBase has also missed out this meta relationship.

5.3.3.5 REFERENCING

A referencing relationship is used to relate similar concepts, but it appears under different aliases in different design aspects. This relational concept is important to link up concepts in different tool fragments of a methodology. Referencing is denoted by drawing a v-shaped arrow from the source concept to the target concept. The number of arrow heads depends on the cardinality and a letter 'r' is placed in the middle of the arrow. Figure 5.15 illustrates the referencing between the three models in OMT. We concentrate on the *objectModel* and the *functionModel*. A *process* in a *functionalModel* refers to an *operation* in *objectModel*, *data* refers to *attribute*, whereas both *actor* and *dataStore* refer to an *object*. As in [Rumbaugh 91], *functionalModel* shows 'what has to be done' and the *objectModel* shows the 'doers'.

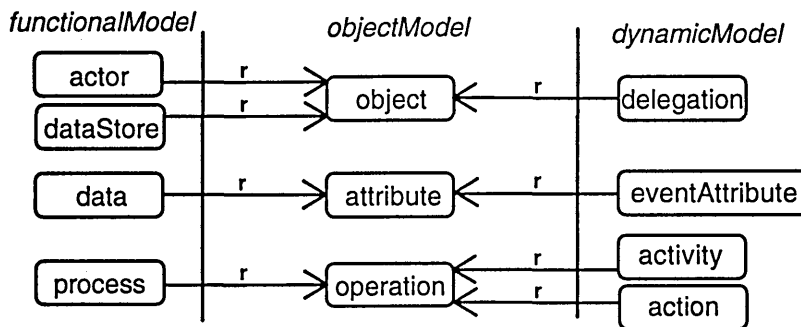


Figure 5.15 OMT: Referencing between Three Fragments

Fortunately all these relationships are uni-directional. However, bidirectional referencing relationship is also possible, for instance the referencing relationship between the *function* concept and the *relationship* concept is bidirectional, as shown in figure 5.16. In addition, an *object* is described in a *dataDictionary*, and the *dataDictionary* refers back to that specific *object*. This association is best represented as one-to-many referencing relationship.

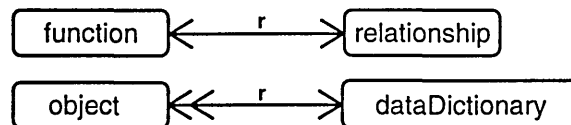


Figure 5.16 OMT: Bidirectional Referencing Relationships

MethodBase supports a *defines* relationship, which is more or less the same as referencing. However, it is more appropriate in a unidirectional relationship which has lost the bidirectional referencing characteristic. IPSYS ToolBuilder provides reference relationship, but in fact it is used for connecting any related concepts for navigation purposes. Figure 5.17 summarises all the notations in our meta product model.

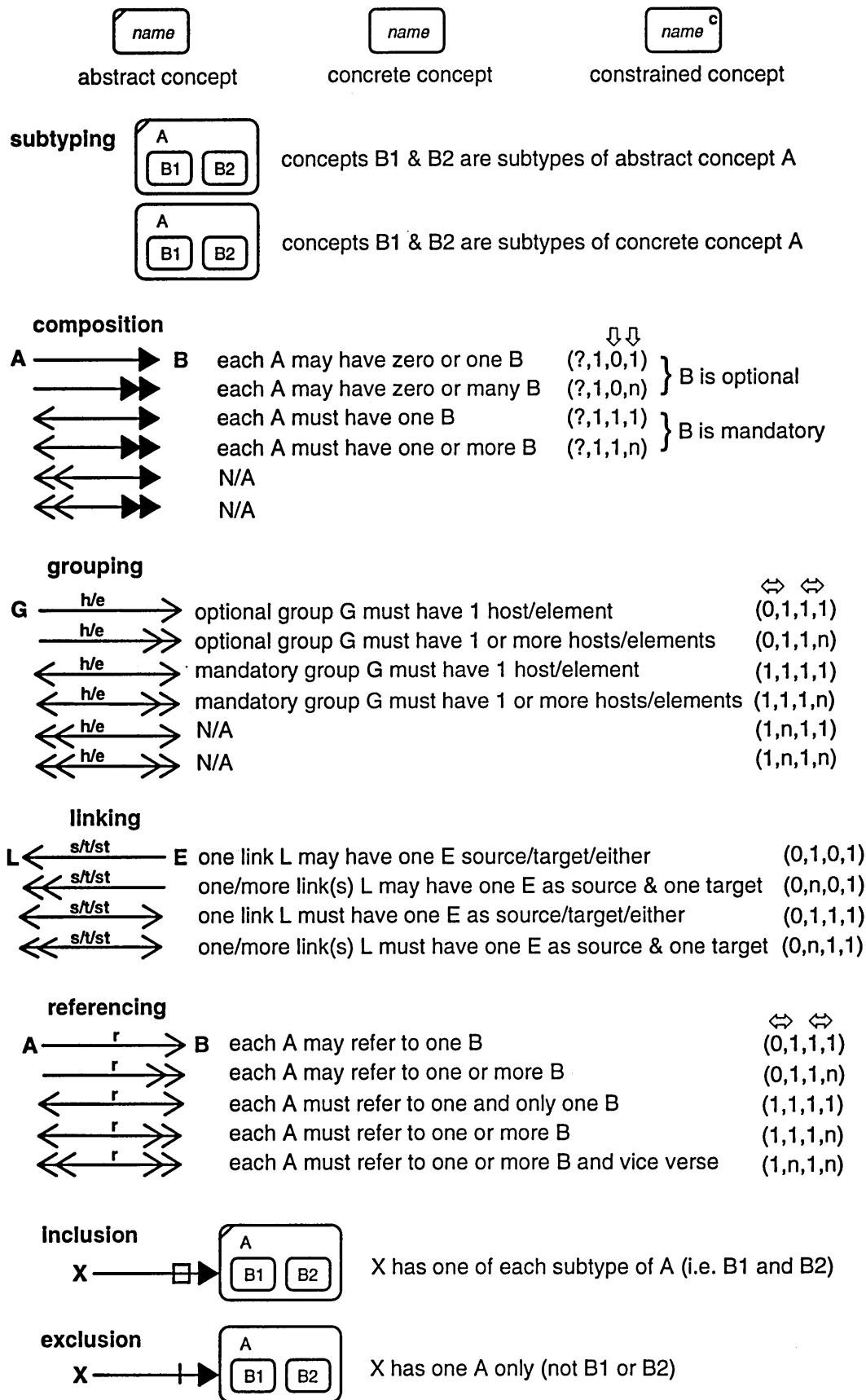


Figure 5.17 Concept Diagram Notations

5.4 ADVANCED CONCEPT MODELLING

Because of the number of concepts in a SDM, concept modelling is a very complicated and tedious job. The crystallisation of models into a common generic representation is also not a simple task. The new model must be concise and precise in its description. This section investigates a number of techniques that have been developed to solve various modelling problems or to reduce the complexity of the model. First of all, we shall investigate the possible relationships between entity concept and fragment concept.

5.4.1 AGGREGATION VS DECOMPOSITION VS REFERENCING

Software development vacillates between applications as well as within a single application. Each method allows a certain flexibility for the user to design within the 'limited' method concepts. In our meta model, there are three possible ways to denote the association between an entity and a fragment, namely aggregation, decomposition and referencing.

Aggregation is denoted as composition. There is an intrinsic asymmetry to the association: one concept is subordinate to the other. An aggregate (method fragment) consists of a number of constituents (method entities). For instance, a *stateDiagram* is an aggregation of its *state*, as shown in figure 5.18a.

Decomposition is depicted as grouping in our product model. A group is a concept, so a host concept (method fragment) may be expanded to an element concept (method entity). In figure 5.18b, a *state* can be decomposed to a *stateDiagram* through the *nestedStateDiagram* concept. However, considering each *state* owns a *stateDiagram*, the relationship is shown by composition as illustrated by the shaded colour in figure 5.18a. A role or a label is required to denote the type of relationships (see figure 5.29c for the two types of decomposition between *state* and *stateDiagram* in OMT). Nevertheless, the concept is hidden in the label of the composition and it is not identified as a concept of the method. This is a significant drawback of this representation. This relationship shall be considered as an optional host-element association rather than a whole-part ownership.

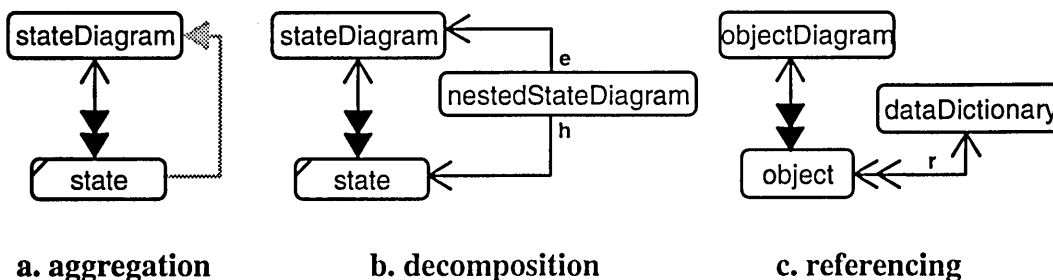


Figure 5.18 Aggregation vs Decomposition vs Referencing

IPSYS ToolBuilder adopts the composition approach for nesting diagrams. This is partly for navigational purposes and partly because of concept ownership management in the entity model. The main objective of the IPSYS ToolBuilder developer is to construct a method CASE tool rather than to formalise a method documentation.

The last type of association is known as referencing, which may also be called ‘defined-by’ relationship. An intrinsic entity concept is further described by another fragment. This is particularly useful when a concept is shared amongst two fragments, and one is a complement of the other. For instance, figure 5.18c shows the *objects* in *objectDiagram* are defined in the *dataDictionary*.

These three associations are distinct in their usages. In summary, aggregation is a whole-part ownership, decomposition is a host-element association and referencing is a defined-by relationship.

5.4.2 COMPOSITION VS REFERENCING

In some cases the representations for composition and referencing have very close meaning. The decision will depend upon a judgment between restriction or flexibility. This can be illustrated by an example. Figure 5.19a shows that the dynamics of an *object* is defined by a *stateDiagram*, the composition is optional. On the other hand, figure 5.19b depicts the relationship by referencing and the *stateDiagram* is owned by some other fragment concept, say a *method*.

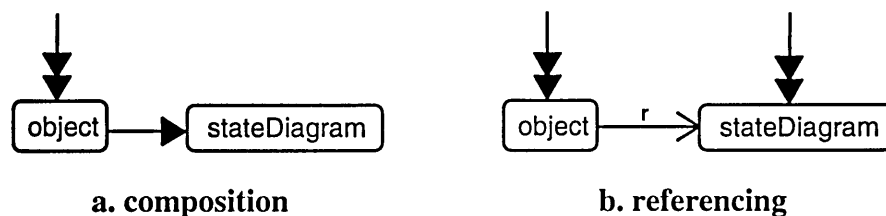


Figure 5.19 Composition vs Referencing

Figure 5.19a describes a *stateDiagram* to be a constituent of an *object*. Every *stateDiagram* must be applied to an *object*. There is a strong dependence in composition. On the other hand, figure 5.19b denotes that an *object* can be further described by a *stateDiagram*. The description is more flexible. Each *object* may refer to a *stateDiagram*, and the *stateDiagram* may describe a number of *objects* rather than just a single *object*. The dependence in referencing is weak.

Either method is permissible. The user has to decide on how specific they wish to be. In most cases, the referencing method is preferred as it provides flexibility in the design method.

5.4.3 MERGING SOURCE AND TARGET LINKS

As mentioned earlier, many link concepts connect entity concepts of the same type. It is convenient to merge the source and target parts together and instead of labelling the parts 's' and 't', use the combined form 'st'. This reduces the complexity of the concept diagram. Figure 5.20 illustrates a situation where the source part and the target part between *transition* and *interState* are merged.

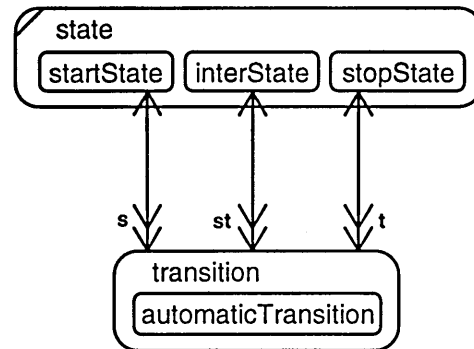


Figure 5.20 Merged Link between *interState* and *transition*

5.4.4 OVERRIDING

The default features of a superconcept can be overridden by its subconcepts. The overriding features can be shown diagrammatically by reproducing the relationship at the subconcept level and denoting the changes explicitly. Figure 5.21 illustrates the *instantiation* concept overriding its superconcept *relationship* and only allowing *instance* to be the target concept of *instantiation*. This technique prevents unnecessary duplication of linking relationships and allows generalisation from the superconcept level. The simplest way of resolving specialisation is to reproduce the common features of subconcepts.

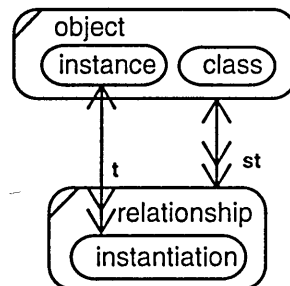


Figure 5.21 OMT: *instantiation* Concept Overrides the Superconcept Relationship

5.4.5 INCLUSION AND EXCLUSION

The relationships between terminal concepts can be easily described, but the relationships between concepts involving subtyping are more complicated, especially those with abstract superconcepts. Overriding is one possible technique to avoid complex representation of non-trivial relationship. In this section, we introduce two more notations to reduce the complexity as well as to resolve possible diagrammatic confusions. Figure 5.22 illustrates four complications that may occur. There is a composition relationship between concept B and a superconcept A, where B is the owner.

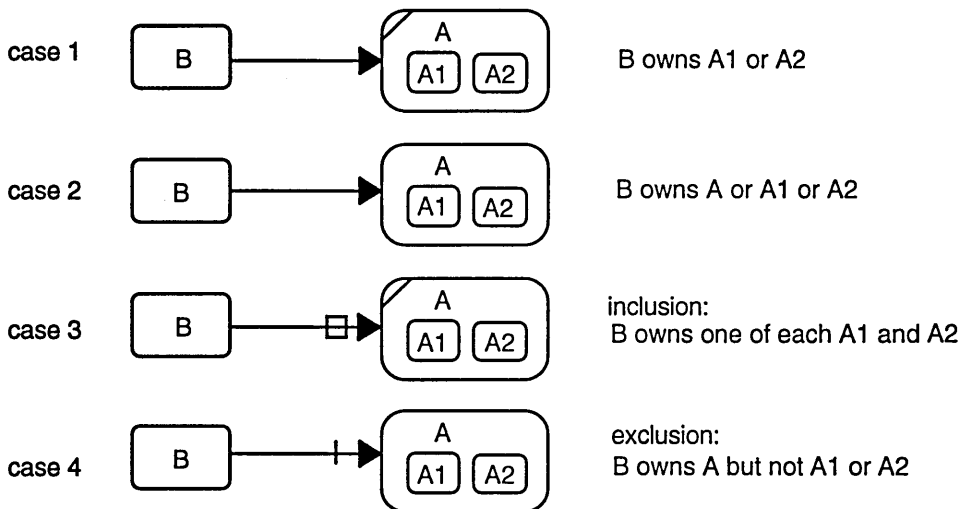


Figure 5.22 Four Cases of Possible Confusions

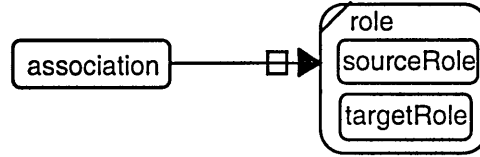
In case one, A is an abstract concept. B cannot directly own A, but it can own one of A's subconcepts, that is either A1 or A2. If the cardinality is multiple, B can have any number of A1 or A2.

In case two, A is a concrete concept, so B can own one of A, A1 or A2. If it is a multiple cardinality, then B can have any number of A, A1 or A2.

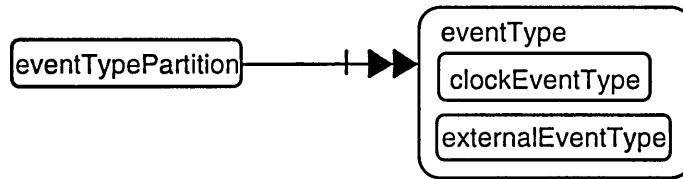
In case three, the square box stands for **inclusion**. That is, B may own one of each A1 and A2. This technique eliminates all arrows from the whole concept to each subconcept components. If the cardinality is multiple, the effect will be the same as case one. The inclusion has no effect.

In case four, the bar is known as **exclusion** (or a cut), which stops inheritance going down to the subconcepts. In the example, B can own a superconcept A but neither subconcept A1 nor subconcept A2. If it is a multiple cardinality, the cut is still valid. That is B can have any numbers of A but not A1 or A2.

Figure 5.23 gives an example for each inclusion and exclusion operation. In OMT, an *association* may contain one *sourceRole* and one *targetRole*, both of these are inherited from the *role* concept. In contrast, in Ptech an *eventTypePartition* contains a number of *eventTypes* but they cannot be *clockEventTypes* or *externalEventTypes*. A cut on the target illustrates this effectively.



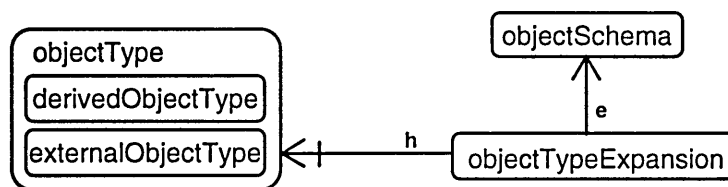
a. Inclusion applied on *association* to *role* Composition



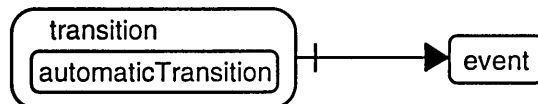
b. Exclusion applied on *eventTypePartition* to *eventType* Composition

Figure 5.23 Examples of Inclusion and Exclusion

The cut technique also applies to all the relationships described in section 5.3.3 apart from subtyping. This is one of the reasons that subtyping is denoted as a subconcept box inside superconcept box. Figure 5.24a illustrates the cut operation in a Ptech grouping relationship. The *objectTypeExpansion* group associates *objectType* as the host and *objectSchema* as an element, but *derivedObjectType* and *externalObjectType* cannot be expanded.



a. Cut Operation in a Grouping Relationship



b. Cut Operation applies on the Source Concept

Figure 5.24 Examples of Cut Operations

In addition, the cut operation can also be associated with the source part. For instance, a *transition* has an *event*, but an *automaticTransition* cannot have any *event*. A cut operation is applied on the source concept *transition*, as shown in figure 5.24b.

IPSYS ToolBuilder avoids confusion between case 2 and case 4 in figure 5.22 by making all superconcepts abstract. This is an alternative option. However, it requires the invention of some new concepts, and this means changing the semantics to meet the needs of the notation. Besides, it makes the concept diagram even more complicated. For example, in order to represent the cut operation illustrated in figure 5.24b, IPSYS ToolBuilder has to introduce a pseudo concept, say *eventTransition*, under the superconcept *transition*, and then to show the composition relationship directly from the *eventTransition*. Figure 5.25 shows the diagram in our concept modelling notation. The IPSYS ToolBuilder entity model produces a similar diagram except the diagonal line in the *transition* superconcept, since all ToolBuilder superconcepts are abstract concepts by default.

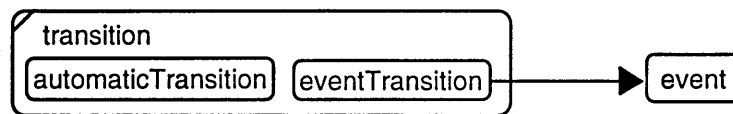


Figure 5.25 IPSYS ToolBuilder Notation for Resolving Cut Operation

5.5 PROCESS MODEL OF PRODUCT MODEL

We present a relatively simple product model of a real method fragment to show the representation power of this meta model. We have chosen the *stateDiagram* of OMT as our example because it is a common fragment amongst methods and it includes a lot of meta modelling notions.

The section also guides the reader through the process of concept modelling. The following seven steps are performed in constructing a product model:

- determine fragments in a method
- identify primary entities in each fragment
- identify relationships among primary entities
- determine properties of existing concepts
- develop complex groupings
- identify referencings between fragments or concepts
- define non-trivial constraint rules

Determining fragments in a method: Each method contains a number of diagram fragments and/or text fragments. For instance, HOOD has a diagram fragment called *hoodDiagram* and a text fragment called *objectDescriptionSkeleton*. These fragments combine together to constitute an aggregate concept of the method. In this example, we have concentrated on a diagram fragment, though a text fragment should have the same modelling steps.

Figure 5.26 shows the three diagram fragments in OMT analysis phrase, they are the *objectModel*, the *dynamicModel* and the *functionalModel*. Normally we can identify the cardinalities of these compositions straight away. In this case, all three models are optional.

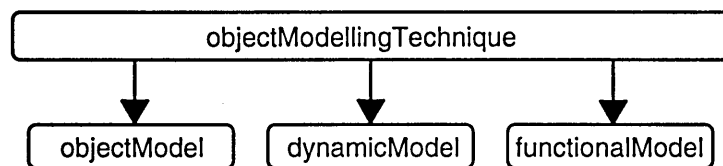


Figure 5.26 Step One: Fragment Concepts

Identifying primary entities in each fragment: Primary entities are distinct concepts in a tool fragment. In a diagram they appear as a node and are shown as a box in the diagram. An OMT *stateDiagram* has only one primary entity that is a *state* concept. However, *state* is an abstract concept with three subtypes, which are *startState*, *interState* and *stopState*. Figure 5.7 shows the *state* concept as well as the *transition* concept, which is to be found in the next step.

Identifying relationships among primary entities: Diagram fragment links are used to represent relationships among all the primary entity concepts defined in the previous stage. These are normally shown as arrows or lines with different heads. An OMT *stateDiagram* has only one link, which is *transition*. Next in our method we have to identify the source and target parts of the link, and denote them as 's' or 't' respectively. All possible paths from source concept to target concept must be determined and any complicated constraints are identified for future reference.

Section 5.3.3.3 gives a detailed discussion about these points and how to apply overriding, inclusion and exclusion as appropriate. Finally, if the source and target parts of a link happen to be the same concept, the 's' and 't' denotation is combined by the merging technique as described in section 5.4.1. Figure 5.20 shows the optimised form of this linking relationship.

Determining properties of existing concepts: After the entities and their relationships have been identified we can look into the secondary concepts, which are the properties of existing concepts. In diagram fragments, these are normally additional notations on nodes or links. They are represented as different types of arrow heads, labels etc.

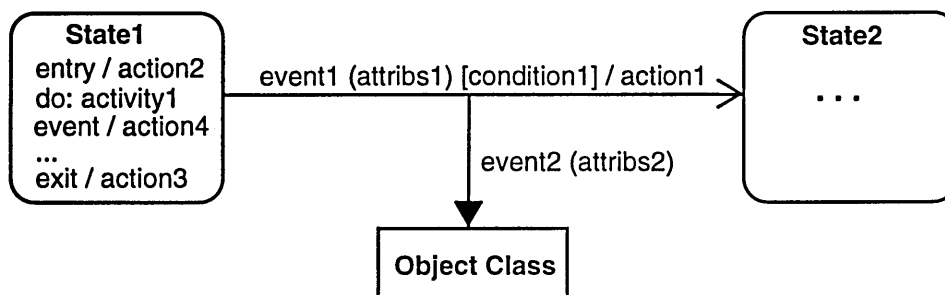


Figure 5.27 OMT: Notation for *state* and *transition* Properties

Figure 5.27 illustrates properties on the *state* and *transition* concepts. A *state* has an *activity*, an *entryAction*, an *exitAction* and a number of *internalActions*, whereas a *transition* has an *event*, an *eventAction*, a number of *eventAttributes* and a *guardCondition*. All these compositions are optional and the actions are subtypes of the abstract concept *action*. A *transition* can have a *guardCondition* with an *event*, but *eventAction* and *eventAttribute* stay together with *event*. Therefore *eventAction* and *eventAttribute* are components of *event* rather than *transition*. Inclusion and exclusion operations are applied as appropriate. Figure 5.28 shows the properties of *state* and *transition* concepts.

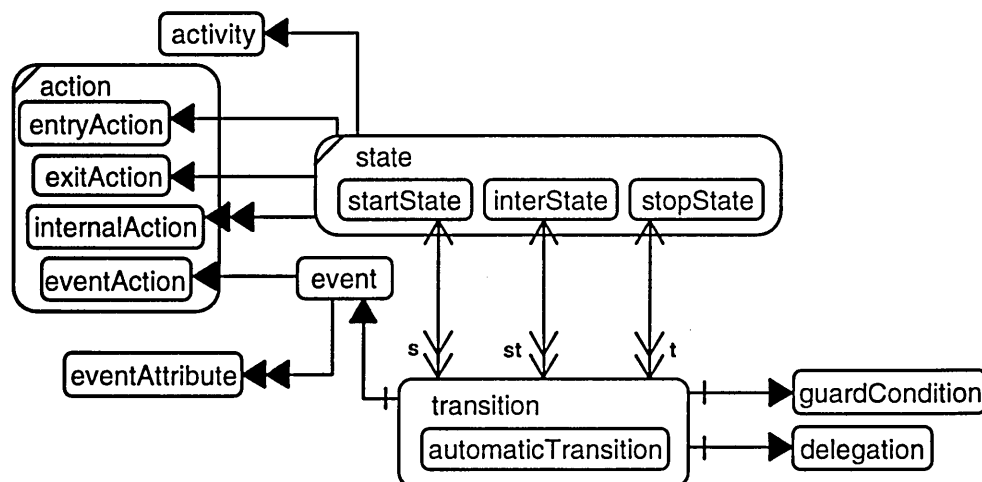


Figure 5.28 Step Four: Properties of Existing Concepts

Determining complex groupings: In this stage we identify all the hidden concepts in the fragment. These concepts do not have a notation, but are represented as group concepts. They are mainly used for navigation or concept transformations, and are of two main types. The first type is where a notation is transformed at the same level, for instance a link to a different form of link. Section 5.3.3.4 describes the OMT *splittingControl* and *mergingControl* grouping. The *splittingControl* grouping allows the transformation of a single *transition* to multiple *transitions*, whereas the *mergingControl* grouping gives the transformation of multiple *transitions* to a single *transition*. Figure 5.12 and figure 5.13 illustrate this grouping modelling technique. The second type is a transformation between different levels, for instance from an entity concept to a fragment concept. Figure 5.29a and 5.29b show the two ways to extend a *state* to a *stateDiagram*. The corresponding product model for these transformations is given in figure 5.29c.

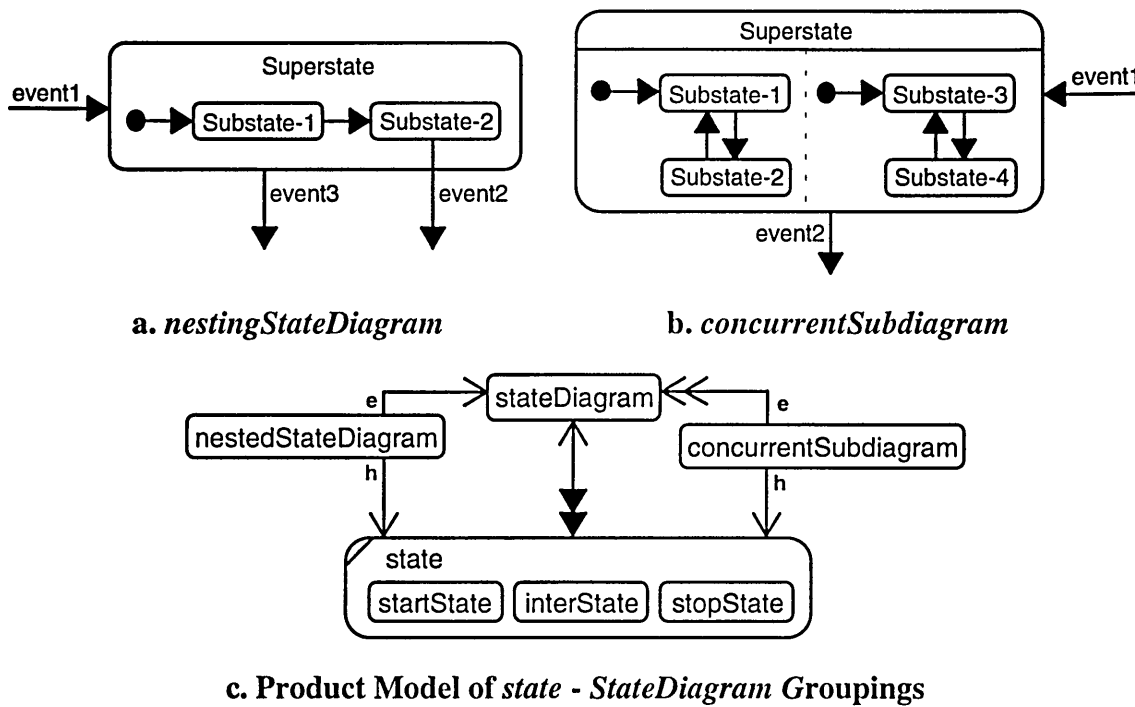


Figure 5.29 Step Five: Complex Grouping

Identifying referencings between fragments or concepts: In this stage we need to identify the referencing relationships discussed in section 5.3.3.5. Again, there are two types of referencing, one within a fragment and one between fragments. An OMT *dynamicModel* does not have referencing relationships within itself, but in its *objectModel* a *discriminator* refers to an *attribute*. Figure 5.15 illustrates the referencing relationships amongst the three tool fragments in OMT.

Let us look at the referencing relationships between an *objectModel* and a *dynamicModel*. A *delegation* from *transition* in the *dynamicModel* refers to an *object* in *objectModel*; an *eventAttribute* refers to an *attribute*, both *activity* and *action* refer to an *operation*. Again advanced modelling techniques are applied to appropriate parts of the model. Combining all results of the previous stages we obtain the product model of *dynamicModel* as shown in figure 5.30.

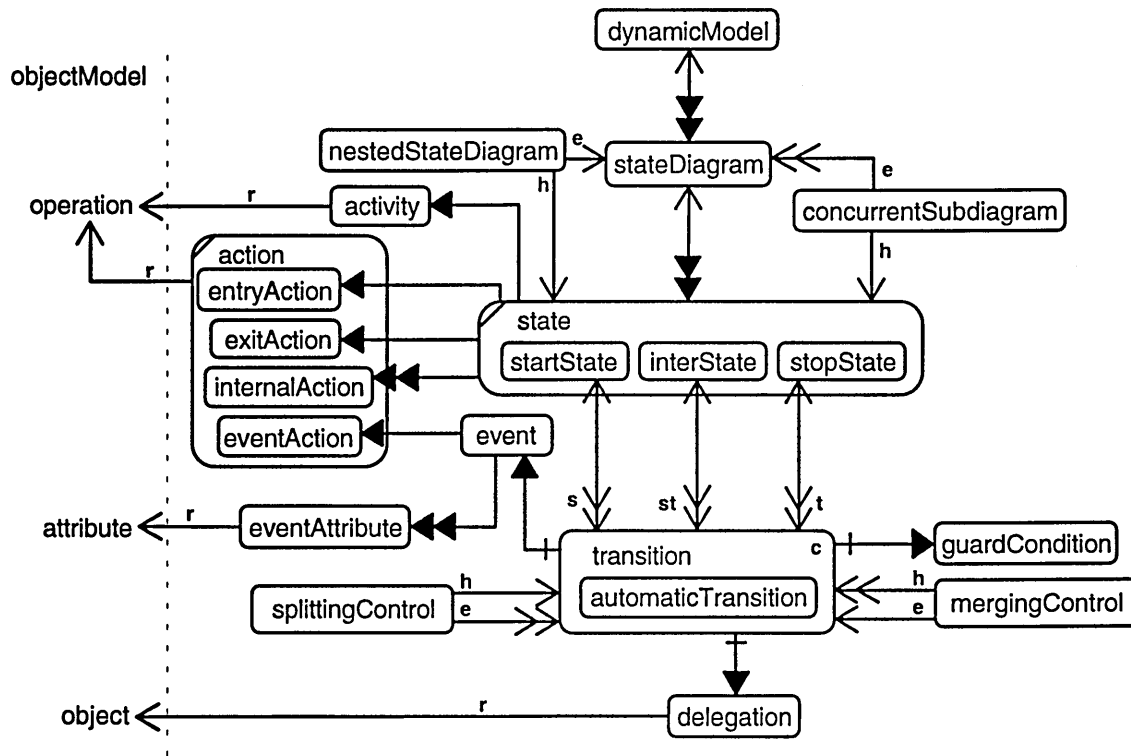


Figure 5.30 Final Step: a Complete Product Model of *stateDiagram*

Defining non-trivial constraint rules: This stage defines any extra constraints between concepts that cannot be represented in the concept diagram. We suggest that they should be formulated as rules by Method Specification Language. As shown in section 5.3.3.3, a constraint rule is required in the transition concept. That is used to avoid a *transition* flowing from the *startState* directly to the *stopState*. A constrained concept is denoted by a letter 'c' at the top right corner of the concept box. These concept rules or constraint functions are stored in the rule fields of the corresponding heuristic record (see chapter seven for actual details). The textual representation of the concept rule is also shown later in chapter eight.

5.6 ZOOM HIERARCHY

The complexity of a concept diagram depends primarily on three issues. Firstly, the number of meta concepts within a method can directly affect the complication of a model. Secondly, the cohesion factor of a fragment can alter the concept structure. A highly cohesive fragment produces a large number of meta relationships in a fragment. These relationships may cause many complex links between meta concepts. Finally, the complexity also depends on the coupling factor of a fragment. Contrary to cohesion, fragment coupling affects the number of meta relationships between fragments. If this factor is high, a large number of references occur, resulting in long and nested links in the diagram.

In order to reduce the complexity of the structure and to allow the developer to focus on individual features of the model we introduce a zoom facility. There are two zoom types and each of them forms a hierarchical structure on its own. They are known as *zoom by detail* and *zoom by feature*. The first one gives different details of the overall method, whereas the second one shows part of the method by individual feature.

5.6.1 ZOOM BY DETAIL

The three modes of this zoom dimension are known as overview mode, middle mode and detail mode. Each mode shows a method in a different level of detail. The **overview mode** gives a brief summary of a method by showing all fragment concepts. This is particularly useful when counting the number of fragments or viewing the interrelationship of individual fragments. The **middle mode** shows all fragments and all primary entity concepts, which may also include group concepts and link concepts. This mode is used if the secondary concepts or the complete concept model is not of interest. Finally the **detail mode** displays all meta concepts of the method.

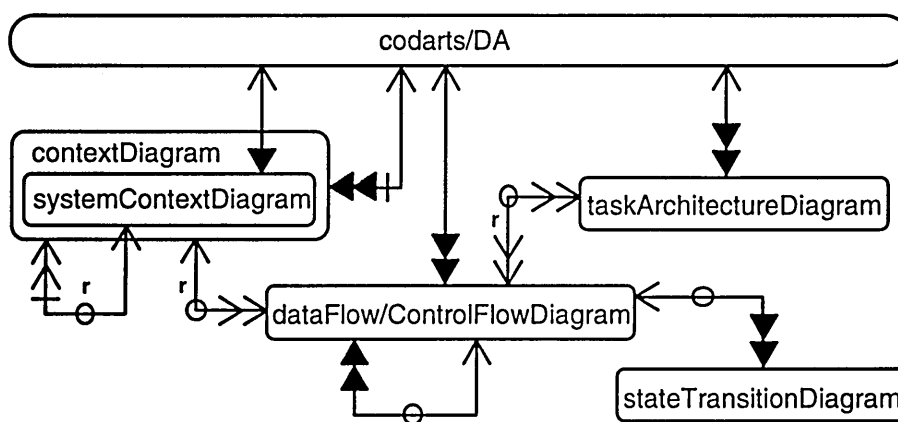


Figure 5.31 Codarts/DA: Overview Mode

Figure 5.31 illustrates the overview zoom mode of Codarts/DA. Bubbles are placed on the relationships to denote an interim concept or a list of cascade relationships between fragments. For instance, in the example the bubble shown on the composition relationship between *dataFlowControlFlowDiagram* and *stateTransitionDiagram* is a *controlTransformation* (entity) concept, whereas the bubble on the referencing relationship between *systemContextDiagram* to *contextDiagram* is a *contextDiagramDecomposition* (group) concept. If there is a list of different relationship types between fragments, the interim relationship will be shown as a referencing relationship, with the cardinalities still shown on the relationship.

Moreover, inclusion and exclusion operations can still be applied to the overview mode. For instance, each design has a number of *contextDiagrams* but there is only one *systemContextDiagram*. However, *systemContextDiagram* is a subconcept of *contextDiagram*, so a cut is placed on the the target side of the composition between *codarts/DA* (method concept) and *contextDiagram*.

5.6.2 ZOOM BY FEATURE

Similar to zoom by detail, this zoom dimension also has three modes: method mode, fragment mode and entity mode. These zooms are based on different types of feature in a method.

Method mode displays all the meta concepts and meta relationships of a method, which has the same effect as detail mode on the other zoom dimension.

Fragment mode displays the meta concepts and meta relationships of just a selected fragment of the method. This mode is useful when the developer concentrates on the details of a single fragment, rather than inter-fragment relationships. Figure 5.30 illustrates the *dynamicModel* fragment of OMT under this mode.

Lastly, the **entity mode** shows all meta concepts related to a selected concept, which can be a fragment concept, an entity concept or a group concept or a link concept. Figure 5.26 and figure 5.12 demonstrate a *method* fragment concept and a *nestedStateDiagram* group concept of OMT under this mode respectively.

5.7 PRODUCT MODEL OF CONCEPT DIAGRAM

Product models of SDMs can be represented graphically by concept diagrams, so the *conceptDiagram* fragment can be considered as a diagram fragment in a meta modelling method. This fragment in turn can be represented by the product model. Figure 5.32 shows the product model of the concept diagram by this modelling technique. The following points are noted:

- The *abstractConcept*, the *constrainedConcept* and the *sharedConcept* (see chapter eight) are the subtypes of the *concept* supertype. An *abstractConcept* is a *concept* that does not have any instances; a *constrainedConcept* is bound with a *constraintRule*, whereas a *sharedConcept* must refer to a *dissectionSet* (see chapter eight).
- Each *conceptDiagram* can zoom into a number of smaller scale *conceptDiagrams* by using the *zoomIn* concept, and a number of *conceptDiagrams* can zoom out to a larger scale *conceptDiagram* by the *zoomOut* concept (see section 5.6).
- The *abstractConcept* overrides the source part of a *subtyping* between *concept* and *relationship*, since the *abstractConcept* can only be a source concept of *subtyping*.
- Each *relationship* has one of each *cardinality* and *role*, but the *direction* concept only applies to the *referencing* relationship.
- The superconcept of *inclusion* and *exclusion* is not labelled. In fact, they are just notations to reduce complexity of a *conceptDiagram*.
- The *relationship* concept is a constrained concept because the source and target parts of relationship normally depend on its type. For instance, a *dissectionSet* can only be used to relate fragment concepts (see chapter eight).
- Finally, *optionality* does not actually show in the *conceptDiagram*, but it can be derived from the minimum cardinalities of a meta relationship.

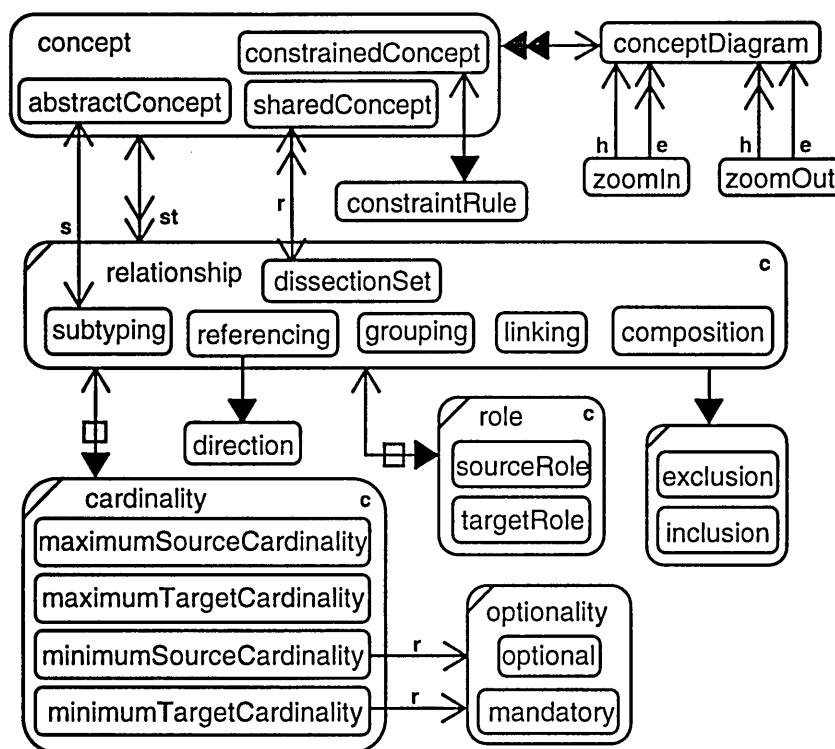


Figure 5.32 Product Model of Concept Diagram

5.8 CONCLUSION

In this chapter we presented a formal, structural and rigorous product model for SDMs. This meta model produces a unified documentation of method concepts. The model includes three main ingredients: meta concepts, meta relationships and relationship properties. The model is represented graphically by a concept diagram and textually by a method specification language. We have also illustrated the steps to be taken to form this diagram, and have explained a zooming technique to focus on the individual detail of a model. The product model is developed as part of this research project. We have tried out this product model in five SDMs, namely HOOD, OMT, Booch OOD, Ptech and Codarts/DA.

6. PROCESS MODEL

This chapter complements the method concept discussion in the last chapter. It documents the dynamic aspects of meta modelling, by describing the tasks that need to be undertaken to create a product model, and in particular, *when* these tasks should be applied. The method concepts are structured to integrate with method tasks, the latter denoting the software development processes. Each task comprises a task function and context parameters of the function are represented by concept tokens. The overall model in method representation is known as a **process model**.

6.1 INTRODUCTION

A process model describes those aspects of a system concerned with the sequencing of operations, such as events that mark changes, sequences of tasks, states that define the context for tasks and the organisation of tasks and triggers. The model also describes those aspects of a system concerned with transformations of concepts, mappings, constraints and task dependencies. Therefore, in principle, the process model is a series of tasks (normally in a network structure) arranged in the order of creating instances of concepts in method level. Software development as a process is a ‘creative’ activity since it involves a lot of decision making and task modelling. The process is also highly incremental and iterative.

We have investigated a number of process modelling techniques. However, many techniques describe process models at a software level rather than at a method level. This chapter presents a generic model to represent the method processes. We also suggest the following points:

- The process structure of method model should be relatively simple and flexible.
- The processes of individual fragments in a method should be arranged in such a way that they can be dissected and shared amongst SDMs, which can even be customised methods.
- The heuristic system must take an important role in the system, since the guidance and rules should be embedded in each method development process.
- The concepts described in the product model must also play a vital part in process modelling.
- The other major issue is the ordering of processes. The design processes of a method should be arranged in a particular order, and the corresponding pre- and post- conditions of each process should be checked at each stage of development.

The organisation of this chapter is as follows. The next section discusses different types of processes in meta modelling, section 6.3 describes a few basic issues of process models. Three preliminary approaches are introduced in section 6.4. Then we present the task functions, the task sequence and the task modelling in section 6.5, section 6.6 and section 6.7 respectively. Section 6.8 shows the meta process model and section 6.9 looks at the meta meta model. The last section gives a summary of this chapter.

6.2 META MODELLING PROCESSES

We have identified three types of process in our meta modelling approach. It is important to distinguish between them before starting the discussion on process modelling. The three types of process are the **method process**, the **metaCASE process** and the **CASE tool process**. Figure 6.1 illustrates these processes in three levels.

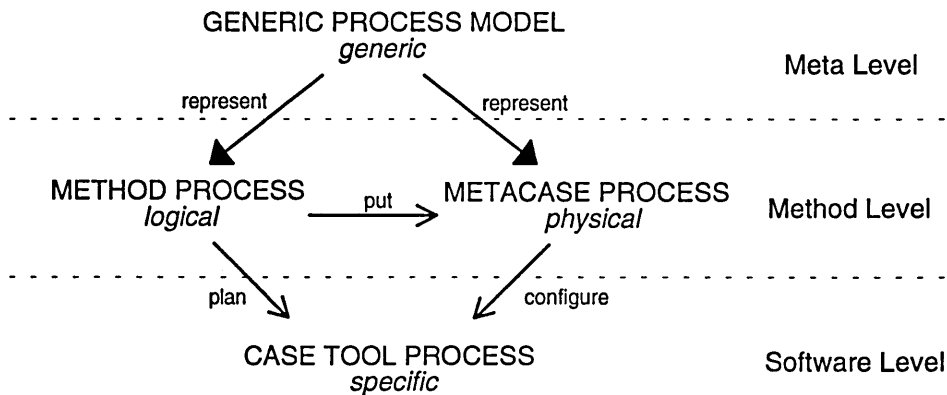


Figure 6.1 Meta Modelling Processes

The generic process model that is described in this chapter is in the meta (top) level. It is a generic meta model, and aims to represent the processes in the method level.

In the method level there are two types of models, namely the method process and the metaCASE process. A method process describes the development process that are presented in a SDM, whereas a metaCASE process describes how to map the method process into a metaCASE tool. Therefore there is a relationship between these two processes. The method process must be converted and placed into the metaCASE tool. We denote the relationship between these two processes by the put association as shown in figure 6.1.

The bottom level is the software level, where a CASE tool is generated from a metaCASE tool. The CASE tool process depends on the method process, as well as the facilities of the metaCASE tool that provide a configurable process.

The details of these three types of process are discussed in the following subsections.

6.2.1 METHOD PROCESS

A method process is the logical process described in a SDM to show the design sequence of the method. Certain literature refers to this as a ‘cookbook approach’, because in theory, a developer can follow the design steps to obtain the software products. However, a rigid approach to design usually leads to inflexible and largely useless design products. Creativity and common senses are required in addition to the process (see section 6.3 for details). The process of a method is more of an incremental and iterative process, in which the products of design gently unfold over time.

Step	Activity
1	identify the classes and objects at a given level of abstraction
2	identify the semantics of these classes and objects
3	identify the relationships among these classes and objects
4	implement these classes and objects

Table 6.1 Booch OOD: Method Process

For instance, the process of Booch OOD [Booch 91], as shown in table 6.1, is a four step design sequence. The first column in the table denotes the design step number and the second column gives the activity description. Booch OOD supports the incremental and iterative process of round-trip gestalt design. This is an incremental process: the identification of new classes and objects causes the developer to refine and improve upon the semantics of and relationships among existing classes and objects. It is also an iterative process: implementing classes and objects often leads us to the discovery of new classes and objects whose presence simplifies and generalises the design.

The Booch OOD has a very simple process model (it comprises only one level and four steps). However, showing the design sequence in tabular form can be a very useful tool in process modelling, especially if the conditions and operations of method processes are considered. The process model of a method is closely related to the product model of the method. For instance, in the Booch OOD example, class, object and relationship are method concepts defined in the product model. The method process is mainly called to determine these concepts. The later sections in this chapter will illustrate that the method concepts described in the product model can be used as input and output tokens to the method process.

As the aim of this research is to develop a generic model to represent a SDM, method process is the main modelling object amongst the three types of processes. Nevertheless, our generic model is powerful enough to represent the metaCASE process as well.

6.2.2 METACASE PROCESS

We described the method process in the last section, that is a logical process model of a SDM. In this section, we discuss the process in a metaCASE tool, which is a physical process model of the method. A metaCASE process is highly dependent on the semantics of a metaCASE tool, such as the data model to represent method concepts and the functional model to describe the method processes. A metaCASE process is the channel to express method semantics to metaCASE semantics, and all these semantics comprise of both product and process models. Therefore, the metaCASE process can be easily clarified into two phases. The first phase involves mapping the logical semantics to the physical semantics and the second phase deals with placing the semantics into the tool.

The technique of mapping method semantics into the metaCASE tool is discussed in chapter eleven. We concentrate on the process of putting method semantics into the metaCASE tool in this section. This metaCASE process describes a sequence of steps that a developer can follow to gradually place the physical semantics into the chosen metaCASE tool. Although this metaCASE process is not a SDM process, we find our meta model also handles this process model efficiently.

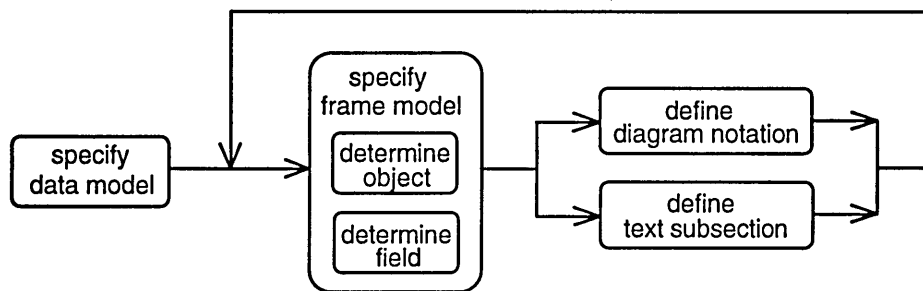


Figure 6.2 IPSYS ToolBuilder: An Example of MetaCASE Process

Figure 6.2 illustrates an example of IPSYS ToolBuilder's [IPSYS 92] metaCASE process. The round-corner boxes denote tasks and the v-shaped arrows represent the control flows in the process model. A developer starts the metaCASE process by specifying the data model of the method, and then develops the diagram and text frames of the required tool. The last step is to define the appropriate diagram notations and text subsections for each frame. The developer can carry on creating more frames. Apart from the incremental and iterative process already mentioned in the last section, this model also introduces the idea of cascade and parallel processes. This is a cascade process: the specification of a new frame requires the formation of shared objects and fields, which then require the definition of the diagram notations and text subsections. Each step refines and describes the details of the previous step(s). It is also a parallel process: the frame specification consists of a number of graphical and textual notations, which in theory can be defined concurrently.

6.2.3 CASE TOOL PROCESS

After the method semantics are placed into the metaCASE tool, it is ready to generate a specific CASE tool of the method. Hence, the generated tool is embedded with the method concepts and method processes introduced to the metaCASE tool. In the software level, the CASE tool is specific to a particular method (or a number of methods if tool integration is provided) and the tool can be used to develop software of a certain domain. However, the metaCASE tool may provide some facilities that allow the process to be refined and/or configured. Therefore, the CASE tool process is first planned from the method process defined in early stage, and then the metaCASE tool may allow the developer to configure the generated process. These relationships are represented by the *plan* and *configure* associations in figure 6.1.

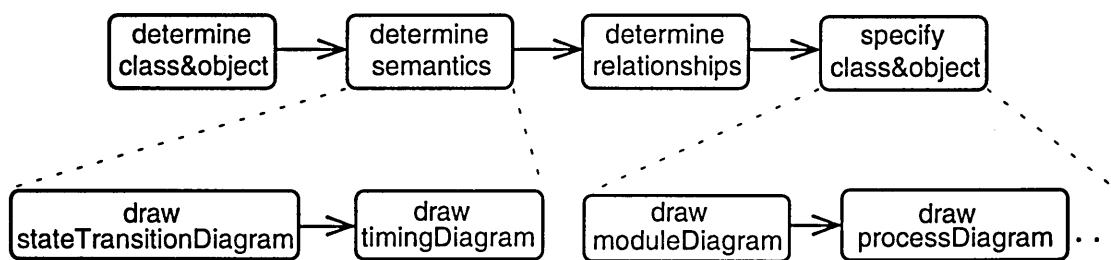


Figure 6.3 Booch OOD: the Configurable Process

Let us look at an example of configuration by refinement. In figure 6.3, the top four tasks show the process of Booch OOD, which has been tabulated in table 6.1. This part of the figure shows the method process. Booch OOD does not precisely define the activities of each step. For instance, the second step aims to establish the meanings of the classes and objects identified from the first step. This technique involves writing a script for each object, which defines its life cycle from creation to destruction, including its characteristic behaviours. This description is incomplete and ambiguous. However, the developer can configure the process, so that it can be refined to document the semantics of each key abstraction in *timingDiagram* and *stateTransitionDiagram*. This mechanism can also be applied to the other tasks as shown in the figure.

The CASE tool configuration is a major topic on its own [Fisher88], and it is outside the boundary of this research. Therefore, CASE tool processes or configurable processes will not be discussed in the rest of this thesis.

Nevertheless, these diagrams are used to describe the process model of a method graphically. It is good enough to show the sequence of processes, but this does not denote the dependencies between the processes. In later sections, we will develop an enhancement of this diagram to address the problem. The enhanced diagram is known as a **task diagram**.

6.3 BASIC ISSUES

Unlike the product model, the process model may be incomplete or ambiguous. We shall look at three basic issues of process modelling, which are creativity, explicitness and grainsize.

We have investigated the process model of a large number of SDMs. Some methods only provide a set of product notations and give no direct description about design sequence, which allows the developer to form their own tasks or process model. For instance, OOSD [Wasserman 90] is a method without a process model. Some other methods give minimal information on the process model. For example, Booch OOD is a fairly 'concept-rich' method, but its process model of the method is described in only ten pages.

The process model requires a lot of **creativity** as well as common sense. A method engineer has to construct tasks from the concepts in the product model. However, it is possible to refer to other methods which have similar fragments. For instance, Booch OOD does not have a precise description about the formation of *stateTransitionDiagram*. The process model of a similar fragment, say *stateDiagram* in OMT [Rumbaugh 91], can be borrowed for guidance. Also, common sense is a good approach. For instance, in order to form a *dataDictionary*, the *objects* of the system must first be identified. Therefore the task *insert(object)* must occur before the task *specify(object,dataDictionary)*.

This brings us to the second issue of process modelling, that is **explicitness**. An explicit task is a task recorded directly in a SDM, whereas an implicit task is not mentioned but has an implication that it must be carried out. Since all concepts must be created by the tasks in the process model, there is an implication to determine each and every concept in the product model. For instance, *nestedStateDiagram* is a group concept in the *dynamicModel* fragment of OMT. However, there is no description about constructing a lower level *stateDiagram* in the process model. An implicit task must be inserted to identify this feature, which implies the formation of a lower level *stateDiagram*.

The last issue is about the process **grainsize**. Our process model supports decomposition, so there are different levels of abstraction about the process. If the description is brief or vague then it is known as a coarse grain process. For instance, the basic process model of Booch OOD shown in table 6.1 is a coarse grain model. On the other hand, if the description describes a very precise and self sufficient step then it is called a fine grain process. In our model, we specify tasks as far as the operational level. That is when the task cannot be decomposed any more, but an operation must be carried out. For instance, insert a new *object* to a list in *insert(object)* task or add a state icon to the *stateDiagram* in *draw(state)* task. It is important to note that the grainsize must be consistent throughout a method.

6.4 THREE PRELIMINARY APPROACHES

When investigating meta modelling techniques, we found three preliminary approaches towards process modelling. These are the menu driven approach, the event sequence approach and the frame-based navigation approach. These approaches appear in both meta modelling research [Saeki 93] and in metaCASE tools [Smolander 91] [IPSYS 92].

6.4.1 MENU DRIVEN APPROACH

A menu driven approach can place emphasis on incremental change, by putting up a different set of options. It can be thought of as a basic (or simplistic) version of frame based navigation (described later). When an option is chosen from the menu, the required operation performs. A new set of options is created, which is actually a list of possible consequent steps. This menu driven approach is used in most database systems, because the number of steps are predefined and limited. This approach forms a hierarchical menu structure that defines the navigation possibilities to subsequent processes. Jumpers may be set to link a menu from one node of the hierarchy to another. Figure 6.4a illustrates a cascade menu system. If the first option of menu A is chosen, menu B will form, whereas if the last option is selected, menu C will form. Though the number of options varies in different menus, they are predefined for each individual menu. Nevertheless, this menu driven approach is not ideal for meta process modelling. It is too rigid and cannot accommodate the creative nature of the software development process. For instance, the number of options in a menu may vary according to the recent steps. A fixed size menu in each state is not a desirable solution.

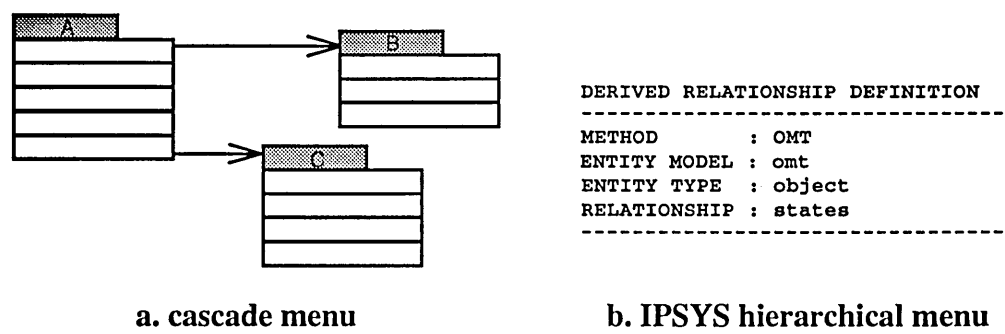


Figure 6.4 Menu Driven Approach

IPSYS ToolBuilder supports this menu driven approach. Figure 6.4b shows an IPSYS four level menu structure, which are method, model, entity and relationship levels. Each item in a level links to a specific menu and the options of a menu are related to an individual item. For instance, selecting the value *omt* of the *Entity Model* item gives the options such as *Show Entity Model Diagram* and *Show Entity Model Definition* etc.

6.4.2 EVENT SEQUENCE APPROACH

An event sequence approach is based on the formation of event lists. Each event in the list triggers a specific operation required by the method. A number of events are arranged in a sequential order and execute accordingly. The operation can be any software development based function.

The following diagram demonstrates an event sequence. Before the trigger, figure 6.5a, a list of events A1, A2... is on the sequence. When the event A1 is triggered, the corresponding operation is performed and then a list of new events, B1, B2 and B3, is inserted to the remaining sequence. It must be noted that there are various ways to reform the sequence after an event execution. Figure 6.5b shows the two most common ways, which are *insert to front* and *insert to end*. Other ways are *context limiting*, *specificity ordering*, *size ordering*, *data ordering*, *refractoriness* etc. These techniques are similar to conflict resolution in an expert system.

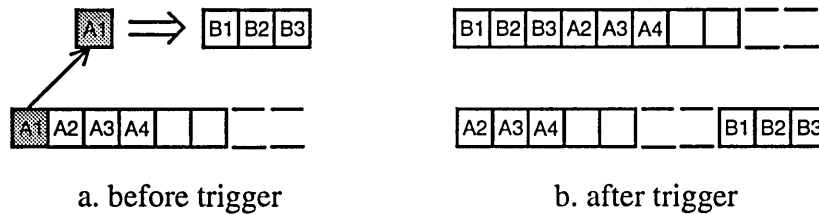


Figure 6.5 Event Sequence Approach

A disadvantage of this approach is that the sequence may consist of highly coupled events, which is a group of interrelated events in the same method. They may make event management and fragment sharing more difficult.

This problem can be solved by introducing concept tokens to the process model. These tokens are based on the concepts in the product model. They appear as context parameters in task functions. An operation is a task function attached with a set of preconditions and postconditions, which monitor the control flow of the model. A precondition expresses a group of tokens that the system must possess in order to carry out the operation, whereas a postcondition expresses a group of tokens that the system will receive after the operation.

In addition, there is a need to deal with token refinement, that is to refine or redefine the meaning of tokens. For instance, state may mean different things in different methods. There are two ways to overcome this. One way is to generalise the commonality of the concept in a hierarchical form. The other way is to give them different labels; *state(omt)* and *state(booch)* for instance.

6.4.3 FRAME BASED NAVIGATION APPROACH

A frame based navigation approach allows a control flow amongst text frames and diagram frames. The navigation options (or menu) are bound to individual items in the current frame, as shown in figure 6.6a. Preconditions and triggers can be attached to each state operation. This approach looks simple as long as the navigation path is formed. Since the control information is encapsulated in the objects, the branching can be made relevant to the nature of the object. Moreover, the number of options bound to the object is flexible and it will be less than a single combined menu as in the menu driven approach. However, the layout of this frame based navigation graph may be different in various frames.

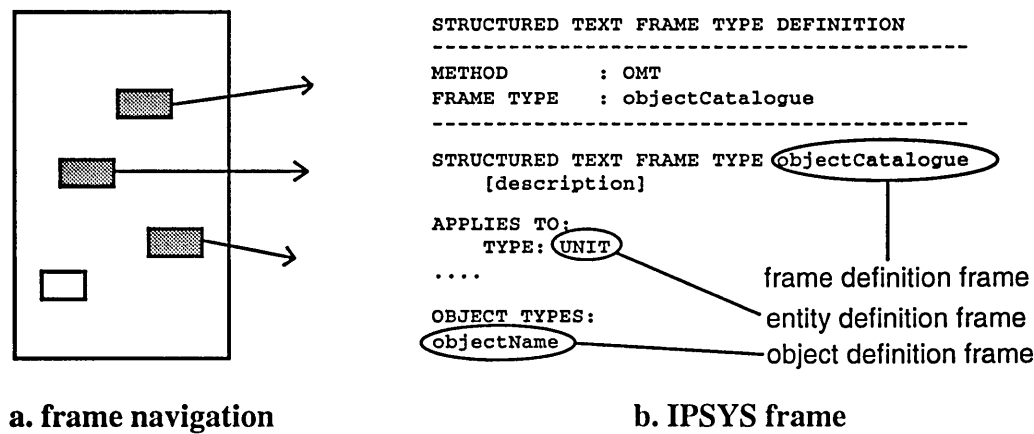


Figure 6.6 Frame Based Navigation Approach

IPSYS ToolBuilder supports this frame based navigation approach. Figure 6.6b shows part of the structured text frame type definition *objectCatalogue*, and several items on the frame allow navigation to other frames. For instance, if the entity item *UNIT* is selected, the entity definition frame is shown, whereas if the object item *objectName* is chosen, the object definition frame is displayed.

IPSYS ToolBuilder allows this navigational effect to be inherited to the generated CASE tool. All frames, objects and fields can be bound with corresponding operations, that is frame operation, object operation etc. These operations may be accompanied by preconditions which guard the action or by triggers to force other executions. The operations appear as options in a menu, when the related items are chosen from the frame. Therefore they are used for navigational purposes, and that becomes the control structure of the system. The destination of the operation is always another frame in the system.

In order to achieve the above three approaches, a Ptech-like [Martin 92] event schema diagram is introduced to denote the process model. This is discussed in section 6.7 task modelling.

6.5 TASK FUNCTIONS

As mentioned in chapter four, a process model is comprised of a number of tasks, which are executed by task functions. The behaviour of a task function can be summarised in figure 6.7.

- A task function may be either an optional or a mandatory operation in a design sequence. For instance, in Codarts/DA method [Gomaa 93], if an application is non-distributive the *perform(distributedSystem)* task can be ignored (as shown in figure 6.7a).
- The tasks may also be mutually recursive, that is they loop round to trigger one another. For instance, in the Ptech method [Martin 92], the *insert(event)* task from an *eventTrace* may induce an *object* to be filled by the *specify(object)* task, which may, in turn, create more *events*. Figure 6.7b illustrates a two step cycle, but more complicated loops may occur in method processing.
- A special case of mutual recursion occurs when both tasks have the same function, that is a recursion applies on the same task. Such iteration is known as single recursion. For instance, in OMT, a *state* discovered by the *insert(state)* task may cause the identification of other aggregate states or substates (as shown in figure 6.7c).

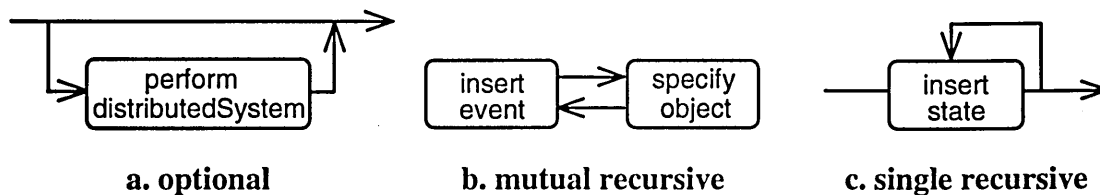


Figure 6.7 Behaviour of Task Functions

To conclude, it is difficult to identify every single development path (or possibility) in the process model. In order to provide enough - but not too crude - assistance, the process model must be kept as simple as possible. Therefore, instead of constructing the navigational paths, we concentrate on describing the necessary requirement for the task and the expected outcomes of the task. These are the precondition and postcondition of a task function respectively. As most of these tasks aim to develop the concepts in the product model, these conditions are mainly composed of the method concepts. The parameter in a task function is known as *contextParameter*, which is a *conceptToken* to monitor control flow. The capability of *conceptToken* will be shown in following sections.

From our method investigation, we have identified nine types of task function. They are the *perform*, *do*, *draw*, *insert*, *delete*, *modify*, *adjust*, *retype* and *specify* functions. The first two are composite task functions and they take the task name as their parameter. On the other hand, the last seven functions always have concept(s) as their parameter(s), so they are also known as **concept functions**.

In addition, the *perform*, *do* and *draw* functions are composite functions, which can either decompose or refine into other task functions. However, the other six task functions are mainly for operating method concepts. They are also called **terminal task functions**. These terminal functions are always associated with heuristics to carry out the activity. Each composite function must be eventually represented by these terminal functions.

We use OMT and Codarts/DA for the following illustrations, since they require most of these functions.

6.5.1 perform FUNCTION

The *perform* task function processes a predefined task sequence. This is the way to implode other tasks within the current sequence. The context will pass to the called sequence and return the result to the current sequence. The system will normally follow the heuristics as defined in the called sequence, although additional heuristics (usually constraints) can be attached to this sequence. This function is useful if part of the process has been declared separately or within another method. For instance, the Codarts/DA process model is pre-processed by the Cobra process model as its analysis phase. The task function *perform(cobra)* in Codarts/DA model will call up the Cobra sequence.

Furthermore, a *perform* function can always decompose into a number of task functions (see section 6.7.4 for details). Thus the associated heuristic normally gives the definition the task rather than provides guidance for the activity.

6.5.2 do FUNCTION

The *do* function processes a number of subfunctions. Unlike the *perform* function, only one of the subfunctions is carried out. The *perform* function is similar to a subroutine call and can represent graph structures, whereas *do* is a block of 'case' statements (such as those in C or Pascal programming language). The *do* function may refine to a list of alternative tasks (see section 6.7.5). It should be used whenever a design decision is made out of a number of choices, which include the option of not to take any choice at all. For instance, the *do(verifyAssociation)* task function in OMT can refine to the following tasks:

- adjustAssociationBetweenEliminatedClass
- deleteIrrelevantAssociation
- retypeAssociationToOperation
- modifyTernaryAssociation
- deleteRedundantAssociation
- retypeRedundantAssociationToDerivedAssociation
- modifyMisnamedAssociation
- insertRoleName
- retypeAssociationToQualifiedAssociation
- insertMultiplicity

6.5.3 draw FUNCTION

The *draw* function constructs a graphical presentation. It is a composite task function, because a diagram requires many design decisions which are defined by the *insert* functions. For instance, the *draw(contextDiagram)* function of Codarts/DA includes the determination of *externalEntity*, *subsystemTerminator*, *dataTransformation* as well as *dataFlow* in the diagram. Therefore the *draw* function calls upon the appropriate task functions, such as *insert(X)*, *delete(X)* or *modify(X)* concept functions to perform necessary operations on the element *X* in a graphic editor (these functions are defined in the following subsections). The *draw* function launches a graphic editor with all appropriate icons pre-defined by the graphical representation of the product model. Context-sensitive help or guide-rules can be provided for this function.

6.5.4 insert FUNCTION

The *insert* task function provides design decision. It is the most common function in any design task. It may be used to obtain concrete domain entities, such as *object*, *function* and *task*, or to compose system definitions such as *systemDecomposition*, *messageCommunication* and *eventSynchronisation*. The *insert* function is usually supplied with a rule or a criterion in its heuristic field. In a manual system, it provides a list of items or decisions. Specific examples of *insert* functions within Codarts/DA are: *task*, *informationHidingModule*, *object*, *function*, *systemDecomposition*, *subsystemDecomposition*, *messageCommunication* and *eventSynchronisation*. An insert function takes the element being created as its context parameter, for instance *insert(task)*.

6.5.5 delete FUNCTION

The *delete* task function inverts the result of the *insert* task function. It may be used to erase any obtained design entities as created or modified by other task functions. A *delete* function is normally associated with a design rule in its heuristic record (see chapter seven). If the design condition is satisfied, the *delete* function can be fired. Again, the delete function takes the element being deleted as its context parameter. In our *do(verifyAssociation)* example above, there are two entries (*deleteIrrelevantAssociation* and *deleteRedundantAssociation*) of the *delete(association)* task function.

6.5.6 modify FUNCTION

This function allows the developer to revisit the pre-determined elements with a new stimulus. The *modify* function is to optimise an element, especially after there have been a number of related tasks performed since the element is created. Normally, heuristic guidance or rules are

linked to the task. A *modify* task function takes the element being deleted as its context parameter. In the *do(verifyAssociation)* example, the two entries of the *modify(association)* task function are *modifyTernaryAssociation* and *modifyMisnamedAssociation*.

6.5.7 adjust FUNCTION

An *adjust* function is similar to a *modify* function, except that the element which causes the change is declared. In other words, this function is to check or verify the pre-determined elements against other types of recently identified elements. An *adjust* function takes two elements as context parameters and heuristic guidance must be provided for the task. For instance, in the *adjustAssociationBetweenEliminatedClass* task of our *do(verifyAssociation)* example, if one of the classes in the association has been eliminated then the association must be eliminated or restated in terms of other classes. The corresponding *adjust* task function is expressed as *adjust(association,class)*, which reads 'adjust association by class'.

Besides, in Codarts/DA, the task function *adjust(task,informationHidingModule)* allows the consolidation and integration of the relationships between elements of the two types.

6.5.8 retype FUNCTION

This function allows a pre-determined element to change to a different type. The element could have been created previously by the *insert*, *specify* or *draw* task function, so this function must launch a text window or a graphic window according to the nature or current state of the element. Again a *retype* function takes two elements as context parameters and the heuristic guidance must be provided for the task. The function *retype(X,Y)* stands for change the type of element from *X* to *Y*. For instance, in the three retype tasks of the *do(verifyAssociation)* function example listed below, the first one is expressed as *retype(association,operation)*.

```
retypeAssociationToOperation  
retypeRedundantAssociationToDerivedAssociation  
retypeAssociationToQualifiedAssociation
```

6.5.9 specify FUNCTION

The *specify* task function fills in a template, form, specification, document or explicit code. Both *specify* and *draw* are common task functions. The *specify* function handles textual descriptions and the *draw* function deals with graphical presentation. In a *specify* function, the developer is provided with a text editor. Heuristic rules may be used, to guide the input process, to enforce referential integrity, or to provide a batch check of the completeness and consistency at any appropriate check-point. A *specify* function may take either one or two

parameter(s). In a single parameter function, the context is being specified such as *specify(systemConfiguration)* in Codarts/DA. In a two parameters function, the first concept is specified into the second one. For instance, *specify(object,dataDictionary)* in OMT means to specify the determined *object* in a *dataDictionary*.

6.6 TASK SEQUENCE

Having defined the task functions of the process model, we can then illustrate the approach with a real method example and a couple of scenarios concerning the model. Table 6.2 shows the task sequence of the OMT analysis phase. It is defined within OMT, but we restructure it into our process model format. Each row of the table represents a task, and each task is given a code number, which intends to show the task decomposition, as shown in the first column.

No	Task	Context Parameter	Precondition	Postcondition	Heuristic
1	perform	analysis	requirement	analysis	heuristic for OMT-OOA
1.1	perform	objectModel	requirement	objectModel	heuristic for object modelling
1.2	perform	dynamicModel	requirement	dynamicModel	heuristic for dynamic modelling
1.3	perform	functionalModel	requirement	functionalModel	heuristic for functional modelling
1.1.1	insert	object	requirement	object	identifying object class, keeping right class (p 153-156)
1.1.2	specify	object	object	dataDictionary	describing each object class (p 156)
1.1.3	insert	association	object	association	identifying association, keeping right association (p 156-161)
1.1.4	insert	attribute	object	attribute	identifying attribute, keeping right attribute (p 162-163)
1.1.5	insert	inheritance	object	inheritance	refining with inheritance (p 163-165)
1.1.6	modify	association	association		testing access path (p 166)
1.1.7	insert	module	object	module	grouping classes into modules (p 168-169)
1.2.1	insert	scenario	requirement	scenario	preparing a scenario, interface format (p 170-172)
1.2.2	insert	event	scenario	event	identifying event (p 173)
1.2.3	insert	eventTrace	event	eventTrace	identifying event trace (p 173)
1.2.4	draw	stateDiagram	eventTrace	stateDiagram	building a state diagram (p 173-179)
1.2.5	adjust	event object	stateDiagram object		verifying consistency, match events between objects (p 179)
1.3.1	insert	data	requirement	data object	identifying input and output values (p 180)
1.3.2	draw	dataFlowDiagram	data object	operation dataFlowDiagram	building data flow diagram (p 180-182)
1.3.3	specify	operation	operation		describing function (p 182) from object model, event, state action and activity, function, shopping list, simplifying operation (p 183-185)
1.3.4	insert	constraint	object	constraint	identifying constraints between objects (p 183-184)
1.3.5	specify	optimisation	operation		specifying optimisation criteria (p 183)

Table 6.2 OMT: Analysis Phase Sequence

We demonstrate how to read the table, using task 1.2.2 as an illustration. A task function comprises the second and third columns. In the example the task function is *insert* and the context to be determined is *event*. The function is normally written as *insert(event)*. However, there may be more than one parameter in a function, for example task 1.2.5 *adjust(event,object)*. The only requirement of task 1.2.2 is the *scenario* token, which is shown in the precondition column. A list of *events* is produced after the function execution, and this token is denoted in the postcondition column. There may be more than one token for these conditions, for instance task 1.3.2 *draw(dataFlowDiagram)* requires both *data* and *object* tokens and produces the *operation* and *dataFlowDiagram* tokens. Moreover, a task may have the precondition but no postcondition: for example task 1.3.3 *specify(operation)* does not produce any token. The last column shows heuristic information about the task. For task 1.2.2 this is *identifying event* given in page 173 from the reference book [Rumbaugh 91].

The code number shows the level of task decomposition. For example, task 1.2.2 is a third level decomposition. It is part of task 1.2 *perform(dynamicModel)* and which in turn is part of task 1 *perform(analysis)*. This task decomposition technique is discussed in section 6.7.4.

Now we shall look at two scenarios of this OMT analysis phase sequence. The first scenario concerns the definition of *object* in *objectModel* and the second scenario concerns the *state* in the *stateDiagram*.

In the first scenario task 1.1.1, the *insert(object)* function, takes a basic *requirement* as input, and triggers a text editor known as *objectCatalogue*. The *objects* are determined and placed in the catalogue, then an *object* token is set as shown in the postcondition. Task 1.1.2 *specify(object)* function checks the precondition, which is the *object* token. It then takes the *objectCatalogue* as input and opens up a *dataDictionary*. The key fields are filled by the *objects* in the *objectCatalogue* and the description of *objects* can be entered. A new token *dataDictionary* is formed. Tasks 1.1.3 to 1.1.7 continue to construct the *objectModel*.

In the second scenario, task 1.2.1 to task 1.2.3 form a list of *events* in an *eventCatalogue*. Task 1.2.4 *draw(stateDiagram)* checks the precondition, which is the *event* token. Then the function takes the *eventCatalogue* as input and launches a graphic editor to depict the *stateDiagram*. The editor should provide all icons required for the diagram as described in the product model. The *draw* function will initiate *insert* tasks to identify *state*, *transition*, *guardCondition* etc., and the *stateDiagram* token is formed. Then, in order to execute task 1.2.5 *adjust(event, object)* function, the task 1.1.1 must be completed since one of the preconditions is the *object* token. In this task, the *events* in *eventCatalogue* are matched between *objects* in *objectCatalogue* so as to verify consistency of the *stateDiagram*. When a change is made on the *eventCatalogue*, the corresponding *stateDiagram* will show on the graphic window to allow possible modifications.

6.7 TASK MODELLING

In most methods, *stateTransitionDiagram* (STD) and *dataFlowDiagram* (DFD) are the main tools for describing the dynamic and functional behaviour of a software system. However, for method process modelling we find that a combination of the features from both tools is required. The model must have the ability to depict *state*, *event* and *trigger* in an STD, which show the state-based dynamic features. It must also be able to depict *operation* (known as *process* in DFD), *dataFlow* and *controlFlow* in a DFD to denote data transformation aspects. Therefore we introduce a variation of the Ptech event schema (see section 2.4.4) to denote the meta level process model graphically.

Since the model is not actually an event schema, we shall refer to it as **task diagram** and the approach to develop such a diagram **task modelling**. The next two sections define the meaning of a task, a trigger and a concept flow in the process model. Then we shall describe, with illustrations, task diagrams, task decomposition, task refinement and parallel tasks.

6.7.1 A TASK

In our meta model, a task provides the means to rigorously describe assisted software process models. This task provides mechanisms that support the description of generic software process models which can be incrementally and repeatedly instantiated in order to produce particular software process models for specific applications. In a process model, a task is an autonomous process, and is defined as an **operation** with an optional guard **condition**. A condition describes the precondition and an operation formulates the postcondition. Each task is represented by a **state** in the process model and it may end with an **event** to **trigger** other tasks of the system. In Ptech terminology, the state before an event is called *eventPrestate* and after an event is known as *eventPoststate*.

The notation of a process model is best represented by a Ptech-like event schema notation as shown in figure 6.8. Apart from the basic notions, *conceptFlow* is introduced to denote input and output flows of concepts to the task. In order to distinguish it from a *trigger*, a *conceptFlow* is depicted by a thin arrow, whereas a *trigger* is depicted by a thick arrow.

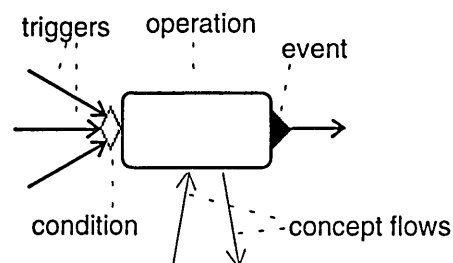


Figure 6.8 Process Model: Task

We shall now demonstrate how the three preliminary approaches of process modelling mentioned in section 6.4 can be described by this task notation.

For the *menu driven approach*, a menu is an event state (prestate) and each option is an event that triggers an operation. Consequently it changes into another event state (poststate), which is in effect another menu.

For the *event sequence approach*, each event in the queue is a trigger to an operation. The priority mechanism can be handled by guard conditions. The repeated and optional features in the list can also be implemented through the triggers.

Finally, the *frame-based navigation* falls into this process model naturally. Similar to the menu driven approach, each frame is an event state, with predefined navigation paths scattered in the frame. Therefore each context node (for instance an entity type or an object type in IPSYS ToolBuilder) can be a trigger to another frame.

6.7.2 TASK TRIGGER AND CONCEPT FLOW

The aim of this section is twofold: firstly to explain the trigger rule in Ptech and the task trigger in the process model, and secondly to introduce the concept flow for the concept token.

Let us first review the trigger rule in the Ptech method. Just as an event triggers an operation, so a trigger is a cause-and-effect link. Specifically, the trigger takes the underlying object of an event and determines those object(s) required to invoke an operation. As illustrated in figure 6.9a, each *triggerRule* has three basic components: an *eventType* (the cause), an *operation* (the effect) and a *function*. The function takes the causal event's underlying object and maps it to those objects being passed as arguments to the operation it invokes. Figure 6.9b shows the classic example of a vending machine: whether or not a sale can be completed depends on its control condition, which is the function defined in the rectangle box. The highlighted words in the function box denote the three underlying objects, each of these objects refers to one of the three incoming event types.

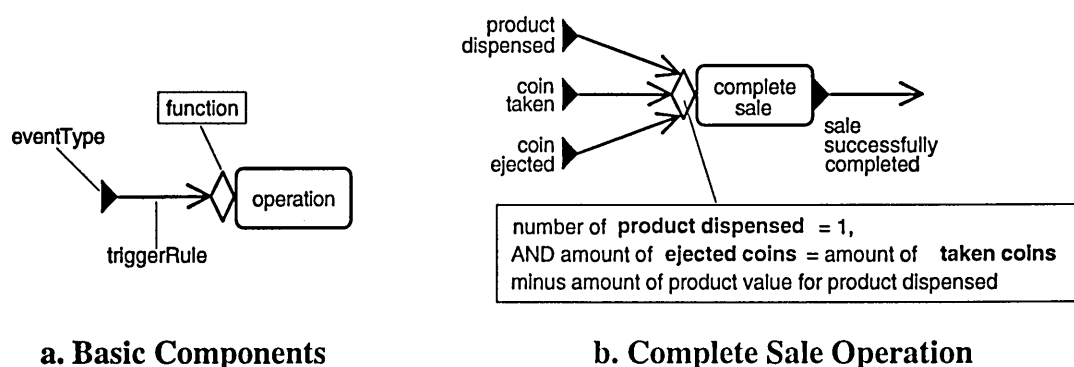


Figure 6.9 Ptech: Trigger Rule

A Ptech event schema diagram is a very powerful tool for specifying behaviour, which is important in software process modelling. A trigger is an event based mechanism that takes the underlying object of an event and invokes an operation. Although the underlying object (or attribute) is implicit in the diagram, this information is shown in the function instead. This function allows us to form specific and complicated control conditions, for instance the *complete sale* example shown in figure 6.9b.

However, the underlying objects found in a method process model are discrete concepts from those found in a product model. And since there is no complex guard condition in a method, the function of a trigger rule is inappropriate. Therefore, we show these objects explicitly as concept tokens and link them to tasks by **concept flows**. These tokens precisely denote the entry condition of a task and they are side-products of a task function. Hence the definition of a trigger is changed, the basic components of a trigger in the process model are shown in figure 6.10a. An event causes a trigger to invoke a task. However the operation will not be executed until all concept tokens in the condition are collected. This operation is actually one of the task functions described in section 6.5. With this new definition, a trigger is like a preceded-by relationship amongst tasks, whereas the concept flows from the input and output relationships between concept tokens and task functions.

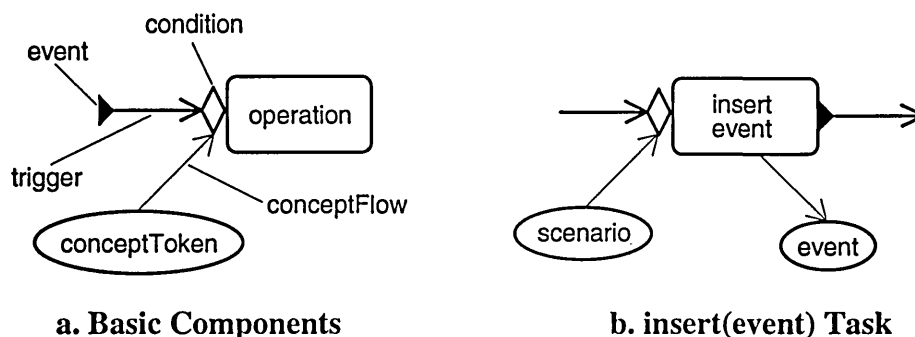


Figure 6.10 Process Model: Task Trigger and Concept Flow

Besides their significance as mentioned above, the concept flows are also vital components in the process model. They depict the preconditions and postconditions of tasks in the diagram. A concept flow points to the condition box representing the corresponding concept token that invokes the task function. In other words, instances of the concept are inputs to the task. On the other hand, a concept flow running out of an operation box states that an instance of the concept has been created or modified and the related concept token is available.

Figure 6.10b denotes the task 1.2.2 *insert(event)* from the OMT analysis phase sequence shown in table 6.2. In the method process model, it is depicted by a series of these tasks arranged together and shown in a task diagram.

6.7.3 TASK DIAGRAM

A task diagram describes the task sequence of a method. The diagram is a directed graph in which nodes represent tasks and arcs represent *triggers* (depicted as thick arrows) between tasks. *Concepts* and *conceptFlows* (depicted as thin arrows) are introduced to the task diagram. A *conceptFlow* is a *dataFlow* running in or out of a *task*. The corresponding data is a *conceptToken*, which refers to a concept in the product model. A *conceptToken* is represented by an oval shape with the concept name inside it.

Figure 6.11 shows the OMT *dynamicModel* task diagram. When a *conceptFlow* points to a *condition* box, the *conceptToken* is required to execute the foregoing task. For example, in the figure, *scenario* is required in both the *insert(event)* task and the *insert(eventTrace)* task. On the other hand, if a *conceptFlow* points to a *conceptToken*, the preceding task will produce an instance of the *concept* and the corresponding *conceptToken*. Again in the figure, the *draw(stateDiagram)* task builds an instance of *stateDiagram*.

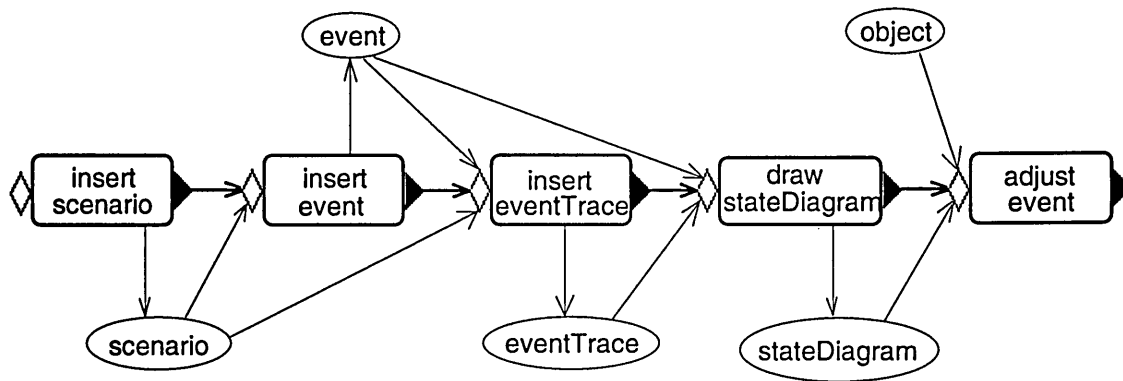


Figure 6.11 OMT: *dynamicModel* Task Diagram

It is possible to form a task diagram out of the task sequence as shown in table 6.2. The task function and *contextParameter* are depicted as a task box. The tokens in the precondition and postcondition columns are shown as *conceptTokens* with *conceptFlows* in and out of tasks. The code number illustrates task decomposition, which is discussed in the next section.

6.7.4 TASK DECOMPOSITION

The task diagram also supports decomposition of tasks. An *operation* can be decomposed into a lower level task diagram showing the detailed structure of the operation. This task decomposition will form a task hierarchy of the method. For example, figure 6.12 illustrates the decomposed task model of the *draw(stateDiagram)* task shown in figure 6.11. One must be careful about the feedback loops, a new stimulation is probably required for each iteration. In the following example, the feedback may be caused by discovering a new *eventTrace*.

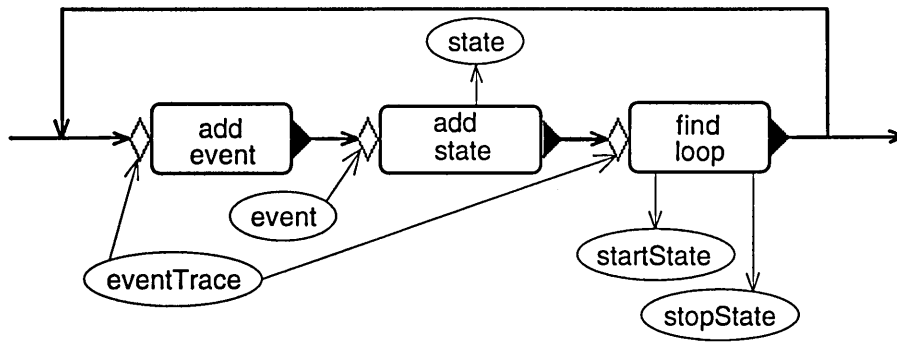


Figure 6.12 OMT: *draw(stateDiagram)* Task Diagram

The main objective of a task diagram is to show task sequence, which includes task conditions and event triggers. Concept tokens should be placed in the right level of abstraction in the task decomposition hierarchy. For instance, *startState* and *stopState* are outputs from *draw(stateDiagram)*, but they only display in the decomposed level as shown in figure 6.12 and not on the top level. That is because these elements are not of interest in the global view of the *dynamicModel* task diagram. In addition, task decomposition performs an AND-sequence of task functions. That is all the tasks in the lower level task diagram are carried out. This decomposition may happen only in the *perform* or the *draw* task function.

6.7.5 TASK REFINEMENT

Task refinement also classifies task diagram into different levels of abstraction. However it performs an OR-logic of task functions, that is only one of the lower level tasks is carried out. A task refinement is used for describing alternative design decisions are available. It is denoted by the *do* task function in higher level task diagram. For instance, figure 6.13 illustrates part of the refined task diagram of the *do(verifyAssociation)* as discussed in section 6.5.2 (a complete version is shown in figure E.3).

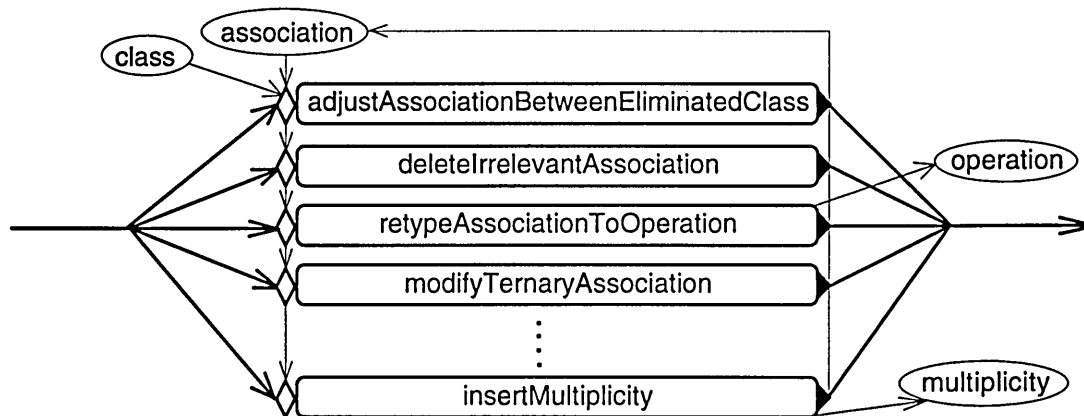


Figure 6.13 OMT: *do(verifyAssociation)* Task Diagram

6.7.6 PARALLEL TASKS

So far, all tasks discussed above have concerned sequential software processes, that is only one task in execution at a time. In a sequential system, a process can be fired when all its required *conceptTokens* are ready and all preceding tasks have completed. However, some SDMs emphasise certain tasks can be processed at the same time. These tasks are known as **parallel tasks**. There are two main reasons for this concurrency:

- The parallel tasks (or even sequences of task) have totally separate semantics, that is no direct or indirect connection between the tasks. This is common in the tasks for loosely-coupled method fragments. These fragments have only weak relationships (i.e. referencing) between them but other strong associations (i.e. such as grouping or shared concepts in dissection sets) are not allowed. For example the three tool fragments in OMT are loosely-coupled. The three respective tasks for performing *objectModelling*, *dynamicModelling* and *functionalModelling* can then be executed concurrently.
- In contrast, the tasks that are highly-coupled may require concurrency so that they can complement one another to accomplish the interrelated design considerations. Figure 6.14 illustrates an example of this situation. The three related tasks are concerned about identifying elements which are *class*, *association* and *attribute* in the system. The common requirement of these tasks is the *problemStatement* token, however for each element identified it is used to determine more other elements. These highly integrated tasks are physically processed in parallel.

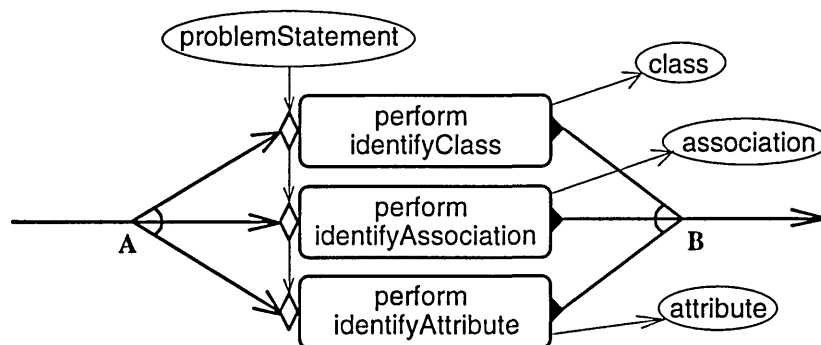


Figure 6.14 OMT: Parallel Tasks in Identifying Elements

This set of parallel task are best denoted by lining them up as in the figure 6.14. In order to distinguish from the task refinement (described in the previous subsection), we place an arc amongst the triggers as shown in point A and point B. Therefore these arcs denote the task synchronisation. Point A represents a splitting of control and point B describes a merging of control. These ideas are borrowed from the control modelling technique by *transition* of *stateDiagram* in OMT (see figure 5.13a).

6.8 META PROCESS MODEL

In this section we demonstrate a task sequence to constitute a process model, in other words the process model of process modelling. We show that it is very useful to structure a process model as a task sequence, which we present in a tabular form as shown in table 6.2. The following description uses *perform(dynamicModel)* as an example to show the order of task modelling. This illustration includes tasks 1.2.1 to 1.2.5 of the OMT analysis phase sequence.

There are five main steps in process modelling:

- identifying tasks with the associated operations
- determining event triggers amongst the tasks
- identifying concept tokens required in current abstraction
- determining inflows and outflows of concept tokens
- identifying tasks requiring decomposition and repeating the previous four steps

Identifying tasks with the associated operations: In those methods that provide design steps or sequences, the top level tasks are normally carefully defined. Most tasks can be formulated into the first six task functions (or operations) given in section 6.5. Then for each operation appropriate context parameters are associated, these *conceptTokens* can be interchanged with the *concepts* in product model. Table 6.3 demonstrates how to identify tasks and operations in OMT's *dynamicModel*:

No	Task	Context Parameter	Original Description
1.2.1	insert	scenario	prepare scenarios of typical interaction sequences
1.2.2	insert	event	identify events between objects
1.2.3	insert	eventTrace	prepare an event trace for each scenario
1.2.4	draw	stateDiagram	build a state diagram
1.2.5	refine	event, object	match events between objects to verify consistency

Table 6.3 Step 1: OMT *dynamicModel* Tasks and Operations

Determining event triggers amongst the tasks: Each task in a sequence is an iterative process. In this step event triggers amongst tasks are identified. These triggers may appear in different forms, such as cascade tasks, parallel tasks or recursive tasks. Guard conditions are not implemented at this stage, so the triggers present just the order of task execution. In our *dynamicModel* example the task sequence is in a linear form as shown in figure 6.15.

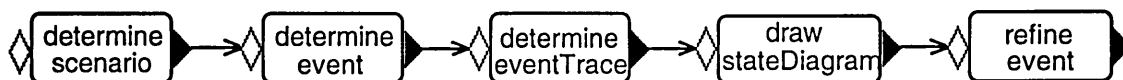


Figure 6.15 Step 2: OMT *dynamicModel* Task Sequence

Identifying concept tokens required in current abstraction: In this step the precondition and postcondition of each task is considered. The concepts required for the task are shown as precondition tokens. The concepts formed after the task are shown as postcondition tokens. These tokens relate the tasks in the sequence. It is important to keep these tokens as concise as possible so that the tasks can remain in the same level of abstraction. Table 6.4 illustrates the concept tokens of OMT *dynamicModel*. Notice that the context parameters are not necessarily equivalent to the tokens in either the task precondition or the postcondition. However, if the function is *insert*, *draw* or *specify*, the consequence of the function is most likely to be the context parameters described, as shown in task 1.21 to task 1.2.4.

No	Task	Context Parameter	Precondition	Postcondition
1.2.1	insert	scenario	requirement	scenario
1.2.2	insert	event	scenario	event
1.2.3	insert	eventTrace	scenario, event	eventTrace
1.2.4	draw	stateDiagram	event	stateDiagram
1.2.5	refine	event, object	stateDiagram, object	-

Table 6.4 Step 3: OMT *dynamicModel* Concept Tokens

Determining inflows and outflows of concept tokens: After identifying the concept tokens, determining the input/output concept flows is simply a matter of mapping from the precondition and postcondition columns of the task sequence table to the task diagram. An input concept flows to a condition box shows the precondition, whereas an output flows from an operation box depicts the postcondition. Figure 6.11 shows the complete top level task diagram of OMT *dynamicModel*.

Identifying tasks requiring decomposition and repeating the previous four steps: Task decomposition (section 6.7.4) is an important notion in process modelling. A decomposed task diagram gives a low level of abstraction about the task, so the detail concept tokens of the task are displayed. Figure 6.12 illustrates the decomposed *draw(stateDiagram)* task.

6.9 META META MODEL

It would prove useful to show the meta model of both the product model and the process model, which is the meta meta model. The product model discussed in the previous chapter can be described graphically by concept diagram. However, the process model discussed in this chapter can be tabulated in the task sequence as well as depicted in the task diagram. In order to reduce unnecessary confusion, table 6.5 shows a map for this meta meta model. It reads as '*the concept diagram of product model is shown in figure 5.32*', which is the shaded area in the table. This figure is shown in chapter five and others are displayed in this section.

Meta Model	Product Model	Process Model
Concept Diagram	Figure 5.32	Figure 6.17
Task Sequence	Table 6.6	Table 6.7
Task Diagram	Figure 6.16	Figure 6.18

Table 6.5 A Map for Meta Meta Model

The formation of a product model is demonstrated in section 5.5. It is a seven step task sequence. Table 6.6 shows that the tasks in the product model are fairly independent. Apart from the incremental sequence, the only requirement is the *concept* token. The task diagram of the product model is shown in figure 6.16. It is worth noting that the precondition to start the product modelling is the knowledge acquisition of method engineering, which is discussed in chapter ten.

Task	Context Parameter	Precondition	Postcondition
insert	fragment	-	concept
insert	entity	concept	subtyping, composition
insert	linking	concept	subtyping, linking
insert	property	concept	subtyping, composition
insert	grouping	concept	grouping
insert	referencing	concept	referencing
specify	constraintRule	concept	constraintRule

Table 6.6 Task Sequence of Product Model

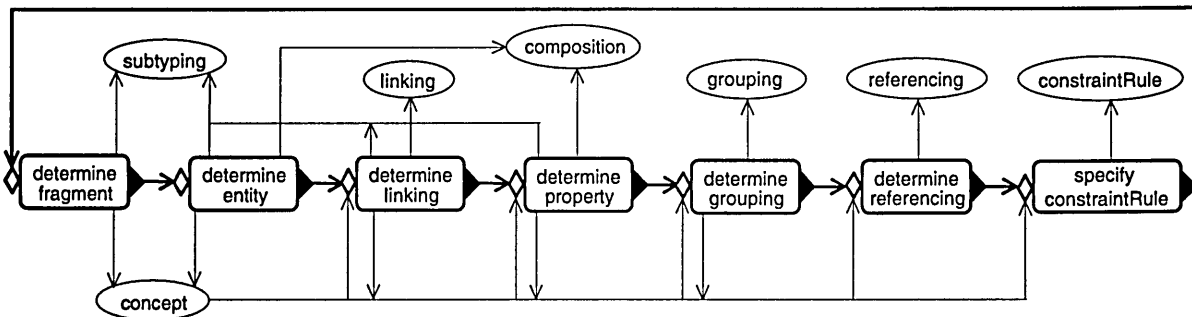


Figure 6.16 Task Diagram of Product Model

Figure 6.17 shows the concept diagram of a process model. The following points are made about this product model:

- Each *task* has an operation, which is one of the nine task functions. These functions are denoted as subtypes of the *taskFunction*.
- There are two *constrainedConcepts* in the process model. The *contextParameter* is constrained since it depends on the type of task function, whereas *conceptFlow* is a dataflow from *task* to *conceptToken* or vice versa.
- The *event* that follows an *operation* defines the *eventPrestate* and *eventPoststate* of a *task*.

- A *trigger* is a link connecting an event to a *taskFunction*. If the *taskFunction* is conditional, the *event* will trigger the *condition* instead.
- The *taskDiagram* and *conceptDiagram* are the two main fragments of the meta model. The *taskDiagram* relates to the *conceptDiagram* through the *conceptToken*, which itself refers to a *concept* in a *conceptDiagram*.

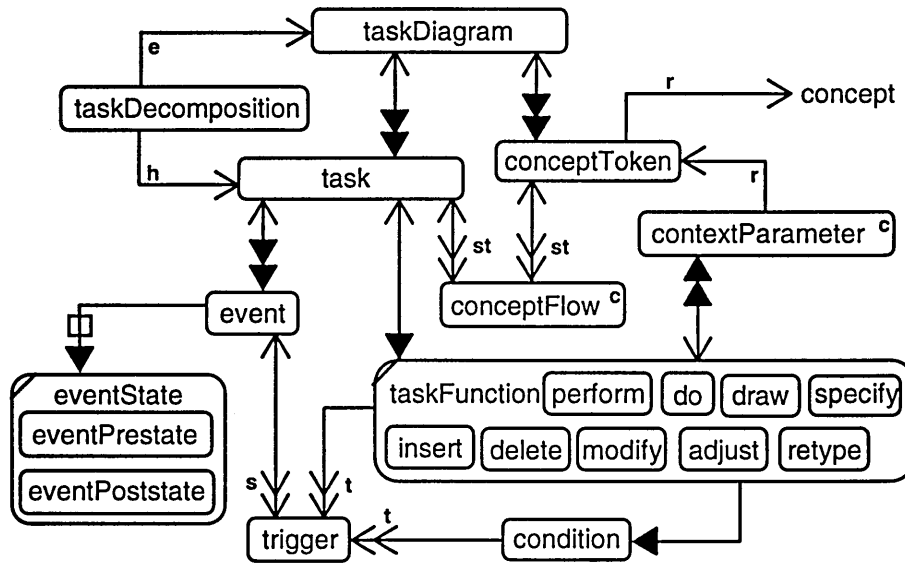


Figure 6.17 Concept Diagram of Process Model

Task	Context Parameter	Precondition	Postcondition
insert	task	-	task, operation, condition
insert	trigger	task, condition	trigger, event
insert	conceptToken	task, condition	conceptToken
insert	conceptFlow	task, conceptToken	conceptFlow
insert	taskDecomposition	task	taskDecomposition

Table 6.7 Task Sequence of Process Model

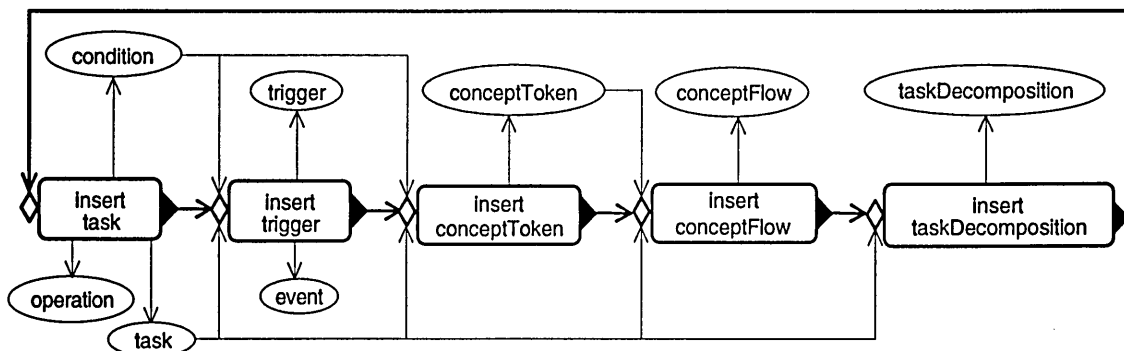


Figure 6.18 Task Diagram of Process Model

Section 6.8 demonstrates the formation of a process model, which can be described by a five step task sequence as shown in table 6.7. Figure 6.18 depicts the process model in a task diagram.

6.10 CONCLUSION

This chapter presented a generic process model of SDMs. Unlike the product model, the process model is loosely defined in order to provide flexibility and freedom for developer creativity. This is needed because most SDMs have only a coarse grain description on method processes, and some of them are described implicitly. The structure of a task and different types of task functions are identified. Task sequence is also introduced to document the process model in a tabular form, which maps easily to a task diagram. This diagram is based on the Ptech event schema notion, with extensions to model *conceptToken* and *conceptFlow*.

We have tried out this process model with five methods. The result is encouraging, since the generic representation has sufficient capability to capture method processes as well as allowing the customisation of fine grain processes.

Chapter five and this chapter describe the structural and behavioural aspects of method models in graphical form. The textual form is presented by a unique method specification language. In the next chapter, we look at the heuristic model that accompanies the product and the process models.

7. HEURISTIC MODEL

Chapter five and six introduce the two main models in GMR. However, the method model is incomplete unless it is accompanied with the heuristic information embedded in the method. This chapter is concerned with a pragmatic approach to incorporate method heuristics in our meta model.

7.1 INTRODUCTION

Heuristics can be considered as experience that is obtained by domain expertise. Some heuristics give a detailed description of the method concepts. This may be the definition of a particular semantic or guidance to use the information. Other heuristics outline the processes of software development. They may provide assistance of the task involved or design decisions needed to be considered. Since these semantics are closely related to the concepts and techniques, they should be documented as part of the SDM.

Both ALF-MASP and SOCRATES address the existence of method heuristics (see chapter three for details), but they treat heuristics lightly in their meta model representation. Besides, most metaCASE tools, such as ToolBuilder, simply assume the semantics are well-known to the tool user (see chapter eleven). There is no specific denotation of heuristics.

[Clancey 85] presents a heuristic classification model for characterising different kinds of problems. The emphasis is on problem-solving reasoning and particularly in psychology rather than software development. However, the idea is notable in considering different types of heuristics in SDM. [Causse 93] defines a heuristic level of description for heuristic control knowledge which is based on the control roles central to the approaches of KADS. The KADS views of domain level and reasoning level correspond to our product model and process model respectively. Although heuristics have a close relationship with the two levels (models), we believe that it should not be treated as an interface between them in meta modelling. GMR describes a heuristic as a pragmatic structure which can be attached to other semantics of the method. It provides guidance and defines the usage of the associated semantic. This structure has been tested and evolved from the five chosen methods (see appendices F and G).

This chapter is organised as follows. Section 7.2 defines the structure of method heuristics and classifies it as concept heuristic or task heuristic. The three fields of a heuristic record: *text*, *rule* and *link*, are described in sections 7.3, 7.4 and 7.5 respectively. Section 7.6 illustrates the mapping of the heuristic model to the other two models and section 7.7 gives a conclusion.

7.2 METHOD HEURISTIC

The method heuristic is one of the important structures of a single method, because it is the kernel of any semantic system. The heuristic system is the semantic assistant embedded in both the product model and the process model. It can be a definition of a diagram or of a notation. Alternatively it could be a design guidance or a design rule of a software development process. Therefore they are not prescriptive.

There are two types of heuristic: rules and criteria (guidance text). However, to enhance the method heuristic, we introduce links to enhance the cross referencing mechanism of heuristic records in the semantic knowledge base. Since a method heuristic must refer to a semantic of the product model or the process model, each heuristic record is addressed by the name of the semantic in their respective model. In addition, each record contains three fields which are the *text*, the *rule* and the *link*. Figure 8.10 (in next chapter) presents a full BNF grammar of the heuristic specification language, here is the simplified definition of a heuristic record:

```
heuristic HeuristicName ;  
text      HeuristicDescription ;  
rule      Conditions [=> Action] ;  
link      ListOfHeuristicNames ;
```

These fields are explained in more detail in the subsequent sections. Essentially, any heuristic whose action is determined by pre-conditions and is definitive in its logic may be automated in a production rule system. Such heuristics are described by the rule subfield. Where it is possible, this is the most desirable subfield to use, since such decisions can then be made (optionally) transparent to the user. This hides the complexity, shortening development time and can reduce errors by permitting automation. Any heuristic which is ambiguous, open to interpretation, context dependent or optional has to be described by the text subfield. The link subfield is rather special and empirical experience has shown it to be necessary, in documenting the heuristics of a method.

Before any further description, it is essential to identify different types of method heuristics. From the nature of method heuristics, we classify them into two categories: *concept heuristic* and *task heuristic*. They are defined in the following two subsections and further descriptions are given in the discussions in sections 7.3 to 7.5.

7.2.1 CONCEPT HEURISTIC

Chapter two states that different concepts may be used to describe various semantics in software development. It is important to give a clear specification of each concept that has a definitive meaning apart from the concept name itself. Therefore the description of a concept with necessary examples are significant heuristics of the method. This information is stored in the *text* field of a heuristic record, which is known as a concept heuristic.

The *rule* field of a concept heuristic is usually empty. If the concept is constrained (section 5.3.2.6), then the field is used to record the constraint rules for consistency checking. This rule is triggered whenever any concept functions associated with that concept instance are fired. This rule is generally a conditional rule without action; or otherwise the action of the rule is that the concept instance itself does not exist. For instance, the following heuristic record of the *dataFlow* concept in OMT, the rule restricts any *dataFlow* instance not to connect between an *actor* instance and a *dataStore* instance directly. (see section 7.4)

```

heuristic dataFlow ;
text    'Data flow is the connection between the output of one object or process and the
        input to another.' ;
rule    not( source(actor) and target(dataStore) ) or
        not( source(dataStore) and target(actor) ) ;
link    actor, dataFlowDecomposition, dataFlowDiagram, dataStore, process ;

```

In a concept heuristic, the *link* field contains a list of concepts that are concerned with the heuristic. This link is usually denoted by a relationship of the concept in the product model. However, other non-directly associated concepts may also occur. They provide the cross referencing mechanism in defining the meaning and/or constraint of the original concept.

7.2.2 TASK HEURISTIC

A task describes a software development process based on the concepts recorded in the product model. The process model shows *what* are the interrelationships between tasks and defines the structure of an individual task. Each task is comprehended by a task heuristic that provides a detail specification of *how* can the task be used. The following heuristic record illustrates the task¹ *taskCohesionCriteria* in Codarts/DA:

```

heuristic taskCohesionCriteria ;
text    'Task cohesion criteria address whether and how transformations should be grouped
        into concurrent tasks.',
        'There are four types of task cohesions: temporal cohesion, sequential cohesion,
        control cohesion and functional cohesion.' ;
rule    'Two transformations are constrained so that they cannot execute concurrently and
        hence must execute sequentially' => do(taskCohesion) ;
link    temporalCohesion, sequentialCohesion, controlCohesion, functionalCohesion ;

```

The *text* field supports the task with a textual description, which can be a general guidance or criterion for the execution of the task. Besides, it acts as an English specification of the *rule* field. It can be displayed as help information to assist the software developer with the task.

The *rule* field of a task heuristic denotes a design decision as a production rule. The antecedent of a rule comprises the AND-OR statements of a design situation, whereas the consequence may trigger a task function.

¹ It must be clear about the difference of tasks in the method model (by GMR) and the software model (say by OMT). A method task is a software development process, but a software task describes an application process.

The *link* field includes a list of semantics that relate to the specific task. This semantic could be a concept, a task or another heuristic. A link to a concept is normally presented as the input or output concept flow in the process model, whereas a link to a task is denoted by the task trigger, decomposition and refinement in the task diagram (see chapter six). However, the main aim of this *link* field is to record the highly coupled task heuristics together. These heuristics have to be considered simultaneously. It is because firstly they may accompany one another's design decision, and secondly they may even contradict each other. This is illustrated by the *taskStructuringCriteria* of Codarts/DA method² in section 7.5.

7.3 HEURISTIC TEXT

A heuristic text is a criterion that is a software development idea. It is recorded as a list of strings in the text field. Most heuristics include this field to describe their rules and links. A criterion can be a definition, an explanation or a guideline.

For a concept heuristic, the criterion is usually presented as a definition appearing in the glossary or main text of the literature. In the illustration of the concept *dataFlow* in section 7.2.1, the description of *dataFlow* in OMT is given.

For a task heuristic, more complicated information may be recorded. Using the Codarts/DA example from section 7.2.2, the first text clause in the *taskCohesionCriteria* defines what it is. The second text clause explains that the *taskCohesionCriteria* can be further categorised into four different types, which are *temporalCohesion*, *sequentialCohesion*, *controlCohesion* and *functionalCohesion*. These criteria are connected by explicit links listed in the link field (see section 7.5).

In the *functionalCohesion* task heuristic shown below, the second text clause (the bolded text) serves as a general guideline to apply the heuristic rule (see section 7.4). This information is important in assisting user to select an appropriate criterion in the according task.

```

heuristic functionalCohesion ;
text  'Function cohesion occurs when there are one or more functions that are closely
      related and only one function may be executed at any one time.' ,
      'Function cohesion is the weakest form of task cohesion, it should be used
      primarily when it supports the other forms of cohesion.' ;
rule  'The data traffic between two functions is high and having them as separate tasks
      could increase the system overhead.'
      or
      'A group of functions all operate on the same data structure or the same I/O tasks.'
      or
      'The transformations are related both temporally and functionally.'
      => modify(task) ;
link  taskCohesion, temporalCohesion, sequentialCohesion, controlCohesion ;

```

² Both OMT and Codarts/DA are heuristic 'rich' methods.

The use of a SDM requires decisions about the application domain to be made. If these decisions are triggered by heuristic rules they can be automated. Other decisions rely upon user discretion and require input from the user. During system automation or user determination, heuristic text can be used as help text. It is attached to the process model and the product model, so that context-sensitive messages can be presented to the user.

Whether one is building a new method, customising an existing method or simply using a method, the heuristic model being described in this chapter is equally applicable. The help window allows the user to search through the knowledge base and to examine the heuristic text of related items via the heuristic links.

7.4 HEURISTIC RULE

A heuristic rule is like a production rule in an expert system. A rule contains two parts, an antecedent and a consequence. When the antecedent is fulfilled, the consequence is triggered. For instance, an entity must have at least one relationship with other entity in the system, otherwise it must be outside the system.

The standard form of a rule is *if Condition then Action* or *Antecedent => Consequence* or in the prolog predicate form *Deduction :- Pattern*. We adopt the second form (as shown in the *taskCohesionCriteria* example) since it is considered to be more adaptable as a data field in our heuristic specification language. Both antecedent and consequence accept textual form or functional representation. They are separated by the reserved symbol ' \Rightarrow '.

It is proposed to introduce three logic keywords into the heuristic rules, namely: **not**, **and**, **or**. The **not** keyword can be used for logic inversion, and it must be placed before a clause. The last two keywords are conjunction keys, and examples of use are shown in the *functionalCohesion* above in section 7.3. These keywords are placed between antecedents. The precedence of these keywords follows the order as listed above. This work is intended to be exploratory.

There are fundamentally three levels of heuristic rules.

- The lowest level regards coercions or limitations, for instance multiplication or referential integrity in the OMT. They may be obvious rules to the human designer but still need to be recorded. The user should be warned or even have entries rejected if the input invalidates these rules. In addition, these rules may appear in the form of *antecedent* only. Therefore they are known as *constraints* or *conditions*, and they must hold valid at all times. An example of this is the constraint rule of *dataFlow* concept shown in section 7.2.1.

- The *functionalCohesion* rule is considered as the middle level. These rules help the user to make design decisions or to select a suitable system configuration. There are usually many such rules in most SDM. The enforcement of these rules is not as ‘strong’ as the lowest level ones, but the user is required to follow the guidance. The choices made will be logged, so that potential problems can be traced through an audit trail.
- The highest level rules deal with design strategy. For instance: *IF the application regards real-time processing and has a lot of Computation blocks THEN use a task architecture diagram in Codarts/DA*. These level of rules are strategic knowledge in KADS, which is considered as outside the scope of this research (also see section 12.3).

7.5 HEURISTIC LINK

The main usage of any heuristic link is to cross reference between different heuristics and this helps to navigate through the network of heuristics. An example of an OMT heuristic network³ is shown in figure 7.1. The notations used in the figure are explained within the dotted box. All heuristics are under a concept heuristic, known as *objectModellingTechnique*. The diagram also illustrates that a method heuristic network is mainly divided into two sub-networks: one for the concept heuristics and the other for the task heuristics. These two sub-networks are linked together by the concept flows as represented in the task diagrams.

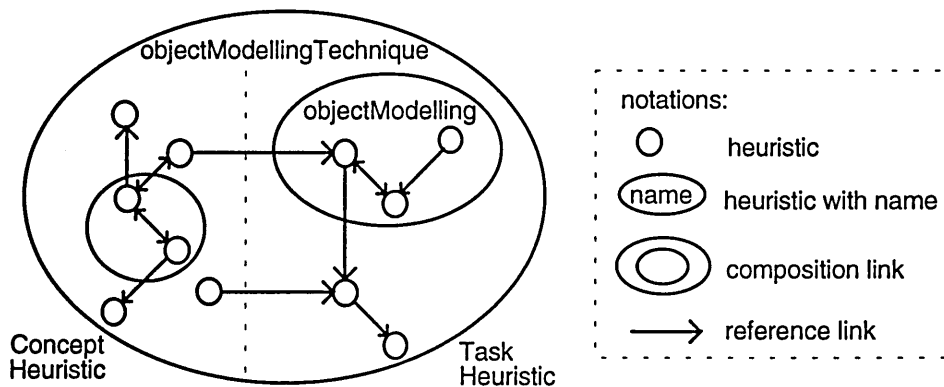


Figure 7.1 OMT: Heuristic Network

We classify the two types of heuristic link: composition link and reference link. The former can only appear in one side of the network, whereas the latter also provides links between the sub-networks. Although the heuristic specification language treats them as reference pointers only, it is useful to differentiate between the two types for semantic clarity.

³ Each heuristic must have a name, but the heuristics in the figure are depicted as circles in order to reduce complication.

7.5.1 COMPOSITION LINK

A composition link is used to connect a criterion to its branches, especially in the definition of relevant issues. For instance in Codarts/DA, the *objectStructuringCriteria* introduces five different *object* concepts: an *externalDeviceI/OObject*, a *userRoleObject*, a *controlObject*, a *dataAbstractionObject* and an *algorithmObject* as shown in figure 7.2. Each of these objects has their own criteria, which are also combined as part of the *objectStructuringCriteria*. The denotation of composition link simply encloses the branch-heuristics⁴ as the example shown in figure 7.2.

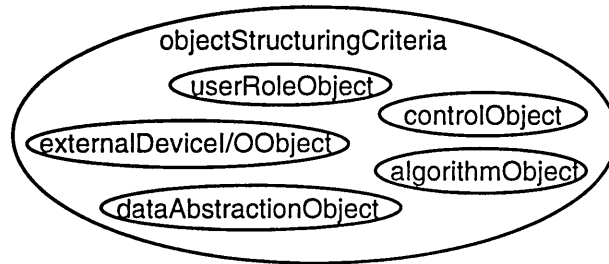


Figure 7.2 Codarts/DA: *objectStructuringCriteria* Composition Link

Figure 7.3 depicts another illustration of composition links amongst task heuristics: the *taskCohesionCriteria* and the *taskInversionCriteria* in Codarts/DA. As mentioned in section 7.7.2, the *taskCohesionCriteria* comprises four branches of cohesion criteria. The *taskInversionCriteria* also consists of three types of inversion criteria of concurrent tasks. Therefore the relationships can be denoted by the composition links as in the following diagram.

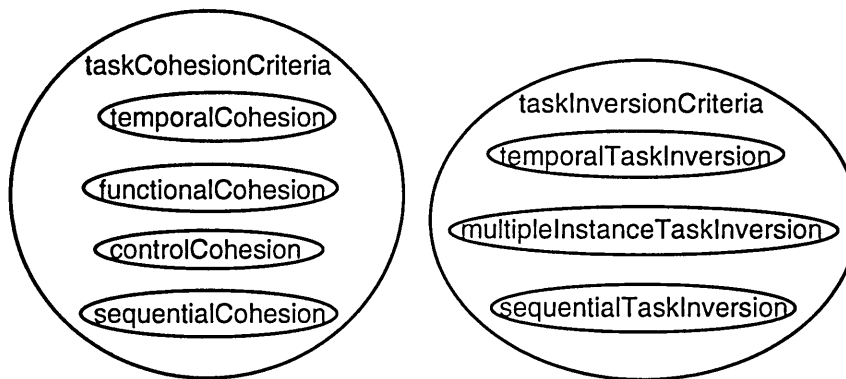


Figure 7.3 Codarts/DA: *taskCohesionCriteria* and *taskInversionCriteria*

⁴ In order to avoid confusion with the subtyping and the composition denotations in the product model, we use the stem-branch relationships to represent the hierarchical structure, instead of parent-child or whole-part relationships.

7.5.2 REFERENCE LINK

A reference link connects all other associations between heuristics, apart from the composition links. For instance, each of the three conditions in the *functionalCohesion* (shown in section 7.3) is related with the other three *taskCohesionCriteria*: the *temporalCohesion*, the *sequentialCohesion* and the *controlCohesion*. Therefore they are interconnected by the reference links as shown in figure 7.4 below. A reference link is depicted by a v-shaped arrow pointing towards the receiving heuristic symbol. In most cases, a reference link is a bi-directional relationship so the link has arrows at both ends.

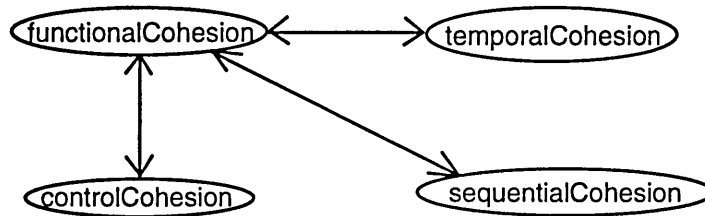


Figure 7.4 Codarts/DA: Reference Link of *functionalCohesion*

In addition, a referencing link may also relate contradicting heuristics. For instance, the *temporalCohesion* and the *functionalCohesion* repudiate the *temporalTaskInversion* in the *taskInversionCriteria*, as shown in the heuristic record below. The according reference links must be provided in making an appropriate design decision. Figure 7.5 depicts these links.

```
heuristic temporalTaskInversion ;
text    "temporal task inversion is similar to the weak forms of temporal cohesion, two or
        more periodic, periodic I/O and/or periodic temporally cohesive tasks are combined
        into one task" ;
rule    "temporal transformations that are functionally related into a task"
        and
        "the tasking overhead is considered too high"
        => modify(task) ;
link    taskCohesionCriteria, temporalCohesionCriteria, functionalCohesionCriteria ;
```

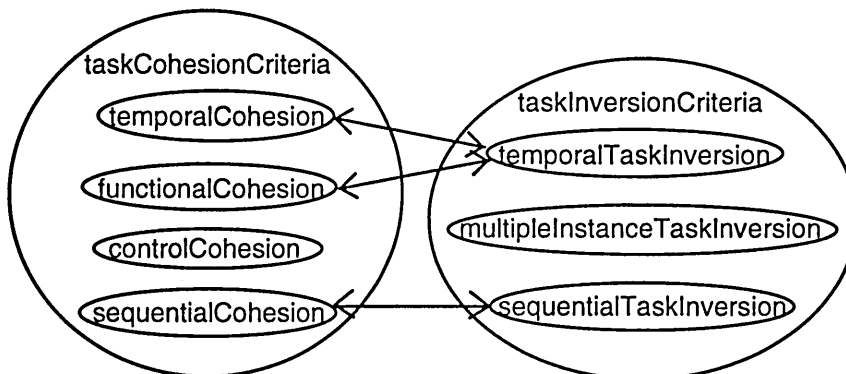


Figure 7.5 Codarts/DA: *taskStructuringCriteria* Reference Link

7.6 MAPPING TO THE TWO MODELS

The heuristic model is one of the three models in GMR. This section demonstrates the mapping of the heuristic model to the other two models, which are the product and the process models. We first look at the mapping of method heuristics and then the mapping of heuristic links.

For the concept heuristic, each concept in a product model has exactly one heuristic, such as the OMT action concepts shown in figure 7.6. The mappings are depicted by thick solid arrows. The association is maintained by sharing the same semantic token, which is the concept name in the product model and the heuristic name in the heuristic model.

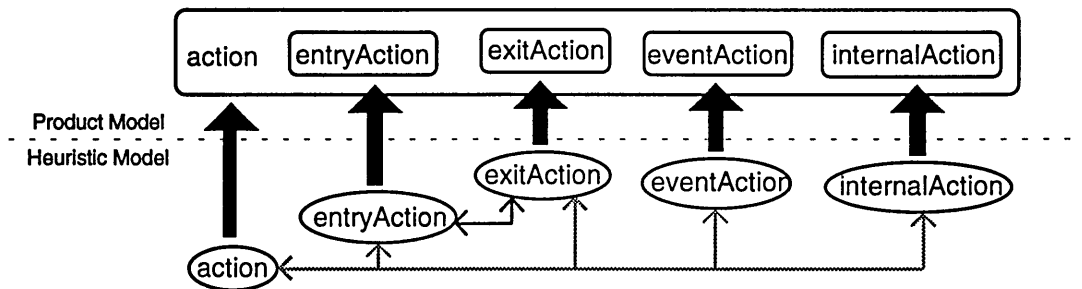


Figure 7.6 OMT: Mapping Heuristics to Concepts in Product Model

However, for the task heuristic, each task is attached with the appropriate heuristic (by name) and that heuristic may extend to link to other task heuristics. For instance the *insert(task)* function and the *insert(informationHidingModule)* function are accompanied with the *taskStructuringCriteria* and *moduleStructuringCriteria* respectively. On the other hand, the heuristic can refer back to its corresponding task by the consequence part of the heuristic rule field. In the *temporalTaskInversion* criteria (shown in the previous section), the heuristic relates to the task function *modify(task)*.

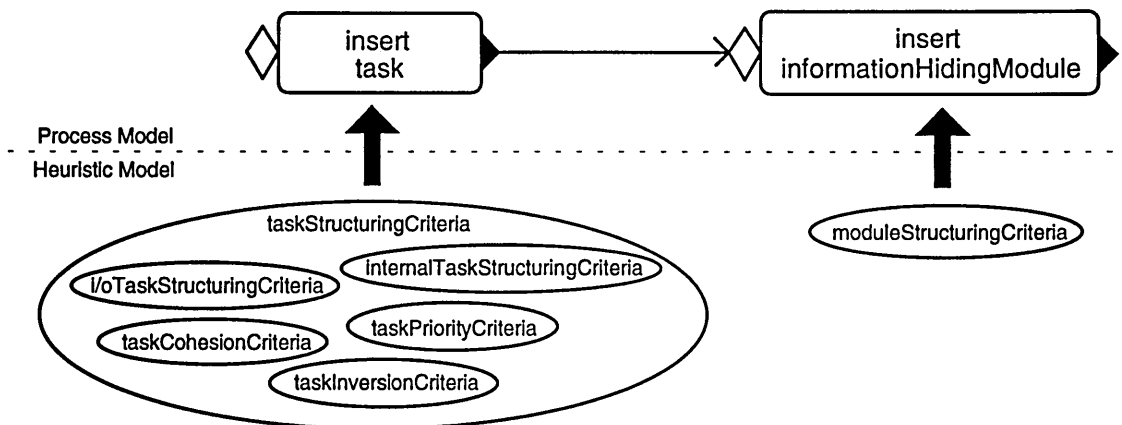


Figure 7.7 Codarts/DA: Mapping Heuristics to Tasks in Process Model

The mapping of links in the other two models to heuristic links can be summarised in table 7.1. However, the reference link may be used to connect semantics that do not have a direct relationship in their respective model. For example, figure 7.6 illustrates that the *entryAction* heuristic refers to the *exitAction* heuristic in order to give a better definition of the two actions. Therefore the reverse mapping, that is from the two models back to the heuristic model, may not be valid.

Heuristic Model	Product Model	Process Model
composition link	concept subtyping concept composition	task decomposition task refinement
reference link	concept linking concept grouping concept referencing	concept flow

Table 7.1 Mapping Heuristic Links to Product and Process Models

It is deliberate policy to allow dual representation in the heuristic model with respect to the other models. Since for some heuristic-rich methods, such as Codarts/DA, it is easier to show the relationships between heuristics in a single model than to refer to separate models. Furthermore, if we reinforce the extra associations of heuristics in other models, for instance the *entryAction* and *exitAction* link mentioned above, the product model has some additional non-conceptual information presented.

7.7 CONCLUSION

This chapter illustrates how to model heuristics in SDMs by identifying the three fields in a heuristic record. These fields are heuristic text, rule and link. This chapter classifies heuristic into two types, which are concept heuristic and task heuristic. The mapping of such a heuristic model to the other two models in GMR is also demonstrated.

8. METHOD REPRESENTATION

Similar to a software model, a method model describes the structural and behavioural aspects of a SDM. The encapsulated information about the SDM is known as method representation. This representation is based upon a generic meta model to document the product, the process and the heuristic of the SDM comprehensively. In this chapter the method representation of GMR is presented. Some extra modelling considerations are also discussed.

8.1 INTRODUCTION

In meta modelling, the semantics of a method should be represented as concepts or techniques. The ideas of individual fragment, entity and property should be denoted so that they can be used to define the specific task operations and dependence of software development processes in the method. The heuristic guidance and rules should also be included in order to provide substantive assistance for the method engineer. All these semantics should be specified to capture the characteristics of the method. In other words, the method representation is a concise and precise documentation of the SDM. Hence the generic model needs to handle various presentation techniques found in method models.

Chapter three shows various attempts to represent SDMs from different viewpoints, but none of them is satisfactory. There is in the literature no established way of deriving such a model based on capturing method techniques rather than specific methods. In this research we propose such a model and try it out on five SDMs to demonstrate the expressiveness of method representation. These methods are Booch OOD, Codarts/DA, HOOD, OMT and Ptech (see chapter two for the overviews and justification). They cover a large spectrum of software modelling techniques and emphasise various aspects of systems development.

The concept diagrams of these five methods and the task diagrams of OMT are attached in this thesis as appendices D and E respectively. The detailed description of individual representations of these methods is less significant and they will not be given in this thesis. However, in this chapter the pragmatic model of GMR is discussed. A common specification language is used for the three models of GMR that is known as *Method Specification Language* (MSL). The complete BNF grammar of MSL is also given.

The organisation of the chapter is as follows. The next section gives an overview of method representation by GMR. Section 8.3 describes some additional considerations in this method representation. The grammar of the three method components in GMR is presented in section 8.4. Section 8.5 shows Prolog clause formats corresponding to the MSL grammar, and the last section gives a conclusion.

8.2 OVERVIEW OF GMR REPRESENTATION

In our generic representation, the semantic information of a single method comprises three parts. The product model describes the method concepts (chapter five); the process model describes the method tasks (chapter six); and the heuristic model gives the method guidance and rules (chapter seven). These components represent the method both graphically and textually. The grammar of each model is documented by a uniform method specification language (MSL), which can then translate to Prolog predicates for execution.

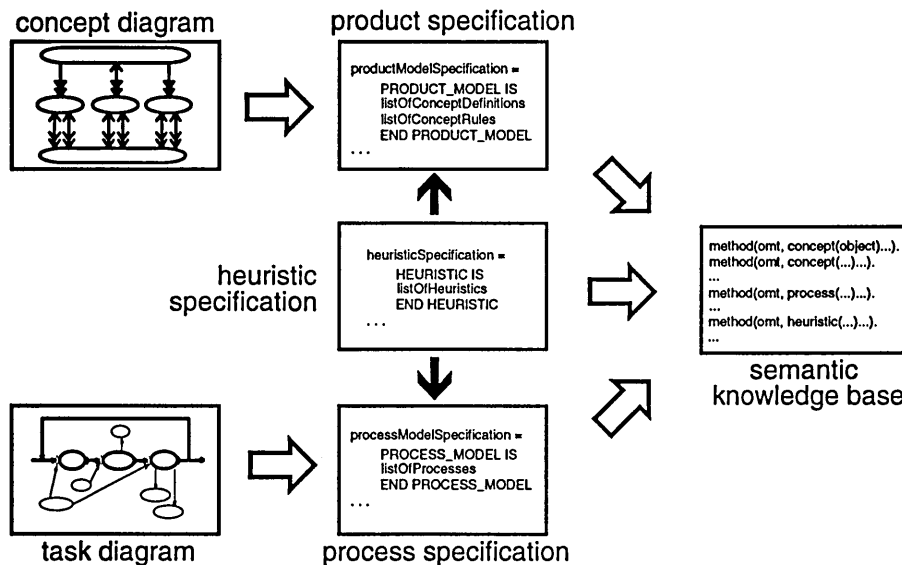


Figure 8.1 GMR Method Development

Figure 8.1 illustrates the method development by GMR. After knowledge acquisition of a SDM (see chapter ten) the method product and the method process are identified and depicted by concept diagrams and task diagrams respectively. Each of these diagrams is then converted into the corresponding specification as shown by the hollow arrows in the figure. The method guidance and rules are also documented as the heuristic specification. These processes of information are linked directly to the related concepts or tasks. Finally, all these specifications are compiled into Prolog predicates.

The descriptions of individual constituents are given in the previous three chapters. In this section, we present the enhancement of the meta models and discuss the significance of such method representation. The next subsections introduce the meta modelling techniques based on dissection sets which are followed by shared concepts between fragments. The representation of various peculiar method relationships, such as derived relationship, multiple inheritance and delegation are discussed. Then a further description on the text fragment is given. Finally, a complete structured grammar of the MSL is presented in the last section.

8.3 ADDITIONAL CONSIDERATIONS

This section introduces some extra representation considerations that have not been discussed in the previous chapters. They include dissection set, shared concept, other concept relationships and text fragment.

8.3.1 DISSECTION SET

In chapter five, the product model introduced five method concept relationships to define different types of association between method concepts. These relationships can be considered as primitive (or meta) relationships in the concept diagram, and they should be differentiated from the method relationships. For instance, *aggregation*, *generalisation* are method relationships between *objects* in OMT. However, they are shown as method concepts in the meta level, and the relationship between these concepts and *object* concept is described by the linking meta relationship. All these relationships are denoted as a link between entity concepts or between an entity concept and a fragment concept.

In a SDM, a concept diagram may contain a number of fragments to describe various aspects of a system. Normally subtyping and referencing are the common concept relationships between method fragments. For examples, in Codarts/DA the *systemContextDiagram* is a subtype of *contextDiagram*; and in Booch OOD each *stateTransitionDiagram* refers to a *stateTransitionTemplate*. In this section we introduce another relationship which only happens between fragment concepts. This relationship is to deal with the common concepts across fragments. Since it is very important in fragment dissection and it copes with one or more method concepts between two fragments, the association is known as a **dissection set**.

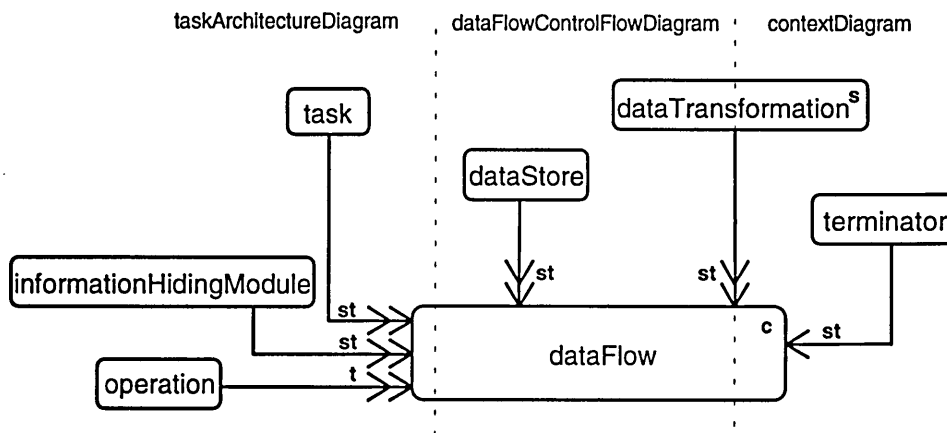


Figure 8.2 Codarts/DA: *dataFlow* and *dataTransformation* Concepts

Figure 8.2 illustrates the overlapping concepts of different fragments in Codarts/DA [Gomaa 93]. The *dataFlow* is a link concept to relate entity concepts in each of the three diagram

fragments. For a *taskArchitectureDiagram*, a *dataFlow* connects a *task* and an *informationHidingModule* or it links up a *task* to an *operation*¹ of an *informationHidingModule*. For a *dataFlowControlFlowDiagram*, a *dataFlow* relates a *dataTransformation* and a *dataStore*. However, a *dataTransformation* can also connect with a *terminator*² by a *dataFlow* in a *contextDiagram*. Due to these complex conditions in the *dataFlow* concept³, a set of constraint rules is required to present these relationships and this is denoted by the label 'c' in the *dataFlow* box (refer to section 5.3.2.6). The dotted lines show the division between fragments. The line between the *dataFlowControlFlowDiagram* and *taskArchitectureDiagram* cuts through the *dataFlow*, which means that the *dataFlow* is a concept applied to both fragments. The dissection set is $\{dataFlow\}$. The line separating *contextDiagram* and *dataFlowControlFlowDiagram* cuts through the *dataFlow* and the *dataTransformation* concepts, which means both of them can be shown in these two fragments. Hence, the dissection set is $\{dataFlow, dataTransformation\}$.

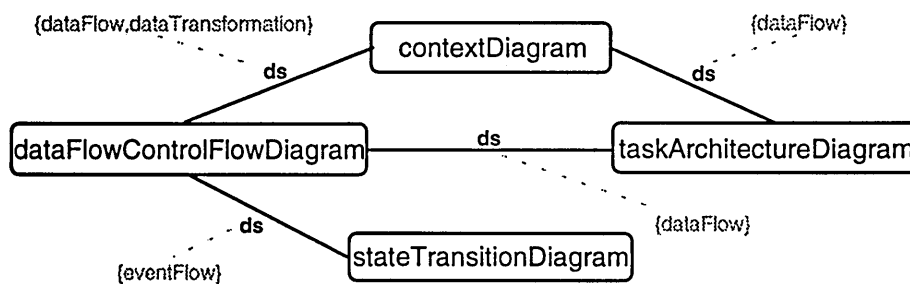


Figure 8.3 Codarts/DA: Dissection Sets

A dissection set is denoted by a line joining the two related fragment concepts with the label 'ds' on it. Figure 8.3 demonstrates the four dissection sets as they appear in Codarts/DA. In addition the figure shows the two possibilities of common concepts in a dissection set.

- Firstly, it simply represents the concepts that literally appear in both fragments, and the instances of the concept which have no interrelationships at all. In the Codarts/DA example, *dataFlows* which appear in different diagrams are separate tangible links.
- Secondly, it shows a continuation of concept modelling between fragments. Again in the example, a *dataTransformation* in the *contextDiagram* is further described in the *dataFlowControlFlowDiagram* and the decomposed diagram of itself.

¹ In this case, the *dataFlow* presents that the *task* invokes an *operation* of the *informationHidingModule*. Therefore the *operation* can only be a target concept in this linking relationship.

² This is a very special case of linking relationship. Since each *terminator* can only have one link to the *dataTransformation*, this linking relationship is denoted as either a source part or a target part of a *dataFlow*.

³ Since all links can have alternative source and target parts, they are shown as optional linking relationships.

8.3.2 SHARED CONCEPT

A shared concept can be considered as a by-product element of the dissection set. A concept is 'shared' if it is a common concept between fragments and each instance of the concept is a component of the fragments. Therefore shared concepts are the strongest links between fragments. For instance, in figure 8.2, each *dataTransformation* in a *contextDiagram* must also be denoted by or decomposed into a *dataFlowControlFlowDiagram*, therefore *dataTransformation* is a shared concept. On the other hand, *dataFlow* is a tangible concept that happens to relate entity concepts in various fragments, which means *dataFlow* is semantically disconnected between the fragments; hence it is not regarded as a shared concept. This notion is depicted by the label 's' in the concept box. If the shared concept is also constrained, the two labels can be combined as 'sc'.

This sharing feature between fragments can also propagate along the method relationships of a shared concept. In a *hoodDiagram* an *object* can have more than one *constrainedOperation* and each of these *operations* may have a *constrainedByLabel*. On the other hand, in the *objectDescriptionSkeleton* the *objectControlStructure* section describes the detail of each *constrainedOperation* including *constrainedByLabel*. Therefore even though *object* and *objectControlStructure* do not have a direct relationship with *constrainedByLabel*, it is still a shared concept between the two fragments. In figure 8.4, the dotted line shows part of the dissection between the two fragments. The dissection on the *constrainedOperation* concept propagates through the composition relationship to the *constrainedByLabel* concept.

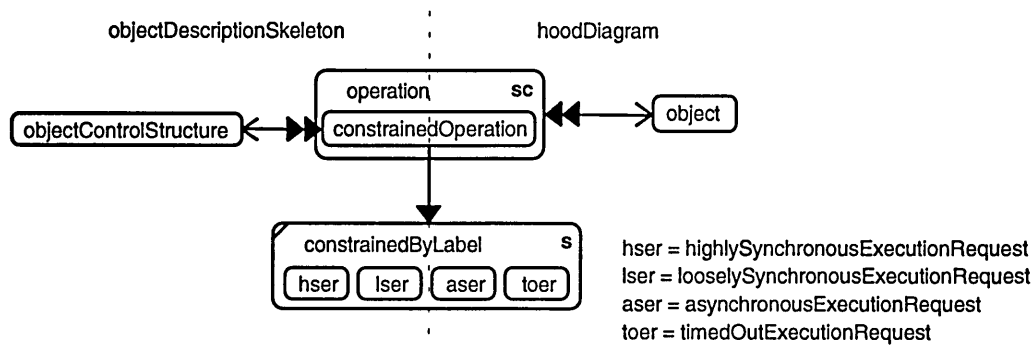


Figure 8.4 HOOD: *constrainedOperation* Shared Concept

If the tool fragments of a SDM are closely related, the shared concept becomes a very useful technique to avoid repeated concept descriptions to individual fragments. These highly coupled fragments occur when a method has different viewpoints or emphasis on the same set of concepts. A typical example is the HOOD method. A HOOD system is represented by the graphical fragment called *hoodDiagram* and each object is further documented by the textual fragment known as *objectSkeletonDescription*. Since all the concepts of an *object* shown on

the *hoodDiagram* are also described in corresponding sections of the *objectSkeletonDescription*, most of these concepts are denoted as shared concepts. Section 8.3.4 gives a detailed discussion of this example.

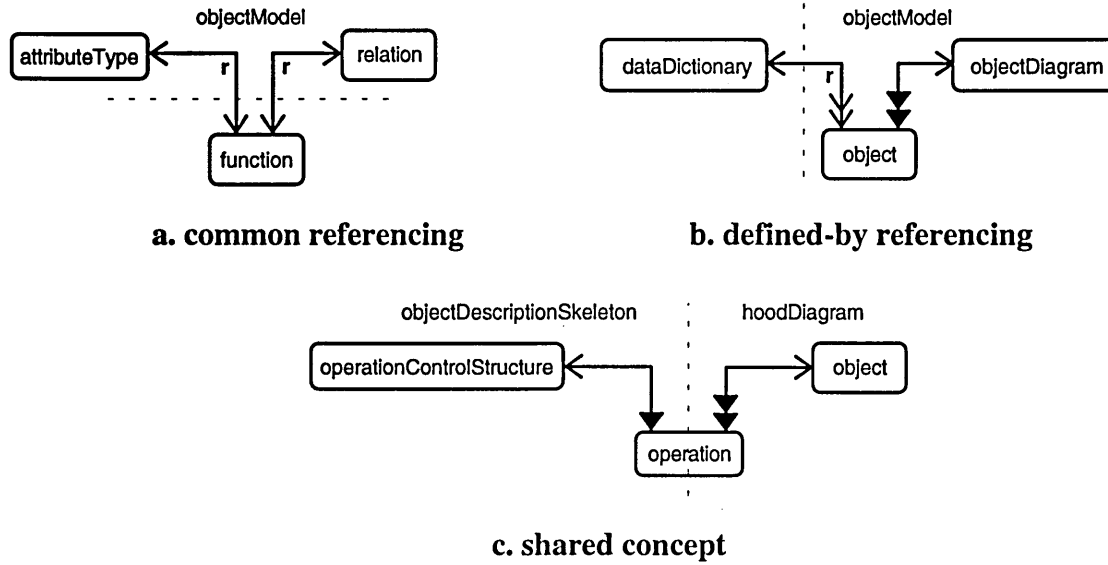


Figure 8.5 Common Concepts between Fragments

With the definition of shared concept, various combinations of fragment coupling by common concepts can be identified. Figure 8.5 summarises the three types of common concepts between fragments.

The first type is a fragment coupling based on referencing relationships to a common concept. In other words, two concepts individually refer to a third concept which is not part of the structure of the first two concepts. This is the most loosely coupled type. Normally the common concept comes from another fragment or simply a multi-valued entity concept of the method. For instance, in Ptech, both *attributeType* and *relation* refer to a *function*, as shown in figure 8.5a. The *function* is a multi-valued concept, which can be stored as clauses in a database. The dotted-line is the dissection line of the concepts.

Figure 8.5b illustrates a different type of coupling. The *dataDictionary* further defines the *objects*, which is part of the structure of an *objectDiagram*. Therefore the referencing relationship only acts as a pointer to the structure of *objectModel*. There is a large difference of dependence to the fragments, hence the dissection line is on the referencing relationship.

However the strongest type of coupling is denoted by shared concept. The common concept is part of the structures of concepts from different fragments. The degree of dependence is similar in both fragments. Figure 8.5c shows that both *object* in *hoodDiagram* and *operationControlStructure* in *objectDescriptionSkeleton* shared the same instances of *operation*, so the dissection line falls on the shared concept itself.

8.3.3 OTHER RELATIONSHIPS

In chapter five we identified the five method concept relationships, which are subtyping, composition, linking, grouping and referencing. The first two are common in software methods, whereas the last three can be considered as specific associations in meta modelling. We must be careful to note that they are meta relationships for modelling method concepts and not the software concepts in the method level. Therefore we look at the method modelling perspective rather than the software modelling perspective. Moreover, the meta product model is intended to represent the method concept. If the software concept is not presented by a method, i.e. no formal definition of approach, this meta model cannot be applied. We illustrate these points by the following relationships: *derived relationship*, *multiple inheritance* and *delegation*.

8.3.3.1 DERIVED RELATIONSHIP

Derived relationships are very common in software modelling. In OOSA, it is denoted by a correlation table or an association object. By definition, this relationship occurs when a software concept is based on other software concepts. Some methods elaborate this to derived attribute, derived class and derived association. The classic example of derived attribute is the attribute *age* of the object *person* derived from the *currentDate* minus the *birthDate*. OMT identifies these derived elements and shows them by adding a diagonal bar as shown in figure 8.6. Sometimes derived association is known as objectification. For instance, the relationship between *husband* and *wife* is objectified to an object called *marriage*. However, all these objects or derived relationships are software concepts rather than method concepts.

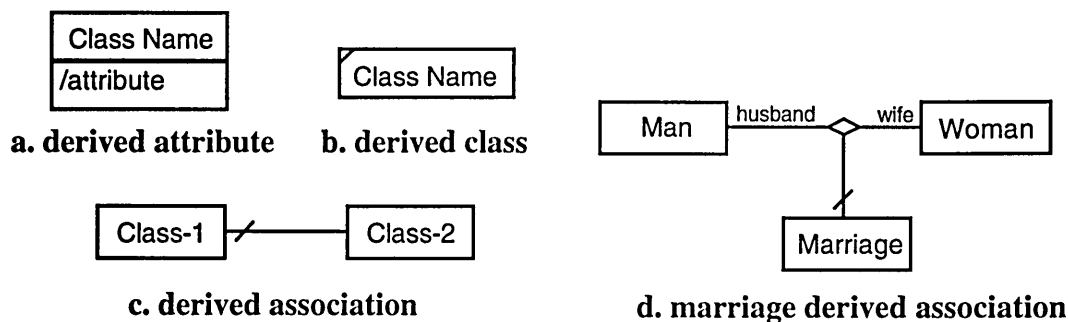


Figure 8.6 OMT: Derived Relationship

Therefore general derived relationships only apply to the software level and not to the method level. The definition of a derived relationship may be presented as a method concept but not as a meta concept relationship. In the next subsection we illustrate a special type of derived relationship in the method level, which is multiple inheritance.

8.3.3.2 MULTIPLE INHERITANCE

Multiple inheritance is a method concept which can also be considered as a derived relationship in the method level. In the object-oriented paradigm, when an object inherits features from more than one object, the relationship between the object and other objects is known as multiple inheritance.

There are two viewpoints for looking at multiple inheritance. The following figures depict these viewpoints graphically by entity relationship diagrams. The ovals represent the entities and the rectangles represent the relationships, where 'MI' stands for *multipleInheritance* and 'I' stands for *inheritance* (or *singleInheritance*). In the first viewpoint, multiple inheritance is described as a single method concept which relates a number of superobjects to a subobject, as shown in figure 8.7a. In the second viewpoint, the relationship is considered as an aggregate of single inheritances having the same subobject, as depicted in figure 8.7b. Therefore multiple inheritance is defined by a group of concepts, which includes the subobject and the single inheritance relationships.

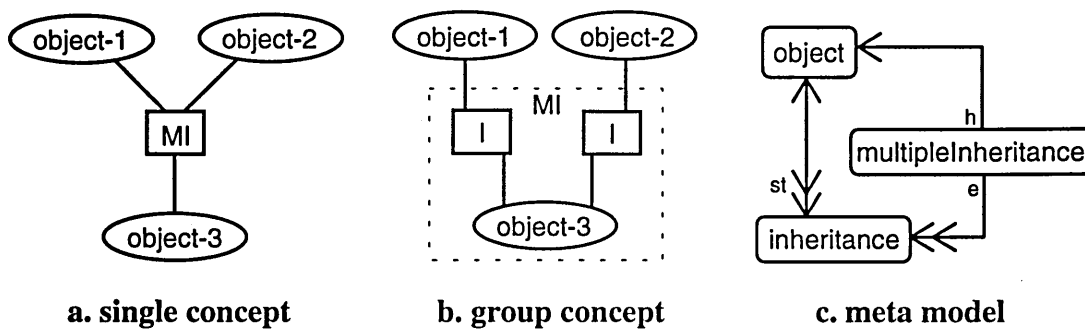


Figure 8.7 Multiple Inheritance

Our meta model adopts the second viewpoint. The reasons are threefold. Firstly, single inheritances break down the relationship into binary concept relationships and this concept relationship is very important in the method model. Secondly, multiple inheritance is normally depicted by multiple 'single inheritance' notations, which can be upward arrows, down arrows, surrounding boxes etc. Therefore, multiple inheritance is an implicit concept without notation. Thirdly, the definition of *multipleInheritance* concept is initiated from a subtype *object* concept combined with a number of inheritance relationships.

Thus the *multipleInheritance* concept is described as a group concept with *object* as host and *inheritance* as element. This grouping relationship is shown in figure 8.7c. Since there is only one *object* instance, the cardinality is (0,1,1,1) in the host part. In the element part, at least two of *inheritance* instances are required in the group, so the cardinality is noted as (0,1,2,n).

8.3.3.3 DELEGATION

Some software languages introduce a delegation mechanism into an object. Delegation is an implementation mechanism in which an object, responding to an operation on itself, forwards the operation to another object. In object-oriented languages it is a mechanism by which methods may be attached directly to instances and where the method resolution is performed by searching a chain of instance pointers rather than by searching a class hierarchy. The objects involved in delegation are normally known as actors [Tello 89].

The factorial algorithm is good for illustrating this because the delegation between actors that can accomplish a recursive factorial is simple. When a call is made to the generic factorial actor with a number n , a *factorial(n)* actor is formed. The actor will then compute the value by instantiating another actor of itself to solve *factorial($n-1$)*. The process keeps recursing until an actor calls itself to find the factorial of 1, which is defined to be 1 by a test condition. This chain reaction of delegation is depicted in figure 8.8.

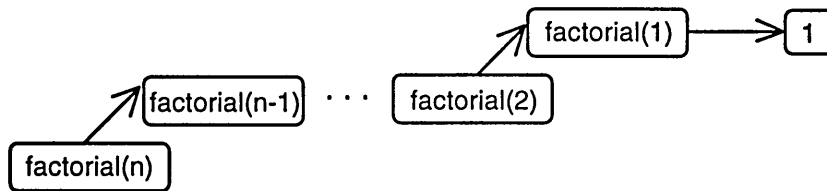


Figure 8.8 Actor Delegation

Delegation is a common implementation mechanism in actor based languages. This concept is a well defined relationship between actors but it is only considered as a technique. No formal SDM has found for these actor based languages. Since *delegation* is a 'floating' concept, it is considered as outside the scope of this research⁴ (see also chapter two).

8.3.4 TEXT FRAGMENT

In most of the discussion so far, diagram fragments have been used for illustration. However some SDMs also provide text fragments as method tools to describe software models. The *objectDescriptionSkeleton* in HOOD and all the templates in Booch OOD are examples of text fragments. A diagram fragment is good for describing various meta modelling techniques and ideas. In general, a text fragment is used as a comprehensive documentation of the pictorial concepts and usually it provides an extension to the graphical presentation. Therefore all the techniques in diagram fragments can also be applied to text fragments, and normally a text fragment has a very strong relationship with the underlying diagram fragment.

⁴ However, it is strongly suspected that *delegation* can be simply be expressed as a relationship in GMR.

Let us consider the two fragments in HOOD (see section 2.4.2): the graphical representation is called a *hoodDiagram* and the textual representation is known as an *objectDescriptionSkeleton* as illustrated in figure 8.9. A *hoodDiagram* is used to describe a network of HOOD objects or a parent *object* with all its descendants, whereas an *objectDescriptionSkeleton* is used to document various parts concerning an *object*. Hence, the *objectDescriptionSkeleton* has a one-to-one referencing relationship to the *object*. From the Figure 8.9, various concepts from the *hoodDiagram* are comprehensively documented in different sections of the corresponding *objectDescriptionSkeleton*. Since most of the basic concepts defined by the method are shown in the diagram fragment, the text fragment adds on details of individual concepts. Therefore the relationships between the concepts in different fragments are recorded by the 'defined-by' referencing relationship. For instances, *requiredInterface* defines required *objects* and *formalParameters*; *providedInterface* declares *operations* and *operationSets* provided by the *object*, whereas each *operationControlStructure* refers to one *operation* etc. Nevertheless, the internal structure of each fragment is still based on the concept modelling technique discussed in chapter five. Appendix D.1 shows a complete concept diagram of the HOOD method.

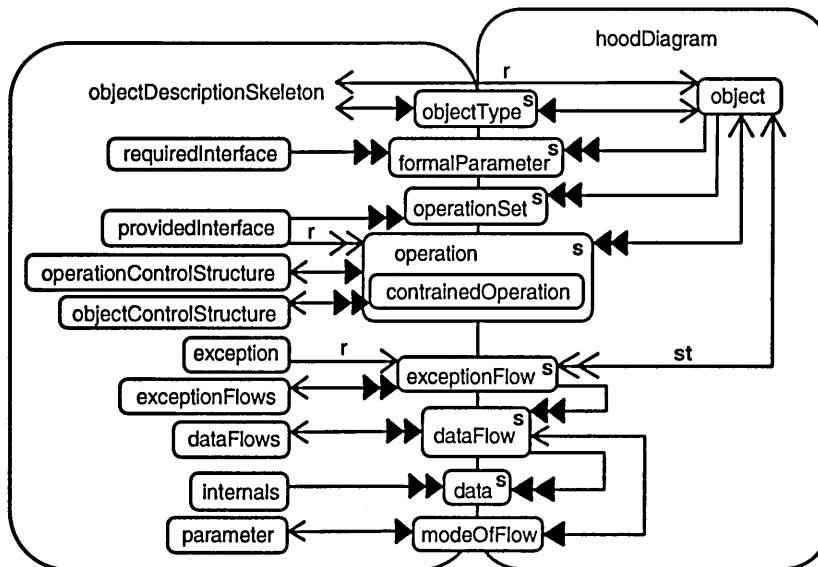


Figure 8.9 HOOD: Relationships Between Graphical and Textual Fragments

Moreover, the task modelling technique of diagram fragments can still be applied to text fragments. Each subtask may be used to specify the details of each section in the text fragment. The general information is outlined by a preceding task, which is the formation of the corresponding graphical representation of the model. Hence this concept specification can be considered as a refinement stage from the predetermined concept definition. Furthermore, each of these tasks and concepts can be accompanied by design heuristics.

8.4 METHOD SPECIFICATION LANGUAGE

In the discussion of various components of a method (those are in chapter four to seven), we use the graphical representation to depict the notions and the relationships of method semantics. It is because diagrammatic form is easier to illustrate the modelling techniques of GMR. This section gives a complete definition of the textual representation. However, it must be clear that the textual form is a complement to the graphical form. This textual form of the GMR method model is by means of a specification language known as Method Specification Language (MSL). It is capable of denoting semantics in the product model, the process model as well as the heuristic model.

We follow the BNF grammar rules as described in [Schreiber 93] and the basic set of construct interpretations are shown in table 8.1. In addition, comment lines are preceded by the symbol ‘%%’.

Construct	Interpretation
<code>::= * + [] <> .</code>	symbols that are part of the BNF formalism
<code>X ::= Y</code>	the syntax of X is defined by Y
<code>[X]</code>	zero or one occurrence of X
<code>X*</code>	zero or more occurrences of X
<code>X+</code>	one or more occurrences of X
<code>X Y</code>	one of X or Y (exclusive-or)
<code><X></code>	grouping construct for specifying scope of operators
<u>symbol</u>	predefined terminal symbol of the language
<i>symbol</i>	user-defined terminal symbol of the language
symbol	non-terminal symbol of the language

Table 8.1 BNF Grammar Rule

A SDM is defined as a composite of three distinct models, which are the product model, the process model and the heuristic model. Each model is further specified in different sections as shown in figure 8.10⁵.

The meta model has been tried out on the five chosen methods listed earlier. These method representations are used in the method evaluation that are discussed in the next chapter. A complete listing of OMT product, process and heuristic specifications in MSL is attached as appendix F of this thesis. These MSL statements are input to LPA Prolog and then intended to be converted into SKB format (the Semantic Knowledge Base described in chapter four). In addition, the OMT MSL statements are converted to Prolog clauses and they are shown in appendix G.

⁵ The definition of concept rules follows the standard declaration of Prolog Clauses. Due to the complexity it is not shown in the figure. Detailed examples of concept rules can be found in appendix G.

```

methodDefinition ::=
    method methodName
        productModelDefinition
        processModelDefinition
        heuristicModelDefinition
    endMethod .

%% DEFINITION OF PRODUCT MODEL

productModelDefinition ::=
    productModel
        listOfConceptElements
    endProductModel .

listOfConceptElements ::= < conceptElementDefinition >+ .
conceptElementDefinition ::=
    conceptDefinition | referencingDefinition | dissectionSetDefinition .

conceptDefinition ::=
    concept conceptName [abstract] :
        [subtypeOf subtypeCategory <_ conceptName>* ;]
        [groupingDefinition | linkingDefinition]
        [listOfCompositionDefinitions] .

subtypeCategory ::= method | fragment | diagram | text | group | link | conceptName .

groupingDefinition ::=
    host conceptName cardinalitySet ;
    member conceptName cardinalitySet ; .

linkingDefinition ::=
    <source conceptName <_ conceptName> cardinalitySet ;>+
    <target conceptName <_ conceptName> cardinalitySet ;>+ .

listOfCompositionDefinitions ::= <compositionDefinition ;>+ .
compositionDefinition ::=
    property conceptName cardinalitySet .

referencingDefinition ::=
    reference conceptName conceptName cardinalitySet ; .

dissectionSetDefinition ::=
    dissectionSet fragmentName fragmentName ;
    [ dissectionConceptDefinition <_ dissectionConceptDefinition >* ] ; .

dissectionConceptDefinition ::= [shared] conceptName .

cardinalitySet ::=
    [inclusionSymbol | exclusionSymbol]
    [ minSrcCar _ maxSrcCar _ minTarCar _ maxTarCar ] .

inclusionSymbol ::= #.
exclusionSymbol ::= !.

minSrcCar ::= 0 | 1 | 2 | n .
minTarCar ::= 0 | 1 | 2 | n .
maxSrcCar ::= 1 | 2 | n .
maxTarCar ::= 1 | 2 | n .

```

%% DEFINITION OF PROCESS MODEL

```

processModelDefinition ::=
    processModel
        listOfTaskDefinitions
    endProcessModel .

listOfTaskDefinitions ::= < taskDefinition >* .
taskDefinition ::=
    task taskName operationDefinition ;
        [precond listOfSemanticTokens ;]
        [postcond listOfSemanticTokens ;]
        [taskCompositionDefinition | taskRefinementDefinition] .

operationDefinition ::=
    operation <taskFunctionDefinition | conceptFunctionDefinition> .

taskFunctionDefinition ::=
    <perform | do> ( taskName ) .

conceptFunctionDefinition ::=
    <<draw | insert | delete | modify> ( conceptName )> |
    <<adjust | retype | specify> ( conceptName , conceptName )> .

taskCompositionDefinition ::=
    compose [parallel] listOfTasks ; .

taskRefinementDefinition ::=
    refine listOfTasks ; .

listOfTasks ::= [ taskName <i taskName>* ] .
semanticToken ::= conceptName | taskName .
listOfSemanticTokens ::= semanticToken <i semanticToken> .

```

%% DEFINITION OF HEURISTIC MODEL

```

heuristicModelDefinition ::=
    heuristicModel
        listOfHeuristics
    endHeuristicModel .

listOfHeuristics ::= < heuristicDefinition >* .
heuristicDefinition ::=
    heuristic heuristicName ;
    text heuristicTextDefinition ;
    [ rule heuristicRuleDefinition ; ]
    [ link heuristicLinkDefinition ; ] .

heuristicTextDefinition ::= 'heuristicDescription' .
heuristicRuleDefinition ::= ruleCondition <or ruleCondition>* => ruleAction .
heuristicLinkDefinition ::= heuristicName <i heuristicName>* .

ruleCondition ::= 'conditionStatement' <and 'conditionStatement'>* .
ruleAction ::= taskFunctionDefinition | conceptFunctionDefinition .
heuristicName ::= semanticToken .

```

Figure 8.10 BNF Definition of MSL

8.5 PROLOG CLAUSE FORMAT

This section presents a complete set of Prolog clauses corresponding to the MSL grammar described in the previous section. The formats of clauses in each model are shown with respect to their OMT statements.

In the **product model**, there are six types of clauses:

```
concept(ConceptName, ListOfSuperConceptNames, Concrete) .
property(ConceptName, PropertyName, CardinalityList) .
source(LinkName, ListOfSourceConcepts, CardinalityList) .
target(LinkName, ListOfTargetConcepts, CardinalityList) .
reference(ConceptName1, ConceptName2, CardinalityList) .
dissection(FragmentName1, FragmentName2, ListOfConceptNames) .
```

The examples of OMT MSL statements are:

```
concept dataFlowDiagram ;
  subtypeOf diagram ;
  property actor [1,1,0,n] ;
  property dataStore [1,1,0,n] ;
  property process [1,1,1,n] ;
concept dataFlow ;
  subtypeOf link ;
  source actor, dataStore, process [0,n,0,1] ;
  target actor, dataStore, process [0,n,0,1] ;
  property data [1,1,1,n] ;
reference actor object [0,1,1,1] ;
```

and the converted Prolog clauses are as follow:

```
concept(dataFlowDiagram, [diagram], concrete) .
property(dataFlowDiagram, actor, [1,1,0,n]) .
property(dataFlowDiagram, dataStore, [1,1,0,n]) .
property(dataFlowDiagram, process, [1,1,1,n]) .

concept(dataFlow, [link], concrete) .
property(dataFlow, data, [1,1,1,n]) .
source(dataFlow, [actor, dataStore, process], [0,n,0,1]) .
target(dataFlow, [actor, dataStore, process], [0,n,0,1]) .
reference(actor, object, [0,1,1,1]) .
```

There are a few points to note about the product model:

- The group concept and the link concept share the common *source* and *target* clauses presentation (the host being the source and the element being the target). The concept type is distinguished by the *superconcept* field of the *concept* definition.

```
concept concurrentSubdiagram ;
  subtypeOf group ;           ⇒ concept(concurrentSubdiagram, [group], concrete)
  host state [0,1,1,1] ;      source(concurrentSubdiagram, [state], [0,1,1,1]) .
  element stateDiagram [0,1,1,n] ; target(concurrentSubdiagram, [statediagram], [0,1,1,n]) .
```

- The statement with *inclusion* symbol is expanded to represent all possible outcomes. For instance the *role* property in *association* concept is converted into two clauses:

```
concept association ;
...                               ⇒ property(association, sourceRole, [1,1,0,1]) .
property role # [1,1,0,1] ;      property(association, targetRole, [1,1,0,1]) .
```

- The statement with exclusion symbol is converted to a constraint rule:

```
concept transition ;
...
property event ! [0,1,0,1] ;
```

⇒ %% in the heuristic rule field of event
not(owner(automaticTransition)) .

- The statement with overriding is represented by all possible combinations. For instance, the target clauses of *instantiation* overrides the original *relationship* clause:

```
concept instantiation ;
subtypeOf relationship ;
target instance [0,n,1,1] ;
```

concept(instantiation, [relationship], concrete) .
⇒ source(relationship, [object], [0,n,1,1]) .
target(instantiation, [instance], [0,n,1,1]) .

In the **process model**, there are four types of clauses:

```
task(TaskName, TaskFunction, Precondition, Postcondition) .
compose(TaskName, ListOfTaskName) .
refine(TaskName, ListOfTaskName) .
parallel(ListOfTaskName) .
```

The examples of OMT MSL statements are:

```
task objectModelling perform(objectModelling) ;
precond [problemStatement] ;
postcond [objectModel] ;
compose [identifyClass, identifyAssociation, identifyAttribute, organiseInheritance,
testAccessPath, verifyObjectModel, groupClassIntoModule] ;
```

and the converted Prolog clauses are as follows:

```
task(objectModelling, perform(objectModelling), [problemStatement], [objectModel]) .
compose(objectModelling, [identifyClass, identifyAssociation, identifyAttribute,
organiseInheritance, testAccessPath, verifyObjectModel, groupClassIntoModule]) .
```

- The heuristic name may be different from the task name of the same clause.

In the **heuristic model**, there are only two types of clauses:

```
heuristic(HeuristicName, HeuristicText, HeuristicLink) .
rule(HeuristicName, ConditionStatement, ActionStatement) .
```

The examples of OMT MSL statements are:

```
heuristic deleteIrrelevantClass ;
text 'If a class has little or nothing to do with the problem, it should be eliminated.' ;
rule 'A class has little or nothing to do with the problem' => delete(class) ;
link class, object ;
```

and the converted Prolog clauses are as follows:

```
heuristic(deleteIrrelevantClass,
'If a class has little or nothing to do with the problem, it should be eliminated.',
[class, object]) .
rule(deleteIrrelevantClass,
'A class has little or nothing to do with the problem', delete(class)) .
```

- It must be noted that other heuristic links may be added to the resultant list of heuristic links by the language compiler.
- At the moment, both the condition and action field of a heuristic rule accept either a textual statement or a task function.

8.6 CONCLUSION

To conclude, the chapter discusses some additional modelling considerations to enhance the method representation. GMR formulates a common representation for all semantics, including concept, task and heuristic. Both the MSL grammar and the Prolog clause format of GMR are presented with illustrations.

9. METHOD EVALUATION

In the last chapter, five chosen methods were used to demonstrate the generic model. That exercise provided an opportunity to evaluate various significant aspects of our meta modelling technique. This chapter discusses the method evaluation with special focus on method comparison, fragment dissection and selection of methods.

9.1 INTRODUCTION

There is in the literature no established way of deriving a model based on capturing method techniques rather than specific methods. In this research we propose such a model and try it out on five SDMs to demonstrate the expressiveness of method representation. Some significant remarks are made about the meta model in this chapter.

The GMR is not just a generic model for method representation, it gives a better channel to compare the strengths and weaknesses of SDMs. The model formulates a common notion of method product and method process, which results in ease of dissecting tool fragment. This gives the possibility of multi-view specifications or method integration. The techniques of method representation, comparison and fragment dissection are described in this chapter. In addition, selection of method is not only based on method capabilities but most importantly on the experience of the user and external considerations.

The next section presents details of the technique for fragment dissection. Section 9.3 shows the advantage of comparing software methods by this meta modelling technique. Section 9.4 gives a discussion about selecting suitable methods.

9.2 FRAGMENT DISSECTION

Having defined a uniform method representation, a single non-redundant semantic knowledge base can be produced. It is the sole repository of all development information. Thus a CASE tool can be constructed to permit multiple alternative perspectives on development approaches or to enable alternative focuses [Short 91]. The principle is known as method integration. It is a collection of techniques and tools that enable different system development perspectives to be used in a consistent manner. There are three main advantages of method integration:

- It enables multiple views on a problem space. Software systems can be developed with a consistent approach even if they contain components that use different technologies. An example is that of the object structure in a lift control system which may be viewed in an

OOA diagram, whereas the process structure may be best viewed through the dataflow diagram in structural analysis.

- It allows various methods to be used at different life cycle stages. This is important where a customer demands a certain software development approach. For instance, a requirement language is requested in an aircraft project during analysis, but HOOD and Ada must be used for design and implementation.
- It provides method evolution in software development. Newer approaches are often restrained due to the difficulty of changing culture of software developers or the transition from one development technology to another. For example, the benefits of object orientation could be introduced gradually to a team of trained analysts to produce knowledge based systems that work on extracts from operational database systems.

Two key points need to be achieved to support method integration. Firstly, a common meta model for an integrated CASE tool that can support multiple perspectives must be defined. This is, of course, one of the major objectives of the generic SDM representation in this research, and such a model has been described in chapter five to seven. Secondly, the method concepts or components necessary to allow multiple perspectives must be identified and dissected out as a discrete tool from the SDM. In meta modelling, such components are the constituents of methods that provide the integration and they are normally known as a tool fragment or as a method fragment.

Our meta model is designed with special care, to enable fragment dissection, that is to help us 'cut out' tool fragments from a method. Each fragment comprises of two components; the product to describe method concepts and the process to develop method tasks. Hence it is important to dissect the fragment products as well as the fragment processes. The different techniques in product dissection and process dissection are presented in the following two subsections.

9.2.1 PRODUCT DISSECTION

A method is described in terms of a number of tool fragments. Though these fragments may have different optionality or multiplicity in that method, they can be dissected as discrete constituents to support multiple viewpoints across methods in a particular application. Such tool fragments are normally interrelated by shared concepts or by intermediate relationships. These items must be determined to fully dissect a fragment out of the method context. Figure 9.1 depicts the possible relationships between fragments. The diagram shows a method with two fragments, namely *fragment A* and *fragment B*. For simplicity, the details within each fragment are not shown, however inside the *fragment A* box, the five possible relationships among concepts are given.

The three notations between the two fragments depict the relationships between method fragments. They are the dissection set, referencing relationship and grouping relationship, which are known as the ‘cuts’ between the fragments.

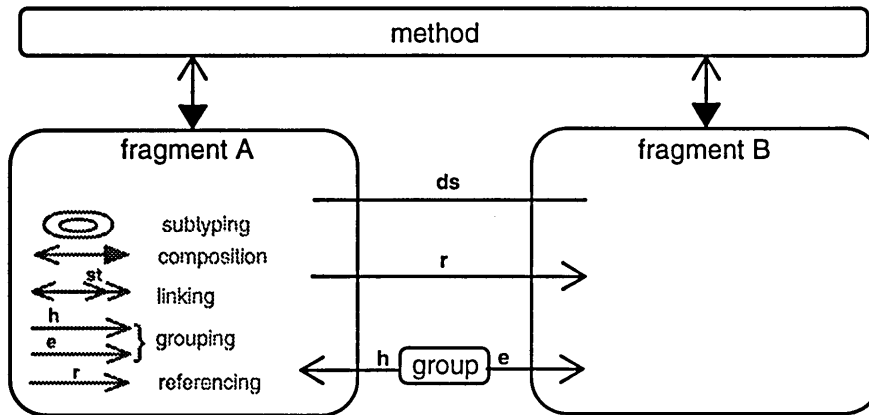


Figure 9.1 Method Product Dissection

The first type is described by a dissection set as introduced in section 8.3.1. It is an important tool to dissect fragments based on the common concepts. These concepts can be either a link concept or an entity concept, as demonstrated by the Codarts/DA concepts *dataFlow* and *dataTransformation* respectively in figure 8.2. The other usual concept in a dissection set is the property concept; that is a concept that has owners from different fragments of a method. For instance, in HOOD a *modeOfFlow* can be a property of a *dataFlow* in a *hoodDigram* fragment or a *parameter* in an *objectSkeletonDescription* fragment as shown in figure 8.9. Hence *modeOfFlow* is an element in the dissection set between the two fragments. This type of dissection relationship also handles the shared concepts. A shared concept is a method concept described with different levels of detail or viewpoints in different fragments, and therefore it can be considered as a special type of concept in a dissection set. The *dataTransformation* concept in Codarts/DA mentioned above is a typical illustration, see section 9.3.2 for details.

There are two ways to dissect fragments by referencing relationships. In the first case, it relates concepts with equivalent semantic meaning of fragments in the same method. For example, in OMT an *activity* of a *state* in *dynamicModel* must refer to an *operation* of an *object* in *objectModel*. This ‘refer-to’ relationship links up entity concepts in different fragments. The other case describes the relationship between two concepts, where one concept is further defined by another concept. For instance, an *object* concept in *objectModel* is described by another fragment called *dataDictionary*. This ‘defined-by’ relationship associates an entity concept with another fragment concept. These referencing relationships form the second type of cuts in product dissection.

Figure 9.2 shows the OMT product dissections. OMT has three tool fragments, which are shown as dotted rectangles in the diagram. All the dissection cuts are uni-directional references running towards the entity concepts in *objectModel*. Hence it is clearly seen that the *dynamicModel* and *functionalModel* describe certain viewpoints of the system and the *objectModel* is proved to be the most important fragment among the three.

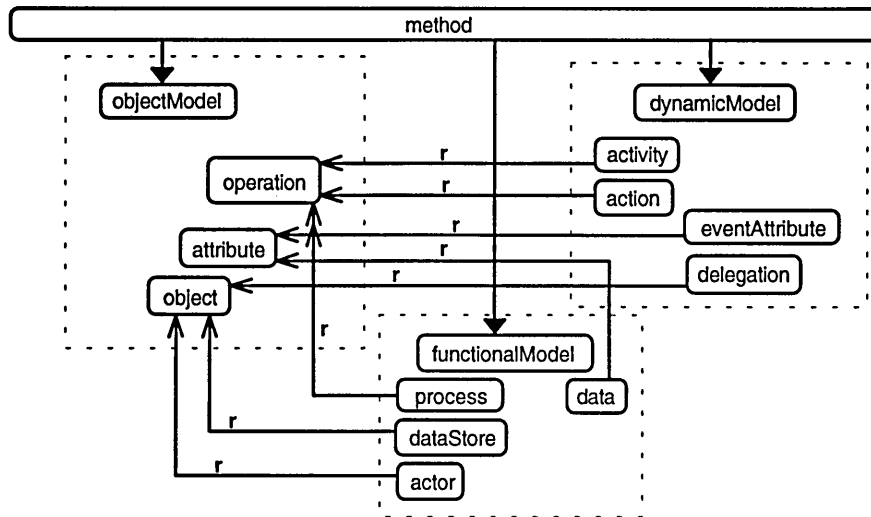


Figure 9.2 OMT: Product Fragment Dissection

The last type of product dissection is denoted by grouping relationships. In dissection, a group concept is normally a decomposition concept or an extension concept of an entity concept. In other words, the host concept is an entity described in one fragment and the element concept is the expanded fragment concept. This product dissection is illustrated by the Ptech method as shown in figure 9.3. In the figure, *productDecomposition* group concept identifies the cut between *objectSchema* and the owner of the *product* concept which is *objectFlowDiagram*, whereas the *activityDecomposition* group concept determines the cut between *eventSchema* and the owner of the *activity* concept which is again the *objectFlowDiagram* fragment.

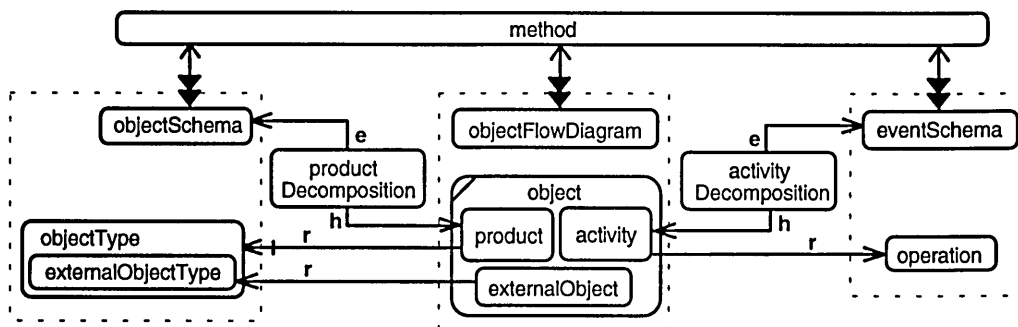


Figure 9.3 Ptech: Product Fragment Dissection

The figure also demonstrates dissection cuts amongst the tool fragments as referencing relationships. Each *activity* in *objectFlowDiagram* refers to an *operation* in *eventSchema*, and a *product* denotes an *objectType* in *objectSchema*, whereas an *externalObject* can only relate to an *externalObjectType*. In conclude to this product dissection layout, *objectFlowDiagram* provides an overview of the system.

The three main points in determining fragment cuts are:

- For any dissection set, a number of concepts are common among the fragments.
- For any referencing relationship, the two related concepts must be in different fragments.
- For any grouping relationship, the host concept is an entity concept and the element concept is a different fragment concept.

The following points are noted to explain why the other relationships cannot form a cut:

- For any subtyping relationship, both the superconcept and subconcept semantically and notationally belong to the same tool fragment. However, if the subtyping relationship belongs to fragment concepts (for instance *systemContextDiagram* is a subtype of *contextDiagram* in Codarts/DA), the situation is more complicated. The complete fragment dissection requires the handling of inherited features from the superconcepts.
- By definition, when a concept owns another concept both of them are in the same tool fragment. Thus composition is an intra-fragment relationship. Nevertheless there is one exception, that is the *method* aggregate fragment. It comprises of a number of fragment concepts, so the composition relationships between *method* concept and any fragment concepts are definite cuts in product dissection.
- For any linking relationship, the link concept is used to associate the source and target entity concepts in the same fragment, so the link concept is also semantically owned by the fragment concept. Hence linking is also an intra-fragment relationship and cannot represent a cut for fragment dissection.

9.2.2 PROCESS DISSECTION

The last subsection shows that the fragment product can be dissected easily by determining a small set of concept relationships between fragments. However the fragment process must also be dissected to enhance the formation of tool fragments cut out by the product dissection. Unlike the fragment product, the fragment process can be scattered through the method process. And since the tasks in the method process may interrelate to one another, the process dissection is more difficult than product dissection. The dissection also has to consider the input and output information that may occur within the tasks, such that the relationships between fragment processes are identified.

Nevertheless there are some methods that have very clear cuts in a fragment process. The three diagrams in figure 9.4 illustrate the three method process models where the fragment dissections are relatively simplistic. It means that the process of each fragment is presented as individual tasks. These method processes are HOOD, OMT and Ptech respectively. For the benefit of explanation, the diagrams only show the top level tasks and all concept flows among these tasks are omitted. It is because the only dependence is the concept tokens produced by the preceding task.

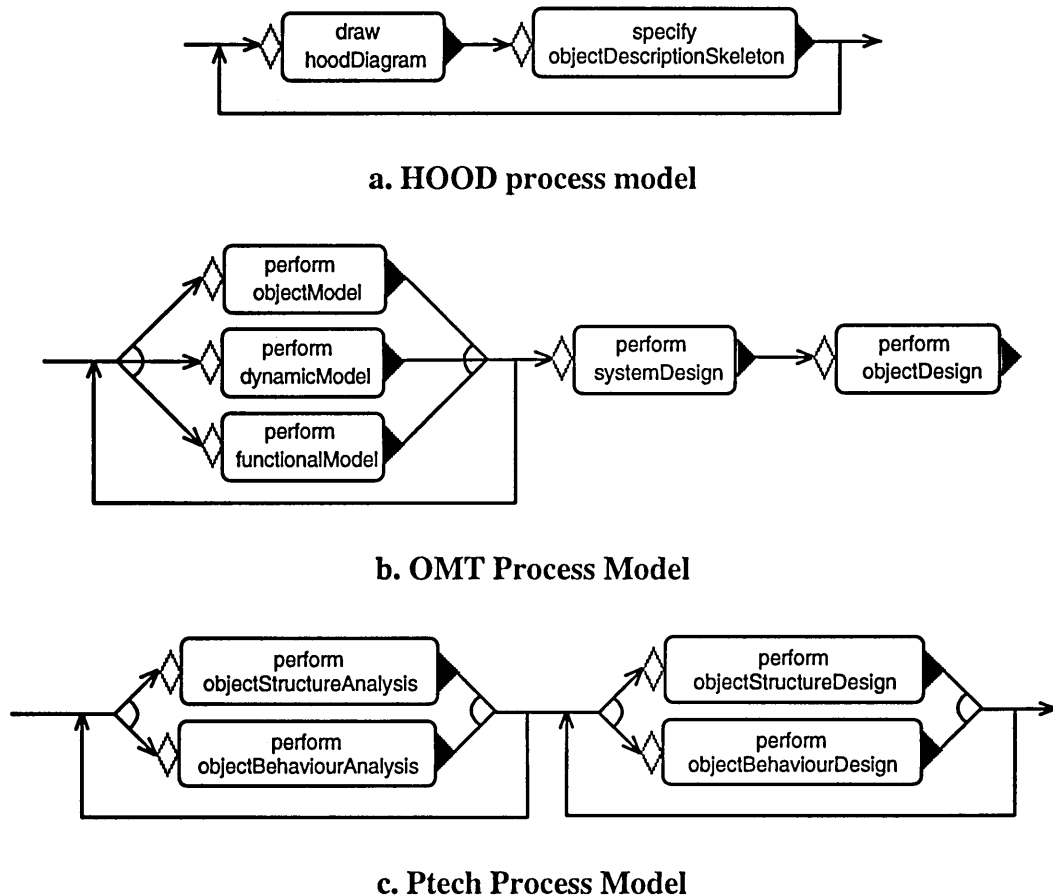


Figure 9.4 Simple Process Dissection

HOOD has only two fragments. It is more of an incremental, iterative process. The two main tasks are *draw(hoodDiagram)* and *specify(objectDescriptionSkeleton)* as shown in figure 9.4a. The former task consists of identifying *objects*, *operations* and *relationships* among them, this information is shown by drawing the *hoodDiagram*. The latter task takes the *hoodDiagram* and maps down to the slots in the *objectDescriptionSkeleton*. Then the developer has to specify the further details of the object, such as data types, pseudo-codes etc. These two steps recur continually until all the objects are specified.

OMT has very clearly defined each fragment process. The three analytical models can be determined in parallel regardless of the order. It then proceeds with the *systemDesign* and finally follows with the *objectDesign* as depicted in figure 9.4b. Each of these fragments can be dissected out. For instance, the *dynamicModel* is independent of the other models, though there are a few reference links to the *objectModel*. These uni-directional links are also present in the fragment product as referencing relationships. These links are not used as requirements for other fragment processes, but a mapping process is needed to connect the related software concepts after both models have been declared.

Ptech describes information engineering as two sides of a pyramid, one side concerned with the data in the business area and the other side concerned with the functions in the model¹. Ptech applies *objectStructureAnalysis* and *objectBehaviourAnalysis* at various levels of the pyramid, whereas the *objectStructureDesign* and *objectBehaviourDesign* come at a later stage. Thus the method can be denoted as a two stage process as shown in figure 9.4c. The *objectSchema* fragment and the *eventSchema* fragment are determined by the *objectStructureAnalysis* and the *objectBehaviourAnalysis* respectively.

Some fragment processes are difficult to dissect from the original development method, such as when a process spreads over a number of tasks or a task comprises the construction of a few fragments. Before introducing the new dissection technique, let us recall the aim and definition of a method fragment.

To support method integration, a method fragment must be an autonomous tool or technique comprised of its own concept structure, task structure and heuristic information. Both product and process models of the fragment must be self governing or unconstrained, although they may have a limited interface to the external development environment.

In a fragment product this interface is formed by a set of concept relationships, whereas in a fragment process this interface composes of two lists of concept tokens. One list denotes the required concept tokens for the process and the other list describes the concept tokens produced by the process.

For example, the Codarts/DA process model comprises of seven steps as shown in figure 9.5². The *perform(cobra)* task identifies *objects*, *functions* and determines graphical fragments such as *systemContextDiagram* and some *contextDiagrams*, *controlFlowDataFlowDiagrams* and *stateTransitionDiagrams*. The succeeding task *perform(distributedSystem)* deals with *systemDecomposition*, *subsystemDecomposition* and *systemConfiguration*. It also identifies a list of *subsystems*.

¹ Ptech uses the term 'methods'. We change it to 'functions' in order to avoid unnecessary confusion.

² To simplify the task diagram for illustration, a number of concept tokens have been deliberately missed out.

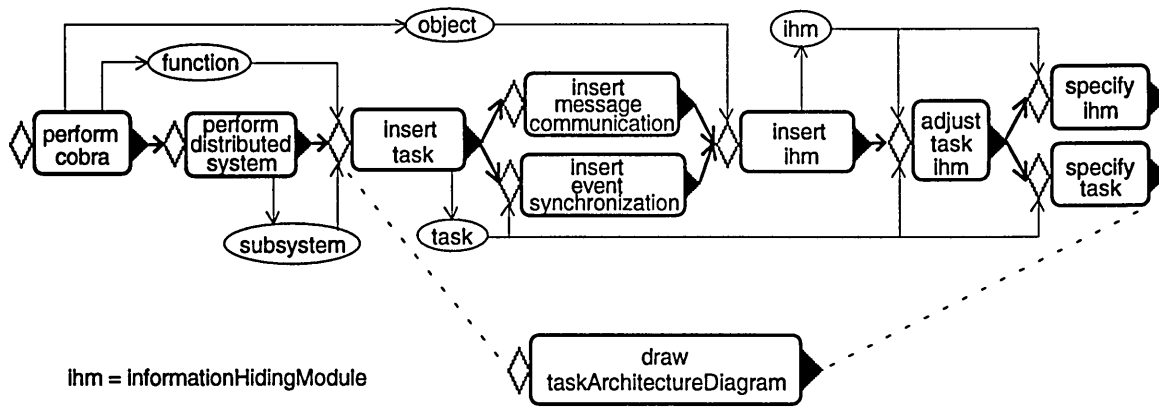


Figure 9.5 Codarts/DA: Process Model

After that, the five following steps concentrate on task structuring and information hiding module structuring. All these steps contribute to the formation of *taskArchitectureDiagram*. Therefore these tasks can be composed into a pseudo-task *draw(taskArchitectureDiagram)*, as shown by the dotted lines in the figure. The effect is known as **task composition** (the reverse of task decomposition described in section 6.7.4). The task composed is called a **composite task**. Although the *draw* function is used for the composite task in this example, the *perform* function is also common in task composition. The subsequent step is to determine the precondition and postcondition of the task, i.e. the concept token inflows and outflows. The basic guidelines to identify these concept tokens are recorded as follows:

- The precondition comprises concept tokens that are required by each internal task minus those tokens produced by themselves.
- The postcondition consists of concept tokens that are produced by each internal tasks minus the precondition of the composite task.

In the example, the required tokens form the list $\{function, object, subsystem, task, ihm\}$. Since both *task* and *ihm* tokens are produced by internal tasks, the overall precondition of the task is the list $\{function, object, subsystem\}$. Thus the tokens produced from the composite task are $\{task, ihm, taskArchitectureDiagram\}$.

Task composition is a technique which may not truly represent the method process, but it is definitely important in process dissection. Figure 9.6 shows an example of Booch OOD. All four design steps described by Booch can be composed into two pseudo tasks *draw(classDiagram)* and *draw(objectDiagram)*. As mentioned in chapter six, a coarse grain method process should allow user defined tasks, such as the *draw(stateTransitionDiagram)* task and the *draw(timingDiagram)* task which can be developed out of *insert(semantic)* task. In this case, the two new tasks can be treated as subtasks of the composite tasks, and the corresponding task information should be omitted in fragment dissection.

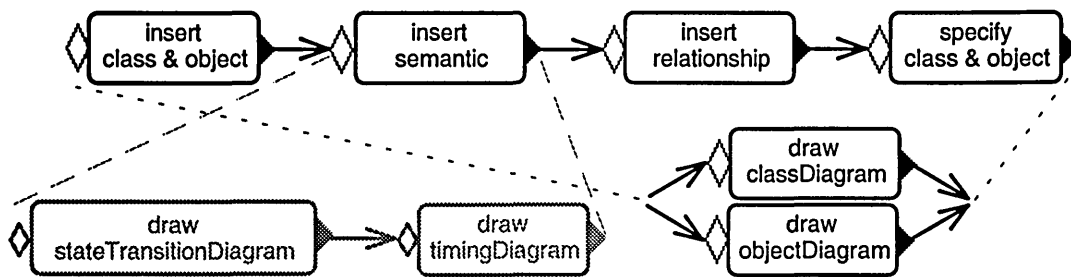


Figure 9.6 Booch OOD: Process Model

The third constituent of a tool fragment is the heuristic guidance, which is documented by the heuristic specification. Since each concept or task predicate has a reference pointer to the corresponding heuristic clause, they can be stored together as a module. Thus no formal dissection is required for the method heuristic.

9.3 METHOD COMPARISON

One of the main advantages of our generic method representation is to provide a formal channel to compare methods. The uniformity enforced by the meta model, the products and processes of SDMs, is ideal for method comparison. A method engineer can use the result of comparison to choose a method or a tool fragment to suit their particular problem domain and environment. Thus, method comparison also can support the integration of methods.

Some method comparisons are made on the basis of adaptability towards a specific application. For instance, the evaluation of object-oriented analysis and design methods in [Cribbs 92] aims to quantify the strength and weaknesses of eight OOA/D methods that best meet the needs of Alcatel Network System's development organisations. This evaluation is beneficial to the organisation as well as to the developers who work in a similar problem domain or environment. Nevertheless, this technique biases towards a region of software applications or certain requirements in the system. The result is far from satisfactory.

In contrast, some method comparisons are based on a set of predetermined concepts or techniques. These are normally found in the literature introducing a method. For instance [Gomaa 93] introduces Codarts/DA and the method is compared with five other SDMs. In order to show the excellence of the method, a 'tailor-made' example is formulated, i.e. the *Cruise Control and Monitoring System*, with all distinct concepts in Codarts/DA. Therefore the described method stands out from the other SDMs. This style of 'method comparison' is, of course, ideal for illustrating some new technologies. In practice, some concepts in the compared SDMs are hidden by the comparison, so it is not a 'true' comparison.

In the following subsections, two types of 'true' method comparisons are introduced. They are known as **numerical comparison** and **fragment comparison**. These techniques are

direct outcomes of the uniformity of GMR. We also show the results of comparing the five chosen methods based on these techniques.

9.3.1 NUMERICAL COMPARISON

This comparison is self explanatory. Numerical comparison is based on the number of fragments, products, processes and heuristics, to compare the strength and weaknesses between methods or fragments in a method or modelling aspects within a method. For method product, the comparison uses the number of concepts and relationships. For method process, the comparison uses the number of tasks, triggers and concept flows. Numerical comparison provides general information of a method which can be used as a guide when choosing a suitable method in a specific application or a tool fragment in method integration. In addition, it compares the process models and heuristics amongst SDMs. Most method comparisons or method evaluations fail to do this.

There are various ways to obtain numerical comparisons, the following list suggests some meaningful choices:

- **Number of fragments / concepts / tasks / heuristics in a method** gives general information about the method. This information provides the influence of a method such as its significant perspective in describing a particular method product or introducing a particular method process.
- **Number of concepts / tasks / heuristics per fragment or in a specific fragment** gives the weight of the fragment from a focal point of view. If a particular fragment of the method consists of numerous concepts or heuristics, the emphasis of the method may lay on that fragment. For instance, the main focus of Codarts/DA is to construct the *taskArchitectureDiagram*. This information can be also given in percentage form, i.e. the percentage of concepts in a method which uses to describe a particular fragment.
- **Number of concepts / tasks / heuristics in data model / function model / state model** illustrates the strength of describing such a model by the method. It allows one to show whether the method has an inclination towards any of the three basic modelling techniques in software development. Again, this information can be recorded by percentage to compare various models.
- **Number of relationships per concept in a fragment or number of triggers per task in a fragment** shows how strong the entities in a fragment are related. This is also known as fragment cohesion. It provides the information about the complexity of the respective concept or task in a fragment.

- **Number of relationships between fragments in a method product** describes how strongly the fragments in a method are related. This is known as the fragment coupling, it shows the ease of fragment dissection.
- **Number of top level tasks** gives a brief overview of the high level breakdown in the process model of a method.
- **Decomposition factor of process model** gives the total number of composite tasks (i.e. *perform* and *do* functions) in the method. This factor shows the weighting of the complex components in the method (on the other hand it reflects the number of terminal tasks). This information can be recorded by percentage of tasks in the method.
- **Number of heuristics in method product / method process** shows the overall weighting of the components, which can be used to compare methods. This can also be denoted as a percentage to compare between the method product and the method process.

	Booch OOD	Codarts/DA	HOOD	OMT	Ptech
number of fragments	6	4	2	3	3
number of concepts	120	78	52	81	69
data model	77.0 (64%)	-	-	45.0 (55%)	28.5 (42%)
function model	32.0 (27%)	8.5 (19%)	-	11.0 (14%)	27.5 (40%)
state model	11.0 (9%)	9.0 (20%)	-	25.0 (31%)	27.5 (40%)
fragment cohesion	1.36	1.30	1.51	1.28	1.47
function cohesion	1.10	1.35	-	1.91	1.58
state cohesion	1.48	1.11	-	1.60	1.33
fragment coupling	5.33	2.75	6.00	2.67	1.67
number of tasks	11	36	13	71	27
top level tasks	4	14	4	4	4
composite tasks	11 (100%)	7 (22%)	5 (38%)	19 (27%)	11 (41%)
number of heuristics	131	123	65	126	96
number per fragment	21.8	30.75	32.5	42.0	32.0
% in product model	91.6%	63.4%	80.0%	50.0%	71.9%
% in process model	8.4%	36.6%	20.0%	50.0%	28.1%

Table 9.1 Numerical Comparison of the Five Chosen Methods

Table 9.1 illustrates a number of general numerical comparisons on the five chosen SDMs. The following remarks and evaluations are drawn from the comparisons:

- Booch OOD has six fragments (*class diagram, object diagram, state transition diagram, timing diagram, module diagram* and *process diagram*). Codarts/DA has four fragments (*context diagram, data flow/control flow diagram, state transition diagram* and *task architecture diagram*). HOOD has two fragments (*HOOD diagram* and *object description skeleton*). OMT has three fragments (*object model, state model* and *functional model*). Ptech has three fragments (*object flow diagram, object schema* and *event schema*).

- b. The number of concepts per fragment averages around 18 to 28 and the number of concepts in a method mainly depends on the number of fragments.
- c. Codarts/DA does not directly denote a data model, but it does specify concurrent objects as *information hiding modules*. These are addressed in the *task architecture diagram*.
- d. HOOD combines all aspects of software modelling in a *HOOD diagram* and the *object description skeleton* is a textual extension to automate Ada code. Therefore there is no differentiation of various viewpoints in HOOD.
- e. In Ptech, an *object flow diagram* gives an overview of the system, an *object schema* describes the structural aspect and an *event schema* describes the behavioural aspect. The event schema can be considered as a combination of a function model and a state model.
- f. Booch OOD puts nearly two-thirds of concepts in data modelling (class and object diagrams). OMT also emphasises the data model and the percentage for the state model is relatively high. Ptech has well balanced structural and behavioural models.
- g. The product fragment cohesion is in average between 1.10 to 1.90 relationships per concept. This does not give much information. However the product fragment coupling, which is the number of relationships between fragments, shows that Booch OOD (5.33) and HOOD (6.00) are highly coupled-fragment methods. Fragment dissection seems uneasy with these two methods.
- h. Both Booch OOD gives very brief process model descriptions, as all the tasks denoted are composite tasks (i.e. 100% in the decomposition factor). In other words, there is no terminal task function explicitly defined by this method.
- i. Codarts/DA and OMT give a general outlines of each task in software development. A fair amount of refining steps are described as terminal task functions in the method.
- j. Ptech only provides a very detailed process of the *object behaviour analysis* (23 tasks out of the overall 27 tasks), however there are no instructions for the *object structural analysis* or *object structural design* or *object behaviour design* at all.
- k. Booch OOD, Codarts/DA and OMT are heuristic 'rich' methods as they have over 100 heuristics described. Moreover, Booch OOD concentrates on defining its 120 concepts rather than giving guidance on the tasks. HOOD is particularly weak in term of illustrating the method heuristics.
- l. Both Codarts/DA and OMT give a well-balanced heuristics on the product and process models. However, the other three methods seem to describe what are the method concepts rather than to show how to model these concepts by method tasks. The extreme case in the numerical comparison is the Booch OOD which has over 90% of method heuristics concerning the concepts in the product model.

9.3.2 FRAGMENT COMPARISON

Most method engineers find it more useful to compare how the actual concepts in various methods describe a similar viewpoint of a software system. Since the meta model requires all method concepts to be denoted in the concept diagrams, it is very easy to compare methods by viewing their perspectives of certain aspects. This aspect could be a data model, a function model or a state model. These are normally described as tool fragments. Hence this type of comparison is known as fragment comparison. Fragment comparison is a useful tool in method integration because it allows the designer to choose the most suitable fragment amongst the methods available. The table shown can be used as a checklist in the selection.

Table 9.2 demonstrates the fragment comparison based on various state model fragments in different SDMs. A '•' mark in the grid denotes that the corresponding concept is included in that method, and the letter next to the mark refers to the notes presented below.

- a. Software dynamics are modelled by various tool fragments in different SDMs. Booch OOD and Codarts/DA uses *state transition diagram*, OMT uses a *state diagram* in the *dynamic model*, Ptech uses an *event schema*. HOOD describes *control* in an *object control structure*, but this is not classified as a state model so is not shown in the table.
- b. Ptech is the only method that describes an *event type*, which includes *classification events*, *coalesce events*, *creation events*, *declassification events*, *decoalesce events*, *reclassification events*, *termination events* and *tuple-substitution events*.
- c. Different methods denote *process* in different ways: Booch OOD by *action* (same as *operation*, *method* or *message*, which is further described in the *class diagram*), Codarts/DA by *event* to trigger *transformation* (*input event* or *output event*, which is described in the *task architecture diagram*), OMT by *action* or *activity* (*action* includes *entry action*, *exit action*, and *event actions*, which are all referred to as *operation* in *object model*) and finally Ptech by *operation* (or known as *activity* in *object flow diagram*).
- d. A *control condition* in Ptech has a similar effect as *condition* in Codarts/DA and *guard condition* in OMT. However, Ptech describes it graphically whereas Codarts/DA and OMT describe it in textual form.
- e. A *trigger rule* in Ptech has the similar effect as *transition* in Booch OOD, Codarts/DA and OMT, but *trigger rule* is a more powerful tool. It takes the underlying *object(s)* of an *event* and determines those *object(s)* required to invoke an *operation*.
- f. Booch OOD, Codarts/DA and OMT all describe *state*. However Booch OOD and OMT differentiate *start state*, *inter state* and *stop state*. Ptech describes *state* as *event prestate* and *event poststate*.
- g. An internal action in OMT is similar to an internal operation in Codarts/DA.

Concept	Booch OOD	Codarts/DA	OMT	Ptech
<i>action</i>	•c		•c	
<i>activity</i>			•c	
<i>classification event</i>				•b
<i>clock event type</i>				•
<i>coalesce event</i>				•b
<i>condition</i>		•d	•d	
<i>control condition</i>				•d
<i>creation event</i>				•b
<i>declassification event</i>				•b
<i>decoalesce event</i>				•b
<i>delegation</i>			•	
<i>dynamic model</i>			•a	
<i>entry action</i>			•c	
<i>event</i>	•	•	•	•
<i>event action</i>			•c	
<i>event attribute</i>			•	
<i>event generalisation</i>			•	
<i>event partition</i>				•
<i>event poststate</i>				•f
<i>event prestate</i>				•f
<i>event schema</i>				•a
<i>event state</i>				•f
<i>event subtyping</i>				•
<i>event trace</i>			•	
<i>event type</i>				•
<i>exit action</i>			•c	
<i>external operation</i>				•
<i>input event</i>		•		
<i>inter state</i>	•f		•f	
<i>internal action</i>			•g	
<i>internal operation</i>				•g
<i>merging control</i>			•	
<i>nesting event schema</i>				•
<i>nesting state diagram</i>			•	
<i>operation</i>				•c
<i>operation subtyping</i>				•
<i>output event</i>		•		
<i>reclassification event</i>				•b
<i>scenario</i>			•	
<i>splitting control</i>			•	
<i>start state</i>	•f		•f	
<i>state</i>	•f	•f	•f	
<i>state aggregation</i>			•	
<i>state generalisation</i>			•	
<i>state transition diagram</i>	•a	•a		
<i>stop state</i>	•f		•f	
<i>termination event</i>				•b
<i>transition</i>	•e	•e	•e	
<i>trigger rule</i>				•e
<i>tuple substitution event</i>				•b

Table 9.2 Fragment Comparison based on State Modelling

9.4 SELECTION OF METHOD

Designing effective software specifications is important for developing high quality software since specification and design phases are the early steps in the software development process. To support the software development many excellent SDMs, such as structural analysis & design methods and object-oriented analysis & design methods, have been developed. A good method should present a product model and a process model concisely and precisely. Although most methods have a detailed description of the product model, they give minimal information about the process model.

There are two problematic situations which can occur in both models. These are the **hidden detail** and the **missing information**. The ‘hidden detail problem’ happens mostly in method products. For instance the OOSA method described in [Shlaer 88] introduces the state model and the process model³ of the software system but gives no formal description about these two models. The notational concept can only be discovered implicitly from the given examples. Some methods have summary pages to present all concepts introduced by the method, which does reduce the possibility of hidden detail. The ‘missing information problem’ occurs in many method processes. Booch OOD only describes four top level design steps, and many tasks are mentioned vaguely in the method, whereas the OOSD method introduced in [Wasserman 90] is a method with no process model at all. The solution is to allow developers to customise their own concepts and tasks in the ‘unfinished’ method. Creativity and common sense are required. The integrated CASE tool must also provide facilities to handle this information correctly.

On the other hand, some SDMs demand that every individual detail is described and that all supporting fragments must be relatively complete. This approach is required for consistency checking or software automation. For example, HOOD is a design method that assists software modelling by developing *hoodDiagrams*. The description of each object is enhanced by the *objectDescriptionSkeleton*, which is used for Ada code generation.

In general, among the five methods investigated, OMT and HOOD seem to be well structured methods: the method product, method process and heuristics are all well-defined.

However, to choose a suitable method for an application in a restrained problem domain and/or environment does not only depend on the capabilities of the method. Other issues have to be taken into consideration. Some of these are listed as follows:

- The most dominant point in choosing a method is the resource’s availability. This resource may include human-based resources, software-based resources and hardware-

³ In OOSA, state model is depicted by state transition diagram and process model is shown as dataflow diagram. Hence the process model mentioned here refers to the software process of a system, and not the method process in meta modelling.

based resources. The experience of the development team is a vital factor in choosing a method, since the initial overhead of adopting a new method is often large. The availability of programming languages or access to CASE tools and environments are also important points to consider.

- The next point is the ease of use and the ease of learning a method. In order to use a capable software method efficiently, sufficient education or training must be provided. Learning by working is also a possible way of adopting new technology, but this adds to the cost of the development process. Sometimes the price of a CASE tool is more significant than the suitability of the underneath method.
- Often, the golden rule is 'simple is the best'. A simple method may not have comprehensive concepts for general development purposes, but it may be suitable to serve the need of certain projects. A complicated method may provide more features to denote software development, but usually there is a higher cost to pay.
- Finally, the suitability of a method depends on the amount (or percentage) of concepts, tasks and heuristics matching the requirement of the system. If there is no complete match, a best-fit algorithm may have to evaluate the suitability of various methods available. The weight of each element is given to show the priority in selecting methods and a scoring mechanism has to be invented. In this case, GMR can be utilised.

9.5 CONCLUSION

This chapter describes the threefold benefits of GMR. Firstly, it provides a generic model to document SDMs formally and systematically. In view of developing an automated SDM system, the representation is designed in an executable form for future extension. Secondly, GMR is also developed with method integration in mind. This allows method fragments to be shared, since dissection of these fragments can be easily worked out using this model. Finally, due to the uniformity of representation, GMR provides a better channel to compare methods or to choose appropriate fragments based on concepts.

10. KNOWLEDGE ACQUISITION OF METHOD MODELS

The development of knowledge based systems is difficult and time consuming. The acquisition of the knowledge necessary to present a certain method's semantics is considered to be one of the main bottlenecks in knowledge engineering. This chapter clarifies some of these difficulties. It then demonstrates how method knowledge can be extracted from a distinct set of acquisition media. The techniques include inspecting, fabricating and verifying (IFV) method models. An illustration of knowledge elicitation of OMT is presented.

10.1 INTRODUCTION

Two major modelling methods have developed in knowledge acquisition research: the Personal Construct Psychology (PCP) method [Kelly 55] [Shaw 93] and the KADS¹ method [Schreiber 93] [Tansley 93]. The PCP approach focuses on the derivation of the formal modelling framework for knowledge engineering and the translation of this framework into a formal knowledge base. The KADS approach focuses on the formal modelling framework and its translation into a computational knowledge base. KADS is intrinsically a modelling approach with a series of seven models: the organisational model, the application model, the task model², the model of cooperation, the model of expertise, the conceptual model and the design model [Schreiber 93]. The four knowledge levels are the domain level, the inference level, the task level and the strategic level (see appendix C for a brief description of KADS).

This chapter does not intend to give a complete description of the process of knowledge engineering. Instead, our major interest is in the modelling of expertise in the domain level of knowledge. The aim of this research is to investigate a generic representation for software development methods, although the ultimate goal of meta modelling may lead to the implementation of tools to manipulate the semantic knowledge base.

¹ KADS is an abbreviation that has lost its original interpretation. In [Schreiber 93], it stands for *Knowledge Analysis and Documentation System*, whereas in [Tansley 93], it means *KBS Analysis and Design Support*. The former emphasises the principled approach to model expertise and the latter stresses the various steps of knowledge-based systems analysis and design. Nevertheless, the common intention of KADS is a set of techniques to assist the knowledge engineering aspect of system development.

² The task model in KADS (or MIKE) must not be confused with the task diagram in our process model (chapter six). The task model carries out a decomposition of basic problem-solving tasks in KBS, whereas our task diagram depicts the development steps of a software method into the task structure.

This chapter concentrates on the knowledge acquisition required to develop the method models. The next section introduces the problems and techniques in knowledge acquisition and expertise transfer. Section 10.3 discusses the three main types of acquisition media in method knowledge acquiring. A detailed practical guide of method knowledge elicitation is given in section 10.4. Section 10.5 sketches the outline of elicitation in method models with illustrations based on OMT. Finally section 10.6 gives a brief summary of the chapter.

10.2 METHOD KNOWLEDGE ACQUISITION

The human activity is not easily accessible to awareness. A study on expertise-transfer processes among scientists reported that some knowledge may not be accessible through the expert: he/she may not be able to express it, but also he/she may not be aware of its significance to his activities [Gaines 87]. This can also occur when acquiring an expert's knowledge about a software development method (hereinafter known as 'method knowledge'). The method knowledge being sought is expertise about denoting system products and processes. The expertise acquired is more formally described than with many other domains. For instance, many software methods adopt structured-text or diagrammatical representations in their formalisation. Knowledge acquisition at this level of abstraction is relatively simple, however, the implicit expertise or critical presumptions involved in software modelling are more difficult to capture.

10.2.1 KNOWLEDGE ACQUISITION PROBLEMS

The main problems in accessing method knowledge are:

- Some software techniques only become apparent under certain conditions. Much modelling experience cannot be expressed easily in speech or literature, but the method knowledge may only be acquired when the situation occurs. For instance, the *softwareArchitectureDiagram* fragment concept in Codarts/DA is a slight variation of the *taskArchitectureDiagram* fragment concept, and it is only illustrated in the Robot Controller System case study at the very end of the book [Gomaa 93, page 352].
- Sometimes the method knowledge is incomplete or incorrect. There can be implicit situational dependencies that make explicit expertise inadequate for performance. This point was discussed in chapters four, five and six. On the other hand, experts may make explicit statements which do not correspond to their actual behaviour, and these lead to incorrect performance. For example, the *startState* of a *stateTransitionDiagram* shown in [Booch 91, page 272] has a number of incoming transitions, which validates the original definition of a *startState* concept.

- Many concepts have presumptions to support the technique, and the expert may not be able to transmit the expertise because he/she is unable to evaluate them. This presumption also appears implicitly to the knowledge engineer. In some situations, the overall process of expertise acquisition is made considerably harder than it ought to be. For instance, the process model of *stateTransitionDiagram* is not clearly defined in Booch OOD: there is an implicit presumption of an *eventTrace* concept in this particular scenario.
- An apprentice may not be able to convert his/her comprehension of a method into a model. Likewise, the method knowledge may not be applicable even when expressed in natural language.

10.2.2 KNOWLEDGE ACQUISITION TECHNIQUES

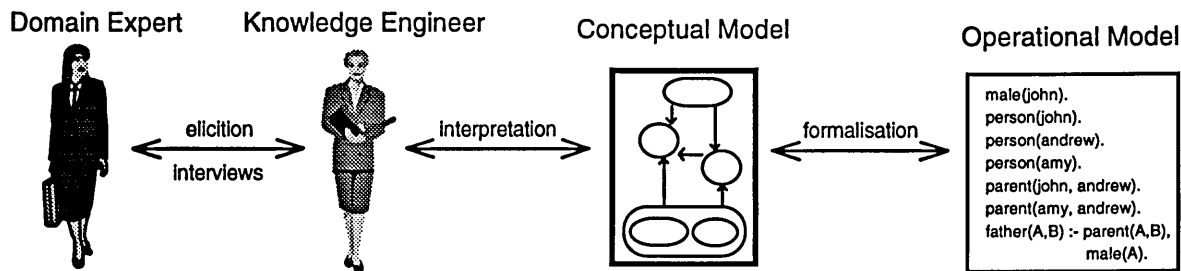
In the development of knowledge engineering methods and rapid prototyping techniques the emphasis has been on knowledge acquisition and hence on the transmission of expertise. A rich variety of techniques are available for method knowledge transfer:

- Expertise may be transmitted through examples. An expert may be able to transmit a skill by showing his own performance without necessarily understanding the basis of his expertise. The expert may not even be aware of the skill required. Through examples, the implicit expertise can be shown clearly to the apprentice. Alternatively, knowledge may be acquired by analogical reasoning. The transfer of models and skills from one situation to another is an important source of knowledge. Most literature about software development methods illustrate their techniques by a number of case studies. For instance, [Booch 91] offers a set of five complete design examples encompassing a diverse selection of problem domains. Each example is illustrated by a different object-based or object-oriented language.
- A trainer may be able to induce expertise by indicating correct and incorrect behaviour without necessarily understanding the skills in detail or himself being expert in its performance. In method knowledge engineering, expertise may also be acquired by comparing the semantics adopted by different methods. The distinctions indicate the significance of the semantics in the method. For instance, *stateGeneralisation* is an important concept in OMT that does not appear in similar fragments of other methods.
- Expertise may also be acquired by the application of general laws and principles to new situations: the use of physical laws and systemic principles to generate specific expertise is the basis of scientific and engineering expertise. This is particularly useful in method knowledge acquisition, since all methods are bound to some design rules or principles.

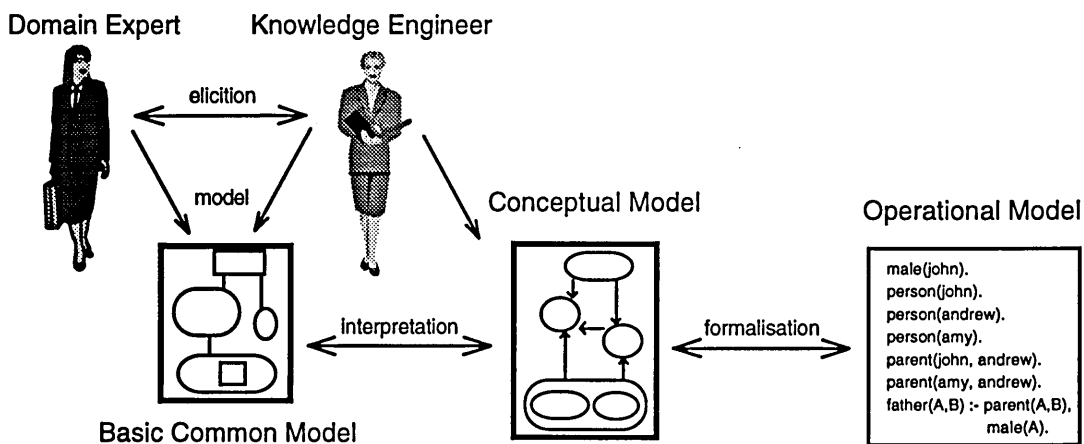
10.3 KNOWLEDGE ACQUISITION MEDIA

Many researchers claim that knowledge acquisition involves at least three activities. Firstly, **knowledge elicitation** obtains an informal model of the knowledge from the domain expert. This is normally carried out by the knowledge engineer interviewing one or more problem domain experts. Secondly, **knowledge interpretation** transforms the elicited data using some conceptual frameworks, such that the model is more applicable for analytical and prototyping purposes. And lastly, **knowledge formalisation** is the stage to conceptualise the model in a form that the program can use the knowledge [Gaines 93] [Neubert 93].

The prevalent argument of expertise-transfer shows that a linear incremental knowledge engineering approach is not a competent approach. As depicted in figure 10.1a, the domain expert has no direct engagement in the knowledge modelling processes. The general approach should be based on a mutual agreement, that is the domain expert should be involved in the construction of a basic domain model with the knowledge engineer, who will further interpret the agreed model into an informal conceptual model of the application. From this model an executable operational model can be obtained. Figure 10.1b illustrates the revised approach.



a. Simple Linear Model



b. Mutual Agreement on the Knowledge Domain

Figure 10.1 General Approach for Knowledge Acquisition

The situation in method modelling is very different. Ideally, the knowledge engineer communicates directly with the domain experts, for example with the method developer(s) or experienced users of one or more specific methods. This way, the implicit knowledge may be identified and the critical presumptions can be clarified. Unfortunately, this is not always the case in method engineering since it is a very difficult task to identify 'experience' method users, and even harder to obtain an interview with a particular method developer. Therefore the method knowledge engineer has to rely on other communication media, such as written or machine-readable materials publicly available. There are, in general, three relevant types of knowledge acquisition media for method knowledge:

- **Literature**, such as Journals and books, is the major source of a method description. Most well-known software development methods have at least one publication either in paper form or compiled into a book. This material gives a comprehensive documentation of the method model. It transforms the abstract software development ideas, such as modelling concepts and experiences, into a more understandable and applicable form when expressed in written language. A book may discuss various methods (such as [Graham 94]), and a method may be described in a number of books (such as OOA in [Coad 90] & [Coad 91]). Different versions of a method may also be described in newer editions of a book. For instance, the Booch method in [Booch 94] is very different from the Booch OOD in [Booch 91].
- **CASE Tools**. Good case tools for a specific method or a group of methods form the second type of knowledge acquisition media. The better developed methods usually have one or more CASE tool(s), for instance Booch OOD, OMT, OOA, Objectory, Ptech etc. These CASE tools demonstrate the software development techniques that accompany the method. Thus, the knowledge engineer can have direct experience of the method by using the tool. However, the choice of tool is important - some CASE tools introduce deviations to the original method. For example, the OOATool dedicated for Coad OOA is a proficient tool, since Peter Coad is one of the consultants in Object International where the tool was developed. Nevertheless, this is normally the most expensive acquisition media among the three types, and it may not be feasible to buy a tool just for the sake of eliciting knowledge of a particular method.
- The last type of knowledge acquisition media is a **Training Course** or a **Tutorial** as provided by academic institutes or commercial industries. There are courses for teaching software developers so that they are familiar with certain SDMs. For instance, the Rational company, where the ROSE tool for Booch OOD was developed, organises seminars for their CASE tool. This type of knowledge acquisition medium can be considered as a secondary expertise consultation.

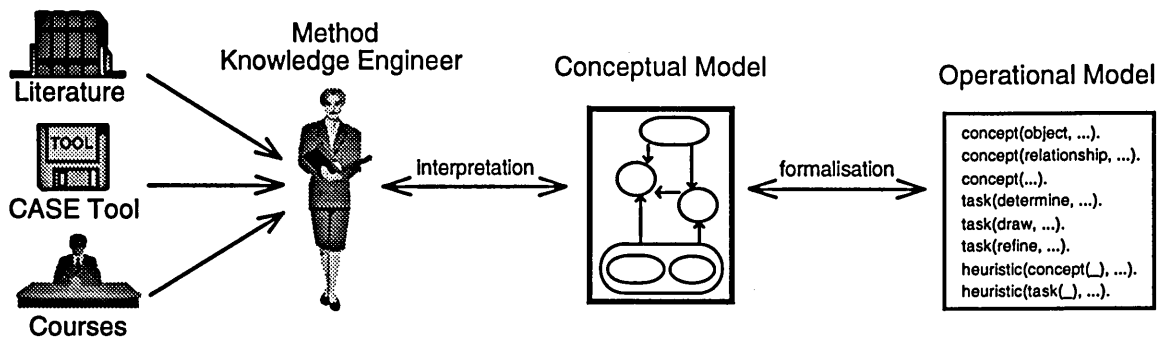


Figure 10.2 Specific Approach for Method Knowledge Acquisition

Figure 10.2 depicts a specific approach for method knowledge acquisition. We refer to these media as method acquisition media, because they embody a different set of knowledge acquisition tools and techniques. The following two subsections present the pros and cons of using these method acquisition media in obtaining method knowledge.

10.3.1 ADVANTAGES OF METHOD ACQUISITION MEDIA

The major problem in knowledge acquisition is eliciting expertise through interviewing domain experts. An expert may not be able to transmit the knowledge explicitly because he/she is unable to express it verbally in natural language. However, their knowledge might be accessible through written means. Pictures, diagrams and tables may be used to illustrate complex concepts. Also, the table of contents, index, glossary, summary or even references available in the method acquisition medium may provide useful searching directories.

For the last two types of acquisition media described in the previous section (the CASE tools and the training courses), method knowledge may also be acquired by trial and error learning. This is the basic inductive knowledge acquisition process that is always in operation although heavily overlaid by the social transfer processes.

10.3.2 DISADVANTAGES OF METHOD ACQUISITION MEDIA

Method acquisition by literature has its drawbacks. It does not have the advantage of being led by an expert tutor. Another problem is the lack of interaction with the outside world. It can also be difficult to gain any practical experience of using the method knowledge, though an intelligence computer-based training (CBT) toolkit may compensate for this deficiency of interactivity. Furthermore, good documentation is required in these method acquisition media, as they are treated as ‘secondary’ experts in the knowledge domain. The documents need to be well written. Ideally, a ‘primary’ expert still needs to be available to solve any queries or to clarify any implicit presumptions that cannot be answered by the knowledge acquisition media.

10.4 ELICITATION OF METHOD KNOWLEDGE

Elicitation of a method in software development is one of the most crucial activities in method knowledge acquisition. Expertise transfer cannot be simply considered (or modelled) as an engineering task or as a technical discipline, since it depends upon communication skills and techniques. Communications may be verbal as well as visual. However some practical guidance or experience is available when extracting significant information from the knowledge domain. There are three basic questions to be answered in knowledge elicitation:

- A. What is the information (knowledge) of interest to be elicited?
- B. Where is the knowledge mentioned stored?
- C. How can it be extracted from this storage?

We discussed knowledge acquisition media (question B) in section 10.3. Identification of essential information from a specific knowledge domain is a vital process in elicitation. In our case, the method model may include information such as modelling concepts, design steps, instruction in tasks, business rules etc. It also depends on the level of abstraction being described or required in the final intended system. Therefore, the description cannot really be separated from the discussion on extracting knowledge, that is to procure information from a massive amount of materials (or in computational terminology - raw data). Both question A and question C are considered at the same time in this section.

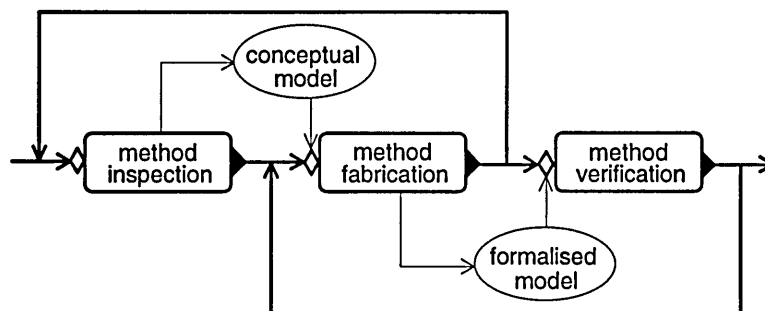


Figure 10.3 Top Level Process Model of Method Elicitation

Our approach towards the elicitation of method knowledge can be summarised by a three stage process model as depicted in figure 10.3. The stages are method inspection, method fabrication and method verification. **Method inspection** conceptualises the method's semantics from one or more knowledge acquisition media into a conceptual model, which can then be denoted in graphical or textual form. **Method fabrication** transforms the model into a practical (or executable) form so that it can be used by an applicable tool. The result is a formalised model known as a semantic knowledge base. **Method verification** checks the validity of the individual items in the semantic knowledge base. It also validates the entire

formalised model as a whole. Detectable mistakes can be identified and/or rectified by verification rules introduced from the meta model. This three stage model is abbreviated as the **IFV model**.

As with the phases of the software development life cycle, these method elicitation stages do not have clear boundaries. They are disjoint in terms of process and overlapping in terms of time. In other words, different stages have distinct tasks in knowledge elicitation but the respective tasks may occur simultaneously. For example, method fabrication may happen as long as a conceptual model is formed from method inspection, even though it is incomplete or erroneous. The amended formalised model may check through existing method verification rules if they are available. These three method elicitation stages are described separately in the following subsections.

10.4.1 METHOD KNOWLEDGE INSPECTION

Method inspection is the analytical stage of knowledge elicitation. The objective is to conceptualise the method's semantics from the accessible knowledge acquisition media. Since literature is the major source in method knowledge elicitation and the availability of the other two media depend on their application, the following discussion focuses on literature as the basic medium for knowledge acquisition. However, similar techniques can be applied to other materials available for inspection.

The product generated from this stage is known as the **conceptual model**³, it comprises a set of diagrams and structured texts. The diagrams incorporate concept diagrams and task diagrams which denote the product model and process model of the method respectively. The structured text gives the preliminary form of the method specification statements. This conceptual model is passed onto the method fabrication stage to produce the formalised model. Method inspection incorporates three sequential and incremental substages. They are known as cursory reading, chronic reading and conclusive reading as shown in figure 10.4.

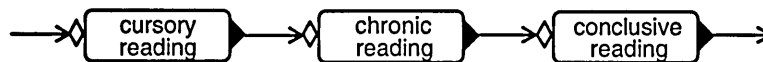


Figure 10.4 Method Knowledge Inspection

³ Conceptual model should not be confused with concept diagram in the product model. The former describes the overall modelling principles and the strategic ideas of a method, whereas the latter captures the notions of the method used to present a software system only. In other words, a concept diagram is one of the basic elements in a conceptual model.

10.4.1.1 CURSORY READING

In this substage the knowledge engineer scans the literature (a book or a journal) to obtain an overview of the method. The crucial method semantics are normally presented in various distinct parts of the literature. These distinct parts include the introduction, table of contents (chapter headings and subheadings), glossary and index etc. From a method knowledge engineering point of view, the basic modelling approaches and/or some major strategic ideas should be identifiable in a cursory reading of these sections.

The **introduction** section normally gives information about the foundation of the approach or the problem domain denoted by the method. For instance, by reading the identified sections, one can easily find out that HOOD is an object-based method, whereas Ptech is a set-based object-oriented approach. Therefore the knowledge elicitation can be done around the main focus of the respective methods or in a coherent manner.

The **table of contents** usually gives a logical sequence to interpret the method, although some literature places the materials in an inappropriate order. Sometimes the literature provides a map (in introduction) for the reader to extract particular information of the method or to ignore materials that the reader might already know. For instance, in [Gomaa 93], three methods are described, namely Codarts, Adarts and Codarts/DA. Since Codarts/DA is more general than Adarts and it is superset of Codarts, it is chosen for investigation. Furthermore, a list of design methods is introduced before the presentation of the proposed methods. The inspection of these methods, including Adarts, is optional, so the corresponding chapters or sections that are noted in the table of contents can be skipped. In addition, the structure of the chapters and sections provides the differentiation and emphasis of the method. For example, from the layouts of the chapter headings (and subheading), it can be readily identified that Booch OOD comprises six main diagrams (class, object, module, process, state transition and timing diagrams) and four steps (identify classes & objects, semantics, relationships and then implement classes & objects).

Most literature provides a **glossary** to summarise the key ideas introduced by the method in alphabetical order. It is normally presented at the back of the literature and usually there is a description given to each key idea. A glossary is an important source in eliciting method semantics, since the description field gives a formal definition to each key semantic. For example in HOOD⁴, *dataFlow* is defined as '*a flow of data in parameters of operations between used and using objects*'. A data (semantic) dictionary can be built up from this information. The descriptions can also provide information towards the heuristic model based

⁴ In order to improve readability, hereinafter in this section, the names of software development methods also denote the respective literature that described the methods. For instance, Booch OOD is from the literature [Booch 91], Codarts/DA is from [Gomaa 93], HOOD is from [Robinson 92], OMT is from [Rumbaugh 93] and Ptech is from [Martin 92].

on key semantics of the method. However, there may also be some abbreviated names or non-method based ideas. For instance, in the glossary of Codarts/DA, AAD (the first item) is only an abbreviation of *adaArchitectureDiagram* so is not a concept; nor is Abstract Data Type (the second item), which is a general idea in object-orientation rather than specific to the method. In addition, the glossary may not contain all key method semantics or it may combine a few concepts into one terminology. For example, in the glossary of OMT, *aggregation* is described as a general concept of a part-whole relationship, though OMT identifies aggregation of states and aggregation within a state (*concurrentSubdiagram* and *nestedStateDiagram* concepts in *dynamicModel*). Thus the glossary is only a brief summary of key ideas rather than a complete listing. Nevertheless, it should cover most of the important semantics of the method and acts as a checklist for the final conceptual model (see the conclusive reading subsection).

Similar to the glossary, the **index** section presents most of the key ideas of a method. Some significant semantics can be deduced from the entries in the section. In addition, it provides a referential link from the entry to the main body of the literature. For example, in OMT, the index entry of *action* has a list of sub-entries as shown below (the numbers give where the corresponding entries can be found in the main text and the bold number shows the location of the concept definition):

- action **92-93**, 131-132
 - entry and exit 101-102
 - internal 102
 - notation for 93
 - on a state 101
 - on a transition 93

With the help of references to the main text, the above example can be read as '*an action is a notational concept of three different types, namely entry action, exit action or internal action; it can appear either on a state or on a transition*'. Therefore, the index is a powerful tool to associate various ideas on top of relating them to the main text. The above example demonstrates the association of entity concepts in subtyping relationship as well as the ownership of action concept itself. Again, the index section is only a guide to identify method semantics, it is unlikely that the section provides a complete set of key semantics in the method.

In the discussion so far the method elicitation techniques are based on the product model. The semantics of process model can also be identified in the cursory reading of such as in the introduction, table of contents and sometimes even in glossary. However, the distinction is not as clear as the semantics of product model. For instance, the subheadings in Booch OOD chapter six, *The Process*, show that there are four main tasks in the process model and each of them has corresponding activities and products. However, the heuristic knowledge is normally not obtainable in cursory reading.

10.4.1.2 CHRONIC READING

In this substage, the knowledge engineer inspects the main text in depth until all key semantics are extracted from the literature. This may include reading the main body of the literature sufficient times to gain a clear understanding of all illustrations given in examples, diagrams, tables and possibly the bulletin points. By the end of chronic reading, a preliminary conceptual model should be sketched for further inspection in conclusive reading. Knowledge acquisition from textual information is fairly similar to that of verbal communication with domain expert [Wielinga 93]. However, the elicitation based on literature has a few distinct characteristics:

Obviously, the **main text** provides the details of semantic knowledge that are crucial to the method. Some of them are implicit or hidden information, it is impossible to extract them other than by a chronic reading of the text itself. For knowledge elicitation of software methods, special care must be taken to maintain the right level of semantic abstraction. Task decomposition in process modelling is a good illustration. In the horizontal dimension all tasks of a composition should have the same level of abstraction. In the vertical dimension the levels of decomposition should provide the key tasks presented by the method. Furthermore, some non-diagrammatic information can only be described in main text. This includes the heuristic guidelines, design rules and concept constraints. In Codarts/DA, a large proportion of the text is focused on the description of structuring criteria about subsystem, object, function, task and information hiding module. On the other hand, a HOOD diagram has a number of design rules associated with its concepts. For example, rule (U-2) is that: *'passive objects shall not use each other in a cycle'*. These textual materials cannot be shown diagrammatically as product or process models, but they should be represented in our heuristic model.

The other important knowledge elicitation tool is **illustration**. Most illustrations of method models given in the literature, are designed to demonstrate specific semantics so are given as enhancements to the description in the main text. There are various ways to illustrate ideas such as examples, diagrams, tables, charts, etc. For example, in the KADS approach, a road map (like a Gantt chart project plan) is used to show the activities of the method against time. It is easy to indicate dependence or complex relations by graphical representation. An illustration not only makes the expertise more readable and understandable but also gives a concrete presentation of abstract ideas - the concept of a *stateTransitionDiagram* is not understandable until there is a tangible example of it. Sometimes ideas do not appear in the main text, but are found in illustrations.

Bulletin points are normally used to show a number of important ideas about the semantics, they are rather like different sections in a chapter. For instance, OMT uses bulletin points to present the criteria for discarding redundant and irrelevant classes. Bulletin points may also be used to describe a number of tasks in a logical order. For example, the process model of

HOOD is given by four bulletin points with the heading as shown below. Each of these bulletin points is allocated to a subsection of the chapter.

- *Phase 1. Problem definition. ...*
- *Phase 2. Development of a solution strategy. ...*
- *Phase 3. Formalisation of the strategy. ...*
- *Phase 4. Formalisation of the solution. ...*

A **case study** (occasionally known as an application) is another important source of knowledge inspection in chronic reading. Nowadays most software development methods have at least one case study at the end of the literature. A case study describes a complete or a large part of the software development life cycle, rationally proving the methods to be applicable to a real world situation. Therefore a case study can also be considered as an extensive illustration of the method. Sometimes case studies are used to show the adaptability of the method to various software or hardware platforms. For instance, Booch OOD includes five applications in the last part of the literature. In doing so it is demonstrated that the modelling technique can be applied to the according five programming languages (Smalltalk, Object Pascal, C++, CLOS - Common Lisp Object System and Ada).

10.4.1.3 CONCLUSIVE READING

Conclusive reading is the last but not the least stage in method inspection. A relatively complete conceptual model of the method may have been gained by chronic reading, but human construction and/or predilection may cause the method model to be erroneous. The aim of this stage is to rectify any detectable faults that may be induced by the knowledge engineer in previous stages. This is done by studying various conclusive parts of the literature conscientiously. The implicit semantics in the main text are also validated in this stage.

In most literature there is a **summary section** (page, chart, table, etc.) either placed on the inner covers (such as Booch OOD and OMT) or in a chapter (such as Codarts/DA and Ptech) of the literature. The summary section gives a concise presentation of the method's semantics. Most of these semantics are notational concepts, but they may also depict other semantics implicitly. For instance, the inner covers of Booch OOD clearly summarise the semantics of six diagram fragments in the product model as well as four main tasks in the process model. All these semantics should have been captured in the conceptual model, so the summary section can be used to check and to finalise the main features of the method model. There is also another way to utilise the summary section. That is, to employ it in cursory reading, to construct a preliminary model which forms the basis in developing the entire conceptual model. However, using the summary section for validation only, may reduce unnecessary bias induced in the early stage of knowledge elicitation. On the other hand, the **glossary** section is

reviewed to check all key semantics are presented correctly in the conceptual model. Fine adjustment may mean referring back to the main text (through the **index**) to rectify the mistaken structure.

Furthermore, by considering the method semantics as a whole in conclusive reading, a more coherent model is obtained. For instance, unification of common concepts in product model is combined together. The values of *concurrency* (*sequential*, *blocking*, *active*) in *classDiagram* and *objectDiagram* of Booch OOD can be combined to share same properties.

10.4.2 METHOD KNOWLEDGE FABRICATION

During knowledge inspection, method semantics are elicited and captured in a conceptual model comprising a preliminary form of concept diagram, task diagram and heuristic model. These items are then transformed into a structured text format defined by the method specification language (MSL). The textual form of the method model is known as a formalised model. The MSL statements can be modified, rectified and finally compiled into Prolog clauses for further use. All the activities described above contribute to the construction of a knowledge base for method semantics, or in other words, fabricating method knowledge. As noted earlier, a knowledge based system or CASE tool can also be derived from this semantic knowledge base. The following process model summarises this:

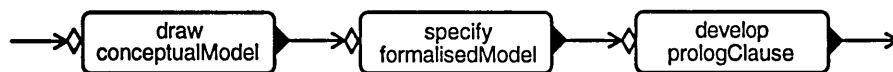


Figure 10.5 Method Knowledge Fabrication

The knowledge representation of method models was discussed in chapter eight. This section only concentrates on describing the significance of each stage of knowledge fabrication.

A **conceptual model** presents method ideas straight from knowledge acquisition media. The main interest in conceptualisation is to capture abstract ideas into concrete models such that they are descriptive as well as manageable. A conceptual model denotes method knowledge in a more readable and editable manner. Since knowledge inspection is a long and error-prone task, ‘ease of change’ is important in method modelling.

Likewise, a **formalised model** enables the presentation and modification of the method models captured. It also shows method semantics that cannot be denoted by the conceptual model. For instance, a complete description of heuristics can only be shown in a formal presentation. A formalised model also permits error checking and detection (section 10.4.3 discusses five types of verification) so that a rectified method model can be conceived. In addition, a knowledge engineer may modify a formalised model and then reverse engineer back to the conceptual model.

An **executable form** of method semantics is useful in constructing tools or systems operating on the knowledge base. For this reason Prolog is an ideal language in our application, though certain knowledge engineering tools (such as KEE) may also suit the job. A better performance and/or a smaller storage size is usually the consequences of a compiled form, however this is of only minor concern in this research.

10.4.3 METHOD KNOWLEDGE VERIFICATION

After knowledge fabrication, the method model may contain a number of errata. They may have been introduced during method inspection, such as missing implicit semantics or generating contrary ideas. Mistakes might also have been introduced in the fabrication stage, such as typing errors or inconsistent information.

The knowledge specified must be verified so that it presents a true representation of the method. Simple rules can be formalised or even embedded in the MSL compiler to identify detectable mistakes. Meaningful error messages are shown to enable the knowledge engineer to rectify discovered faults. This process in knowledge elicitation is known as **method knowledge verification**.

It must be clear that this verification can only rectify the errata introduced by the knowledge engineer. It cannot validate the method semantics as presented in the acquisition media. In other words, knowledge verification helps to obtain or maintain the legitimate semantics shown in the source. It is almost impossible to validate if a method model is a true and complete representation of the intended method. Furthermore, a lot of human understanding is done on exposition of knowledge, and most misinterpretation can never be remedied unless the originator is involved in the validation process.

Five techniques are identified for method knowledge verification. They are *correctness*, *completeness*, *contradiction*, *consistency* and *contrast*. Figure 10.6 illustrates that these checks can be executed simultaneously and there is no dependence on one another. They are discussed separately in the following subsections.

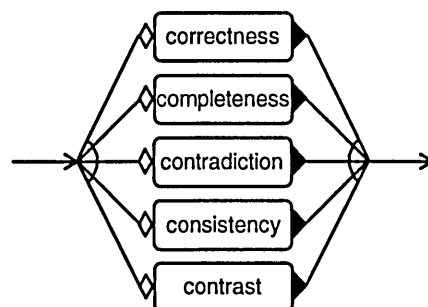


Figure 10.6 Method Knowledge Verification

10.4.3.1 CORRECTNESS

The word 'correctness' may have many different interpretations (it can even stand for a combination of all the following techniques). In method verification, it is interpreted as an agreement with the definition of the meta model, that is to check that the method model does not violate the meta-knowledge of the generic model. There are constraints on each meta-concept in a method representation. These constraints can be formulated as rules in the MSL compiler, so all these mistakes can be rectified by the knowledge engineer. Two basic forms of correctness are described below:

- *Self identity.* All concepts must have their own declaration (or entry) in the method model, so that every semantic is registered in the knowledge base. This rule detects spelling errors and duplications of semantic entries. It is done by maintaining a symbol table of all identifiers (names of semantics) and checking the validity of the semantic name in the MSL statements during compilation.
- *Min-max cardinality pair.* A cardinality pair denotes the minimum and maximum cardinalities of a relationship. Most concept relationships have two cardinality pairs, one for the source part and the other for the target part (see chapter five). By definition, the maximum cardinality must be larger than the minimum cardinality. This error can be indicated by examining each cardinality pair.

10.4.3.2 COMPLETENESS

Similar to correctness, completeness deals with the verification of the method model against the meta-knowledge of the generic model. However it considers the constraints between two concepts rather than the definition of a single concept. These constraints can also be included in MSL compilation checking. The incomplete semantics of a method model are indicated. Three common forms of completeness are shown as below:

- *Link completeness.* Each link concept must have exactly one source part and one target part (one-to-one in cardinality pair). If any of those parts has more than one choice, the cardinality pair of each potential concept must be defined as zero-to-one.
- *Fragment completeness.* A fragment without any entity is a void concept. Therefore the MSL compiler should detect that each fragment concept must have at least one composition relationship to a valid entity concept.
- *Subtyping and overriding.* A subtyping relationship inherits properties of the supertype to the subtype. An overriding feature rewrites (or overrules) this characteristic. In the case of multiple inheritance, overriding must be used to indicate the overlapped properties.

10.4.3.3 CONTRADICTION

Unlike correctness and completeness, contradiction involves two or more semantics. The usual situation is the relationship between two semantics affecting a third semantic. It also handles problems of cycles amongst semantics. This verification can also be implemented as constrained rules to detect faults during MSL compilation. These semantic faults are indicated to the knowledge engineer. Two types of contradiction are shown as below:

- *Ownership problem.* There are two types of hierarchies in the concept diagram: subtyping and composition. The diagrammatic denotation has virtually avoided cyclic problems in subtyping relationships, as recursive subtyping is logically invalid in meta modelling. However, these problems may happen in composition relationships. By definition, a cyclic ownership is possible in some cases, but a warning message may be generated to notify the knowledge engineer.
- *Referential integrity.* The referencing relationship is an important technique to denote the dissection points between fragments in a method. However, the reference direction must be compatible and any unnecessary recursive loop should be avoided.

10.4.3.4 CONSISTENCY

Most methods have a main theme to their semantics. This theme gives a coherent picture of the method and it must be highlighted in the method model accordingly. Therefore the verification of consistency is sometimes known as coherency. For instance, Ptech is based on the theme of relating *object* and *event*. An *objectType* comprises a number of *event operations*, and an *eventType* takes the underlying *object* in a *trigger* to invoke an *operation*. This consistency must be coherent in the product model as well as the process model of the method. For example, Codarts/DA is based on the modelling of *informationHidingModule* and *task*. A *systemContextDiagram* is constructed to identify the *dataTransformations* in a more detailed *dataFlowControlFlowDiagram*, from which the *controlTransformations* are used to develop *stateTransitionDiagrams*. Finally, all these semantics contribute to define various *tasks* and *informationHidingModules* in the *taskArchitectureDiagram*. Consistency is a useful tool to check that the theme of a method is logically presented in the knowledge base. However it is impossible to describe this formally as constraint(s).

10.4.3.5 CONTRAST

Contrast is the opposite of consistency. Instead of looking at the coherency of a method's semantics, it compares its semantics with the corresponding semantics in other methods. Hence the key semantics of the original method emerge in the contrast. For example, Ptech

models concurrency by event synchronisation of states, whereas Codarts/DA models it by a concurrent task. For a second example, the semantics described by *stateTransitionDiagram* in OMT are different from those of similar fragments in other methods. These semantics include concepts such as *stateGeneralisation* and *concurrentSubdiagram*. OMT also distinguishes the difference of *action* and *activity*. Therefore, all these distinctions must be emphasised in the method model. Contrast is a technique to divert the knowledge engineer from taking a single-minded modelling viewpoint and instead to look at the bigger scope of method family. It is normally used to check whether the formalised model presents the distinct semantics of other methods or not. Again this is very difficult to model as an executable rule.

10.5 METHOD MODEL ELICITATION

The last section described the elicitation of semantic knowledge of method models with an inclination towards ‘how they can be found from literature’. This section focuses on the method elicitation of the three constitutes in a meta model, which are the product model, the process model and the heuristic model. OMT is chosen for this illustration, since it covers most of the elicitation techniques. A complete listing of OMT MSL statements is given in appendix F for validation.

10.5.1 PRODUCT MODEL ELICITATION

Most of the method concepts can be identified in the cursory reading stage of inspection - the glossary usually lists all the main concepts. Product model elicitation should include as candidates all concepts except the implementation constructs (such as null, overloading, self, weak typing) and general ideas (such as invariant, method, object-oriented, programming-in-the-large). Looking into the main text, the candidate concepts are collated with their definition, and the meta relationships and properties are identified. It is important to unify the terminology because a concept may be called differently in various places in the literature. The general rule is to choose the shortest meaningful name and to keep the main concepts only. For instance, the concepts:

- *actor* means actor object;
- *operationPropagation* is a more specific name than propagation;
- constraint actually denotes two different concepts: *associationConstraint*, *classConstraint*

Some concepts are hidden in section headings (such as *entryAction*, *exitAction*, *internalAction*) or illustrations (such as *concurrentSubdiagram* and *dataFlowDecomposition*). They must be extracted and defined explicitly in the heuristic model to avoid ambiguity. For instance, the following MSL statements denote the *concurrentSubdiagram* concept and the corresponding concept heuristic.

```
concept(concurrentSubdiagram, [group], concrete) .
source(concurrentSubdiagram, [state], [0,1,1,1]) .
target(concurrentSubdiagram, [stateDiagram], [0,1,1,n]) .
```

```
heuristic(concurrentSubdiagram, [state, stateDiagram],
'Concurrency within the state of a single object arises when the object can be partitioned into
subsets of attributes or links, each of which has its own subdiagram.') .
```

Referencing relationships and dissection sets are more global meta relationships in method modelling, they only come out in the conclusive parts. For instance, in the end of each chapter of [Rumbaugh 91] describing the analytical models is presented the relations between fragments with the subsection headings:

5.7 Relation of Object and Dynamic Models

6.6 Relation of Functional to Object and Dynamic Models

The knowledge engineer has to model segments of the concept diagram and gradually place them together. A simple data dictionary is maintained, which eventually becomes the heuristic model. When doing this, the engineer should always be asking the questions, '*Does the model have enough concepts to describe the software semantics in this example?*' and '*Are the concepts truly reflective of the emphasis in the method?*'. The emphasis should be on the overall product model rather than on refining the individual details of concepts and relationships. The glossary and index should be checked again for any omissions or errata. The method model must also be verified with the semantics described in case studies and summary pages. The entire process is a continual iteration; different parts of a model are often at different stages of completion. If a deficiency is found, the engineer must go back to an earlier stage and correct it. Some refinements can only come after the process model elicitation.

10.5.2 PROCESS MODEL ELICITATION

Unlike the product model, the process model is usually ill-defined or full of presumptions. The elicitation is mainly based on the sequential description of the software development steps. Even the glossary does not give much information. This is true even for top level tasks (such as *analysis*, *systemDesign*, *objectDesign* in OMT). The main text shows a list of tasks in linear order, but special care must be taken over recursive loops and repeated patterns. For instance, the bulletin points⁵ of object modelling:

- 1 identify objects and classes
- 2 prepare a data dictionary
- 3 identify associations between objects
- 4 identify attributes of objects and links
- 5 organise and simplify object classes using inheritance

⁵ For the sake of clarity, the bulletins are shown together with numbers in this illustration.

- 6 verify that access paths exist for likely queries
- 7 iterate and refine the model
- 8 group classes into modules

The second point ‘prepare a data dictionary’ is in fact applied to classes, associations, attributes and operations. Therefore the actual sequence of tasks should be 1 2 3 2 4 2 ... etc. The seventh point ‘iterate and refine the model’ is really a loop-back to any of the previous stages rather than a call back to the first step every time. Moreover, the first point in the above example is accompanied by heuristics mentioned in the two following sections ‘identifying object classes’ and ‘keeping the right classes’. With the ‘prepare a data dictionary’ step, there are three steps to complete the composite task *identifyClass*. The pre- and post- conditions of the task are identified from the heuristic statement as shown below:

```
task identifyClass perform(identifyClass) ;
    precondition [problemStatement] ;
    postcondition [class] ;
    compose [insertClass, verifyClass, specifyClass] ;

heuristic identifyClass ;
text    'Identifying relevant classes from the application domain. Objects include physical
        entities as well as concepts; avoid computer implementation constructs.' ) .
link    class, object ;
```

The *insertClass* and *specifyClass* tasks are terminal, whereas the *verifyClass* task is refined based on the heuristic given in ‘keeping the right classes’ section. If there is no terminal task, it may not be wrong but keep it as a composite task with description in the heuristic model. The developer has to customise this with their own action. Furthermore, not all concepts in the product model are described in the process model because they may not be in the same level of abstraction or they may be embedded in the construction of other (composite) concepts. Always prepare the task sequence tables and the task diagrams. The summary section does not normally help in elicitation, so the model is verified by cross-checking the pre- and post- conditions required and acquired by the tasks in different abstraction levels.

10.5.3 HEURISTIC MODEL ELICITATION

Ideally, each method semantic has a heuristic clause to explain its meaning. Most concept descriptions can be found in a glossary, but the implicit concepts, such as group concepts, can be extracted from the corresponding section in the main text. Avoid lengthy reports on the concept, because the developer can always refer to the source of reference. The heuristics of product model should only give the brief definitions of individual concepts, that is a summary of the description in the literature as demonstrated in section 10.5.1 and 10.5.2.

Heuristics of a process model contain rules and guidance of modelling decisions. Although the tasks may be found in various parts of the literature, they are mainly denoted in the main text. High level tasks, such as a composite task, only acquire a brief description of its

functionality as shown earlier. Denote all decision making heuristics shown, even though they may be contradictory. Not all tasks demand a heuristic, but most terminal tasks require a clause for making decisions. Therefore a rule in extracting a process heuristic is to look for the IF...THEN... (or similar) statements. For instance, all the points in the 'keeping the right classes' section are presented in such a format, as illustrated below:

```
heuristic(deleteRedundantClass, [class, object],
'If two classes express the same information, the most descriptive one should be kept.')
```

```
heuristic(deleteIrrelevantClass, [class, object],
'If a class has little or nothing to do with the problem, it should be eliminated.')
```

10.6 CONCLUSION

This chapter has presented the knowledge acquisition techniques for method models. The discussion focused on extracting domain knowledge (as in KADS). There are some fundamental difficulties in method acquisition but a variety of techniques can help to enrich the quality of expertise transfer. Method knowledge engineering employs a distinct set of knowledge acquisition media, because suitable experts are difficult to find and their availability is low. The importance of these specific media is investigated. Some guidelines for knowledge elicitation are also given as practical assistance to the knowledge engineer in capturing method semantics. These techniques include inspecting, fabricating and verifying method models. Various illustrations are given, based on the knowledge acquisition of the OMT method model.

11. MAPPING METHOD SEMANTICS TO METACASE TOOLS

A method model is a specification which is only a ‘theoretical design’ until it can be implemented into an ‘executable tool’. Therefore it is important to show that the method semantics of our meta model (i.e. product model, process model and heuristic model) can be mapped into metaCASE tools. There is an incremental equilibrium between ‘design’ and ‘tool’ in method engineering. Our meta model is a generic representation. It does not depend on any particular metaCASE tool, though the IPSYS ToolBuilder has been chosen for this illustration. Two case studies are used to demonstrate the techniques required. Firstly, in a simple method, Scratch¹ provides an example of a total mapping. Secondly, Booch91 illustrates a partial case study in matching meta components to those in the chosen metaCASE tool.

11.1 INTRODUCTION

As a generic representation of software development methods, the meta model is neither tool nor method dependent. This uniformity is a great advance on the present technology for method engineering. Some of the current approaches are only accomplished by human drudgery or by individual understanding without any genuine structured techniques or models. There are various possible applications of meta modelling techniques:

- The meta model conceptualises complex method semantics, permitting a smoother and more accurate transition to the metaCASE tool model.
- The standardised model allows a set of method metrics to be asserted so that a formulated mechanism can be employed in method comparison. (see section 9.3)
- It supports multiple perspectives in a specific problem domain, since method integration is more effective amongst unified method models. Fragment dissection divides a method into useful components that can then be reconstructed into a customised method.

This chapter elaborates on the first point listed, that of developing a meta model to conceptualise complex method semantics. This is an essential requirement of meta modelling and a formal proof of the meta model applicability. Figure 11.1 depicts the two ways of transforming method semantics into a metaCASE tool model. The dotted arrows depict the

¹ Scratch method is a design method described in IPSYS ToolBuilder 1.3 tutorial [IPSYS 91].

normal way of CASE tool development: a metaCASE user inserts his/her understanding of a method directly from method acquisition media (MAM) into the metaCASE tool. The alternative way is shown by the thick arrows: this approach employs meta modelling techniques in the transformation. Method semantics in MAM are formalised into a method model by IFV and then mapped into the metaCASE tool model. The IFV techniques are presented in chapter ten and this chapter discusses the mapping techniques. The thin arrows in figure 11.1 denote the flows common to both approaches. It is important to emphasise that a software engineer uses the method embedded in the generated CASE tool to produce the final applicational software. In order to distinguish the method models described in different domains, the corresponding models of generic metaCASE tool domain, specific metaCASE tool (i.e. ToolBuilder) domain, metaCASE tool user domain and meta modelling domain are referred as generic tool model, ToolBuilder model, user model and our model respectively.

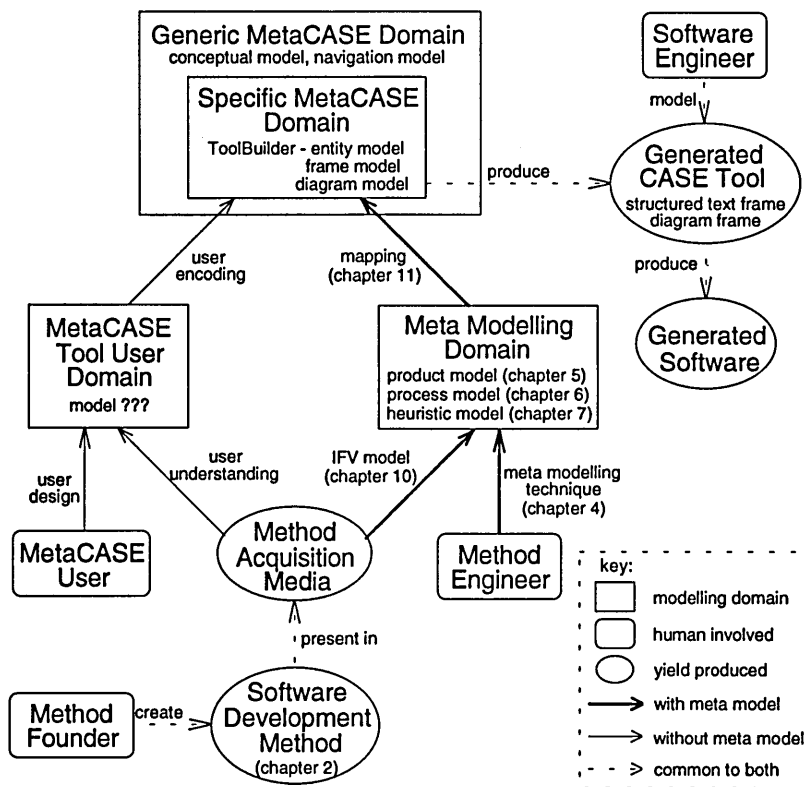


Figure 11.1 Two Ways of Modelling Method in MetaCASE Technology

The chapter is structured as follows. The next section discusses the importance of metaCASE technology in method engineering. It then introduces two case studies. The first case study given in section 11.3 demonstrates the techniques embodied in mapping the method model semantics to the metaCASE tool. Section 11.4 presents the outcomes derived from matching method models formalised with and without meta model in the second case study. The results are drawn to a conclusion in section 11.5.

11.2 SIGNIFICANCE OF METACASE TECHNOLOGY

This section looks at the significance of metaCASE technology in the aspect of method engineering. It begins by illustrating an incremental equilibrium in method engineering, and finishes by introducing two case studies to clarify the relation between our model and the metaCASE model.

11.2.1 INCREMENTAL EQUILIBRIUM IN METHOD ENGINEERING

Philosophers debate about the process of modelling a hierarchy (refer back to figure 1.2). Goodman proposed that it involves a dynamic equilibrium between data and inference rules as shown in figure 11.2: *'a rule is amended if it yields an inference we are unwilling to accept; an inference is rejected if it violates a rule we are unwilling to amend'* [Goodman 73]. This is termed as reflective equilibrium in inductive inference.

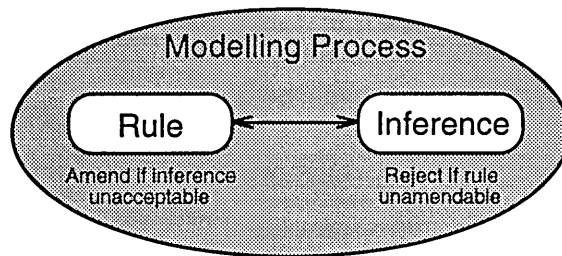


Figure 11.2 Reflective Equilibrium in Inductive Inference

This theory contrasts with the present method engineering realm in a pragmatic fashion as depicted in figure 11.3: *'a design evolves if we implement a tool with better models; a tool develops if we specify a design with better techniques'*. The existing design models restrict the development of tool implementation, whereas the current techniques in a tool restrain the evolution of design specification. This is termed **incremental equilibrium** in method engineering since it advocates a persistent progression. Thus the equilibrium emphasises technology gained under continuous improvement of available designs and tools. Meta modelling embodies a set of techniques to guide the development of method models.

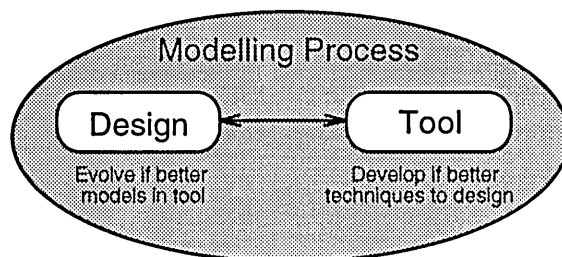


Figure 11.3 Incremental Equilibrium in Method Engineering

Hence it is desirable to denote the method model as a metaCASE tool model. To conclude, the mapping of method semantics to a metaCASE tool is not just a back-end application for meta modelling, but also a necessary front-end process to improve software development methods.

11.2.2 METHOD ENGINEERING IN METACASE

To satisfy the incremental equilibrium in method engineering, the semantic gap between design techniques and tool models² must be reduced. Figure 11.4 suggests that this gap arose during the modelling of a method into a metaCASE tool. The gap is normally induced by an unstructured approach to meta modelling within the tool or by an unsuitable knowledge elicitation technique from the method acquisition media. The first problem is tool-dependent and so is outside the scope of this research. However, chapter ten describes an IFV model to handle the second problem and this chapter extends it with the mapping of our method model to the tool model.

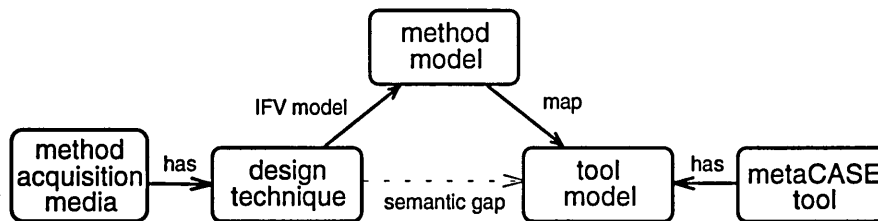


Figure 11.4 Semantic Gap in Method Engineering

11.2.3 TWO CASE STUDIES

Two case studies are introduced to reveal the techniques of mapping method semantics. The first case study demonstrates that there is appropriate technology to map method semantics to the metaCASE tool. A simple method known as Scratch is employed for this illustration. The second case study affirms that the existing work is at the right level of abstraction for present technology; that is, the current meta modelling design techniques agree with the available metaCASE tool method. A tool generated from a more intricate method, Booch91³, is used to outline this matching.

² Tool models mean the method semantic models embodied in a metaCASE tool in order to generate a CASE tool. This is not intended to be a process model to map method semantics to the CASE tool.

³ This method is referred as Booch91 since it is a slightly modified version of the original Booch OOD published in 1991 [Booch 91] (see section 11.4 for details).

Figure 11.5 depicts the work flows of these two case studies. The first work flow is shown by dotted arrows. The meta model conceptualises the Scratch method into our formalised model by using the IFV model described in chapter ten, and then the formalised model is mapped to the semantic model within a metaCASE tool from which the Scratch CASE tool is generated. The second work flow is denoted by dashed arrows. A metaCASE user models the Booch91 method in the tool and the tool-based semantic model (tool model) is matched against our formalised model represented by our meta model using the IFV model again.

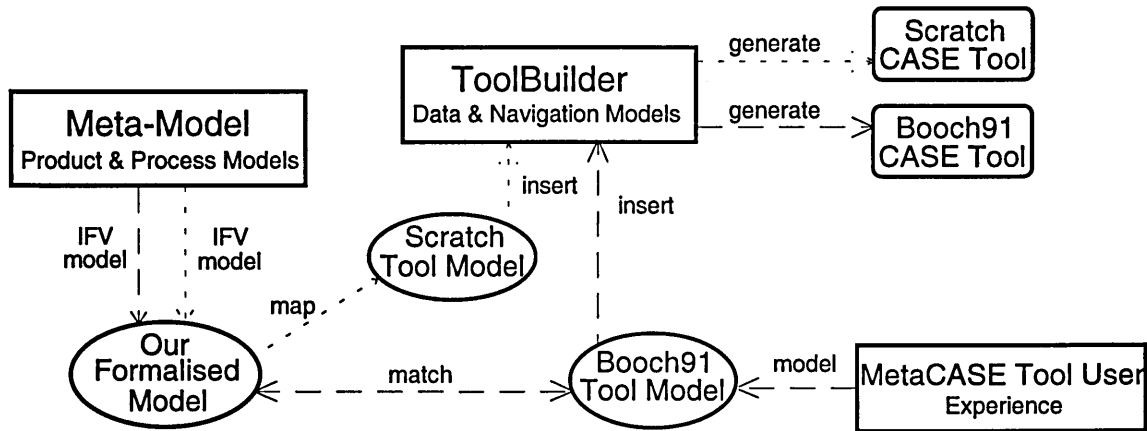


Figure 11.5 Work-Flows of The Two Case Studies

The two respective CASE tools are generated and executed as specified. The aims are summarised as:

- to demonstrate that method semantics can be mapped into a metaCASE tool
- to illustrate that meta modelling is method (and size) independent
- to indicate the discrepancy of method modelling with and without a meta-model
- to show the compromise of logical method design to practical metaCASE tool

The research should ideally consider a number of metaCASE tools. However, due to the limitation of time and the availability of competent tools, the experiment is only exercised on IPSYS ToolBuilder. This tool is known to be widely exploited for both academic and commercial purposes. The method semantics are captured by a mixture of entity models and frames model, which effectively sketch the data model and navigation model in the tool. The embedded programming language is known as EASEL. (refer to appendix B for more details)

Nevertheless, most descriptions and comments in both case studies given in this chapter are made in general to all metaCASE tools. ToolBuilder is used as a typical metaCASE tool to illustrate the mapping techniques and matching outcomes.

11.3 CASE STUDY A - SCRATCH METHOD

Scratch is a simple software development method. It may be considered as a method segment that can be applied to any competent method, such as OMT or Booch OOD. Scratch is an object-oriented method consisting of:

- Object Relationship Diagram - to capture information models;
- State Transition Diagram - to capture dynamic behaviour;
- Object Catalogue - giving a summary of objects in the database;
- Event Catalogue - giving a summary of events in the databases;

Because of the simplicity of Scratch it is an ideal vehicle to demonstrate the mapping techniques to the metaCASE tool. The formalised method model is captured in the Method Specification Language and then is placed into ToolBuilder for experimentation. The generated tool verifies the accuracy of the semantic mapping. The following subsections examine and discuss this with respect to the three constitutes of our method model.

11.3.1 PRODUCT MODEL MAPPING

This subsection demonstrates how the product model techniques are practised in the data model of the target tool. From the list in the previous section, Scratch has two diagram fragments and two text fragments. An *objectRelationshipDiagram* denotes *relationships* between *objects* in the problem domain, whereas a *stateTransitionDiagram* describes the behaviour of an individual *object* in terms of *states* and *transitions*. The *objectCatalogue* and the *eventCatalogue* provides a checklist of the corresponding concepts. Figure 11.6 denotes the overall products of Scratch in a concept diagram.

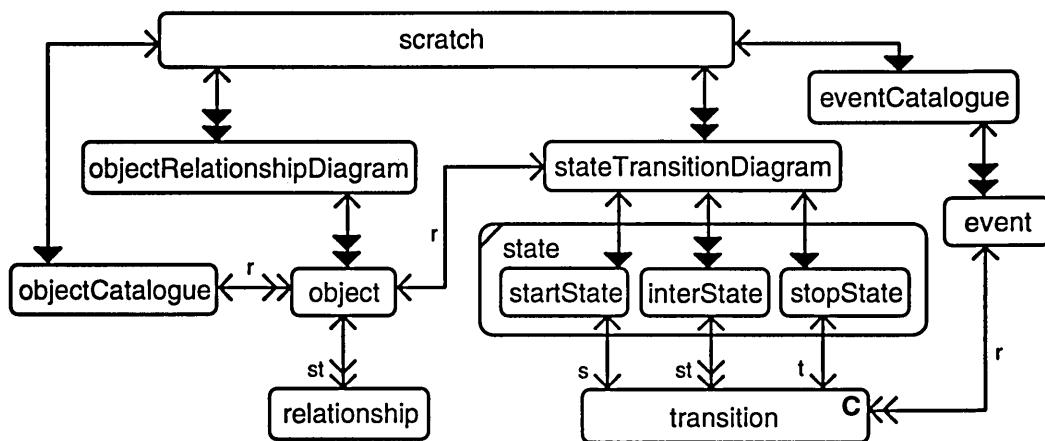


Figure 11.6 Concept Diagram for Scratch

The following points are noted about the concept diagram:

- Scratch is based on the modelling of objects, and its behaviour is denoted by a corresponding *stateTransitionDiagram*. Therefore a referencing relationship is used to depict this association. In addition, due to referential integrity, a software model must have at least one of each diagram fragment, and so one of each catalogue.
- In order to reduce the complexity of object-based relationships, such as inheritance, association, aggregation etc, Scratch denotes a simple concrete meta-relationship. The detail relationship properties, such as roles and cardinalities, are also omitted.
- Most methods have a very vague definition of different types of *state* concepts. In the Scratch example, it simply enforces one *startState* and one *stopState* in each diagram so as to demonstrate a specific semantic handling in the metaCASE tool (see figure 11.8).
- The *transition* concept is a multiple source-types and target-types link. No *transition* may connect a *startState* directly to a *stopState*. This is described by the following two product constraint rules:

$$\text{not}(\text{source}(\text{startState}) \text{ and } \text{target}(\text{stopState})).$$
- An *event* is depicted as a label on one or more *transitions* in the *stateTransitionDiagram* and is fully described in the *eventCatalogue*. This loosely coupled relationship between *event* and *transition* is denoted by a one-to-many referencing relationship.
- The actual system has various structured text detail frames to enable documentary text to be entered (such as *object*, *relationship*, *state*, *transition* and *event* definition frames). Since these frames have no direct semantic effects in the software modelling, they are not shown on the concept diagram.

11.3.1.1 BASIC PRODUCT MAPPING

ToolBuilder captures the data model by means of an entity diagram. Individual entity details are described by the entity type structured text definitions. Each entity is, in fact, a concept denoted by the method and dependencies between concepts are shown by the three primitive relationships: subtyping, composition and reference relationships (see appendix B and section 3.4.3). Some high level entities are denoted as frames, which are described as fragment concepts in the method model. Textual and graphical fragments are mapped as structured text and diagram frames respectively. The attributes associated with each entity are presented as fields in the diagram frame or objects in the structured text frame. Table 11.1 illustrates the mapping of the product model to the ToolBuilder entity model. It classifies three categories of mapping elements: *concept*, *relationship* and *property*.

Category	Product Model in GMR	ToolBuilder Entity Model
concept	fragment concept	frame
	entity concept	entity (node)
	link concept	entity (link)
	property concept	attribute (or field or object)
	group concept	role in composition
relationship	subtyping (supertype can instantiate)	subtyping (supertype is abstract)
	composition	composition
	linking	composition
	grouping (in same fragment)	composition
	grouping (different fragment)	reference
	referencing	reference
	dissection set	-
	-	derived (path, aggregation, user-defined)
property	cardinality	single / set / sequence
	role (fixed)	role (user-defined)
	directional	forward & reverse links
	shared concept	shared object type
	constraint rule	EASEL code (i.e. precondition)

Table 11.1 Mapping Product Model to ToolBuilder Semantics

11.3.1.2 MAPPING METHOD SPECIFIC SEMANTICS

This subsection uses the Scratch method to illustrate the mapping of our product model to the ToolBuilder entity model and refers to the information in table 11.1. Figure 11.7 depicts the complete entity model diagram of the Scratch method; some segments of entity type definitions are shown to emphasise particular points.

- ToolBuilder distinguishes entities as **nodes** or **links**, which are in fact the entity concepts or the link concepts in the product model respectively. Group concepts are not directly shown in the entity model, but they are hidden as **roles** in the substituted composition or reference relationships (see later).
- **Subtyping** in the product model has a similar effect as that in ToolBuilder, except the latter allows concept instances from terminal entities only. In other words, every supertype is an abstract concept and all terminal entities are concrete concepts. The *STATE* concept in the Scratch method is an abstract concept which is accented in capital letters.
- Apart from subtyping, most relationships in ToolBuilder have user defined **role names** (forward and/or reverse name). These role names have no semantic meaning, but they are used to address navigation flows. For instance, the *unit* reverse role of the composition relationship between *UNIT* and *object* indicates the flow from *object* back to *UNIT*.

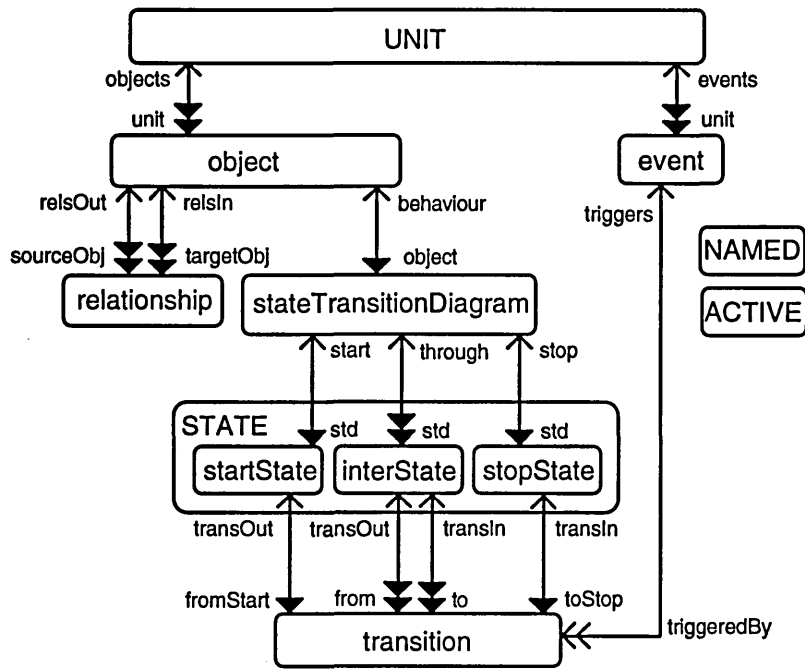


Figure 11.7 ToolBuilder Entity Model Diagram for Scratch

- All ToolBuilder concept relationships can be defined as **bi-directional**, this is shown by a reverse role name and the arrow head(s) on the target side. If the target cardinality has the value '*none*', no reverse link is available. Hence, the **cardinalities** in the product model describe the logical dependence (*zero-one*, *zero-many* etc.) of concept relationships, rather than the physical entries (*single* / *set* / *sequence*) of links. Table 11.2 shows the mapping of cardinality constraints between the concept relationships and the ToolBuilder links.

Concept Relationships		ToolBuilder Links	
source	target	source	target
-	-	-	none
zero-one	zero-one	single	single
one-one	one-one	single	single
zero-many	zero-many	set / sequence	set / sequence
one-many	one-many	set / sequence	set / sequence
many-many	many-many	set / sequence	set / sequence

Table 11.2 Mapping Cardinality Constraints

- Mapping **composition relationships** is straightforward, apart from the compositions of link concepts (described next). Some extra syntax is employed by ToolBuilder to indicate the role, sequence and ownership features, such as the definition in *UNIT* concept:

```
COMPOSITION RELATIONSHIPS
events : Seq Of event INVERSE unit
objects : Owner Of Seq Of object INVERSE unit
```

- ToolBuilder denotes the link concept but a **linking relationship** is not directly distinguished. The relationship is mapped into a composition relationship, although this is not a satisfactory solution (see section 3.4.3). The declarations of the source and target parts are denoted by the **Out** and **In** directives respectively. This is demonstrated by the *relationship* link in the *object* entity definition shown below:

```
COMPOSITION RELATIONSHIPS
relsOut : Out Seq Of relationship INVERSE sourceObj
relsIn : In Seq Of relationship INVERSE targetObj
```

- **Reference relationships** in ToolBuilder are used to associate entities for navigational purposes. They have a similar effect as the referencing relationship in the product model. Referencing relates a concept to another concept that further specified itself (see section 5.3.3.5). The declaration is similar to that of composition. For instance, the reference relationship in the Scratch method depicts a *transition* associated with an *event* label from the *eventCatalogue*.

```
REFERENCE RELATIONSHIPS
triggeredBy : event INVERSE Seq Of triggers
```

- There is no **grouping relationship** in the Scratch method - the relationship between *object* and *stateTransitionDiagram* has a one-to-one referencing dependence. If state decomposition is allowed, a group concept will form between host *state* and the element diagram. ToolBuilder hides this group concept and replaces the relationship by a composition relationship from the host *state* to the diagram.
- There are no **dissection sets** or **shared concepts** shown in the illustration. A dissection set is useful in method integration or fragment dissection, but this semantic does not normally register in the formalisation of a CASE tool. However, we must be able to deal with shared concepts where a concept with identical semantic meaning is presented in different fragments. ToolBuilder handles these concepts by *shared object types*.
- Some product constraints cannot be denoted in the concept diagram, but instead are declared as rules in MSL. These constraints normally lead to special encoding by the tool language, such as EASEL in ToolBuilder. Individual constraints can be recorded in either the **precondition** slot of the add object operation or the **on creation** slot of the entity type definition (see later). However, one might have to modify the entity model to captivate the effect of a constraint. For instance, the only formal way to denote the *startState*-to-*stopState* constraint in an entity model diagram is to introduce two ‘virtual concepts’ distinguishing the source of the *transition* as depicted in figure 11.8. Of course these concepts have no real semantic meaning but they are inserted for the sake of denotation. The current Scratch tool manages the creation of *startState* and *stopState* in the formation of the *stateTransitionDiagram*, which effectively handles the constraint by EASEL code.

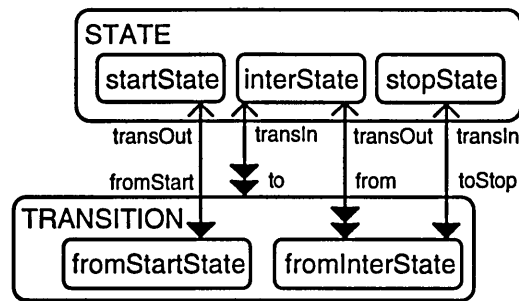


Figure 11.8 ToolBuilder Entity Model of the *startState* to *stopState* Constraint

11.3.1.3 EXTRA SEMANTICS

Before conveying our product model to the metaCASE tool model, extra semantics must be employed to complete the conceptual model in defining the final executable tool. These semantics may be tool specific, user specific or may be extended features for the purposes of process modelling, so they are not provided directly by the method models themselves. The following points illustrate some of these extensions of the Scratch method in ToolBuilder.

- ToolBuilder provides three **prefined entity models**, from which new entities can obtain the built-in operations in the metaCASE tool (see appendix B). The following three entity type definition segments (*UNIT*, *STATE* and *transition*) show the inheritance mechanism for the *DIAGRAM_UNIT*, the *NODE* and the *LINK* entities respectively.

```

ENTITY TYPE UNIT -- A DIAGRAMMATIC FRAME
    IS A ROOT TYPE
    AND IS A predefined . ENTITY
    AND IS A diagram . DIAGRAM_UNIT
    AND IS A diagram . DE_FRAME

```

```

ENTITY TYPE STATE
    IS A ROOT TYPE
    AND IS A predefined . ENTITY
    AND IS A diagram . NODE
    AND IS A scratch . NAMED
    AND IS A scratch . ACTIVE

```

```

ENTITY TYPE transition
    IS A ROOT TYPE
    AND IS A predefined . ENTITY
    AND IS A diagram . LINK
    AND IS A scratch . NAMED
    AND IS A scratch . ACTIVE

```

The tool entity model only gives a **partial declaration of fragments**. For instance the *stateTransitionDiagram* concept is the only fragment denoted. It is shown in order to stress the one-to-one dependence with the *object* concept and to provide a directional path from a structural object to its behavioural state-transitions.

- As mentioned in section 3.4.3, the process model of ToolBuilder is based on a navigation model of frames (fragment concepts) and their interior entities (entity concepts). The composition and reference relationships are the main relationships for this process routing purpose (see section 11.3.2.3). ToolBuilder also provides **derived relationships** to declare extra paths that are not described directly by the product model or ToolBuilder entity model. With this technique, the user can build up navigation routes specific to the required tool environment. The Scratch method illustrates two types of derived relationship. The first type is based on the aggregation of other entity relationships. For instance, the *transitionOut* derived relationship of *transition* is an aggregate of the *transOut* composition relationships from the *startState* and the *interState* as shown below (refer to figure 11.7 for the relationships):

DEFINITION

transitionOut : Seq of transition INVERSE fromState

DERIVED AS :

AGGREGATION

COMPONENTS:

GIVEN: startState

SELECT : transOut

GIVING: transition

AND

GIVEN: interState

SELECT : transOut

GIVING: transition

The second type uses a cascading path of other relationships. For instance, the *unit* derived relationship of *transition* sets up a path to the host *UNIT* entity; through the *fromState* relationship of *transition* (the inverse relationship shown above) and the *unit* derived relationships of *STATE*. The definition is given as below:

DEFINITION

unit : UNIT INVERSE NONE

DERIVED AS :

PATH

COMPONENTS:

GIVEN: transition

SELECT : fromState

GIVING: STATE

THEN

GIVEN: STATE

SELECT : unit

GIVING: UNIT

- ToolBuilder handles referential integrity (as in a database) by means of **propagation** from one entity to another entity through a relationship. A propagation slot accepts EASEL statements which modify contents of other entities by checking that its content is **deleted**, **changed** etc. In Scratch the change of an *event* name propagates to the label of corresponding *transition* through the *triggers* reference relationship.

PROPAGATIONS

WHEN name IS CHANGED

```
foreach transition in transitions := ($[1])."triggers" do
  DE_update_labels(transition);
endfor;
```

- A common example for a user specific extension is the control of naming. For instance, the Scratch method restricts all entity names to be a non-empty alphanumeric string. The attribute type *name* in the *NAMED* entity is defined with the validation code shown below. All named entities in the entity model must inherit from this entity. This control may affect the **validated by**, **updated by** and **show by** slots of the *attribute type definition*.

VALIDATED BY

```
src := $[1];
attr := $[2];
name := $[3];
if matches(name, "^[a-zA-Z][a-zA-Z0-9_]*$") then
  return [TRUE; name];
else
  AI_acknowledgement_prompt (
    "Name must be non-empty alphanumeric string.", "OK");
  return [FALSE; UNDEFINED];
endif;
```

- Following the previous point, non-semantical attributes can be placed in the *entity type definition*. For instance, every *object*, *relationship*, *state*, *event* and *transition* entity in the Scratch method must have a name and a description text. Moreover, the *state* and *transition* entities must have a further section for textual explanation about the actions involved. The *NAMED* and *ACTIVE* abstract superconcepts in the entity model (see figure 11.7) are introduced for these reasons.
- Sometimes it is useful to reinforce concept dependence in the final tool. Apart from checking the right concept type or relationship cardinality etc., a predefined structure can be carried out in various stages of execution. For instance, each *object* in the Scratch method has a *stateTransitionDiagram* to depict its *behaviour*, so it may suggest creating the associated *stateTransitionDiagram* in each formation of *object*:

ON CREATION

```
TBD_release(TBD_create_entity($[1], "stateTransitionDiagram", "behaviour"));
```

and in turn each *stateTransitionDiagram* must have at least one *startState* and one *stopState*. This mechanism provides integrity validation as in a database management system.

ON CREATION

```
start := TBD_create_entity($[1], "startState", "start");
stop := TBD_create_entity($[1], "stopState", "stop");
DE_set_position(start, [256; 32]);
DE_set_position(stop, [256; 640]);
TBD_release(start);
TBD_release(stop);
```

11.3.2 PROCESS MODEL MAPPING

Most metaCASE tools do not represent the method process explicitly. However a task structure can be resolved from the **creation** and **navigation** operations of frames and objects which are based on the entities in the entity model. This structure illustrates the precedence of operations to be carried out in software modelling. Mapping the process model to the metaCASE tool model is done largely by transforming the terminal task functions into these two types of operations in the tool. Various conditions described in each task must be obeyed in the mapping, although the method engineer can form a customised method in the final tool.

11.3.2.1 BASIC PROCESS MAPPING

Table 11.3 shows the mapping of process model components to the semantics of a specific metaCASE tool - ToolBuilder implicitly employs a navigation model to describe task structure (see section 3.4.3). The following points are noted from the table:

- Being a CASE tool, only executable tasks are recorded into the tool semantics. Task functions are mapped to the primitive operations such as *add*, *delete* and *change* etc. in the frame or object menu. The task preconditions and postconditions may be implemented as EASEL statements in the *precondition* and *trigger* slots of the operation respectively.
- The task dependencies are shown by creation and navigation links of concepts described in the product model. In ToolBuilder's terminology, the paths of menu operations are based on the entity relationships that are applied to the object (or frame) operations.
- Most metaCASE tools do not cope with task decomposition or task refinement directly. This is mainly because there are logical dependencies between tasks, and tools are only interested in physical executables. In ToolBuilder, the functions in a composite task should be captured in a single frame, so a navigation to a frame may signify a composition of tasks. In addition, a task refinement embodies the executable task functions in a single concept, and these functions must be denoted as options in a sole frame or object menu.

Category	Process Model in Method	ToolBuilder Navigation Model
task	task	add primary/subordinate, operation
	task function	add, delete, change operations etc.
	task precondition	operation precondition
	task postcondition	operation trigger
dependency	concept token & link	entity relationship
	task trigger	creation/navigation links
	task decomposition	(navigation to a new frame)
	task refinement	(menu options)

Table 11.3 Mapping Process Model to ToolBuilder Semantics

11.3.2.2 FIVE STEPS MAPPING APPROACH

This section describes the detail of techniques for mapping process model based meta models to the metaCASE tool semantic model. The description is presented by a five step mapping approach. Although the ToolBuilder and the Scratch method are used to illustrate the techniques, the same techniques can be applied to other metaCASE tools or methods. The Scratch method has a relatively simple process model. The top level tasks include sketching the *objectRelationshipDiagram* and then the *stateTransitionDiagram* of each critical *object*. The detail specifications in *objectCatalogue* and *eventCatalogue* depend on the availability of *object* and *event* accordingly. Figure 11.9 summarises the method process of Scratch in a task diagram.

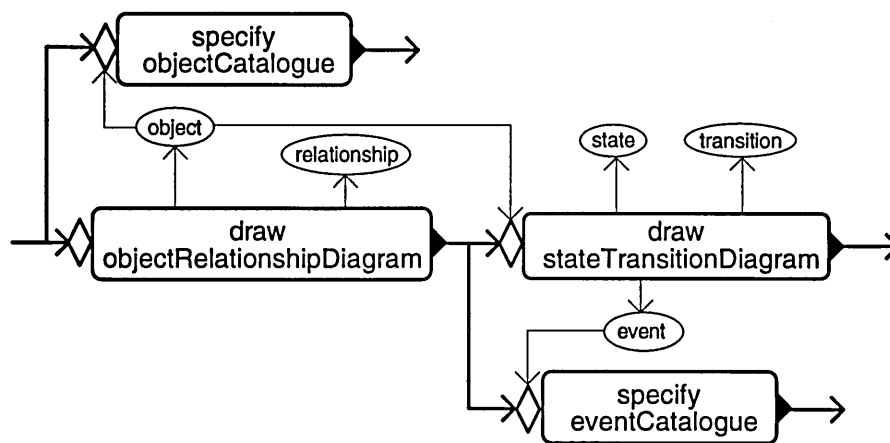


Figure 11.9 Task Diagram for Scratch Method

STEP 1 - IDENTIFYING WHERE EACH CONCEPT INSTANCE ORIGINATES

Each **insert** function in the task sequence is mapped into a 'creation' operation. The concept relationships from the product model give the dependence between concepts, whereas the precedence of tasks in the process model provides the order of execution. These are important guides when determining where each concept instance originates. There may be more than one way to create an entity, but each entity must have at least one creation path. The mapping is simplistic, but each path must comply with the *preconditions* and *postconditions* of the task and the composition relationships in the entity model. For instance, an *object* comprises a set of relationships and the postcondition of the *insertRelationship* task is to form the *relationship* concept token.

```
task(insertRelationship, insert(relationship), [object], [relationship]) .
```

```
heuristic(insertRelationship, [object, relationship],
```

```
'Identify relationships between objects. Any dependency between two or more objects is a relationship; a reference from one object to another is also a relationship.') .
```

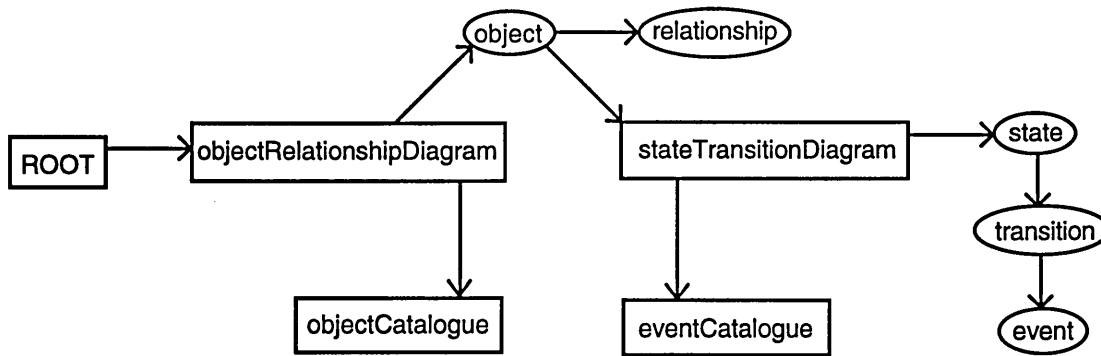


Figure 11.10 Creation Paths of Scratch Method Entities

Figure 11.10 shows the creation paths for the Scratch method entities. A rectangle denotes a frame (a fragment concept), whereas an oval depicts an object (an entity concept). The following points demonstrate the mapping of concealed creation paths based on the associated entity dependence:

- From the ToolBuilder entity model diagram shown in figure 11.7, the creation of a *stateTransitionDiagram* depends on the existence of a corresponding *object*. Therefore the diagram should be formed from its host *object* and not from the *ROOT*.
- In view of the fact that an empty catalogue is possible (say at the very beginning of development), the two catalogues should be preceded by the two diagrammatic fragments accordingly as depicted in figure 11.10. For instance, the formation of *objectCatalogue* is preceded by that of *objectRelationshipDiagram* rather than straight from the *object* concept.
- Scratch defines an *event* as a label of *transition*, therefore a *transition* entity is a logical place to create an *event* instance.

STEP 2 - DENOTING CONCEPT CREATIONS IN THE TOOL

This step is very much tool dependent. ToolBuilder provides three ways to insert method concepts. The first way is by the **on creation** slot in an entity type definition; this technique enables a sequence of object constructions. The second way is through the **propagations** of attributes or relationships. These two ways perform implicit creations inside the tool with specific EASEL codes (as shown earlier).

However, the most common way to insert a new entity is through the **add primary object operation** in a frame menu or the **add subordinate object operation** in an object menu. Unlike the first two ways, these operations require user explicit control. ToolBuilder also allows **precondition** and **trigger** to be inserted for these operations. Precondition detects the validity of the operation by checking the related attribute and/or relationship values, whereas trigger performs a responsive action to the creation. For instance, a Boolean attribute

haveStartState is placed in the *stateTransitionDiagram* object type definition and it is initialised to FALSE until a *startState* is added. A precondition and a trigger is placed in the add object operation of *startState* as follow:

```
ADD OBJECT: startState
[description]
PRECONDITION:
!(value_of_attribute($[1],"haveStartState"))
TRIGGER:
TBD_attribute_update($[1],"haveStartState","TRUE");
```

STEP 3 - IDENTIFYING ADDITIONAL NAVIGATION PATHS

Additional paths are inserted to denote the recursive tasks in software development. Most method processes convey an incremental iterative approach. A hierarchical creation path in metaCASE is insufficient, so the navigation network is enhanced with extra routes and cycles. These tasks should cover all possible terminal functions in the process model, such as *delete*, *modify*, *adjust*, *retype* etc.; apart from the *insert* functions that have been dealt with in the first step. Consider the Prolog clause of task *deleteVagueObject* shown below. An *object* may be found to be vague in its *stateTransitionDiagram* because no *interState* can be defined. Hence, the corresponding *object* in the *objectRelationshipDiagram* must be deleted. A path is placed between the two diagrammatic fragments as shown in figure 11.11⁴. All navigation paths are denoted by thin arrow lines because they are semantically different from the creation paths.

```
task(deleteVagueObject, delete(object), [object, interState], []).
```

```
heuristic(deleteVagueObject, [object, interState],
'An object should be specific. An object without internal states may have ill-defined
boundaries or be too broad in scope.').
```

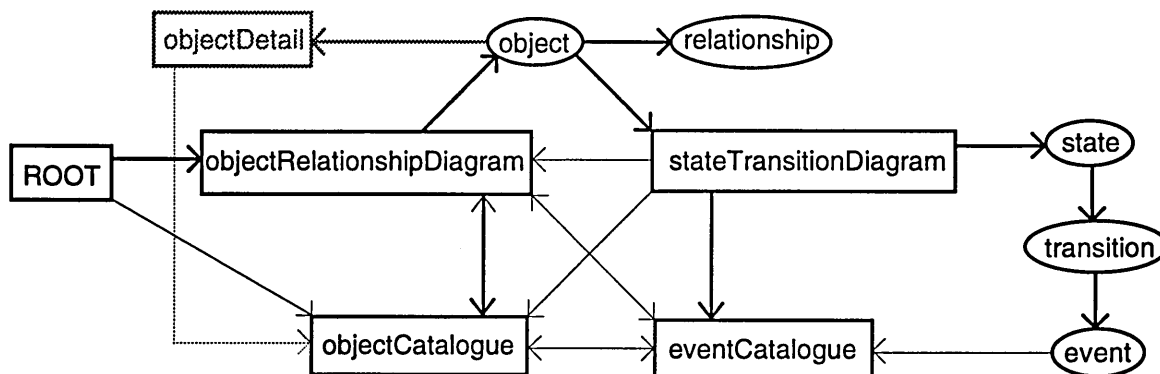


Figure 11.11 ToolBuilder Navigation Model for Scratch

⁴ As mentioned earlier, a number of text fragments are used to describe various entities in Scratch, such as *objectDetail* for *object*, *stateDetail* for *state* etc. These fragments have no vital semantic meaning so they are not depicted in the figure. The *objectDetail* is only shown for a later illustration.

Some paths are method independent but user-customisations. For instance, the initialisation of *ROOT* in ToolBuilder puts up a text window and a diagram window. The developer may find it convenient to place the *objectCatalogue* along side with the *objectRelationshipDiagram* in the beginning, so a navigation path is built from *ROOT* to *objectCatalogue*. In addition, each referencing relationship portrays a loose but significant association between concepts, so a navigation path is normally required. For example, the navigation path between *object* and *objectCatalogue* is composed through the *objectDetail* frame as shown in figure 11.11.

STEP 4 - CHECKING AGAINST SEMANTIC DEPENDENCIES

Before inserting navigation operations into the tool, it is important to check the semantic dependencies of individual tasks. If a path has disregarded the *preconditions* or *triggers*⁵ of the task, the developer has to remove it and replace with other alternative routes that comply with the process model. From the task diagram in figure 11.9, an obvious example in Scratch is the *object* and *stateTransitionDiagram* dependence. Since each *stateTransitionDiagram* is specific to a particular *object*, the only navigation path to a *stateTransitionDiagram* is through its host *object* as shown in figure 11.11.

```
task(drawStateTransitionDiagram, draw(stateTransitionDiagram), [object], []).
heuristic(drawStateTransitionDiagram, [object, stateTransitionDiagram],
'Prepare a state transition diagram for each object with nontrivial dynamic behaviour,
showing events the object receives and sends.').
```

STEP 5 - DENOTING NAVIGATION PATHS IN THE TOOL

Similar to step 2, the denotation of navigation paths is also tool dependent, but there must be a way to represent navigations in the system. ToolBuilder allows the paths to be inserted as navigational operations in the frame type definition, the object type definition and the field type definition. The following code segment shows the declaration of a navigation path in the **operation** slot of an *object* type definition. It defines the path from the host *object* through the *behaviour* reverse composition relationship to the according *stateTransitionDiagram* (see figure 11.7), where the developer is allowed to view as well as to edit the diagram.

```
OPERATION: Show State Transition Diagram
  [description]
HELP TEXT
  To see the object behaviour as a state transition diagram.
FRAME MODE: View/Edit
NAVIGATE TO FRAME: stateTransitionDiagram
ALONG PATH: behaviour
DESTINATION MODE: As-Transaction
```

⁵ The preconditions and triggers discussed in step 4 refer to the task context requirement and task preceding dependence in the process model respectively (see chapter six for details). They must not be confused with the ToolBuilder operation slots in frame/object type definition described in step 2.

11.3.2.3 MAPPING CONCEPT DEPENDENCE TO PROCESS ROUTES

The previous subsections demonstrate that the task triggers in the process model are captured as creation paths or navigation paths (process routes) in the metaCASE tool. The various functions and conditions can also be mapped to tool semantics. Moreover, there is a pattern of mapping the concept dependence in the product model to the entity dependence in the metaCASE tool, and then to the process routes in metaCASE tool. Table 11.4 shows these mappings, and the following points are noted:

- The subtyping relationships are only used to inherit common features between concepts in ToolBuilder. They do not convey to any process routes.
- As shown in section 11.3.1, the composition, linking and grouping relationships in a single fragment are all presented as composition relationships in ToolBuilder. Each of these relationships contributes to a sole creation path.
- The grouping relationships between concepts in different fragments and the referencing relationships describe the significant dependencies in the product model; a sole navigation path must be mapped to illustrate these specific process routes.
- ToolBuilder uses derived relationships to enhance the navigation model with extra routes. They cannot be denoted as creation paths but as navigation paths. These paths are used to fulfil the specific requirements of ToolBuilder or to define user-specific routes. A simple example is the navigation from a *stateTransitionDiagram* to the host *objectRelationshipDiagram* in the Scratch method. Since a single relationship is required in the **along path** slot of the frame operation as shown below, a path derived relationship *unit* must be set up.

OPERATION: Show Object Relationship Diagram
[description]
HELP TEXT
[help]
FRAME MODE: Edit-Only
NAVIGATE TO FRAME: ORD
ALONG PATH: unit
DESTINATION MODE: As-Transaction

Concept Dependence in Product Model	Entity Dependence in ToolBuilder	Process Routes in ToolBuilder
subtyping	subtyping	-
composition	composition	creation path
linking	composition	creation path
grouping (in same fragment)	composition	creation path
grouping (different fragments)	reference	navigation path
referencing	reference	navigation path
-	derived	navigation path

Table 11.4 Mapping Method Semantic Dependence to Tool Process Routes

11.3.3 HEURISTIC MODEL MAPPING

This section discusses the mapping of heuristic clauses into the metaCASE tool, a few of these clauses have been shown in earlier sections. Ideally, a CASE tool provides a hypertext-based system to display heuristics and allows one to associate closely related semantics. There are three aims for heuristics: firstly, give general information of the method; secondly, present context-sensitive guidance of a particular semantic when it occurs; thirdly, provide an error or a warning message if a mistake is detected. Like most metaCASE tools, ToolBuilder does not support such a hypertext system. Although the complex heuristic links cannot be modelled, there are a few ways to show heuristic guidance⁶ inside the tool. The discussion divides into three parts: the concept heuristics, the task heuristics, then the general and error messages.

CONCEPT HEURISTICS

ToolBuilder allows descriptions to be placed in most structured text frames and slots of the method declaration. However, these descriptions are only used as a reminder for method engineer and they will never appear on the generated tool. The ideal location for concept heuristic is the **help text** slot of the corresponding frame type definition or object type definition, since ToolBuilder denotes method concept as a frame or an object. The help information is invoked by a choice on the **frame primary menu** or **object subordinate menu** accordingly. The following concept heuristic of the *stateTransitionDiagram* frame in Scratch can be shown as help text:

HELP TEXT

STATE TRANSITION DIAGRAM

This diagram describes the system behaviour in terms of a diagram showing system states and the transitions that are triggered when events occur. The initial and final states of the system are shown with distinguished symbols. The actions associated with states and transitions are detailed in separate structured text frame.

Related concepts: state, transition, event, event catalogue.⁷

TASK HEURISTICS

When a method help window appears, it includes the help in the concept as well as the help in the navigation operations of the concept. Basic information can be placed in this slot as the example *Show Object Catalogue* operation in *stateTransitionDiagram* demonstrates below:

OPERATION: Show Object Catalogue

[description]

HELP TEXT

To show the catalogue of objects in structured text.

⁶ There is no way to form heuristic links in ToolBuilder, but it is possible to present related heuristic semantics and refer them back to the corresponding locations in the system.

⁷ The *related concept* statement at the end is for reference only, they do not have an actual interactive linking mechanism as described in the heuristic model in the generated CASE tool.

Task heuristics should be shown either passively or at the request of the developer. For instance, no one would like to look at the heuristics of the *verifyObject* task in a pop-up window every time an *object* is created. There are two other ways to manage task heuristics. Firstly, a help **operation** may be placed in the respective concept, so that the information becomes an option in the **object menu**. For instance, the *verifyObject* task heuristics may be displayed in an **acknowledgement prompt** window as declared below:

```
OPERATION: Help on verifying object
TRIGGER
  AI_acknowledgement_prompt ("Delete vague object:\n
  An object should be specific. An object without internal states may have ill-defined
  boundaries or be too broad in scope.\n\n
  Delete implementation construct:\n
  Constructs extraneous to the real world should be eliminated from the object relationship
  diagram.", UNDEFINED);
```

Secondly, the EASEL **output** function presents information in a static message window (see appendix B) along side with the text and diagram windows in ToolBuilder. The **on creation** slot of the *entity type definition* may show the same heuristics but in a passive manner:

```
ON CREATION
  output ("HELP ON VERIFYING OBJECT\n\nDelete vague object:\n ...", INFORMATION);
```

GENERAL INFORMATION AND ERROR MESSAGES

General information may not appear at any time but at the beginning of each new design launched. Therefore, the appropriate location to place the message is the **on creation** slot of the *UNIT* entity definition. For instance, the general information of the Scratch method may be shown by the **acknowledgement prompt** function in *UNIT* entity definition as below:

```
ON CREATION
  AI_acknowledgement_prompt (
  "This is a CASE tool for the Scratch method.\n
  Scratch is a simple object-oriented method, which consists of:\n
  - Object Relationship Diagram : to capture information models;\n
  - State Transition Diagram : to capture dynamic behaviour;\n
  - Object Catalogue : giving summary of objects in database;\n
  - Event Catalogue : giving summary of events in databases.", UNDEFINED);
```

In addition, error messages should necessitate an interactive response to draw the attention to human mistakes or the concept constraints required by the method model. For instance, a macro definition *checkTransitionValidity* may be employed to verify a *transition* by checking the existence of its *fromStart* and *toStop* relationships (refer to figure 11.7):

```
define checkTransitionValidity
  if (DME_rel_defined($[1], "fromStart") & (DME_rel_defined($[1], "toStop"))
    AI_acknowledgement_prompt ("Cannot link start state to stop state!", UNDEFINED);
    return FALSE;
  endif;
  return TRUE;
enddef;
```

Heuristic Model in Method	ToolBuilder Help / Information
concept heuristic	help in frame/object definition
task heuristic	help in operation or EASEL code
heuristic name	frame name or part of operation name
heuristic link	-
-	general information & error messages

Table 11.5 Mapping Heuristic Model to ToolBuilder Semantics

To conclude, the mapping of the heuristic model to ToolBuilder Semantics can be summarised as table 11.5. Since the general information and error messages are specific to each user requirement and also tool dependent, they are not normally included in the heuristic model.

11.3.4 MORE POINTS ON MAPPING SEMANTICS

Apart from the semantic mapping discussed in the above, there are additional points to make on meta modelling and the ToolBuilder metaCASE tool.

- Scratch does not clearly define its internal task structure, including task composition and refinement techniques. However, it is possible to imitate similar tasks from other methods, for instance the *identifyObject* task described here follows the *identifyClass* task in OMT:

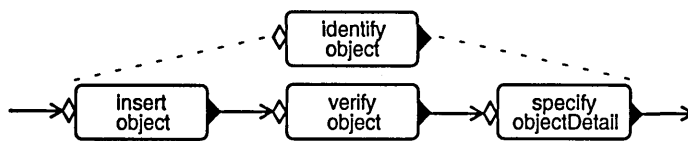


Figure 11.12 Scratch: *identifyObject* Task

- ToolBuilder also supports textual and graphical design of semantic notations, which are the representations of different entities. For instance, the font and the size of an *event* label; the **link style** of a *transition* and the **shape set** of a *state* (see appendix B for the description about shape model and graphics primitives). Nevertheless, these definitions are only the presentation of concepts, they are not important in method modelling.
- ToolBuilder allows a customised format for a text fragment, such as *objectCatalogue* in Scratch. This structure text is declared by a simple grammar in **local subsection** and shared **subsection**, for instance, the following subsection defines the layout of an object name. Again, these declarations are for presentation and they have no semantic interest.

```

Subsection objectName Applies To object
Is Concrete
[description]
name Label "<name>" [ 1 , 20 ] ;
End

```

11.4 CASE STUDY B - BOOCH91 METHOD

In the practical world, most developers may work round a purist method semantic by customising their own 'running version' of the tool model. The main emphasis of this matching exercise is to indicate the discrepancy in method modelling with and without a meta model, and the compromise in logical design to practical tool.

Booch OOD is a well known design method as discussed in chapter two, the corresponding concept diagram and task diagram are shown in appendix D and figure 9.6 respectively. The method embedded in the generated Booch91 tool in this matching experiment is a cut down version of the original Booch OOD. It comprises only the *classDiagram*, *objectDiagram*, *stateTransitionDiagram* and their corresponding structured text templates. This tool is intended to be an executable product and it has an extension to code definition, which makes a bigger contrast with the previous case study. Again, the following discussion is made in view of the product, process and heuristic model matching.

11.4.1 PRODUCT MODEL MATCHING

The discrepancy between the method model and the metaCASE tool model is easily illustrated by comparing our product model with the metaCASE tool model. In ToolBuilder, the tool model is described by an entity model schema. Figure 11.13 shows the entity model diagram of Booch91 developed by a method engineer. The following points are noted in the matching:

- Product modelling depends upon associating concepts by meaningful primitive relationships, but a metaCASE tool is more concerned about objects displayed in frames or navigating entities in the system development process. Hence, **composition** and **referencing** are the two most crucial relationships described. The former is for a creation path and the latter is for an additional navigation path. Other concept relationships are either hidden or replaced by them. In this experiment, the Booch91 tool semantic happens to be described by a purely composition relationship in the entity model diagram.
- IPSYS advises ToolBuilder users to group all possible links and nodes as subtypes of a single entity, so that their dependence can be modelled correctly. The classic example is shown in the *startState* to *stopState* concept constraint of the Scratch method. This requirement is also encountered in the Booch91 entity model. For instance, *classNode* is introduced to embody the *classCategory*, *classUtility* and various types of *classes*; *objectLink* comprises of *outsideSystem* and *insideSystem* links. These abstract concepts are introduced so as to share common features of the subtype nodes or links only. Thus they do not appear in our product model.
- In the declaration of the tool model, some concepts are renamed to suit the new context environment such as *stateLink* is actually referred to *transition* in *stateTransitionDiagram*.

- The *timingDiagram* in Booch is a useful fragment to depict the relative order of *messages* invoked in each *object*. However, many developers are only interested in the *operation* dependence between *objects* which may also directly map into the final code. In addition, the interrelationship of *messages* with time scale references are more difficult to model in ToolBuilder, since it is an entity-relationship (node-link) based modelling tool. Hence, the *timingDiagram* is not included in the Booch91 tool due to this technological limitation.
- Some metaCASE tools also manage code generation. Implementation constructs may be inserted as semantics to associate with concepts already in the model, such that the final tool can act more efficiently. Booch91 declares the hardware *platform*, software *driver* and *application* in the *objectDiagram*. In addition, the corresponding *methods* and *variables* are declared in the *classCompartment* of the host *class*. The following *class method* operation in the *classCompartment* object creates a method marker on the code, which triggers a C macro with embedded EASEL statements.

```

OPERATION: class method
  [description]
  HELP TEXT
  [help]
SELECTION MODE: Single-only
FRAME MODE: View/Edit
TRIGGER:
  create_class_method_marker($[1]);
NAVIGATE TO FRAME: [*frame_type*]

```

- Some relationships can be simplified by **exclusion**, **inclusion** or **overriding**, such as the massive compositions from *classDiagram* to different types of *classNode* and from the *classCloud* to different types of *classLink* in the product model. However, ToolBuilder requires specific role names on each relationship to declare a particular navigation path.
- Most concept constraints are hidden in the product model, but are formally programmed into various parts of the metaCASE tool. For instance, they are declared by **on creation**, **precondition** or **propagations** slots in ToolBuilder. Similarly, the shared concepts are denoted as shared object types in the frame model and not indicated as an entity type.

To conclude, this product model matching illustrates three main reasons for the semantic discrepancy between the two different models. Firstly, a method is a logical model that may not perfectly fit into the physical pattern of a metaCASE tool, so some fine adjustments of the method model are necessary. Secondly, some method semantics may be inappropriate to the developers in their particular work environment and/or problem domain. Hence a true representation of the original method is not really required. Thirdly, additional concepts may be induced into the method such that the tool can perform more effectively for its job. Moreover, human interpretation is another major factor in semantic mismatch; two developers analysing a method will always come up with slightly different models. Since this implicit effect is disputable (see section 10.4.3), it is ignored in the above discussion.

11.4.2 PROCESS MODEL MATCHING

Booch OOD has only a brief description of the method process model, but the concept dependence in the product model governs the task precedence in software development. Such constraints are raised as preconditions and triggers in task functions, which must be denoted in the corresponding tool operations. Section 11.3.2 presents a general approach to map method tasks to the metaCASE tool by addressing the essential creation and navigation paths. In this experiment, the emphasis falls on matching functionality and matching dependence. Before discussion, it is necessary to depict the navigation model of Booch91 as in figure 11.14. A few points are noted about the model:

- The aim of this navigation model is to illustrate the main routes in the tool, hence some details are missed out to reduce the diagrammatic complexity, such as the subtype-entities.
- The navigation model has a very close relationship with the entity model, since the creation paths (thick arrows) are obliged to follow certain composition relationships. Indeed, ToolBuilder does not address tangible frames (such as *designCatalogue*) and fields (such as *designName*) in its entity model, but they are referred by their base entities. This is because ToolBuilder only presents the essential concepts in the entity model, but the navigation model must show each required visible appearance of each entity. For instance, the *designName* field is a presentation of the *design* entity.
- The current version of the Booch91 tool defines *platform*, *application* and *driver* entities as components of *object*. It is believed that they should go under the *design* instead.

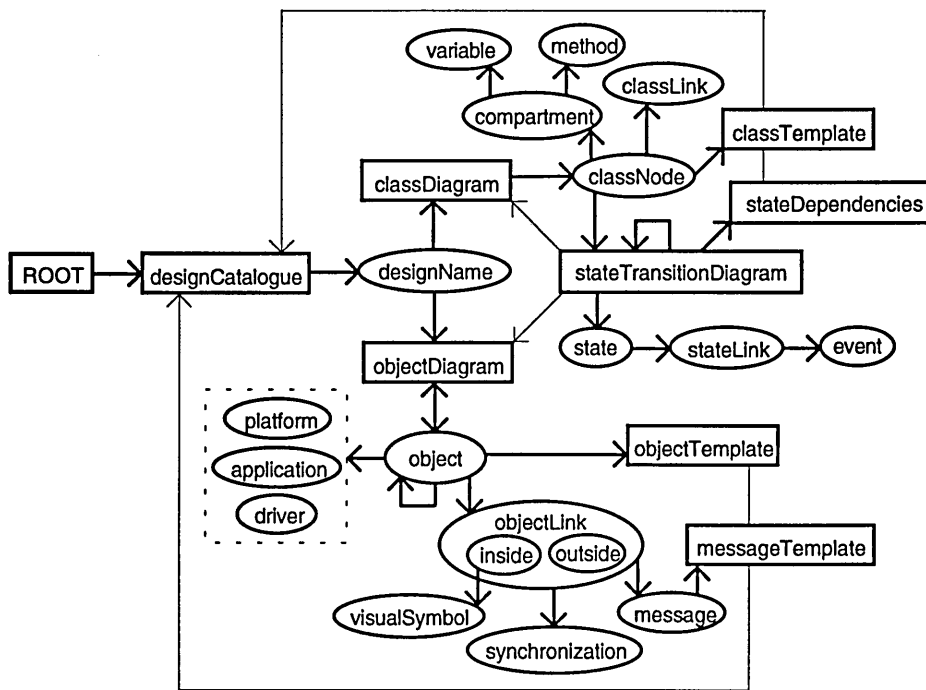


Figure 11.14 Booch91 Navigation Model in ToolBuilder

- The bi-directional creation link between *objectDiagram* and *object* indicates that a group concept (i.e. *objectDecomposition*) is required in the product model. This is because a recursive creation path is not a trivial relationship and it normally describes a different level of concept presentation, such as the different level of *objectDiagram* in this case. There must be a concept incorporated to complete the chain or loop.

MATCHING FUNCTIONALITY

A process model describes tasks by their functions and interconnections (e.g. pre- and post-conditions). All task functions are noted and matched with the ToolBuilder navigation model. A simple classification system is observed in the matching exercise, this will help to understand the different levels of functionality offered by the generated tool. ToolBuilder distinguishes concept as a frame or an object⁸. In order to reduce confusion with the ‘object’ concept in the Booch91 method model, the following discussion refers to the ToolBuilder frame and object as fragment and concept respectively⁹. The three classifications are recorded as below:

- If a task demands a transformation on another fragment, a path must be provided from the current concept or fragment to the target fragment. These task functions are usually represented by the composite tasks such as *perform*, *do* and *draw*. If the context included does not occur, it must follow a creation path otherwise a navigation path must be provided. For instance, the *drawClassDiagram* task signifies a request to launch the *classDiagram* frame, so the invoked concept (*design*) must have a path to that fragment.
- If a terminal task calls for a concept construction such as the *insert* and *specify* task functions, a creation path is obviously required. The *insertClass* task demands a creation path from the current fragment (*classDiagram*) to the *class* concept, whereas the *specifyClass* task demands another path from *class* to *classTemplate*.
- If a task asks for modification of a concept within the same fragment, it does not normally require a path or it can reuse the creation path described in the previous point. These functions, such as *refine*, *delete* and *adjust* etc., have no direct navigation means, but the guidance acquired by the functions are usually presented as task heuristics. Moreover, ToolBuilder checks the validity of existing frames and transverses if necessary. For example, when an *object* is deleted and the corresponding *objectTemplate* is on display, the text window may traverse to the root structured text frame *designCatalogue*.

⁸ Only the presentable form of a concept is considered in the navigation model, so this is referred to as an *object*.

⁹ Strictly speaking, ToolBuilder introduces a third concept type known as field. A field is a terminal concept in a tool fragment. Since the semantic dependence of field has the same effect as object, it is considered as object in this discussion. On the whole, a frame, an object and a field refer to a fragment concept, an entity concept and a property concept defined in our product model respectively (see chapter five).

Table 11.6 summarises the classifications in matching functionality. The target type (second column) is normally denoted as a context parameter in the respective task function.

Function Type	Target Type	Path Type
perform, do, draw	another fragment	creation, navigation
specify insert	another fragment concept in current fragment	creation
refine, delete, adjust etc.	concept in current fragment	-

Table 11.6 Matching Functionality

MATCHING DEPENDENCE

This subsection discusses the task dependence, which will dictate the pre- and post- conditions in the task functions. For instance, the Booch91 method always refers to classes and objects together in the discussion of a design process. Thus, the *drawClassDiagram* and *drawObjectDiagram* tasks have no dependence and they can be performed concurrently. In other words, there is no constraint to bind their order of execution. Moreover, the *insertClass* task function describes the insertion of a *class* in a *classDiagram*, the dependence is denoted in the corresponding MSL statement as:

```
task insertClass insert(class) ;
    precondition [classDiagram] ;
    postcondition [class] ;
```

In the following matching exercise, the dependence is described according to the fragment and concept types. The four categories of dependence are identified as follows, and they are summarised in table 11.7.

- **Fragment-to-concept dependence** - a typical parent and child dependence where a concept is found inside a fragment. This relation always implies a composition amongst the two semantics and a creation path in the tool navigation model. For instance, a *classNode* can only be inserted into a *classDiagram*. The dependence is denoted by the composition relationship between them.
- **Concept-to-fragment dependence** - an unusual dependence that reflects the decomposition of a host concept to another fragment. It is normally represented by a group concept or a referencing relationship. The denotation depends on the nature of the dependence. For instance, the *stateTransitionDiagram* is employed to document the dynamic behaviour of the associated *class*. Therefore it has a clear dependence from the host *class* to the respective *stateTransitionDiagram*. This type of dependence is also applied to the navigation path between a concept with its comprehensive template, such as the *class* concept and its *classTemplate* fragment.

- **Concept-to-concept dependence** - the general connection between two concepts which signifies a creation path if both concepts are in the same fragment. For instance, each *class* comprises properties such as *cardinality*, *concurrency* and *persistence* etc. The dependence is denoted by the composition relationships from the host *class* to the property concepts. The linking relationships can also be used to depict this type of dependence, such as the *stateLink* amongst the *state*. Moreover, if the concepts are from different fragments, the path is shown as a referencing relationship in the product model. There is no such example encountered in the Booch91 method.
- **Fragment-to-fragment dependence** - there are two possibilities for this type of dependence: firstly, a referencing relationship between two closely related fragments; secondly, an association between graphical and textual fragments for mutual clarification. For instance, the dependence between *stateTransitionDiagram* and *stateDependencies*, illustrates a cross referencing relationship between the diagram and structured text fragments respectively.

Dependence	Product Model Relationship
fragment-to-concept	composition
concept-to-fragment	grouping, referencing
concept-to-concept	composition, linking, referencing
fragment-to-fragment	referencing

Table 11.7 Matching Dependence

It must be stressed again that the navigation model is a tool-based process model, which is not actually present in the original method. In other words, a navigation model is a special mechanism to describe tool operations that reflect the task functionalities and dependence. The model is meant to be generic, although some slight adjustments may be required from one tool to another tool. Section 11.3.2 demonstrates a stepwise approach to map method tasks to navigation paths and this section, reinforces the approach by matching the inner parts of a task to a metaCASE tool model.

11.4.3 HEURISTIC MODEL MATCHING

There is no heuristic model in the current version of the Booch91 tool. However, the concept heuristics may be displayed actively in the help window and the task heuristics may be shown passively in the message window (see section 11.3.3 for details).

Nevertheless, this indicates the mistreatment of heuristic information by the method engineer, which makes the CASE tool a notational presentation software package rather than a package inhabited with the full knowledge of the source method model.

11.5 CONCLUSION

MetaCASE technology is very important in method engineering, since it applies meta modelling techniques as well as advocating an improvement of the system development method. The three components in a method model, namely the product model, the process model and heuristic model, are mapped into the semantics of the available metaCASE tool, IPSYS ToolBuilder. The resulting metaCASE tool model is assumed to be portable to other metaCASE tools. Proof of this assumption is a subject for further work.

12. CONCLUSION

This thesis has presented a generic model (GMR) for representing software development methods. The GMR promotes the hypothesis that meta modelling is a strategic and advantageous activity. To conclude this investigation and experimental work, the goals and assumptions are recapped.

12.1 INTRODUCTION

Previous researchers have found that, in general, there are two approaches to unify software development methods; standardising their common interface and enforcing a meta model formalism.

The former is commonly known as method integration, such as CDIF and PCTE. They provide levels of integration to transfer or to share portable components amongst methods. They stress information exchange rather than modelling method semantics. The approach is adequate for passing data or process between methods, but the outcome is far from satisfactory. These standards are surpassed in competing with software platforms, such as OLE and DDE in Microsoft Windows [Boyce 92].

The latter approach includes both meta modelling research and metaCASE modelling. Most research projects in this field specialise on a particular aspect of meta modelling formalism, for instance assisted software processing in ALF-MASP or conceptual task modelling in SOCRATES. The method representation is neither precise nor concise. Method heuristics might be given little or no consideration. Most metaCASE tools do, however, provide an entity-relationship model to describe the concept relationships of methods and tool operations to denote the task functions. Much of the semantics get hidden in the CASE tool language, or sometimes even ignored. In addition, being a CASE tool, the meta model is biased to allow implementation of constructs or ease of coding.

Through our GMR, method semantics can be extracted from an expert and formalised into an executable form stored in a semantic knowledge base. The meta model, including the products, the processes and the heuristics, is shown to be advantageous in method comparison, fragment dissection and in the selection of methods. The potential benefits of this work are transferability between existing methods and the automation of the generation of a CASE tool to support new and evolving 'customised' methods better suited to the requirements of specific applications. It could also form the core part of a future method of an evolutionary prototyping approach to software engineering.

The conclusion is given in two main parts: firstly, the achievements of GMR to date are shown; secondly, the discussion of possible future work where GMR could be utilised in strategic software development.

12.2 CURRENT ACHIEVEMENT

This thesis has discussed the GMR in detail. The three basic components of a method, including their internal components and interrelationships, are supported with both textual presentation (i.e. MSL) and graphical denotation (i.e. diagrams). The model is compiled into Prolog clauses, making extensive experimentation possible. The following points emphasise the current achievements of meta modelling using GMR formalism:

- The GMR model is a uniform representation of software development methods. It supports a standard repository of semantic information about methods that is termed a 'semantic knowledge base'. We validate the model with five common but complex methods. These methods are carefully chosen so as to illustrate a wide spectrum of modelling techniques. And yet, there are some similarities across the methods to illustrate the method comparison. The GMR is also shown to be a practical representation for mapping the logical meta model into a physical metaCASE tool. A proficient tool (i.e. ToolBuilder) and two methods with different levels of complexity are chosen to demonstrate the skills and techniques required.
- One of the major achievements in GMR is fragment dissection. Most meta modelling approaches give no guidance for the dissection of portable fragments but GMR explicitly depicts the 'cuts' of methods. In the product model, the concepts of *grouping* and *referencing* relationships denote the separation of fragments. In the process model, *task function* types and *decomposition* indicate a similar distinction. Not only are the links between fragments highlighted by these features, but their shared concepts are also expressly denoted in GMR.
- Due to the uniformity of GMR, two types of method comparison are available. Firstly, the *numerical comparison* is based on the statistical information from MSL, and it measures the emphasis and complexity of methods. Secondly, the *fragment comparison* evaluates the techniques or modelling features captured in the associated method fragments. This is particularly useful when selecting appropriate fragments for a problem requirement. The internal measuring and matching mechanism has yet to be explored.
- The generic representation forms a backbone for method evaluation, which anticipates the selection or customisation of methods. An evolutionary approach is foreseen for future work, supported by general hints and guidance as revealed from the GMR viewpoint.

- A knowledge acquisition model is invented for eliciting method semantics. It is named 'IFV' to signify its three-stages: *inspection* from acquisition media; *fabrication* of a conceptual form and *verification* of the resultant model. IFV is found to be an effective approach during the acquisition phase of the five chosen methods.

On the other hand, three fundamental questions have to be considered:

- *Is GMR a 'perfect' model for representing methods?* At the very beginning, this research claimed that there is no single perfect method that suits all problem domains and/or work environments. If the statement is true and GMR is a modelling technique at the meta level, then it cannot be an absolute model for representing all methods. However, the answer to the question must be negative. There are many methods in the world. Even at the method level, there are different types of methods to satisfy various needs of application. This research focuses on class-based methods, since these are widely used in the software industry - including our collaborating establishment. We make no claims for other areas in the method spectrum, although KADS (a frame-based method) seems to be amenable to GMR formalism.
- *Can GMR be extended or evolved in the future as technology advances?* The invention of the GMR is based on three factors: our prior experience, an investigation and experimentation. Firstly, our previous experience of class-based modelling gives us a prima-facial reason to suspect that this is a suitable formalism for method specification. Secondly, the investigation of available software development methods and meta modelling techniques gave us precious strategic information to formulate the generic representation. Lastly, the GMR is evolved from a continuous 'trial and error' approach through experimentation with five chosen methods. Technology will not stay static. The above three points show that GMR is designed with up-to-date and relevant modelling techniques and should be well placed to be extended as the technology advances.
- *Is there any formal proof of GMR?* The verification stage of the IFV model introduced five techniques (the 5 'C's) to validate each method representation. However, a full proof [Norcliffe 91] of these models is impossible because of the creative and open-ended nature of software development. At the meta modelling level, proofs are even more difficult. Firstly, it is impossible to have an exhaustive experiment covering all software development methods. Secondly, minor adjustments (new versions) may gradually evolve in each method as the technology advances, or more modelling formalisms may emerge. The proof will never be complete. And finally, most methods have hidden semantics and missing concepts, partially due to the nature of development and partially to allow the designer some freedom to adopt other techniques. The method description especially is normally incomplete. Hence our aim was to obtain a generic and workable representation for the available modelling techniques. We have demonstrated that it functions within the

scope of our chosen area, i.e. the classed-based methods. Our findings, are that the GMR is strong in important areas of meta modelling, in particular in method comparison, fragment dissection, and the selection of methods.

12.3 FUTURE WORK

A number of further research tasks can be identified. Many new projects could be based on an extended practical application of the method representation. The work of the current project could also be continued, with tasks such as:

- Due to the limitation of time, only five methods have been chosen for detailed experiment. It would be possible to ratify GMR further by looking at more methods.
- The main scope of this research lies on class-based software development methods. Other parts of the method spectrum could also be considered to expand the boundary.
- If more metaCASE tools or meta modelling systems become available, it would be worthwhile to extend the investigation to these formalisms.
- The mechanisms used to demonstrate various techniques for method comparison and fragment dissection are fairly primitive. It would be possible to extend the experimentation with better and more sophisticated algorithms, so that the performance and consistency of outcomes are better.

However, the two major areas for research are ‘gathering of information about the problem domain’ and ‘analysis of design decisions’ as mentioned in chapter one. The GMR can be used as the basis for their representation. For instance, *objectOriented* is an analytical concept for design and it comprises four basic elements as shown in the first order predicate below:

objectOriented(abstraction,encapsulation,inheritance,polymorphism).

Elements of the predicate can be decomposed into concepts presented in GMR, such as the *encapsulation* concept which itself embraces three other concepts:

encapsulation(class,attribute,operation).

Factorisation of these predicates is also possible, for example the *objectOriented* predicate can be rewritten as *objectBased* with an extra *inheritance* semantic:

objectBased(abstraction,encapsulation,polymorphism).
objectOriented(objectBased,inheritance).

Hence the complex problem information and analytical data lead to method semantics. Some expert system features can be adopted in the processing, for instance, weighting factors and uncertainty, can be introduced to each element of concern. This numerical information can be useful for matching algorithms in the evaluation and selection of methods. Although it is still

far too early to describe how to carry out the problem analysis, GMR provides a potential direction as the back-end representation of possible semantics. More stimulating ideas are required to fill this uninhabited area within software development.

12.4 CONCLUSION

Meta modelling is an advantageous activity, as it enables methods to be used unambiguously. GMR supports this by providing a uniform representation of software development methods. The benefits include method comparison, fragment dissection, selection and customisation of methods. Apart from extending the exercise illustrated in this thesis with more methods and available metaCASE tools, there is potential for future work which will include the 'gathering of information about the problem domain' and the 'analysis of design decisions'.

APPENDIX A: GLOSSARY

abstract concept - a concept that has no formal notation; this concept shall be inherited by other concept(s) and denoted in a method model.

aggregate fragment - a concept that cannot be classified as either diagram or text fragment, but it is an aggregate of a number of entity concepts.

CASE tool process - a specific process of a method in a generated CASE tool which has been configured by the developer.

chronic reading - the second step of method inspection; detailed understanding of method knowledge acquired.

completeness - a verification of a method model against meta-knowledge that checks the constraints between two concepts rather than the definition of a single concept.

composition relationship - a bidirectional whole-component relationship, sometimes it is known as containment relationship; a concept contains another concept, if the concept is a component of the other, or a concept exists within another concept.

concept - a fundamental idea that can be applied to the development of a software system.

concept diagram - a graphical representation yielded from concept modelling.

concept heuristic - the heuristic information of a concept.

concept relationship - an association between two concepts; in product modelling, concept relationship may be subtyping, composition, referencing, linking or grouping.

concept token - a type of semantic token based on concept.

conceptual model - an outcome generated from method inspection, it comprises of a set of diagrams and structured texts.

conclusive reading - the third step of method inspection; overall understanding of method description.

concrete concept - a concept that is constructed primarily by inheriting from other concept(s) and rarely adds its own concept.

consistency - a coherency check of semantics within a method (opposite to that of contrast).

constrained concept - a concept with a constraint (or constrained) rule.

constraint rule - a formal condition (or rule) in a method.

- context parameter** - a signature of a task function to specify the parameter(s) as semantic token(s).
- contradiction** - a method verification which involves two or more semantics, such as the relationship between two semantics affecting a third semantic.
- contrast** - a comparison check of semantics with corresponding semantics in other methods (opposite to that of consistency).
- correctness** - an agreement with the definition of meta model, that is to check the method model does not violate the meta-knowledge of the generic model.
- cursory reading** - the first step of method inspection; introductory understanding of method description.
- diagram fragment** - an explicit concept, normally known as a graphical tool fragment.
- dissection set** - a set of concepts that appear in both fragments of a method; these concepts must be distinguished in tool dissection.
- entity concept** - a basic concept used to describe an idea (or notion) in a method.
- entity relationship model** - a data model based on entity and relationship; for instance the entity model in ToolBuilder.
- formalised model** - a textual form of method model resulting in method fabrication; it enables us to present and modify the method models captured.
- fragment** - an autonomous tool or technique comprised of its own concept structure, task structure and heuristic information. Both product and process models of the fragment must be self governing or unconstrained, although they may have a limited interface to the external development environment.
- fragment concept** - a high level entity concept, which can be a diagram fragment, a text fragment or an aggregate of other concepts.
- Generic Method Representation (GMR)** - a generic representation of a software development method; it is comprised of three basic models, namely product model, process model and heuristic model.
- generic representation** - a representation that is general to the domain of interest.
- group concept** - an association concept which relates the host and the element concepts in a grouping relationship.
- grouping relationship** - an association between a group concept to its host concept or element concept, which is distinguished by a 'h' or 'e' label in concept diagram respectively.

heuristic - a guidance or a criterion on a method semantic (such as concept or task).

heuristic link - a cross reference between heuristics.

heuristic model - a model to describe the semantic assistant embedded in both product model and process model.

heuristic rule - a formalised rule for representing heuristic.

heuristic text - a descriptive body of a heuristic; it may also be known as heuristic guidance.

IFV model - a model for method knowledge acquisition; IFV stands for the three phases approach: inspection, fabrication and verification

KADS - a structured knowledge engineering approach towards knowledge base systems development.

knowledge acquisition - an expertise acquisition concerning the knowledge of interest.

knowledge elicitation - one of the crucial activities in method knowledge acquisition, which involves direct or indirect communication with the domain expertise;

link concept - an association concept which relates two entity concept instances together, one as its source and the other as its target.

linking relationship - an association between a link concept to its source entity concept or target entity concept, which is distinguished by a 's' or 't' label in concept diagram respectively.

meta model - a model to represent method knowledge by a set of formally defined techniques.

meta modelling technique - a suitable and formally defined technique for the representation of method knowledge.

metaCASE process - a physical process model of a method, which is embedded in a metaCASE tool; it is highly dependent on the semantics of the tool, such as the data model to represent concepts and the functional model to describe task.

metaCASE tool - a CASE tool to develop other CASE tools.

method - a set of semantics and techniques for development; in this thesis it refers to software development method.

method acquisition media - a special set of media for acquiring method knowledge.

method engineering - a systematic engineering approach towards method modelling.

method fabrication - the construction of a method model.

method inspection - the examination of method knowledge.

method integration - method semantics sharing based on information exchange.

- method knowledge engineering** - a systematic knowledge engineering approach towards method modelling development.
- method model** - a model used to represent a specific method.
- method notation** - textual or graphical representation of a method semantic.
- method process** - a logical process in a method to show the design sequence.
- method semantic** - a method concept, task or heuristic.
- method verification** - the justification or checking of a method model.
- methodology** - a study of methods.
- model** - an abstract description of a system.
- modelling** - the system function description in 'logical' terms rather than 'physical' terms; describing what the system does, without giving detail of how it does it.
- navigation model** - a functional model based on navigation; for instance the frame model in ToolBuilder.
- parallel tasks** - the software development tasks that can be processed at the same time.
- process model** - a representation of the dynamic, behavioural aspect of a method.
- product model** - a representation of the static, structural aspect of a method.
- prolog clause** - an executable form of method semantics from the formalised model.
- property concept** - an attribute of an entity concept; it must be owned by an entity concept and itself does not own any concepts.
- referencing relationship** - an association to relate similar concepts, which appears as an alias in a different design aspect.
- semantic knowledge base (SKB)** - a knowledge base to store method semantics.
- semantic link** - an association to represent the semantic token as a task precondition or as a task postcondition.
- semantic token** - a concept token or a task token to describe the semantic relationship between tasks.
- shared concept** - an element of a dissection set; it is a common concept between fragments and each instance of the concept is a component of the fragments.
- software development method (SDM)** - a method used for developing software, which may depend on the problem domain, the work environment or the programming language.

subtyping relationship - a bidirectional relationship between a superconcept and a subconcept; a subconcept inherits the properties from a superconcept.

task - a mechanism that supports the description of generic software process models which can be incrementally and repeatedly instantiated in order to produce particular software process models to specific methods or applications.

task composition - a formation of tasks in a sequential order; the reverse is known as an 'and' task decomposition.

task diagram - a graphical representation yielded from task modelling.

task function - an optional or a mandatory operation in a task sequence.

task heuristic - the heuristic information of a task.

task postcondition - the consequence of a task function in terms of semantic tokens.

task precondition - the guard condition of a task function in terms of semantic tokens.

task refinement - a decomposition of task into a number of potential decisions; it is also known as an 'or' task decomposition.

task sequence - a logical sequence of tasks in a specific method.

task token - a type of semantic token based on task.

task trigger - an event based mechanism that takes the underlying concept of an event and invokes a task function.

text fragment - a structured text or pseudo code fragment, which is normally for detailed description purposes or code generation.

tool integration - method semantics sharing based on data exchange between CASE tools.

ToolBuilder - a metaCASE tool which aims to generate a method based CASE tool.

APPENDIX B. IPSYS TOOLBUILDER

Further to the discussion of the ToolBuilder metaCASE tool in section 3.4.3 and the two semantic mapping case studies in chapter eleven, this appendix presents a more detailed description of IPSYS ToolBuilder. This is supplementary material for those who are not familiar with ToolBuilder, and a reference for other readers. Further information can be found in [IPSYS 92].

B.1 TOOLSET ARCHITECTURE

ToolBuilder is a metaCASE tool for building CASE tools. It is a fully integrated CASE tool that supports any specific software or system development lifecycle. It incorporates both the textual and the graphical presentation of data. In addition it supports multi-user and multi-site operation. Figure B.1 depicts the ToolBuilder integrated windowing system as containing three static frames and sometimes a context-sensitive pop-up window defined by the user.

- **Structured text frame** - manipulates all textual information
- **Graphical frame** - manipulates all graphical notations
- **Messages frame** - displays all messages output by the tool

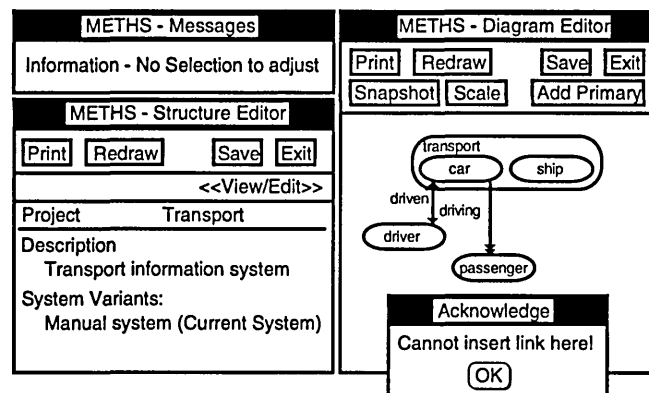


Figure B.1 ToolBuilder Windowing System

- **Acknowledge window** - shows user defined interactive messages (see the above figure)
- **Help window** - shows a collection of help information defined in a frame

ToolBuilder allows programming code to be inserted in various slots of the structured text frames. The language used is EASEL, and the invocation of these slots is referred to as code triggers. Figure B.2 summarises the overall toolset architecture into three models.

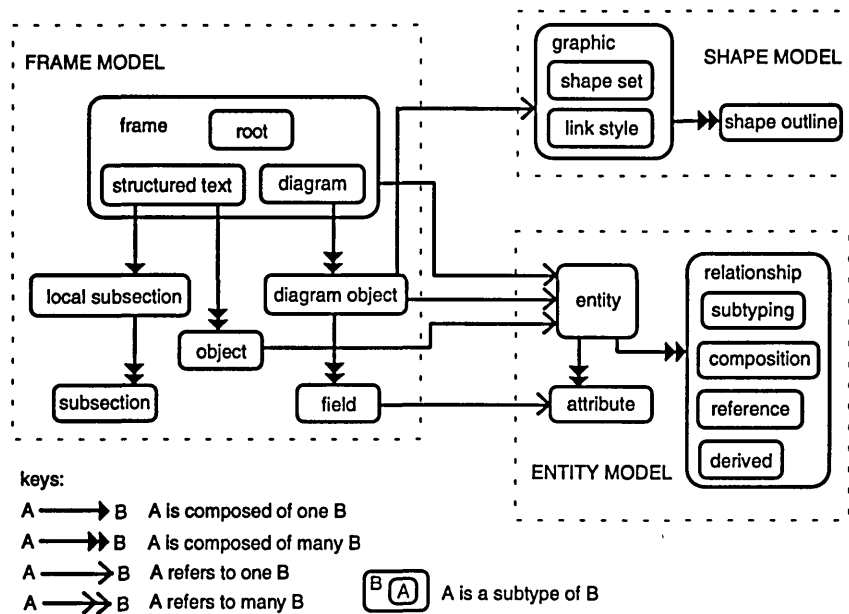


Figure B.2 ToolBuilder: Three Basic Models

- **Entity model** - an ER schema to describe the data model of the generated tool;
- **Frame model** - a set of frames to navigate both structured text and graphical information;
- **Shape model** - a group of shape definitions for diagram objects in the graphical frames.

These models are described in the following three sections. Elements in the models have a literal **name**, and optional **descriptions** can be inserted in various places. In the following description, all slots in the **type definition** portions are 'optional' unless otherwise stated as 'mandatory'.

B.2 ENTITY MODEL

Data modelling in ToolBuilder is based on an **entity model schema**. Data is organised into entities and different kinds of data have different properties known as attributes. A schema is a plan for representing relationships between entities and the associated attributes. The diagram depicting the schema is called an **entity diagram**. The next subsection introduces the three **default entity models** predefined by ToolBuilder, and then describes the basic elements in the entity model: **attribute type**, **entity type** and **relationships**.

B.2.1 DEFAULT ENTITY MODELS

ToolBuilder provides three default entity models - predefined, diagram and document. These entity models grant the necessary inference mechanism to declare other entity models, so they must be inherited by all CASE tools and must not be deleted.

- **PREDEFINED entity model** - defines the predefined entity types, relationships and attributes known to the databases. The predefined attribute types are *Integer*, *String*, *Boolean* and *Date*, whereas the predefined entity type is *entity* itself.
- **DIAGRAM entity model** - controls the graphic presentation. It provides the internal definitions of *DIAGRAM_UNIT*, *DE_FRAME* (editable diagram frame), *NODE* and *LINK* entities, which are inherited by any entity model having a graphical presentation.
- **DOCUMENT entity model** - provides attributes used by the publisher tool and it is inherited by entities which are to be the root of a document.

B.2.2 ATTRIBUTE TYPE

Apart from the four primitive *PREDEFINED* attribute types mentioned in the previous subsection, ToolBuilder allows user defined attribute values. Any extra attribute types are declared in an **attribute type definition** structured text frame, which has the following slots:

- **based on** - based on a previous defined attribute and allows adjustment (mandatory);
- **shown by** - code trigger to show how attribute is displayed in a text frame;
- **shown in edit box by** - code trigger to show how attribute is displayed in an edit box;
- **validated by** - code trigger to validate the entered value;
- **updated by** - code trigger when database is updated.

B.2.3 ENTITY TYPE

There are two types of entities: an **abstract entity** binds concrete entities with common attributes, whereas a **concrete entity** represents a definitive data object. The **entity type definition** structured text frame has the following slots:

- **base type** - (IS A) single inheritance within the entity model (mandatory);
- **inherited types** - (AND IS A) multiple inheritance across different entity models;
- **attributes** - properties of the entity type;
- **composition relationships** - strong associations to form basic building blocks;
- **reference relationships** - weak associations to relate created entities;
- **derived relationships** - relationships built on top of other relationships;
- **subtypes** - other entity types that share common features;
- **propagations** - user defined code trigger upon change, such as modify or delete;

- **on creation** - initialisation code trigger of the entity.

B.2.4 RELATIONSHIPS

ToolBuilder supports four types of primitive relationships: subtyping, composition, reference and derived relationships. Segments of relationship declarations are shown with the diagrams.

B.2.4.1 SUBTYPING RELATIONSHIP

ToolBuilder provides single inheritance (IS A) between entities within the same entity model. Since a supertype entity cannot be instantiated, the relationship is better known as **subtyping**. All features of the supertype entity, such as attributes, relationships and code triggers are inherited to the subtype entities. Each leaf entity must have a base type. If the entity does not inherit from other entity types, then the base type is a default **ROOT TYPE**. Figure B.3 illustrates the two types of inheritance - **IS A** relationship (subtyping) within the entity model and **AND IS A** multiple inheritance across different entity models.

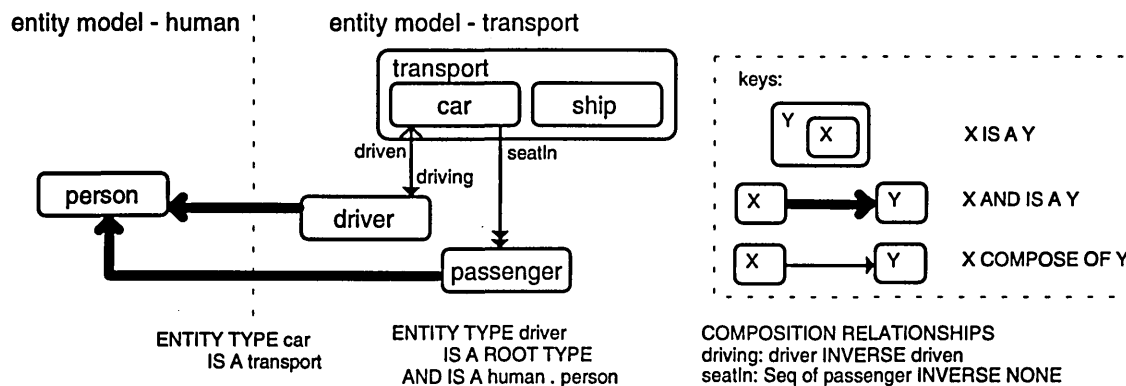


Figure B.3 ToolBuilder: Entity Types and Inheritance

B.2.4.2 COMPOSITION RELATIONSHIP

Composition links entities to form basic building blocks, which form **part-whole hierarchies** in the entity model. When an entity is deleted, all part-entities are also removed. However, an entity can exist without any part-entities. Composition is denoted by a solid head arrow pointing from the whole-entity to its part-entity. ToolBuilder supports **single-valued** (single solid arrow head) and **many-valued** (double solid arrow head) compositions. In addition, the relationship can have a **reverse link** (v-shape arrow head on the target side) from the part-entity back to the whole-entity. For instance, in figure B.3, a *car* has one *driver* but many *passengers*, and the *driver* can refer back to the *car* by the *driven* reverse link.

B.2.4.3 REFERENCE RELATIONSHIP

A reference relationship provides a 'weak-link' between existing entities in the database, because both entities remain even when the relationship is deleted. A reference relationship is depicted as a v-shaped arrow in the entity diagram. Similar to the composition relationship, a reference relationship supports single- or many-value links and reverse links, which are shown by the *toY*, *toZ* and *toF* relationships respectively in figure B.4.

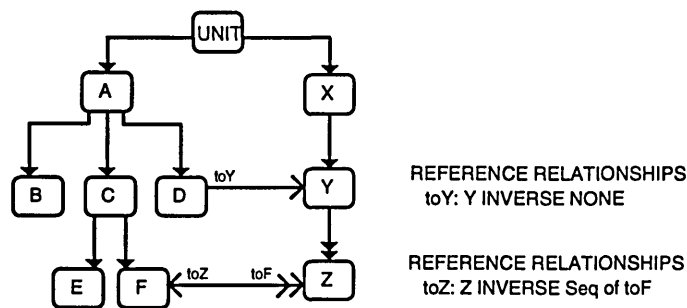


Figure B.4 ToolBuilder: Reference Relationships

B.2.4.4 DERIVED RELATIONSHIP

Derived relationships allow extra-navigation through the database. A derived relationship is not stored explicitly, but is based on composition and reference links, or other derived relationships. Derived relationships are usually not depicted in the entity diagram, but they are shown as dotted arrows in figure B.5 for clarity. The three types of derived relationships are **path**, **aggregation** and **user defined**. These are described below.

PATH - DERIVED RELATIONSHIP

A path is like a relational join. It is based on **cascading component relationships**. For instance, in figure B.5, the *toZ* derived relationship of entity X is 'path-by' the *toY* and *toZ* composition relationships of entities X and Y respectively. This type of derived relationship may also be defined to be followed recursively. A single-valued path composes of only single-valued components, whereas a many-valued path is a set (no duplication) or a sequence (sequenced with possible duplication) of single-valued and multi-valued components.

AGGREGATION - DERIVED RELATIONSHIP

This relationship is formed by **composing together component relationships side by side** and evaluates each applicable component relationship at the entity. It may also be a deferred aggregate of subtype entities. For instance, entities *B1* and *B2* are subtypes of entity *B*, and

the *hostA* derived relationship of the supertype entity *B* is derived from the *hostA* reverse composition links of the subtypes as shown in figure B.5. Both single-valued and many-valued aggregations are provided, and they are updatable if all components are updatable.

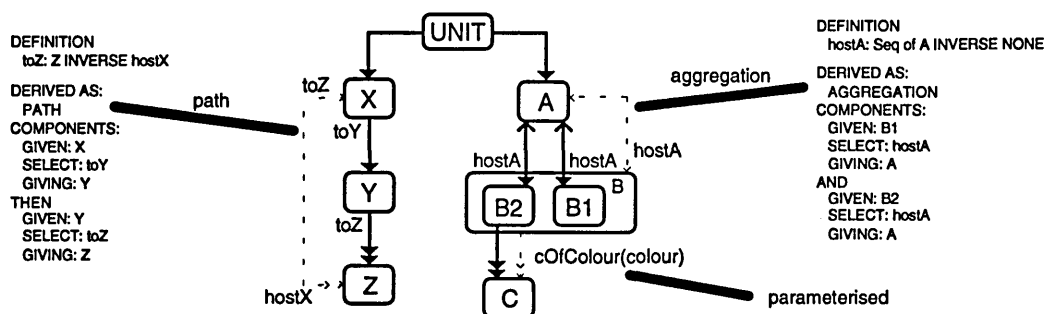


Figure B.5 ToolBuilder: Derived Relationships

USER DEFINED - DERIVED RELATIONSHIP

This relationship allows EASEL code to be inserted whenever the relationship is selected. For a single-valued relationship the user provides TBD_read_rel function, whereas for multiple-valued relationship the user provides TBD_read_rel, TBD_next and TBD_release functions.

ToolBuilder also supports **parameterised** derived relationships in all three types, but EASEL statements must be placed in the relationship definitions. For instance, different entities *C* may be distinguished by the host entity *B2* through the *colour* of the entities, a parameterised function *cOfColour(colour)* can be used in this derived relationship as shown in figure B.5. In addition, the **reverse link** of a derived relationship is permissible if all components in the relationship are bi-directional. Table B.1 summarises the variations of derived relationships.

Type	Cardinality	Options	How to evaluate
path	single-valued	may be <i>parameterised</i> may be <i>recursive</i>	single-valued result of traversing single-valued relationships end to end
path	many-valued	may be <i>parameterised</i> may be <i>set-valued</i> may be <i>recursive</i>	many-valued result of all traversals through single and many valued relationships end to end
aggregation	single-valued	may be <i>parameterised</i> may be <i>deferred</i>	selection of a single valued result from a number of single valued relationships
aggregation	many-valued	may be <i>parameterised</i> may be <i>set-valued</i> may be <i>deferred</i>	aggregation of all results of a number of single or many valued relationships
user-defined	single-valued	may be <i>parameterised</i>	by executing the READ statements
user-defined	many-valued	may be <i>parameterised</i> may be <i>set-valued</i>	by executing the READ, NEXT and RELEASE statements

Table B.1 ToolBuilder: Summary of Derived Relationships

B.3 FRAME MODEL

As shown in figure B.2, there are three types of frames in the frame model. These are root frames, structured text frames and diagram frames.

B.3.2 ROOT FRAME

A root frame is needed for initialisation. The operations in the root frame are the actions invoked by the generated tool when it enters a transaction from the database selection window. These operations are executed in the context of an entity which is the root of the entire database and known as the transaction root. This entity is of the type *UNIT*. Thus, the root frame defines navigation to the initial structured text frame as well as diagram frames.

B.3.3 STRUCTURED TEXT FRAMES

Structured text frames present the textual information about the tool in a structured text window. Every structured text frame has a definition, which includes the following slots:

- **applies to** - particular entity in database (mandatory);
- **help text** - information about the frame which appears with those in the frame operations;
- **object types** - apply to different items in the text frame;
- **shared objects** - same as object types but can be shared amongst structured text frames;
- **operations** - local actions in the frame menu;
- **local subsection** - define appearance of the frame through other subsections (mandatory).

There are two main purposes of frame operations: firstly, navigation to other structured text frame or diagram frame; secondly, adding a structured object in the frame. Each object type (including shared object) defined in a structured text frame is declared by a structured text object type definition which has the following slots:

- **editable/non-editable** - (mandatory);
- **applies to** - particular entity in database (mandatory);
- **help text** - information about the object which appears with those in the object operations;
- **operations** - local actions in the object menu;

Structured text objects are terminal entities in a structured text frame, so the second purpose of frame operations mentioned above is not applicable to object operations. However, the object operations can be used to define code **triggers** and **preconditions** (see later) by EASEL to modify the attributes or relationships of the entity that applied to the structured text object.

The appearance of a structured text frame is defined through a local subsection which may use other subsections. A subsection is a contiguous area of structured text, and common features amongst structured text frames can be stored as a **shared subsection**. There are two types of subsections. An **abstract subsection** only applies to an abstract entity and is used to define how different subtypes are to be represented in different ways, whereas a **concrete subsection** applies to any entity and its structure is defined in terms of:

- **fields** - print a database attribute that applies to the entity in current subsection;
- **text literals** - print a literal string;
- **subsection calls** - invoke another subsection that applies to the same entity as the current subsection;
- **decompositions** - get information from a different entity in a single-valued relationship, results in change of context by navigation;
- **repeating subsections** - similar to decomposition but applies in a many-valued relationship of an entity type;
- **attribute dependent selection** - determine one or more choices based on the value of an attribute;

Fields, text literals and subsection calls can be conditional on relationship states or EASEL preconditions. Figure B.6 illustrates the structure of two concrete subsections.

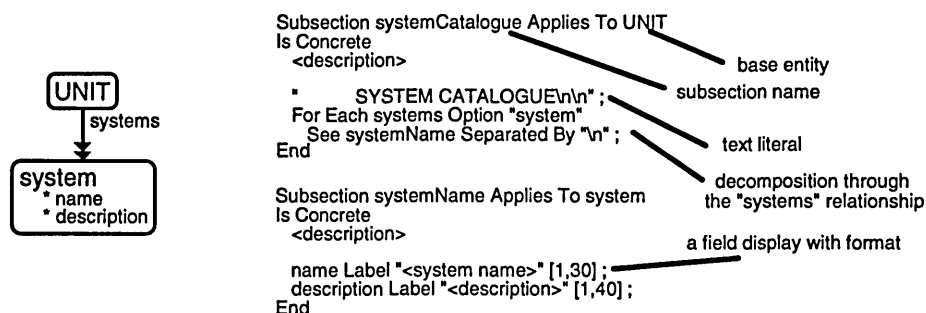


Figure B.6 ToolBuilder: Examples of Subsections

B.3.4 DIAGRAM FRAMES

Unlike structured text frames, diagram frames give the graphical information about the tool in a graphical window. The **diagram frame definition** is as follows:

- **applies to** - the base entity with "AND IS A diagram . DE_FRAME";
- **help text** - show information about the diagram frame in context of the current operations in use;

- **object types** - links to entity model for graphical model;
- **operations** - diagram operation, add primary and frame navigation (see later).

Each **diagram object** shown in the diagram frame must be further specified. ToolBuilder defines a diagram object as a node or a link. The **diagram object definition** is as follows:

- **applies to** - the base entity with 'AND IS A diagram . NODE (or LINK)'
- **presentation** - either a shape set or a formatted text;
- **fields** - handle attributes applied to the base entity;
- **operations** - add subordinate object/field, object operation and navigation (see later);
- **built-in operation specification** - such as selectable, move, delete, add waypoint, etc.

In addition, each **diagram field** of a diagram object type applies to an attribute of the base entity. Similar to diagram frame, both diagram object type and diagram field allow **help text** to be displayed with operation information. The **field definition** is as follows:

- **attribute** - the base attribute;
- **help text** - show information about the field, display with those in operations;
- **presentation** - either a symbolic (only for Boolean attribute) or a formatted text;
- **operations** - field operation or frame navigation;
- **built-in operation specification** - suitable operations similar to diagram object type.

There are four types of menu operation. Firstly, **add primary** inserts a diagram object into the frame. Secondly, **add subordinate** inserts a diagram object or field as a subordinate of another diagram object. Thirdly, the **object (or frame, or field) operation** performs a user operation on the currently selected diagram object. And lastly, **frame navigation** performs a navigation between associated diagram or structured text frames. The first two are simply an option in the menu, the last two share the same structured text definition as follows:

- **help text** - operation help information;
- **selection mode** - single/multiple objects selected;
- **mode** - view only, edit only or view and edit;
- **precondition** - EASEL routine evaluated with menu is constructed;
- **trigger** - EASEL code invoked when menu option is selected;
- **destination frame type** - name of new frame to be loaded;
- **navigation path** - path through database to data for new frame;
- **destination mode** - view, as transaction, edit, as partition (for multi-user).

B.4 SHAPE MODEL

ToolBuilder allows the user to define the presentation of individual diagram objects. This is supported by the **shape model** as shown in figure B.2. Each graphical object is outlined by a number of graphics primitives provided by the tool. Individual lines and shapes can be further specified by the predefined styles as shown in figure B.7.

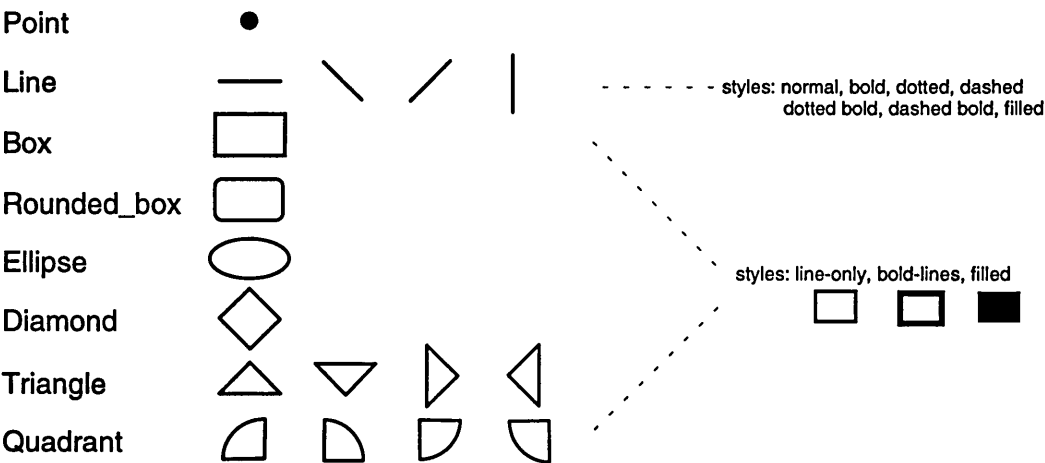


Figure B.7 ToolBuilder: Graphics Primitives

Diagram node objects are denoted by shape sets, whereas diagram link objects are denoted by linkstyles. Figure B.8 depicts some of the possible shape sets and linkstyles composed by these graphics primitives. They are related to the respective diagram objects or fields in the corresponding presentation slots. Predefined shapes and links are associated with a list of **built-in operations** that are appropriate to the graphic type. For instance, the **stretch** operation applies only to node objects and only link objects have **waypoints**.

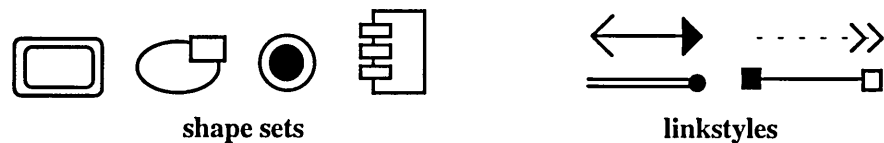
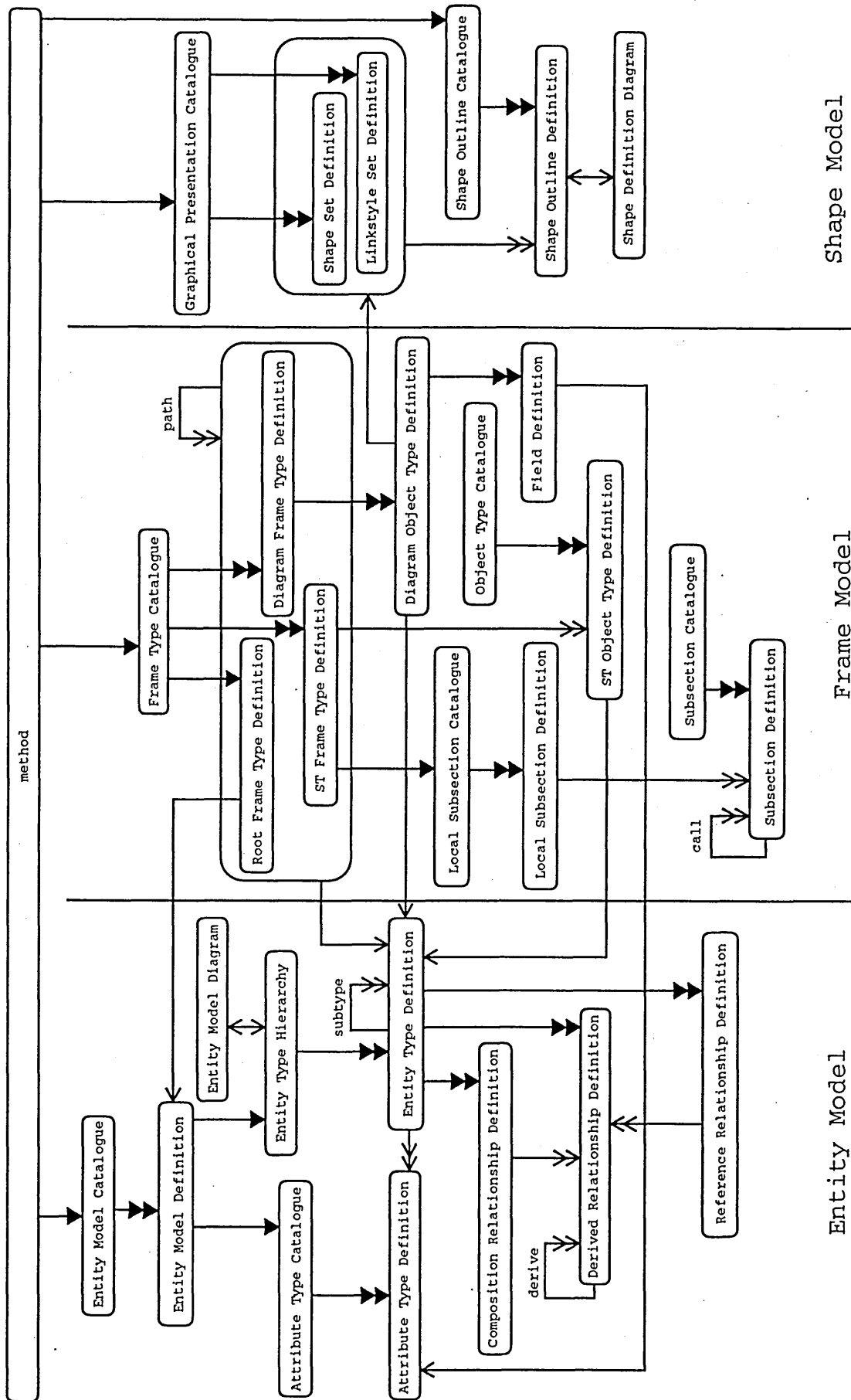


Figure B.8 ToolBuilder: Possible Shape Sets and LinkStyles

B.4 SHAPE MODEL

To conclude, ToolBuilder is an integrated metaCASE tool that comprises an entity model, a frame model and a shape model. These models effectively describe the data model, the navigation model and the presentation of graphics required to generate the intended CASE tool. Figure B.9 depicts the interrelationships of the structured text frames in ToolBuilder.



Shape Model

Frame Model

Entity Model

Figure B.9 Frame Based ToolBuilder MetaCASE Tool

APPENDIX C. THE KADS AND MIKE METHODOLOGIES

The KADS methodology is a major structured **knowledge engineering** approach towards knowledge base systems (KBSs) development. The methodology is described in some detail since it is heavily related to the work in this thesis, such as in the areas of meta modelling, method engineering (chapter three and four) and method knowledge acquisition (chapter ten). Generally speaking, KADS caters for a wider scope of knowledge domain (i.e. any domain), whereas our approach specifically serves the requisite of **method engineering** (or to be more precise ‘method knowledge engineering’).

The MIKE approach [Neubert 93] is an extended version of the KADS methodology. It stresses the use of a **hyper model** for expertise modelling. Some techniques of its knowledge acquisition are also valuable in method engineering. The following two sections give an overview of these two methodologies. The basic principles of each individual approach are outlined in view of the modelling process. Details of method modelling are given in the relevant chapters in the main text of this thesis.

C.1 THE KADS METHODOLOGY

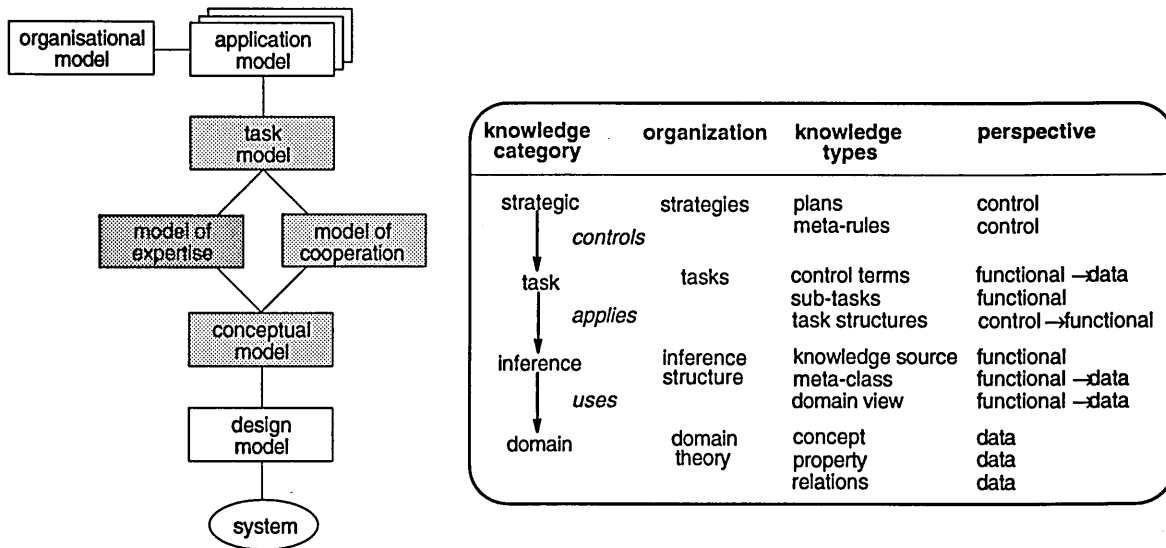
The KADS methodology has been the major structured knowledge engineering approach of knowledge base systems. There are various materials based on KADS, however [Schreiber 93] is the most popular literature in the area. It has been described as ‘the systematic step-by-step model for large scale systems development efforts’.

C.1.1 GLOBAL VIEW

KADS is intrinsically a modelling approach with seven types of model. Each model emphasises certain aspects of the system to be built and abstracts from others and each provides a decomposition of knowledge engineering tasks as shown in figure C.1a.

- An **organisational model** provides an analysis of the socio-organisational environment in which the KBS will have to function.
- An **application model** defines the problems the system should solve in the organisation and what the function of the system will be in this organisation.
- A **task model** specifies how the function of the system, as specified in the application model, is achieved through a number of tasks that the system will perform.

- The **model of cooperation** contains a specification of the functionality of those sub-tasks in the task model that require a cooperative effort between the agents to whom the subtasks have been distributed.
- Building a **model of expertise** is a central activity in the process of KBS construction (as highlighted in figure C.1a). It distinguishes the KBS development from the conventional system development. Its goal is to specify the problem solving expertise required to perform the problem-solving tasks assigned to the system.
- Together, the model of expertise and the model of cooperation provide a specification of the behaviour of the artefact to be built. The model that results from merging these two models is similar to what is called a **conceptual model** in database development.
- The description of the computational and representational techniques that the artefact should use to realise the specified behaviour is not part of the concept model. These techniques are specified as separate design decisions in a **design model**.



a. models of knowledge engineering task b. synopsis of the KADS four-layer model

Figure C.1 KADS Architecture Overview

The next subsection focuses on the model of expertise, although ideas of the models in the circumference (shown as shaded boxes in the figure) are also covered. The framework for modelling expertise in KADS are described as knowledge categories: **domain knowledge**, **inference knowledge**, **task knowledge** and **strategic knowledge** (as depicted in figure C.1b). These generic model components support top-down knowledge acquisition (see subsection C.1.3) with the notion of reusability. Individual knowledge types and perspectives in each layer are described in the corresponding knowledge categories below.

C.1.2 MODEL OF EXPERTISE

KADS describes the resulting model of expertise in a knowledge-level style that is independent of any particular implementation. Each layer of the framework is outlined with examples in the domain of *troubleshooting audio equipment*.

DOMAIN KNOWLEDGE

The domain knowledge embodies the conceptualisation of a domain for a particular application in the form of domain theory. It is based on the epistemological data primitives, which are concepts, properties, two types of relations and structures described below:

- **concept** : the central objects in a domain knowledge and it is identified through its name (e.g. *amplifier*, *speaker system* and *tape deck*);
- **property/value** : defined through their name and a description of the values that the properties can take (e.g. *amplifier* has a property *power* with possible values *on/off*);
- **relation between concepts** : the most common relations are the sub-class relation and the part-of relation; several variants of these two relations exist, each with its own semantics (e.g. *amplifier is-a component*);
- **relation between property expressions** : the relations between property expressions, which are the statements about the values of properties of concepts (e.g. *amplifier:power-button = pressed CAUSES amplifier:power = on*);
- **structure** : a complex object that consists of a number of objects/concepts and relations. (e.g. an *audio system* consists of several *components* and *relations* between these components as a whole).

KADS also introduces a set of domain description notations to depict the domain knowledge as shown in figure C.2. A simple deductive capability would enable a system to handle all solvable problems, but the domain knowledge is considered to be relatively neutral task. This separation of domain knowledge embodying the theory of the domain from its use in a problem-solving process allows the flexible use and reusability of the knowledge.

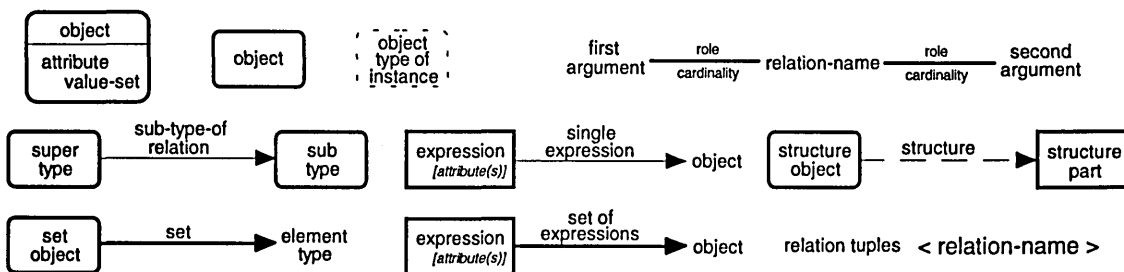


Figure C.2 KADS Graphical Representation of a Domain Description

INFERENCE KNOWLEDGE

The second layer is inference knowledge. An inference specified is assumed to be primitive in the sense that it is fully defined through its name, an input/output specification and a reference to the domain knowledge that it uses. This is illustrated by figure C.3a and the corresponding formal specification of the *decompose* inference is given below (the arrow shows the mapping from inference knowledge onto domain knowledge).

- **knowledge source**¹ : the entity that carries out an action in a primitive inference step (e.g. the *decompose* function is a knowledge source);
- **meta-class**² : the data element that a knowledge source operates on and/or produces, domain objects can be linked to more than one meta-class (e.g. the *system model* refers to the *audio system* whereas the *hypothesis* refers to the *amplifier* in the domain knowledge);
- **domain view** : a specification of how particular parts of the domain theory can be used as a ‘body of knowledge’ (e.g. the *decomposition knowledge* in the example).

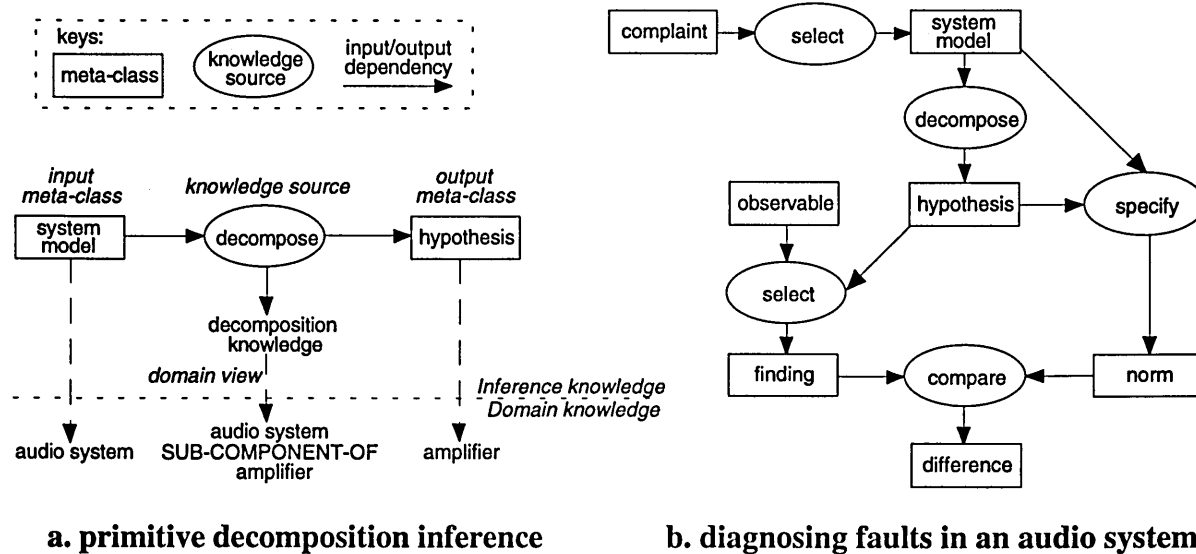


Figure C.3 KADS Inference Structure

knowledge-source decompose

input-meta-class:

system-model → component

output-meta-class:

hypothesis → component

domain-view:

decomposition(system-model, hypothesis) → sub-component-of(component, component)

¹ This term denotes a process that generates an elementary piece of information rather than the corresponding meaning in blackboard architectures.

² It should not be confused with the meaning of *meta-class* in object-oriented systems.

Figure C.3b presents an inference structure for the *audio domain*. The inferences specify a top-down systematic approach that detect sub-models of the audio system which behave inconsistently. KADS also defines the typology of inferences through epistemological categories rather than data-structures. Knowledge sources are categorized as operation types with according argument types as tabulated in table C.1. Moreover, the inference structure only defines the vocabulary and dependencies for control, which are specified as task knowledge.

Operation type	Knowledge source	Arguments
Generate concept/instance	instantiate	concept \rightarrow instance
	classify	instance \rightarrow concept
	generalise	set of instances \rightarrow concept
	abstract	concept \rightarrow concept
	specify	concept \rightarrow concept
	select	set \rightarrow concept
Change concept	assign-value	attribute \rightarrow attribute-value
	compute	structure \rightarrow attribute-value
Differentiating values/structures	compare	value + value \rightarrow value
	match	structure + structure \rightarrow structure
Structure manipulation	assemble	set of instances \rightarrow structure
	decompose	structure \rightarrow set of instances
	transform	structure \rightarrow structure

Table C.1 KADS Typology of Knowledge Sources

TASK KNOWLEDGE

The third category of knowledge contains knowledge about how elementary inferences can be combined to achieve a certain goal. The prime knowledge type in this category is the task. Tasks can achieve a particular goal and many goals may share the same task. A task represents fixed strategies for achieving problem-solving goals, therefore the task knowledge mainly describes the functional side of the knowledge framework. KADS uses the following constructs to describe task knowledge:

- **task** : a composite problem-solving action that can be decomposed into sub-tasks for problem solving (e.g. *systematic-diagnosis* and sub-tasks: *test-hypotheses*, *generate-hypotheses*);
- **control terms** : convenient labels for mapping sets of meta-class elements defined in inference knowledge (e.g. the *differential* is a set of all active *hypotheses*);
- **task structure** : a decomposition into sub-tasks and a specification of the control dependencies between sub-tasks.

The decomposition can involve three types of sub-tasks (figure C.4a) : *primitive problem-solving tasks* (inferences in inference layer), *composite problem-solving tasks* (a task in task layer) and *transfer tasks* (tasks that require interaction with an external agent). In addition, there are four types of transfer tasks. These depend on the flow of ingredient and who takes the initiative (figure C.4b). The dependencies between sub-tasks are described in structured English, such as the task *systematic-diagnosis* specification for the *audio domain* below:

task systematic-diagnosis

goal: find the smallest component with inconsistent behaviour, if one.

input: complaint

output: inconsistent-sub-system: sub-part of the system with inconsistent behaviour

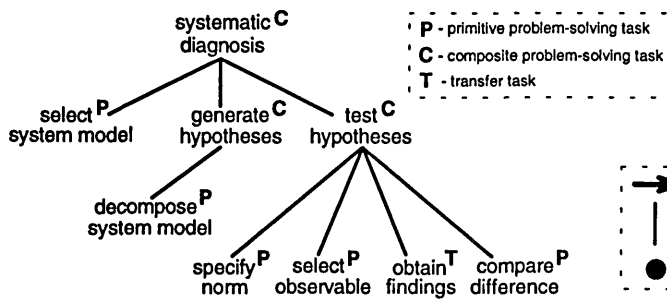
control-terms: differential: set of currently active hypotheses

task-structure:

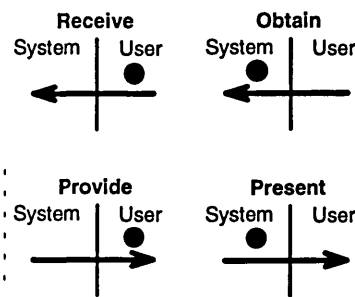
```

systematic-diagnosis(complaint → inconsistent-sub-system) =
  select(complaint → system-model)
  generate-hypotheses(system-model → differential)
  REPEAT
    test-hypotheses(differential → inconsistent-sub-system)
    generate-hypotheses(inconsistent-sub-system → differential)
  UNTIL differential = ∅

```



a. task tree of systematic diagnosis



b. four types of transfer tasks

Figure C.4 KADS Task Structure

STRATEGIC KNOWLEDGE

The fourth category of knowledge is the strategic knowledge which determines which goals are relevant to solve a particular problem. The task knowledge determines how each goal is achieved, whereas strategic knowledge concerns the dynamic planning of task execution (i.e. the control aspect of knowledge). However, most systems developed with KADS use only fixed task decompositions and have little or no strategic knowledge. The study of the nature of strategic knowledge remains mainly a research topic.

C.1.3 KNOWLEDGE REPRESENTATION

The above illustrations of the knowledge framework demonstrate how expertise can be denoted by graphics and/or structured text. The diagrammatic representations of domain knowledge inference knowledge and task knowledge are shown in figures C.2, C.3 and C.4 respectively. KADS also supports textual representation by the *domain description language* (DDL). The BNF grammar rules of concept definition in the domain knowledge is given as follows:

```
concept-def ::= concept concept-name;
               [sub-type-of: concept-name <_ concept-name>*];
               [properties].
properties ::= properties: [property-def <_ property-def>*].
property-def ::= property-name; value-set-def;
               [cardinality-def]
               [differentiation-def;].

value-set-def ::= number | integer | natural | string | boolean | universal |
               number-range(number, number) | integer-range(integer, integer) |
               {string-value <_ string-value>* }.
cardinality-def ::= cardinality: [min nat] [max < nat | infinite>];.
differentiation-def ::= differentiation of property-name(concept-name).
```

The DDL statements allows direct translation onto a set of Prolog predicates, which permits further experimentation on or execution of the domain knowledge. This is illustrated by the *patient-data* concept shown below. The technique is an effective way of organising and representing method concepts in the knowledge based system itself. In fact, a similar type of grammar and predicate structure is adopted to store our generic method representation (refer back to chapter eight).

```
% Translate from DDL statements:
concept patient-data;
concept qualitative-data;
  sub-type-of: patient-data;
  properties:
    fever: {absent, present};
    blood-pressure: {normal, elevated};
    hypertension: {absent, present};

% To Prolog Clauses:
concept(patient_data, []).
concept(qualitative_data, [patient_data]).

property(qualitative_data, fever, [present, absent]).
property(qualitative_data, blood-pressure, [normal, elevated]).
property(qualitative_data, hypertension, [present, absent]).
```

C.1.4 KNOWLEDGE ASQUISITION PROCESS

The description of the various models and knowledge types can be seen as the **product** of KBS construction. With respect to the **process** of KBS construction, KADS provides a

description of phases, activities and techniques for knowledge engineering. A *phase* represents a stage of the engineering process, and it is related to a number of activities that are usually carried out in the phase. An *activity* applies one or more techniques. For instance, the *data collection* activity can be carried out with the *structured interview* technique.

KADS distinguishes two phases in building the model of expertise: knowledge identification and knowledge modelling. The **knowledge identification** is a preparation phase before the actual construction of the model of expertise can begin, whereas the **knowledge modelling** phase builds the model. The relevant activities and techniques for each phase are tabulated in tables C.2 and C.3 respectively.

Products	Activities	Techniques
expertise data	data collection	structured interviews
task model	task analysis	rational task analysis work-flow analysis
	task feature analysis	work-flow analysis protocol segmentation
lexicon	lexicon construction	lexical analysis techniques
glossary	glossary construction	protocol segmentation frame editing
draft domain theory	concept identification	repertory grid card sort
	relation identification	

Table C.2 KADS Knowledge Identification

Products	Activities	Techniques
expertise data	data collection	think-aloud protocols
strategy	-	-
task	interpretation model selection	decision tree
inference	interpretation model selection	decision tree
	model assembly	generic sub-task substitution
domain	model assembly	generic sub-task substitution
	building domain structures	tree diagramming laddering
	domain schema definition	frame editing data modelling techniques segment grouping
model of expertise (above four components)	model validation	functional prototyping protocol analysis
	model differentiation	protocol analysis generic sub-task substitution sub-task expansion
	bottom-up model construction	goal regression forward scenario simulation participant observation

Table C.3 KADS Knowledge Modelling

The detail of activities and techniques are not described in this section, they can be found in the various literature [Taylor 89] [Tansley 93]. Most of them require a close working relationship with the domain expert, such as the *work-flow analysis* technique in knowledge identification, and the *model validation* activity in knowledge modelling. However, this is found to be impractical in method knowledge engineering (refer to chapter ten).

C.2 THE MIKE APPROACH

MIKE (*Model-based and Incremental Knowledge Engineering*) is a knowledge engineering approach based on principles from software engineering and the KADS methodology [Neubert 93]. It defines a framework for eliciting, interpreting and implementing knowledge in order to build a KBS. MIKE is a typical variation of KADS and some of the techniques have been adopted into our IFV model of method acquisition (see chapter ten).

C.2.1 GLOBAL VIEW

The central goal of the MIKE approach is the development of a detailed process model as a means to support the knowledge engineering process. The development process is done in a cyclic, incremental manner where new observations may lead to refinement, modification or completion of the already built-up representations. Various assumptions are made on the basis of principles known from software engineering and from the KADS methodology. The four global phases are *knowledge acquisition*, *design*, *implementation* and *evaluation*. They are repeated according to the different tasks of the application domain. Knowledge acquisition is the main focus of this discussion.

C.2.2 KNOWLEDGE ACQUISITION

Figure C.3 depicts the hierarchical process model of MIKE. The three phases of knowledge acquisition involved are: task analysis, model construction and model evaluation.

- The **task analysis** identifies the overall functionality of the system. Top level system tasks are decomposed into subtasks.
- The **model construction** phase includes the development and interrelation of different models reflecting the desired functionality at different levels of formalisation.
- In the **model evaluation** the specification is evaluated by testing the operational specification using test cases.

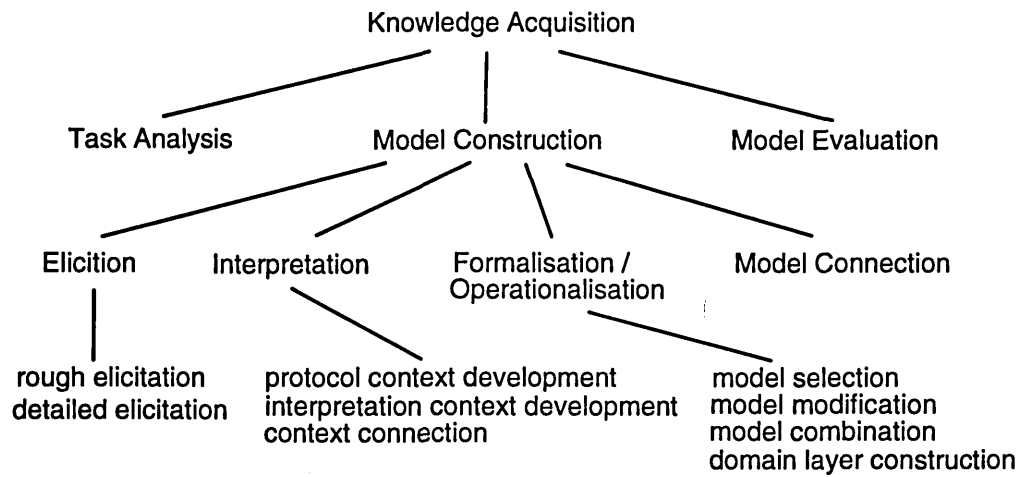


Figure C.5 MIKE Hierarchical Process Model

The MIKE approach also presents a network of knowledge elements by a **hyper model**, which handles the informal knowledge with the ability to structure their description. The knowledge elements are represented as **nodes** (*protocol node*, *activity node* and *concept node*), which are interrelated by **links** (*date link*, *refinement link*, *ordering link*, *dataflow link*, *description link*, *is_a link* and *protocol link*). These nodes and links can be combined into so-called **contexts**, thus establishing a special view on specific semantics in the hyper model:

- A *protocol context* contains all knowledge protocols in order to completely describe the results of knowledge elicitation.
- An *activity context* includes all activity nodes and all refinement links between two activity nodes. It enables the hierarchy of activity nodes and their subactivity nodes to be viewed.
- An *ordering context* includes activity nodes which are related by ordering links.
- A *concept context* encompasses all concept nodes defined in the hyper model.
- A *structure context* is another view on one hierarchy level of activity nodes.
- A *description context* establishes a view on the description links together with their source-nodes and destination-nodes.
- An *interpretation context* includes all contexts (except protocol context) and provides a complete view of the semiformal representation of the expertise.

MIKE provides a direct mapping of this hyper model to KADS through the *model connection* (see later). Each of these contexts is related to a different notion in the *model of expertise* as depicted in figure C.4. For instance, each *concept* in the domain layer refers to a *concept node* in the concept context; a *knowledge source* in the inference layer is denoted as an *activity node* in the ordering context, which in turn represents the task layer in KADS.

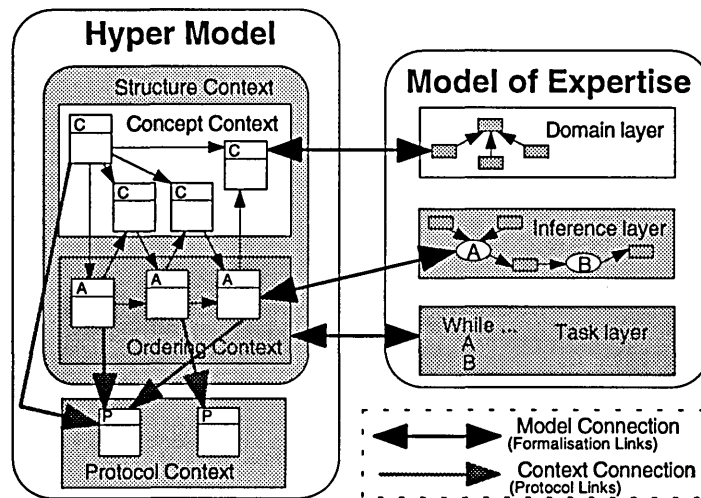


Figure C.6 MIKE: Examples of Context and Model Connections

The following points describe briefly the four model construction subphases, in which the different intermediate models are developed in an integrated environment:

- **Elicitation** means interviewing and observing the expert. MIKE proposes a top-down knowledge elicitation by using adequate, informally described problem solving methods (PSMs).
- The knowledge protocols resulting from elicitation must be analysed during the **interpretation** step. The result of this phase is a semiformal hyper model that describes the main steps of the problem-solving process of the concepts and interactions in the application domain.
- In the knowledge **formalisation** step, the informal expertise in the hyper model is formalised. The formal specification is also operational, thus it can be evaluated by prototyping.
- The relationships between the hyper model and the formal model of expertise are established by a **model connection**. This gives a mapping between formal and informal elements describing the same aspects by using different representation formalisms that are part of the documentation of the model of expertise.

For a more detailed description of MIKE, refer to the paper [Neubert 93].

hood



Figure D.1 HOOD Concept Diagram

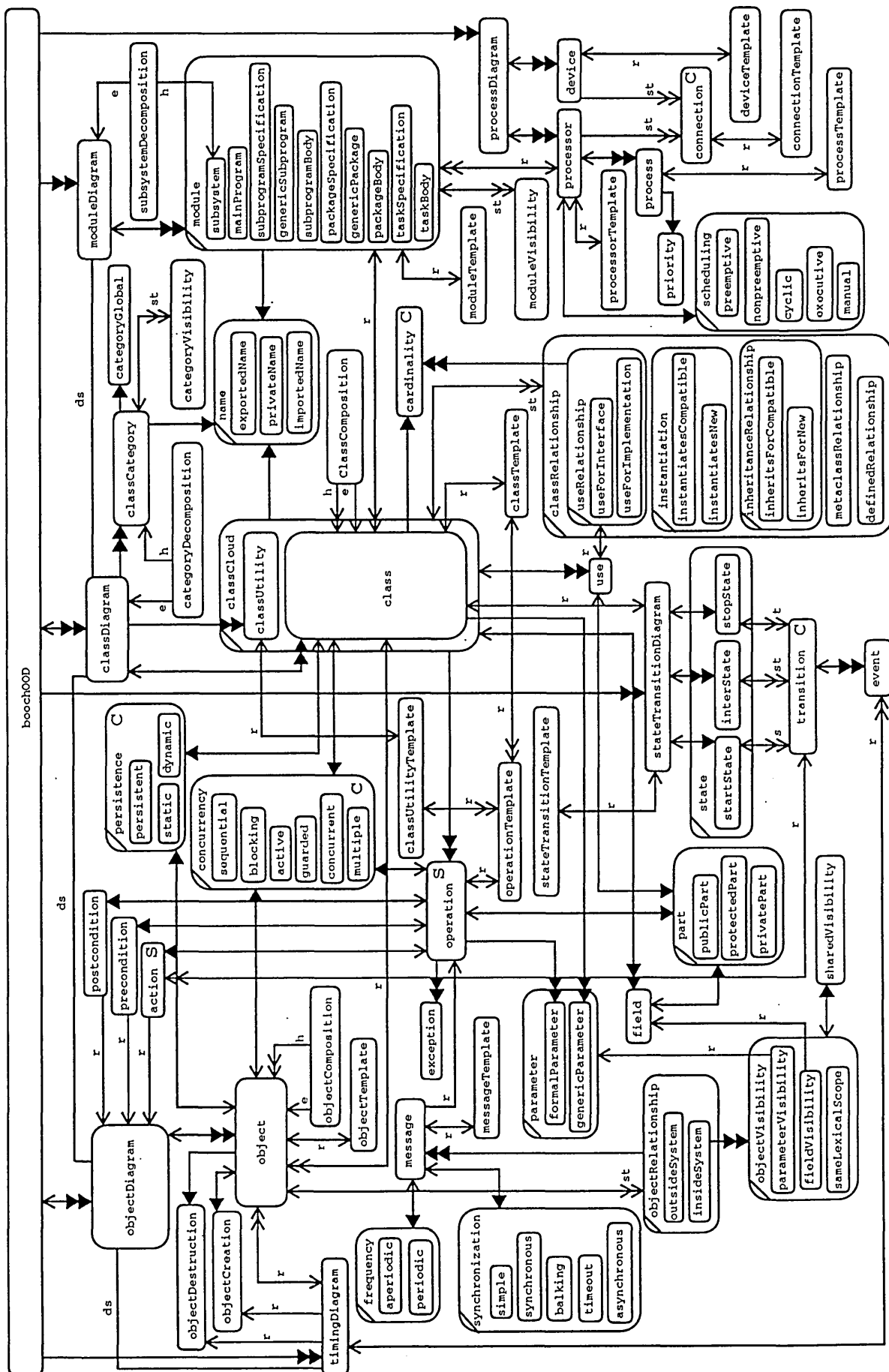


Figure D.2 Booch OOD Concept Diagram

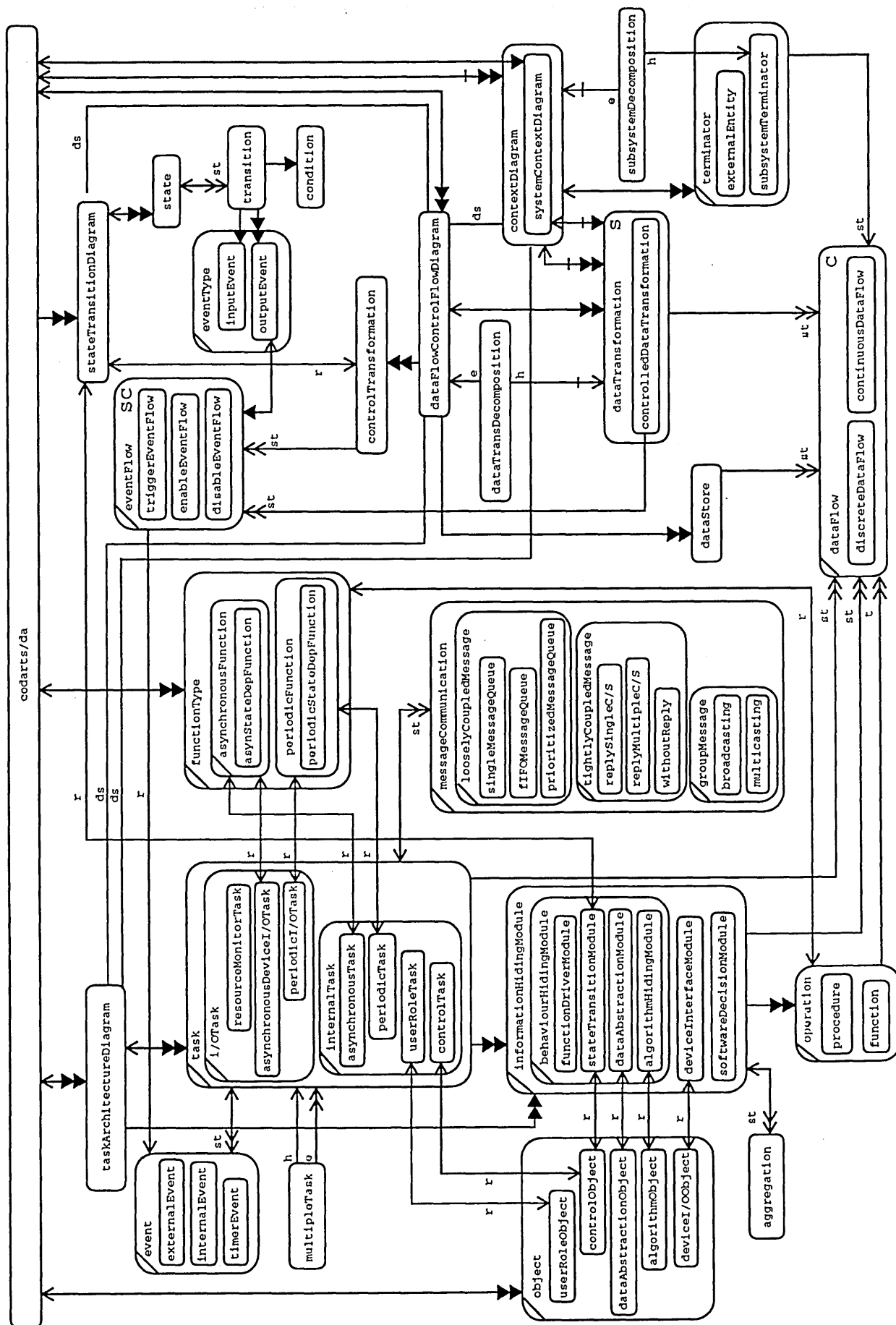


Figure D.3 Codarts/DA Concept Diagram

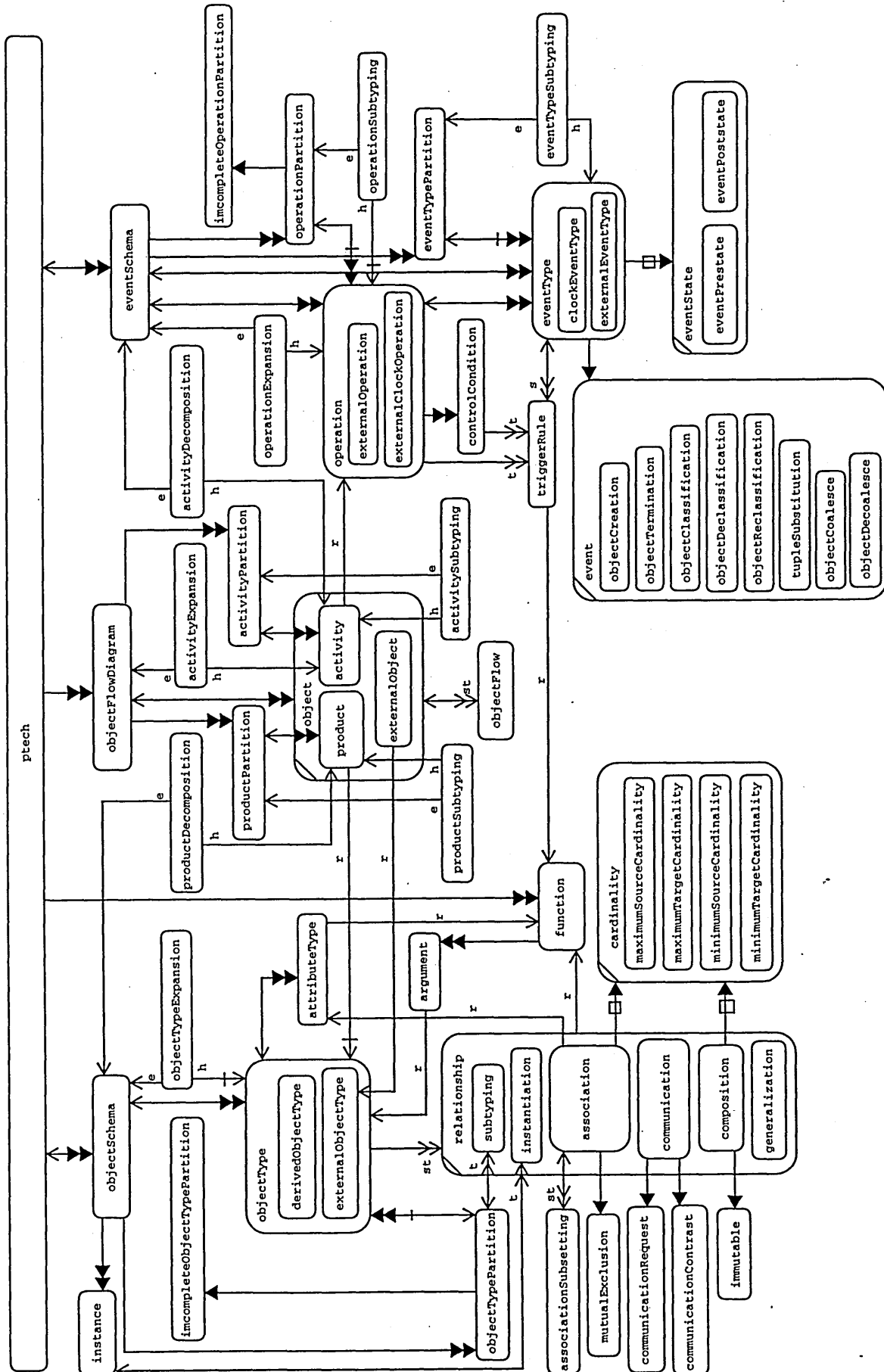


Figure D.5 Ptech Concept Diagram

APPENDIX E: OMT TASK DIAGRAMS

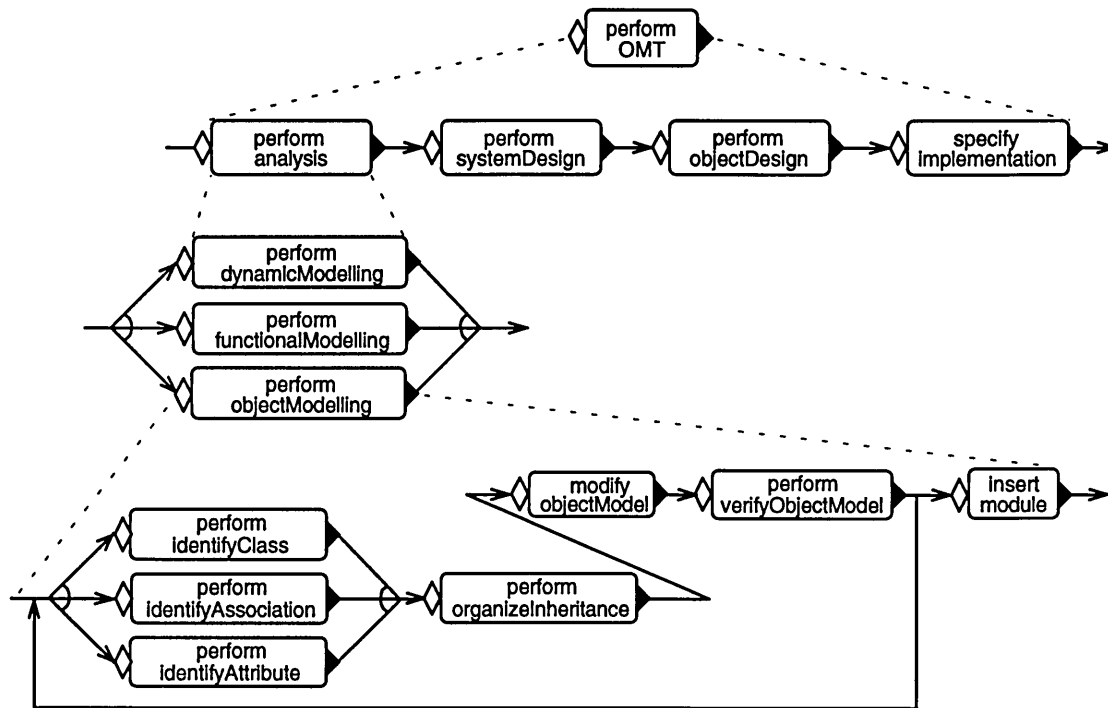


Figure E.1 OMT: Top Level Task Diagram and *objectModelling* Decomposition

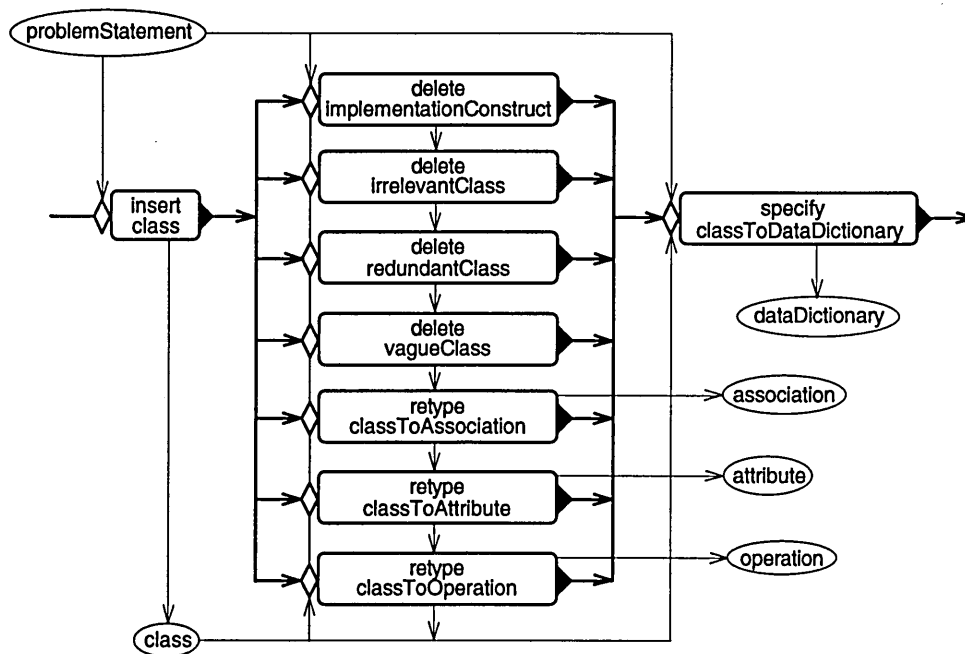


Figure E.2 OMT: *perform(identifyClass)* Task Diagram

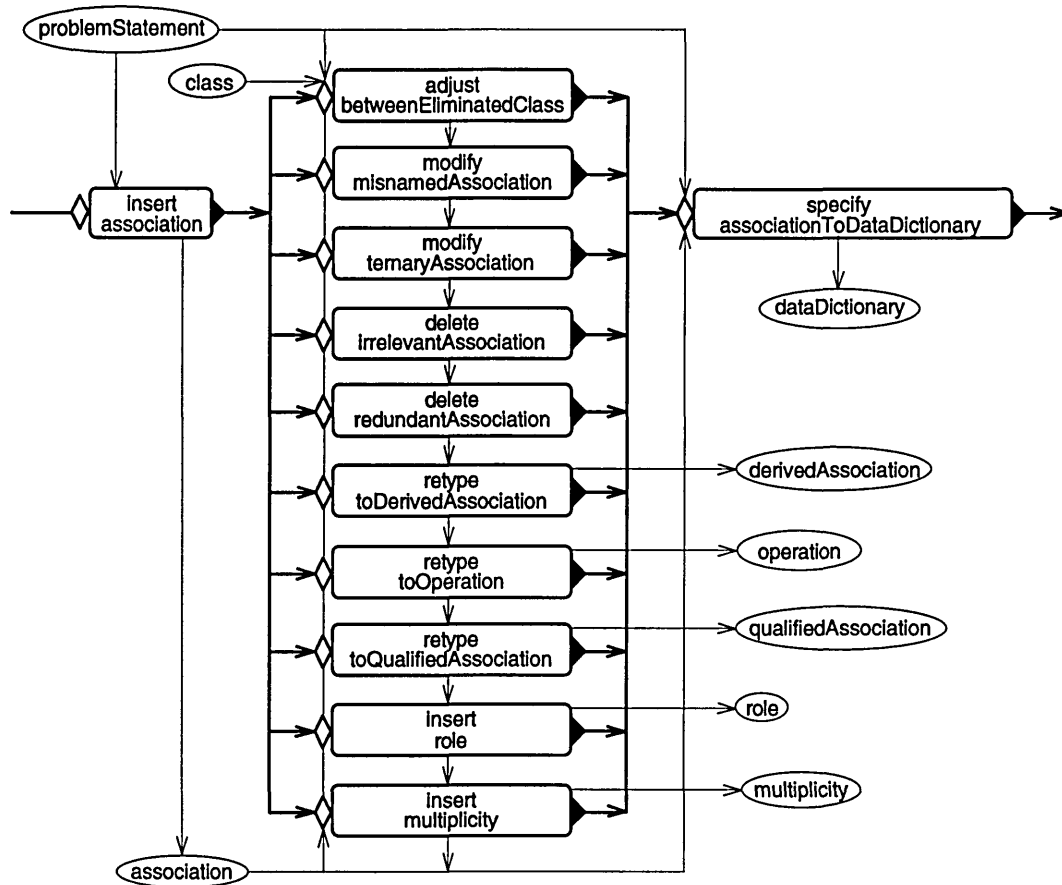


Figure E.3 OMT: *perform(identifyAssociation)* Task Diagram

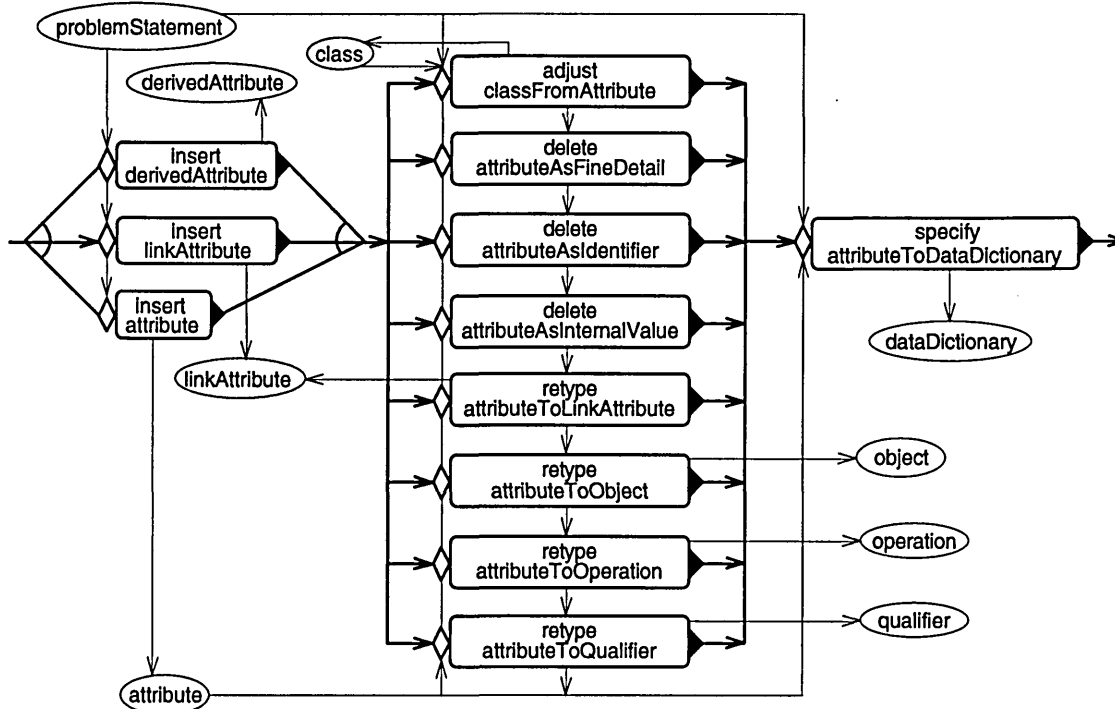


Figure E.4 OMT: *perform(identifyAttribute)* Task Diagram

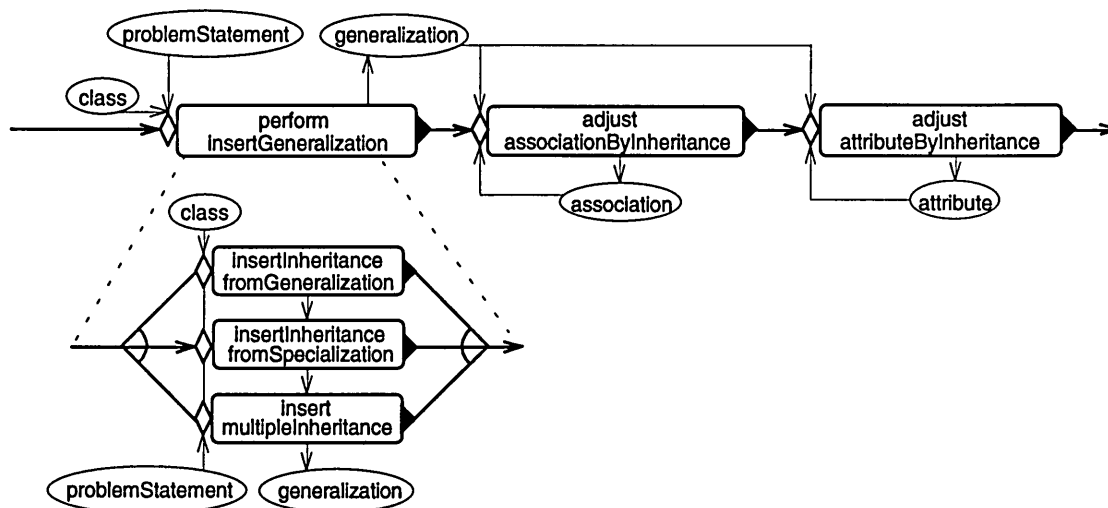


Figure E.5 OMT: *perform(organizeInheritance)* Task Diagram

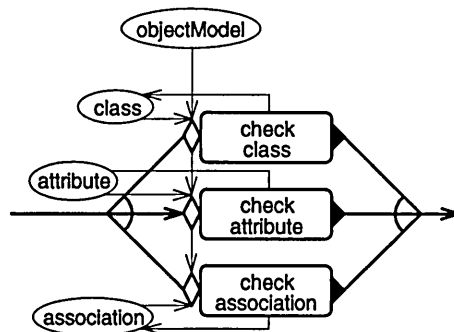


Figure E.6 OMT: *perform(verifyObjectModel)* Task Diagram

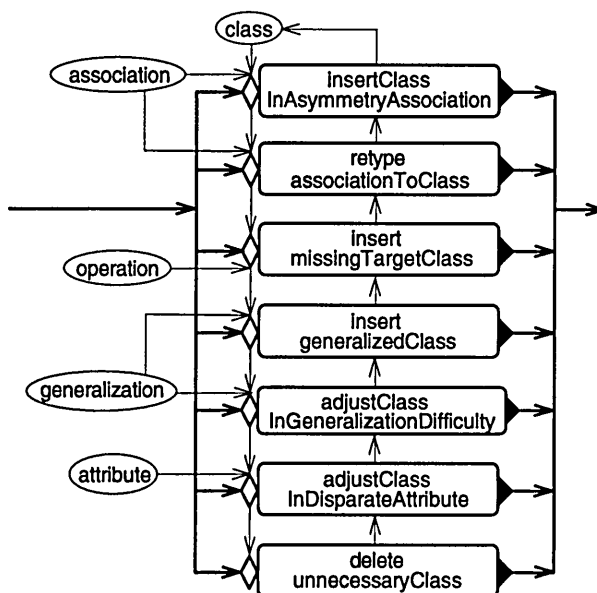


Figure E.7 OMT: *do(checkClass)* Task Diagram

```

graph LR
    attribute(attribute) --> Merge(( ))
    In(( )) --> Merge
    Merge --> retype[retype  
attributeToQualifiedAssociation]
    retype --> Split(( ))
    Split --> qualifiedAssociation(qualifiedAssociation)
  
```

E.4

APPENDIX F: OMT MSL STATEMENTS

%% OMT.MSL

%% Method Specification of Object Modelling Technique [Rumbaugh 91]

%% 8.3.96

method omt

productModel

%% The corresponding concept diagram can be found in diagram D.4 of Appendix D.

```
concept objectModellingTechnique ;
    subtypeOf method ;
    property objectModel [1,1,0,1] ;
    property dynamicModel [1,1,0,1] ;
    property functionalModel [1,1,0,1] ;
concept objectModel ;
    subtypeOf fragment ;
    property module [1,1,1,n] ;
    property objectDiagram [1,1,1,n] ;
    property dataDictionary [1,1,1,n] ;
concept objectDiagram ;
    subtypeOf diagram ;
    property object [1,1,1,n] ;
concept dataDictionary ;
    subtypeOf text ;
concept module ;
    property sheet [1,1,1,n] ;
concept sheet ;
concept signature ;
concept classConstraint ;
concept operation ;
    property signature [1,1,0,1] ;
concept abstractOperation ;
    subtypeOf operation ;
concept classOperation ;
    subtypeOf operation ;
concept derivedOperation ;
    subtypeOf operation ;
concept attribute ;
concept derivedAttribute ;
    subtypeOf attribute ;
concept classAttribute ;
    subtypeOf attribute ;
concept object abstract ;
    property attribute [1,1,0,n] ;
concept instance ;
    subtypeOf object ;
concept class ;
    subtypeOf object ;
    property classConstraint [1,1,0,1] ;
concept abstractClass ;
    subtypeOf class ;
concept derivedClass ;
    subtypeOf class ;
concept metaClass ;
    subtypeOf class ;
```

```

concept relationship abstract ;
    subtypeOf link ;
    source object [0,n,1,1] ;
    target object [0,n,1,1] ;
concept instantiation ;
    subtypeOf relationship ;
    target instance [0,n,1,1] ;
concept generalisation ;
    subtypeOf relationship ;
    property discriminator [1,1,0,1] ;
concept overlappingGeneralisation ;
    subtypeOf generalisation ;
concept discriminator ;
concept association ;
    subtypeOf relationship ;
    property associationConstraint [1,1,0,1] ;
    property linkAttribute [1,1,0,1] ;
    property linkObject [1,1,0,1] ;
    property ordering [1,1,0,1] ;
    property role # [1,1,0,1] ;
    property multiplicity # [1,1,0,1] ;
    property operationPropagation [1,1,0,1] ;
concept derivedAssociation ;
    subtypeOf association ;
concept qualifiedAssociation ;
    subtypeOf association ;
    property qualifier [1,1,0,1] ;
concept role abstract ;
concept sourceRole ;
    subtypeOf role ;
concept targetRole ;
    subtypeOf role ;
concept linkObject ;
concept linkAttribute ;
concept associationConstraint ;
concept multiplicity abstract ;
concept sourceMultiplicity ;
    subtypeOf multiplicity ;
concept targetMultiplicity ;
    subtypeOf multiplicity ;
concept aggregation abstract ;
    subtypeOf relationship ;
    property sourceMultiplicity [1,1,0,1] ;
    property targetMultiplicity [1,1,0,1] ;
    property operationPropagation [1,1,0,1] ;
concept fixedAggregation ;
    subtypeOf aggregation ;
concept variableAggregation ;
    subtypeOf aggregation ;
concept resursiveAggregation ;
    subtypeOf aggregation ;
concept ordering ;
concept qualifier ;
concept operationPropagation ;
reference discriminator attribute [0,1,1,1] ;
reference linkObject object [0,1,1,1] ;
reference linkAttribute attribute [0,1,1,1] ;
reference objectDiagram sheet [1,1,1,1] ;

concept dynamicModel ;

```

```

    subtypeOf fragment ;
    property scenario [1,1,1,n] ;
    property stateDiagram [1,1,1,n] ;
concept stateDiagram ;
    subtypeOf diagram ;
    property startState [1,1,0,1] ;
    property interState [1,1,1,n] ;
    property stopState [1,1,0,1] ;
concept state abstract ;
    property activity [1,1,0,1] ;
    property entryAction [1,1,0,1] ;
    property exitAction [1,1,0,1] ;
    property internalAction [1,1,0,n] ;
concept startState ;
    subtypeOf state ;
concept interState ;
    subtypeOf state ;
concept stopState ;
    subtypeOf state ;
concept action abstract ;
concept entryAction ;
    subtypeOf action ;
concept exitAction ;
    subtypeOf action ;
concept internalAction ;
    subtypeOf action ;
concept eventAction ;
    subtypeOf action ;
concept activity ;
concept automaticTransition ;
    subtypeOf transition ;
concept guardCondition ;
concept delegation ;
concept eventAttribute ;
concept event ;
    property eventAttribute [1,1,0,n] ;
concept eventTrace ;
concept scenario ;
    property eventTrace [1,1,1,1] ;
concept eventGeneralisation ;
    subtypeOf link ;
    source event [0,n,1,1] ;
    target event [0,n,1,1] ;
concept transition ;
    subtypeOf link ;
    source startState, interState [0,n,0,1] ;
    target interState, stopState [0,n,0,1] ;
    property delegation ! [0,1,0,1] ;
    property event ! [0,1,0,1] ;
    property guardCondition ! [0,1,0,1] ;
concept concurrentSubdiagram ;
    subtypeOf group ;
    host state [0,1,1,1] ;
    element stateDiagram [0,1,1,n] ;
concept mergingControl ;
    subtypeOf group ;
    host transition [0,1,1,n] ;
    element transition [0,1,1,1] ;
concept nestedStateDiagram ;
    subtypeOf group ;

```

```

        host state [0,1,1,1] ;
        element stateDiagram [0,1,1,1] ;
concept splittingControl ;
        subtypeOf group ;
        host transition [0,1,1,1] ;
        element transition [0,1,1,n] ;
reference activity operation [0,1,1,1] ;
reference action operation [0,1,1,1] ;
reference delegation object [0,1,1,1] ;
reference eventAttribute attribute [0,1,1,1] ;
reference eventTrace event [1,1,1,n] ;
reference dataDictionary association [1,1,1,n] ;
reference dataDictionary attribute [1,1,1,n] ;
reference dataDictionary object [1,1,1,n] ;
reference dataDictionary operation [1,1,1,n] ;

concept functionalModel ;
        subtypeOf fragment ;
        property dataFlowDiagram [1,1,1,n] ;
concept dataFlowDiagram ;
        subtypeOf diagram ;
        property actor [1,1,0,n] ;
        property dataStore [1,1,0,n] ;
        property process [1,1,1,n] ;
concept actor ;
concept data ;
concept dataStore ;
concept process ;
concept dataFlowComposition ;
        subtypeOf group ;
        host dataFlow [0,1,2,n] ;
        element dataFlow [0,1,1,1] ;
concept dataFlowDecomposition ;
        subtypeOf group ;
        host dataFlow [0,1,1,1] ;
        element dataFlow [0,1,2,n] ;
concept nestedDataFlowDiagram ;
        subtypeOf group ;
        host process [0,1,1,1] ;
        element dataFlowDiagram [0,1,1,1] ;
concept controlFlow ;
        subtypeOf link ;
        source process [0,n,1,1] ;
        target process [0,n,1,1] ;
concept dataFlow ;
        subtypeOf link ;
        source actor, dataStore, process [0,n,0,1] ;
        target actor, dataStore, process [0,n,0,1] ;
        property data [1,1,1,n] ;
reference actor object [0,1,1,1] ;
reference data attribute [0,1,1,1] ;
reference dataStore object [0,1,1,1] ;
reference process operation [0,1,1,n] ;
endProductModel

processModel
%% The corresponding task diagrams can be found in Appendix E.
    task objectModellingTechnique perform(objectModellingTechnique) ;
        precond [problemStatement] ;
        postcond [analysis, systemDesign, objectDesign, implementation] ;

```



```

    compose [analysis, systemDesign, objectDesign, implementation] ;
task analysis perform(analysis) ;
    precondition [problemStatement] ;
    postcondition [analysis] ;
    compose parallel [objectModelling, dynamicModelling, functionalModelling] ;
task systemDesign perform(systemDesign) ;
    precondition [analysis] ;
    postcondition [systemDesign] ;
    compose [...] ;
task objectDesign perform(objectDesign) ;
    precondition [systemDesign] ;
    postcondition [objectDesign] ;
    compose [...] ;
task implementation specify(objectDesign, implementation) ;
    precondition [objectDesign] ;
    postcondition [implementation] ;

task objectModelling perform(objectModelling) ;
    precondition [problemStatement] ;
    postcondition [objectModel] ;
    compose [identifyElement, organiseInheritance, testAccessPath,
        verifyObjectModel, groupClassIntoModule] ;
task dynamicModelling perform(dynamicModelling) ;
    precondition [problemStatement] ;
    postcondition [dynamicModel] ;
    compose [...] ;
task functionalModelling perform(functionalModelling) ;
    precondition [problemStatement] ;
    postcondition [functionalModel] ;
    compose [...] ;

task identifyElement perform(identifyElement) ;
    precondition [problemStatement] ;
    postcondition [objectModel] ;
    compose parallel [identifyClass, identifyAssociation, identifyAttribute] ;
task identifyClass perform(identifyClass) ;
    precondition [problemStatement] ;
    postcondition [class, instance, object] ;
    compose [insertClass, verifyClass, specifyClass] ;
task identifyAssociation perform(identifyAssociation) ;
    precondition [problemStatement, class] ;
    postcondition [association] ;
    compose [insertAssociation, verifyAssociation, specifyAssociation] ;
task identifyAttribute perform(identifyAttribute) ;
    precondition [problemStatement, class] ;
    postcondition [attribute] ;
    compose [insertAttribute, insertDerivedAttribute, insertLinkAttribute,
        verifyAttribute, specifyAttribute] ;

task organiseInheritance perform(organiseInheritance) ;
    precondition [problemStatement, class] ;
    postcondition [generalisation] ;
    compose [insertGeneralisation, adjustAssociationByInheritance,
        adjustAttributeByInheritance] ;
task testAccessPath modify(objectModel) ;
    precondition [objectModel] ;
    postcondition [] ;
task verifyObjectModel perform(verifyObjectModel) ;
    precondition [objectModel] ;
    postcondition [] ;

```

```

    compose parallel [checkClass, checkAssociation, checkAttribute] ;
task groupClassIntoModule insert(module) ;
    precondition [objectModel] ;
    postcondition [module, sheet] ;

task insertClass insert(class) ;
    precondition [problemStatement] ;
    postcondition [class] ;
task verifyClass do(verifyClass) ;
    precondition [problemStatement, class] ;
    postcondition [] ;
    refine [deleteRedundantClass, deleteIrrelevantClass, deleteVagueClass,
        retypeClassToAttribute, retypeClassToOperation, retypeClassToAssociation,
        deleteImplementationConstruct] ;
task specifyClass specify(class, dataDictionary) ;
    precondition [problemStatement, class] ;
    postcondition [dataDictionary] ;

task deleteRedundantClass delete(class) ;
    precondition [class] ;
    postcondition [] ;
task deleteIrrelevantClass delete(class) ;
    precondition [class] ;
    postcondition [] ;
task deleteVagueClass delete(class) ;
    precondition [class] ;
    postcondition [] ;
task retypeClassToAttribute retype(class, attribute) ;
    precondition [class] ;
    postcondition [attribute] ;
task retypeClassToOperation retype(class, operation) ;
    precondition [class] ;
    postcondition [operation] ;
task retypeClassToAssociation retype(class, association) ;
    precondition [class] ;
    postcondition [association] ;
task deleteImplementationConstruct delete(class) ;
    precondition [class] ;
    postcondition [] ;

task insertAssociation, insert(association) ;
    precondition [problemStatement, class] ;
    postcondition [association] ;
task verifyAssociation do(verifyAssociation) ;
    precondition [association] ;
    postcondition [] ;
    refine [adjustAssociationBetweenEliminatedClass, deleteIrrelevantAssociation,
        retypeAssociationToOperation, modifyTernaryAssociation,
        deleteRedundantAssociation,
        retypeRedundantAssociationToDerivedAssociation,
        modifyMisnamedAssociation, insertRoleName,
        retypeAssociationToQualifiedAssociation, insertMultiplicity] ;
task specifyAssociation specify(association, dataDictionary) ;
    precondition [association] ;
    postcondition [dataDictionary] ;

task adjustAssociationBetweenEliminatedClass adjust(association, class) ;
    precondition [association, class] ;
    postcondition [] ;
task deleteIrrelevantAssociation delete(association) ;

```

```

    precondition [association] ;
    postcondition [] ;
task retypeAssociationToOperation retype(association, operation) ;
    precondition [association] ;
    postcondition [operation] ;
task modifyTernaryAssociation modify(association) ;
    precondition [association] ;
    postcondition [] ;
task deleteRedundantAssociation delete(association) ;
    precondition [association] ;
    postcondition [] ;
task retypeAssociationToDerivedAssociation retype(association, derivedAssociation) ;
    precondition [association] ;
    postcondition [derivedAssociation] ;
task modifyMisnamedAssociation modify(association) ;
    precondition [association] ;
    postcondition [] ;
task insertRoleName insert(role) ;
    precondition [association] ;
    postcondition [role] ;
task retypeAssociationToQualifiedAssociation retype(association, qualifiedAssociation) ;
    precondition [association] ;
    postcondition [qualifiedAssociation] ;
task insertMultiplicity insert(multiplicity) ;
    precondition [association] ;
    postcondition [multiplicity] ;

task insertAttribute insert(attribute) ;
    precondition [problemStatement, class] ;
    postcondition [attribute] ;
task insertDerivedAttribute insert(derivedAttribute) ;
    precondition [problemStatement, class] ;
    postcondition [derivedAttribute] ;
task insertLinkAttribute insert(linkAttribute) ;
    precondition [problemStatement, class] ;
    postcondition [linkAttribute] ;
task verifyAttribute do(verifyAttribute) ;
    precondition [attribute] ;
    postcondition [] ;
    refine [retypeAttributeToObject, retypeAttributeToQualifier,
            retypeAttributeToOperation, deleteAttributeAsIdentifier,
            retypeAttributeToLinkAttribute, deleteAttributeAsInternalValue,
            deleteAttributeAsFineDetail, adjustClassFromAttribute] ;
task specifyAssociation specify(attribute, dataDictionary) ;
    precondition [attribute] ;
    postcondition [dataDictionary] ;

task retypeAttributeToObject retype(attribute, object) ;
    precondition [attribute] ;
    postcondition [object] ;
task retypeAttributeToQualifier retype(attribute, qualifier) ;
    precondition [attribute] ;
    postcondition [qualifier] ;
task retypeAttributeToOperation retype(attribute, operation) ;
    precondition [attribute] ;
    postcondition [operation] ;
task deleteAttributeAsIdentifier delete(attribute) ;
    precondition [attribute] ;
    postcondition [] ;
task retypeAttributeToLinkAttribute retype(attribute, linkAttribute) ;

```

```

    precondition [attribute] ;
    postcondition [linkAttribute] ;
task deleteAttributeAsInternalValue delete(attribute) ;
    precondition [attribute] ;
    postcondition [] ;
task deleteAttributeAsFineDetail delete(attribute) ;
    precondition [attribute] ;
    postcondition [] ;
task adjustClassFromAttribute adjust(class, attribute) ;
    precondition [attribute] ;
    postcondition [class] ;

task insertGeneralisation perform(insertGeneralisation) ;
    precondition [problemStatement] ;
    postcondition [generalisation] ;
    compose parallel [inheritanceFromGeneralisation, inheritanceFromSpecialisation,
        insertMultipleInheritance] ;
task adjustAssociationByInheritance adjust(association, generalisation) ;
    precondition [association, generalisation] ;
    postcondition [] ;
task adjustAttributeByInheritance adjust(attribute, generalisation) ;
    precondition [attribute, generalisation] ;
    postcondition [] ;

task inheritanceFromGeneralisation insert(generalisation) ;
    precondition [problemStatement] ;
    postcondition [generalisation] ;
task inheritanceFromSpecialisation insert(generalisation) ;
    precondition [problemStatement] ;
    postcondition [generalisation] ;
task insertMultipleInheritance insert(generalisation) ;
    precondition [problemStatement] ;
    postcondition [generalisation] ;

task checkClass do(checkClass) ;
    precondition [class, objectModel] ;
    postcondition [] ;
    refine [addClassInAsymmetryAssociation, splitClassInDisparateAttribute,
        splitClassInGeneralisationDifficulty, addMissingTargetClass,
        addGeneralisedClass, retypeAssociationToClass,
        deleteUnnecessaryClass] ;
task checkAssociation do(checkAssociation) ;
    precondition [association, objectModel] ;
    postcondition [] ;
    refine [addAssociationInMissingPath, removeRedundantAssociation,
        adjustAssociationInHierarchy] ;
task checkAttribute do(checkAttribute) ;
    precondition [attribute, objectModel] ;
    postcondition [] ;
    refine [retypeAttributeToQualifiedAssociation] ;

task addClassInAsymmetryAssociation adjust(class, association) ;
    precondition [class, association] ;
    postcondition [] ;
task splitClassInDisparateAttribute adjust(class, attribute) ;
    precondition [attribute, class] ;
    postcondition [] ;
task splitClassInGeneralisationDifficulty adjust(class, generalisation) ;
    precondition [class, generalisation] ;
    postcondition [] ;

```

```

task addMissingTargetClass adjust(class, operation) ;
    precondition [class, operation] ;
    postcondition [] ;
task addGeneralisedClass adjust(class, generalisation) ;
    precondition [class, generalisation] ;
    postcondition [] ;
task retypeAssociationToClass retype(association, class) ;
    precondition [association, class] ;
    postcondition [class] ;
task deleteUnnecessaryClass delete(class) ;
    precondition [class] ;
    postcondition [] ;

task addAssociationInMissingPath insert(association) ;
    precondition [association] ;
    postcondition [] ;
task removeRedundantAssociation delete(association) ;
    precondition [association] ;
    postcondition [] ;
task adjustAssociationInHierarchy adjust(association, generalisation) ;
    precondition [association, generalisation] ;
    postcondition [] ;

task retypeAttributeToQualifiedAssociation retype(attribute, qualifiedAssociation) ;
    precondition [attribute] ;
    postcondition [qualifiedAssociation] ;

...
%% The task definitions of Dynamic Model, Functional Model in Analysis Phase,
%% and those in the System Design & Object Design Phases have been skipped here.
...

```

endProcessModel

heuristicModel

```

% CONCEPT HEURISTICS
heuristic abstractClass ;
text    'Abstract class is a class that cannot have direct instances but whose descendants
        can have instances.' ;
link    abstractOperation, class, generalisation ;
heuristic abstractOperation ;
text    'Abstract operation is an operation defined but not implemented by an abstract
        class. The operation must be implemented by all concrete descendent classes.' ;
link    abstractClass, class, generalisation ;
heuristic action ;
text    'An action is an instantaneous operation. Actions are associated with events and are
        usually formal in nature.' ;
link    action, activity, entryAction, exitAction, internalAction, operation ;
heuristic activity ;
text    'An activity is an operation that takes time to complete. Activities are associated with
        states and represent real-world accomplishments.' ;
link    action, operation ;
heuristic actor ;
text    'Actor object is an active object that drives the data flow graph by producing or
        consuming values.' ;
link    dataFlowDiagram, object ;
heuristic aggregation ;
text    'Aggregation is a special form of association, between a whole and its parts, in which
        the whole is composed of the parts.' ;
link    fixedAggregation, resursiveAggregation, variableAggregation ;
heuristic association ;
text    'Association is a relationship among instances of two or more classes describing a

```

group of links with common structure and common semantics.' ;

link associationConstraint, derivedAssociation, linkAttribute, multiplicity, qualifiedAssociation, qualifier, role ;

heuristic associationConstraint ;

text 'Association constraint is a functional relationship of association; a statement about some condition or relationship that must be maintained as true.' ;

link association, linkAttribute, multiplicity, qualifier ;

heuristic attribute ;

text 'An attribute is a named property of a class describing a data value held by each object of the class.' ;

link class, object ;

heuristic automaticTransition ;

text 'Automatic transition is an unlabelled transition that automatically fires when the activity associated with the source state is completed.' ;

link event, guardcondition, transition ;

heuristic class ;

text 'A class is a description of a group of objects with similar properties, common behaviour, common relationships and common semantics.' ;

link abstractClass, classConstraint, metaClass, object ;

heuristic classAttribute ;

text 'Class attribute is an attribute whose value is common to a class of objects rather than a value peculiar to each instance.' ;

link attribute, class ;

heuristic classConstraint ;

text 'Class constraint is a functional relationship of class; a statement about some condition or relationship that must be maintained as true.' ;

link class ;

heuristic classOperation ;

text 'Class operation is an operation on a class, rather than on instances of the class. An instance creation operation is a common example.' ;

link class, operation ;

heuristic concurrentSubdiagram ;

text 'Concurrency within the state of a single object arises when the object can be partitioned into subsets of attributes or links, each of which has its own subdiagram.' ;

link state, stateDiagram ;

heuristic controlFlow ;

text 'Control flow is a boolean value that affects whether a process is executed.' ;

link process ;

heuristic dataDictionary ;

text 'A data dictionary is a textual description of each class, its associations, attributes and operations.' ;

link association, attribute, class, operation ;

heuristic dataFlow ;

text 'Data flow is the connection between the output of one object or process and the input to another.' ;

rule not(source(actor) and target(dataStore)) or
not(source(dataStore) and target(actor)) ;

link actor, dataFlowDecomposition, dataFlowDiagram, dataStore, process ;

heuristic dataFlowDecomposition ;

text 'Data flow decomposition split an aggregate data on a data flow into its components.' ;

link dataFlow, dataFlowComposition ;

heuristic dataFlowDiagram ;

text 'A data flow diagram is a graphical representation of the functional model, showing dependencies between values and the computation of output values from input values without regard for when or if the functions are executed.' ;

link dataFlow, functionalModel, nestedDataFlowDiagram, process ;

heuristic dataStore ;

text 'A data store is a passive object that stores data for later access.' ;

link dataFlowDiagram, process ;

heuristic delegation

text 'Delegation is an implementation mechanism in which an object, responding to an operation on itself, forwards the operation to another object.' ;

rule not(owner(automaticTransition)) ;

link object, transition ;

heuristic derivedAssociation ;

text 'Derived association is an association that is defined in terms of other associations.' ;

link association ;

heuristic derivedAttribute ;

text 'Derived attribute is an attribute that is computed from other attributes.' ;

link attribute ;

heuristic discriminator ;

text 'A discriminator is an attribute of enumeration type that indicates which property of a class is being abstracted by a particular generalisation.' ;

link generalisation ;

heuristic dynamicModel ;

text 'A dynamic model describes the aspects of a system concerned with control, including time, sequencing of operations and interaction of objects.' ;

link scenario, stateTransitionDiagram ;

heuristic entryAction ;

text 'Entry action permits action to be associated with a state, to indicate all the transitions entering the state.' ;

link action, exitAction ;

heuristic event ;

text 'An event is something that happens instantaneously at a point in time.' ;

rule not(owner(automaticTransition)) .

link dynamicModel, eventGeneralisation, eventTrace, scenario, state, transition ;

heuristic eventGeneralisation ;

text 'Events can be organised into a generalisation hierarchy with inheritance of event attributes.' ;

link event ;

heuristic eventTrace ;

text 'An event trace is a diagram that shows the sender and receiver of events and the sequence of events.' ;

link event, scenario ;

heuristic exitAction ;

text 'Exit action permits action to be associated with a state, to indicate all the transitions exiting the state.' ;

link action, entryAction ;

heuristic fixedAggregation ;

text 'Fixed aggregation has an aggregate with a predefined number and types of components.' ;

link aggregation, recursiveAggregation, variableAggregation ;

heuristic functionalModel ;

text 'A functional model describes the aspects of a system that transform values using functions, mappings, constraints and functional dependencies.' ;

link dataFlowDiagram ;

heuristic generalisation ;

text 'Generalisation is the relationship between a class and one or more refined or specialised versions of it.' ;

link discriminator ;

heuristic guardCondition ;

text 'Guard condition is a boolean expression that must be true in order for a transition to occur.' ;

link transition ;

heuristic instance ;

text 'An instance is an object described by a class.' ;

link class, instantiation ;

heuristic instantiation ;

text 'Instantiation is the process of creating instances from classes.' ;

link class, instance, object ;
 heuristic linkAttribute ;
 text 'A link attribute is a named data value held by each link in an association.' ;
 link association, attribute ;
 heuristic mergingControl ;
 text 'Concurrent subdiagrams in a composite state are automatically terminated when the merge transition fires. This is known as merging of control.' ;
 link concurrentSubdiagram, splittingControl, transition ;
 heuristic metaClass ;
 text 'A metaclass is a class describing other classes.' ;
 link class ;
 heuristic module ;
 text 'Module is a coherent subset of a system containing a tightly bound group of classes and their relationship.' ;
 link objectModel, sheet ;
 heuristic multiplicity ;
 text 'Multiplicity is the number of instances of one class that may relate to a single instance of an associated class.' ;
 link aggregation, association ;
 heuristic nestedDataFlowDiagram ;
 text 'A process can be expanded into a lower-level data flow diagram.' ;
 link DataFlowDiagram, process ;
 heuristic nestedStateDiagram ;
 text 'States and events can both be expanded into nested state diagrams to show greater detail.' ;
 link state, stateDiagram ;
 heuristic object ;
 text 'An object is a concept, abstraction or thing with crisp boundaries and meanings for the problem at hand. It is an instance of a class.' ;
 link class, instance, objectDiagram ;
 heuristic objectDiagram ;
 text 'An object diagram is a graphical representation of the object model showing relationships, attributes and operations.' ;
 link object, objectModel, sheet ;
 heuristic objectModel ;
 text 'An object model describes the structure of the objects in a system including their identity, relationships to other objects, attributes and operations.' ;
 link module, objectDiagram ;
 heuristic objectModellingTechnique ;
 text 'Object Modelling Technique is an object-oriented development methodology that uses object, dynamic and functional models throughout the development life cycle. Abbreviated as OMT.' ;
 link dynamicModel, functionalModel, objectModel ;
 heuristic operation ;
 text 'An operation is a function or transformation that may be applied to objects in a class.' ;
 link action, activity, object, signature ;
 heuristic operationPropagation ;
 text 'Operation propagation is the automatic application of an operation to selected objects in a network when the operation is applied to some starting object in the network.' ;
 link aggregation, association ;
 heuristic process ;
 text 'Process is something that transforms data values.' ;
 link controlFlow, dataFlow, dataFlowDiagram ;
 heuristic qualifiedAssociation ;
 text 'A qualified association is an association that relates two classes and a qualifier; a binary association in which the first part is a composite comprising a class and qualifier, and the second part is a class.' ;
 link association, qualifier ;

heuristic qualifier ;
 text 'A qualifier is an attribute of an object that distinguishes among the set of objects at the "many" end of an association.' ;
 link qualifiedAssociation ;
 heuristic recursiveAggregation ;
 text 'Recursive aggregation has an aggregate that contains, directly or indirectly, an instance of the same kind of aggregate.' ;
 link aggregation, fixedAggregation, variableAggregation ;
 heuristic role ;
 text 'A role is a direction across an association, which is particularly useful in dealing with association between objects of the same class.' ;
 link association, object ;
 heuristic scenario ;
 text 'A scenario is a sequence of events that occur during one particular execution of a system.' ;
 link dynamicModel, eventTrace ;
 heuristic sheet ;
 text 'A sheet is the mechanism for breaking large object models into a series of pages.' ;
 link module, objectDiagram ;
 heuristic signature ;
 text 'A signature is the number and types of its arguments and the type of its result.' ;
 link operation ;
 heuristic splittingControl ;
 text 'A transition on an event can split into concurrent parts, one to each concurrent subdiagram. This is known as splitting of control.' ;
 link concurrentSubdiagram, mergingControl, transition ;
 heuristic state ;
 text 'A state is the values of the attributes and links of an object at a particular time.' ;
 link action, activity, event, nestedStateDiagram, stateDiagram, transition ;
 heuristic stateDiagram ;
 text 'A state diagram is a directed graph in which nodes represent system states and arcs represent transitions between states.' ;
 link concurrentSubdiagram, dynamicModel, nestedStateDiagram ;
 heuristic transition ;
 text 'A transition is a change of state caused by an event.' ;
 rule not(source(startState) and target(stopState)) ;
 link automaticTransition, delegation, event, guardCondition, mergingControl, splittingControl, state ;
 heuristic variableAggregation ;
 text 'Variable aggregation has an aggregate with a finite number of levels but a varying number of parts.' ;
 link aggregation, fixedAggregation, recursiveAggregation ;

%% TASK HEURISTICS

heuristic analysis ;
 text 'Analysis is a stage in the development cycle in which a real-world problem is examined to understand its requirements without planning the implementation.' ;
 rule => perform(analysis) ;
 link implementation, objectDesign, systemDesign ;
 heuristic implementation ;
 text 'A stage in the development cycle in which a design is realised in an executable form, such as a programming language or hardware.' ;
 rule => specify(implementation) ;
 link analysis, objectDesign, systemDesign ;
 heuristic objectDesign ;
 text 'An object design is a stage of the development cycle during which the implementation of each class, association, attribute and operation is determined.' ;
 rule => perform(objectDesign) ;
 link analysis, implementation, systemDesign ;
 heuristic systemDesign ;

text 'System design is the first stage of design, during which high-level decisions are made about the overall structure of the system, its architecture and the strategies used to implement the system.' ;
 rule => perform(systemDesign) ;
 link analysis, implementation, objectDesign ;

heuristic identifyClass ;
 text 'Identifying relevant classes from the application domain. Objects include physical entities as well as concepts; avoid computer implementation constructs.' ;
 rule => perform(identifyClass) ;
 link class, object ;

heuristic insertClass ;
 text 'Listing candidate classes found in the written description of the problem. Classes often correspond to nouns. Don't worry much about inheritance or high-level classes; first get specific classes right so that you don't subconsciously suppress detail in an attempt to fit a preconceived structure.' ;
 rule 'listing candidate classes found in the written description of the problem'
 => insert(class) ;
 link class, object ;

heuristic deleteRedundantClass ;
 text 'If two classes express the same information, the most descriptive one should be kept.' ;
 rule 'two classes express the same information' => delete(class) ;
 link class, object ;

heuristic deleteIrrelevantClass ;
 text 'If a class has little or nothing to do with the problem, it should be eliminated.' ;
 rule 'a class has little or nothing to do with the problem' => delete(class) ;
 link class, object ;

heuristic deleteVagueClass ;
 text 'A class should be specific. Some tentative classes may have ill-defined boundaries or be too broad in scope.' ;
 rule 'a class has ill-defined boundaries or be too broad in scope' => delete(class) ;
 link class, object ;

heuristic retypeClassToAttribute ;
 text 'Names that primarily describe individual objects should be restated as attributes.' ;
 rule 'names that primarily describe individual objects' => retype(class,attribute) ;
 link class, object ;

heuristic retypeClassToOperation ;
 text 'If a name describes an operation that is applied to objects and not manipulated in its own right, then it is not a class.' ;
 rule 'a name describes an operation that is applied to objects'
 => retype(class, operation) ;
 link class, object ;

heuristic retypeClassToAssociation ;
 text 'The name of a class should reflect its intrinsic nature and not a role that it plays in an association.' ;
 rule 'name of a class should reflect its intrinsic nature and not a role'
 => retype(class,association) ;
 link class, object ;

heuristic deleteImplementationConstruct ;
 text 'Constructs extraneous to the real world should be eliminated from the analysis model.' ;
 rule 'constructs extraneous to the real world' => delete(class) ;
 link class, object ;

heuristic specifyClass ;
 text 'Isolated words have too many interpretations, so prepare a data dictionary for all classes and objects.' ;
 rule => specify(class,dataDictionary) ;
 link class, dataDictionary ;

```

heuristic identifyAssociation ;
text    'Identify associations between classes. Any dependency between two or more
        classes is an association; a reference from one class to another is an association.' ;
rule    => perform(identifyAssociation) ;
link    association, class ;
heuristic insertAssociation ;
text    'Extract all the candidates from the problem statement, refine and distinguish
        between association and aggregation in a later stage.' ;
rule    'extract all associations from the problem statement' => insert(association) ;
link    association, class ;
heuristic adjustAssociationBetweenEliminatedClass ;
text    'If one of the classes in the association has been eliminated then association must be
        eliminated or restated in terms of other classes.' ;
rule    'one of the classes in the association has been eliminated'
        => adjust(association, class) ;
link    association, class ;
heuristic deleteIrrelevantAssociation ;
text    'Eliminate any associations that are outside the problem domain or deal with
        implementation constructs.' ;
rule    'any associations that are outside the problem domain or deal with implementation
        constructs' => delete(association) ;
link    association, class ;
heuristic retypeAssociationToOperation ;
text    'An association should describe a structural property of the application domain, not a
        transient event.' ;
rule    'an association that describes a transient event' => retype(association, operation) ;
link    association, class, operation ;
heuristic modifyTernaryAssociation ;
text    'Most associations between three or more classes can be decomposed into binary
        associations or phrased as qualified associations.' ;
rule    'associations between three or more classes that may be decomposed into binary
        associations' => modify(association);
link    association, class ;
heuristic deleteRedundantAssociation ;
text    'Omit associations that can be defined in terms of other associations because they
        are redundant.' ;
rule    'associations that can be defined in terms of other associations'
        => delete(association);
link    association, class, keepRedundantAssociation ;
heuristic retypeRedundantAssociationToDerivedAssociation ;
text    'Retype redundant association to derived association if it is useful in the real world
        and in design.' ;
rule    'a redundant association that is useful in the real world and in design'
        => retype(association, derivedAssociation);
link    association, class, deleteRedundantAssociation ;
heuristic modifyMisnamedAssociation ;
text    'Names are important to understanding and should be chosen with great care. Don't
        say how or why a situation came about, say what it is.' ;
rule    'rename association to say what it is' => modify(association);
link    association, class ;
heuristic insertRoleName ;
text    'Add role names where appropriate. The role name describes the role that a class in
        the association plays from the point of view of the other class.' ;
rule    'add role names where appropriate' => insert(role) ;
link    association, class, role ;
heuristic retypeAssociationToQualifiedAssociation ;
text    'Most names are not globally unique, a qualifier distinguishes objects on the "many"
        side of an association.' ;
rule    'add qualifier to distinguish objects on the "many" side of an association'
        => retype(association, qualifiedAssociation) ;

```

link association, class, qualifiedAssociation, qualifier ;
 heuristic insertMultiplicity ;
 text 'Specify multiplicity, but don't put too much effort into getting it right, as multiplicity often changes during analysis.' ;
 rule 'specify multiplicity of associations' => insert(multiplicity) ;
 link association, multiplicity ;
 heuristic specifyAssociation ;
 text 'Isolated words have too many interpretations, so prepare a data dictionary for all associations.' ;
 rule => specify(association,dataDictionary) ;
 link association, dataDictionary ;

 heuristic identifyAttribute ;
 text 'Identify attributes, which are properties of individual objects.' ;
 rule 'identify the properties of individual objects' => perform(identifyAttribute) ;
 link attribute, class ;
 heuristic insertAttribute ;
 text 'Insert attributes that directly related to a particular application, but do not carry discovery of attributes to excess.' ;
 rule 'insert attributes that directly related to a particular application' => insert(attribute) ;
 link attribute, class ;
 heuristic insertDerivedAttribute ;
 text 'Derived attributes should be omitted or clearly labelled. They should not be expressed as operations, although they may eventually be implemented as such.' ;
 rule 'insert derived attribute but not expressed as operation' => insert(derivedAttribute) ;
 link attribute, class, derivedAttribute ;
 heuristic insertLinkAttribute ;
 text 'Identify link attributes, which are properties of the link between two objects, rather than being properties of individual objects.' ;
 rule 'identify attributes that are properties of the link between two objects' => insert(linkAttribute) ;
 link attribute, class, linkAttribute ;
 heuristic retypeAttributeToObject ;
 text 'If the independent existence of an entity is important, rather than just its value, then it is an object. If an attribute appears to be unique, you may have missed the object class that is being qualified.' ;
 rule 'an attribute appears to be an unique object' => retype(attribute,object) ;
 link attribute, object ;
 heuristic retypeAttributeToQualifier ;
 text 'If the value of an attribute depends on a particular context, then consider restating the attribute as a qualifier.' ;
 rule 'value of an attribute depends on a particular context' => retype(attribute,qualifier) ;
 link attribute, qualifier ;
 heuristic retypeAttributeToAssociation ;
 text 'If an attribute select among objects in a set, the attribute qualifies an association.' ;
 rule 'an attribute select among objects in a set' => retype(attribute,association) ;
 link attribute, association ;
 heuristic deleteAttributeAsIdentifier ;
 text 'An identifier is for unambiguously referencing an object in OO-languages, do not list implementation object identifiers in object model.' ;
 rule 'an implementation object identifiers in object model' => delete(attribute);
 link attribute ;
 heuristic retypeAttributeToLinkAttribute ;
 text 'If a property depends on the presence of a link, then the property is an attribute of the link and not of a related object.' ;
 rule 'a property depends on the presence of a link' => retype(attribute,linkAttribute);
 link attribute, linkAttribute ;
 heuristic deleteAttributeAsInternalValue ;
 text 'If an attribute describes the internal state of an object that is invisible outside the object, then eliminate it from the analysis.' ;

```

rule    'an attribute describes the internal state of an object' => delete(attribute);
link    attribute ;
heuristic deleteAttributeAsFineDetail ;
text    'Omit minor attributes which are unlikely to affect most operations.' ;
rule    'minor attributes which are unlikely to affect most operations' => delete(attribute);
link    attribute ;
heuristic adjustClassFromAttribute ;
text    'An attribute that seems completely different from and unrelated to all other attributes
may indicate a class that should be split into two distinct classes.' ;
rule    'attribute that seems completely different from and unrelated to all other attributes in
the class' => adjust(class, attribute);
link    attribute, class ;
heuristic specifyAttribute ;
text    'Isolated words have too many interpretations, so prepare a data dictionary for all
attributes.'
rule    'prepare a data dictionary for all attributes' => specify(attribute,dataDictionary) ;
link    attribute, dataDictionary ;

heuristic organiseInheritance ;
text    'Organise classes by using inheritance to share common structure. Inheritance can
be added in two directions: by generalising common aspects of existing classes into a
superclass or by existing classes into specialised subclasses.' ;
rule    => do(organiseInheritance) ;
link    generalisation ;
heuristic inheritanceFromGeneralisation ;
text    'Generalisation: search for classes with similar attributes, association or operation;
define a superclass to share common features.' ;
rule    'classes with similar attributes, association or operation' => insert(generalisation) ;
link    generalisation ;
heuristic inheritanceFromSpecialisation ;
text    'Specialisation: look for noun phrases composed of various adjectives on the class
name. Enumerated subcases in the application domain are the most frequent source
of specialisation.' ;
rule    'enumerated subcases in the application domain' => insert(generalisation) ;
link    generalisation ;
heuristic insertMultipleInheritance ;
text    'Multiple inheritance may be used to increase sharing, but only if necessary.' ;
rule    'sharing from multiple classes' => insert(generalisation) ;
link    generalisation ;
heuristic adjustAssociationByInheritance ;
text    'Associations must be assigned to specific classes in the class hierarchy. Each one
should be assigned to the most general class for which it is appropriate.' ;
rule    'associations that are not assigned to specific classes in the class hierarchy'
=> adjust(association,generalisation) ;
link    association, class, generalisation ;
heuristic adjustAttributeByInheritance ;
text    'Attributes must be assigned to specific classes in the class hierarchy. Each one
should be assigned to the most general class for which it is appropriate.' ;
rule    'attributes that are not assigned to specific classes in the class hierarchy'
=> adjust(attribute,generalisation) ;
link    attribute, class, generalisation ;

heuristic testAccessPath ;
text    'Trace access paths through the object model diagram to see if they yield sensible
results. Where a unique value expected, there is a path yielding a unique result,
especially for multiplicity "many". If something that seems simple in the real world
appears complex in the model, you may have missed something.' ;
rule    => modify(objectModel) ;
link    association, attribute, class, generalisation ;

```

```

heuristic verifyObjectModel ;
text    'The entire software development process is one of continual iteration; different parts
        of a model are often at different stages of completion. If a deficiency is found, go
        back to an earlier stage if necessary to correct it. Some refinements can only come
        after the dynamic and functional models are completed.' ;
rule    => perform(verifyObjectModel) ;
link    checkAssociation, checkAttribute, checkClass, objectModel ;
heuristic checkClass ;
text    'There are signs of missing objects and signs of unnecessary classes' ;
rule    => do(checkClass) ;
link    association, attribute, class, operation, verifyObjectModel ;
heuristic checkAssociation ;
text    'There are signs of missing associations, signs of unnecessary associations and
        signs of incorrect placement of associations' ;
rule    => do(checkAssociation) ;
link    association, attribute, class, generalisation, operation, verifyObjectModel ;
heuristic checkAttribute ;
text    'There are signs of incorrect placement of attributes' ;
rule    => do(checkAttribute) ;
link    association, attribute, class, operation, verifyObjectModel ;

heuristic addClassInAsymmetryAssociation ;
text    'If there are asymmetries in associations and generalisations, add new classes by
        analogy.' ;
rule    'asymmetries in associations and generalisations' => insert(class) ;
link    association, class, generalisation ;
heuristic splitClassInDisparateAttribute ;
text    'If there is disparate attributes and operations on a class, split a class so that each
        part is coherent.' ;
rule    'disparate attributes and operations on a class' => adjust(class,attribute) ;
link    attribute, class, operation ;
heuristic splitClassInGeneralisationDifficulty ;
text    'If there is difficulty in generalising cleanly, one class may be playing two roles. Split
        it up and one part may then fit in cleanly.' ;
rule    'difficulty in generalising cleanly and one class plays two roles'
        => adjust(class,generalisation) ;
link    class, generalisation ;
heuristic addMissingTargetClass ;
text    'If an operation has no good target class, add the missing target class.' ;
rule    'an operation has no good target class' => insert(class) ;
link    class, operation ;
heuristic addGeneralisedClass ;
text    'If there are duplicate associations with the same name and purpose, generalise to
        create the missing superclass that unities them.' ;
rule    'duplicate associations with the same name and purpose' => insert(class) ;
link    association, class, generalisation ;
heuristic retypeAssociationToClass ;
text    'If a role substantially shapes the semantics of a class, it may be a separate class,
        this often means converting an association into a class.' ;
rule    'a role substantially shapes the semantics of a class' => retype(association,class) ;
link    association, class ;
heuristic deleteUnnecessaryClass ;
text    'An unnecessary class lacks of attributes, operations and associations. Why is it
        needed?' ;
rule    'a class lacks of attributes, operations and associations' => delete(class) ;
link    association, attribute, class, operation ;
heuristic addAssociationInMissingPath ;
text    'If there are missing access paths for operations, add new associations so that
        queries can be answered.' ;
rule    'missing access paths for operations' => insert(association) ;

```

```

link    association, operation ;
heuristic removeRedundantAssociation ;
text    'If there is redundant information in associations, remove associations that do not
        add new information or mark them as derived. If no operations use a path, the
        information may not be needed. This test must wait until operations are specified.' ;
rule    'redundant information in associations' => delete(association) ;
link    association, class ;
heuristic adjustAssociationInHierarchy ;
text    'If role names are too broad or too narrow for their classes, move the association up
        or down in the class hierarchy.' ;
rule    'role names are too broad or too narrow for their classes'
        => adjust(association,class) ;
link    association, class, generalisation ;
heuristic retypeAttributeToQualifiedAssociation ;
text    'If it is need to access an object by one of its attribute values, consider a qualified
        association.' ;
rule    'need to access an object by one of its attribute values'
        => retype(attribute,qualifiedAssociation) ;
link    class, qualifiedAssociation ;

heuristic groupClassIntoModule ;
text    'The last step of object modelling is to group classes into sheets and modules.
        Diagrams may be divided into sheets of uniform size for convenience in
        drawing, printing and viewing. A module is a set of classes that captures some
        logical subset of the entire model. Each association should generally be shown
        on a single sheet, but some classes must be shown more than once to connect
        different sheets. Look for cut points among the classes: a class that is the sole
        connection between two otherwise disconnected parts of the object network. A
        star-pattern is frequently useful for organising nodules: a single core module
        contains the top-level structure of high-level classes. Reuse a module from a
        previous design if possible, but avoid forcing a fit.' ;
rule    'group classes into sheets and modules' => insert(module) ;
link    class, module, sheet ;
endHeuristicModel

endMethod .

```

APPENDIX G: OMT PROLOG CLAUSES

%% OMT.PL

%% Compiled Prolog Clauses of Object Modelling Technique [Rumbaugh 91]

%% PRODUCT MODEL CLAUSES

concept(objectModellingTechnique, [fragment], concrete) .
property(objectModellingTechnique, objectModel, [1,1,0,1]) .
property(objectModellingTechnique, dynamicModel, [1,1,0,1]) .
property(objectModellingTechnique, functionalModel, [1,1,0,1]) .

%% OBJECT MODEL DEFINITION

concept(objectModel, [fragment], concrete) .
concept(objectDiagram, [diagram], concrete) .
concept(dataDictionary, [text], concrete) .
concept(module, [], concrete) .
concept(sheet, [], concrete) .
concept(signature, [], concrete) .
concept(classConstraint, [], concrete) .
concept(operation, [], concrete) .
concept(abstractOperation, [operation], concrete) .
concept(classOperation, [operation], concrete) .
concept(derivedOperation, [operation], concrete) .
concept(attribute, [], concrete) .
concept(classAttribute, [attribute], concrete) .
concept(derivedAttribute, [attribute], concrete) .
concept(object, [], abstract) .
concept(instance, [object], concrete) .
concept(class, [object], concrete) .
concept(abstractClass, [class], concrete) .
concept(derivedClass, [class], concrete) .
concept(metaClass, [class], concrete) .
concept(relationship, [link], abstract) .
concept(instantiation, [relationship], concrete) .
concept(generalisation, [relationship], concrete) .
concept(association, [relationship], concrete) .
concept(aggregation, [relationship], abstract) .
concept(overlappingGeneralisation, [generalisation], concrete) .
concept(discriminator, [], concrete) .
concept(derivedAssociation, [association], concrete) .
concept(qualifiedAssociation, [association], concrete) .
concept(role, [], abstract) .
concept(sourceRole, [role], concrete) .
concept(targetRole, [role], concrete) .
concept(linkObject, [], concrete) .
concept(linkAttribute, [], concrete) .
concept(associationConstraint, [], concrete) .
concept(multiplicity, [], abstract) .
concept(sourceMultiplicity, [multiplicity], abstract) .
concept(targetMultiplicity, [multiplicity], abstract) .
concept(fixedAggregation, [aggregation], concrete) .
concept(variableAggregation, [aggregation], concrete) .
concept(resursiveAggregation, [aggregation], concrete) .
concept(ordering, [], concrete) .
concept(qualifier, [], concrete) .
concept(operationPropagation, [], concrete) .


```

property(objectModel, module, [1,1,1,3]) .
property(objectModel, objectDiagram, [1,1,1,3]) .
property(objectModel, dataDictionary, [1,1,1,3]) .
property(objectDiagram, object, [1,1,1,3]) .
property(module, sheet, [1,1,1,3]) .
property(operation, signature, [1,1,0,1]) .
property(object, attribute, [1,1,0,3]) .
property(class, classConstraint, [1,1,0,1]) .
property(association, associationConstraint, [1,1,0,1]) .
property(association, linkAttribute, [1,1,0,1]) .
property(association, linkObject, [1,1,0,1]) .
property(association, ordering, [1,1,0,1]) .
property(association, sourceRole, [1,1,0,1]) .
property(association, targetRole, [1,1,0,1]) .
property(association, sourceMultiplicity, [1,1,0,1]) .
property(association, targetMultiplicity, [1,1,0,1]) .
property(association, operationPropagation, [1,1,0,1]) .
property(qualifiedAssociation, qualifier, [1,1,0,1]) .
property(aggregation, sourceMultiplicity, [1,1,0,1]) .
property(aggregation, targetMultiplicity, [1,1,0,1]) .
property(aggregation, operationPropagation, [1,1,0,1]) .
source(relationship, [object], [0,3,1,1]) .
target(relationship, [object], [0,3,1,1]) .
target(instantiation, [instance], [0,3,1,1]) .
reference(discrimator, attribute, [0,1,1,1]) .
reference(linkAttribute, attribute, [0,1,1,1]) .
reference(linkObject, object, [0,1,1,1]) .
reference(objectDiagram, sheet, [1,1,1,1]) .
reference(dataDictionary, association, [1,1,1,3]) .
reference(dataDictionary, attribute, [1,1,1,3]) .
reference(dataDictionary, object, [1,1,1,3]) .
reference(dataDictionary, operation, [1,1,1,3]) .

```

%% DYNAMIC MODEL DEFINITION

```

concept(dynamicModel, [fragment], concrete) .
concept(stateDiagram, [diagram], concrete) .
concept(state, [], abstract) .
concept(startState, [state], concrete) .
concept(interState, [state], concrete) .
concept(stopState, [state], concrete) .
concept(action, [], abstract) .
concept(entryAction, [action], concrete) .
concept(exitAction, [action], concrete) .
concept(internalAction, [action], concrete) .
concept(eventAction, [action], concrete) .
concept(activity, [], concrete) .
concept(automaticTransition, [transition], concrete) .
concept(guardCondition, [], concrete) .
concept(delegation, [], concrete) .
concept(eventAttribute, [], concrete) .
concept(event, [], concrete) .
concept(eventTrace, [], concrete) .
concept(scenario, [], concrete) .
concept(eventGeneralisation, [link], concrete) .
concept(transition, [link], concrete) .
concept(concurrentSubdiagram, [group], concrete) .
concept(mergingControl, [group], concrete) .
concept(nestedStateDiagram, [group], concrete) .
concept(splittingControl, [group], concrete) .
property(dynamicModel, scenario, [1,1,1,3]) .

```

```

property(dynamicModel, stateDiagram, [1,1,1,3]) .
property(stateDiagram, stateState, [1,1,0,1]) .
property(stateDiagram, interState, [1,1,1,3]) .
property(stateDiagram, stopState, [1,1,0,1]) .
property(state, activity, [1,1,0,1]) .
property(state, entryAction, [1,1,0,1]) .
property(state, exitAction, [1,1,0,1]) .
property(state, internalAction, [1,1,0,3]) .
property(event, eventAttribute, [1,1,0,3]) .
property(scenario, eventTrace, [1,1,1,1]) .
property(transition, delegation, [1,1,0,1]) .
property(transition, event, [1,1,0,1]) .
property(transition, guardCondition, [1,1,0,1]) .
source(eventGeneralisation, [event], [0,3,1,1]) .
target(eventGeneralisation, [event], [0,3,1,1]) .
source(transition, [startState, interState], [0,3,0,1]) .
target(transition, [interState, stopState], [0,3,0,1]) .
source(concurrentSubdiagram, [state], [0,1,1,1]) .
target(concurrentSubdiagram, [stateDiagram], [0,1,1,3]) .
source(mergingControl, [transition], [0,1,1,3]) .
target(mergingControl, [transition], [0,1,1,1]) .
source(nestedStateDiagram, [state], [0,1,1,1]) .
target(nestedStateDiagram, [stateDiagram], [0,1,1,1]) .
source(splittingControl, [transition], [0,1,1,1]) .
target(splittingControl, [transition], [0,1,1,3]) .
reference(activity, operation, [0,1,1,1]) .
reference(action, operation, [0,1,1,1]) .
reference(delegation, operation, [0,1,1,1]) .
reference(eventAttribute, attribute, [0,1,1,1]) .
reference(eventTrace, event, [1,1,1,3]) .

%% FUNCTIONAL MODEL DEFINITION
concept(functionalModel, [fragment], concrete) .
concept(dataFlowDiagram, [diagram], concrete) .
concept(actor, [], concrete) .
concept(dataStore, [], concrete) .
concept(process, [], concrete) .
concept(data, [], concrete) .
concept(dataFlowComposition, [group], concrete) .
concept(dataFlowDecomposition, [group], concrete) .
concept(nestedDataFlowDiagram, [group], concrete) .
concept(controlFlow, [link], concrete) .
concept(dataFlow, [link], concrete) .
property(functionalModel, dataFlowDiagram, [1,1,1,3]) .
property(dataFlowDiagram, actor, [1,1,0,3]) .
property(dataFlowDiagram, dataStore, [1,1,0,3]) .
property(dataFlowDiagram, process, [1,1,1,3]) .
property(dataFlow, data, [1,1,1,3]) .
source(controlFlow, [process], [0,3,1,1]) .
target(controlFlow, [process], [0,3,1,1]) .
source(dataFlow, [actor, dataStore, process], [0,3,0,1]) .
target(dataFlow, [actor, dataStore, process], [0,3,0,1]) .
source(dataFlowComposition, [dataFlow], [0,1,2,3]) .
target(dataFlowComposition, [dataFlow], [0,1,1,1]) .
source(dataFlowDecomposition, [dataFlow], [0,1,1,1]) .
target(dataFlowDecomposition, [dataFlow], [0,1,2,3]) .
source(nestedDataFlowDiagram, [process], [0,1,1,1]) .
target(nestedDataFlowDiagram, [dataFlowDiagram], [0,1,1,1]) .
reference(actor, object, [0,1,1,1]) .
reference(data, attribute, [0,1,1,1]) .

```

```
reference(dataStore, object, [0,1,1,1]) .
reference(process, operation, [0,1,1,3]) .
```

%% PRODUCT MODEL RULES

```
transition(T) :- source(T, S1, _), startState(S1), target(T, S2, _), not stopState(S2) .
automaticTransition(A) :- property(A, D, _), not delegation(D) .
automaticTransition(A) :- property(A, E, _), not event(E) .
dataFlow(D) :- source(D, A, _), actor(A), target(D, DS, _), not dataStore(DS) .
dataFlow(D) :- source(D, DS, _), dataStore(DS), target(D, A, _), not actor(A) .
```

%% PROCESS MODEL

```
%% task(TaskName, FunctionName(ContextParameters), Preconditions, Postconditions) .
%% compose(TaskName, SubtaskNames) .
%% refine(TaskName, SubtaskNames) .
```

```
task(objectModellingTechnique, perform(objectModellingTechnique), [problemStatement], [analysis,
systemDesign, objectDesign, implementation]) .
```

```
task(analysis, perform(analysis), [problemStatement], [analysis]) .
task(systemDesign, perform(systemDesign), [analysis], [systemDesign]) .
task(objectDesign, perform(objectDesign), [systemDesign], [objectDesign]) .
task(implementation, specify(objectDesign, implementation), [objectDesign], [implementation]) .
compose(objectModellingTechnique, [analysis, systemDesign, objectDesign, implementation]) .
```

```
task(objectModelling, perform(objectModelling), [problemStatement], [objectModel]) .
task(dynamicModelling, perform(dynamicModelling), [problemStatement], [dynamicModel]) .
task(functionalModelling, perform(functionalModelling), [problemStatement], [functionalModel]) .
compose(analysis, [objectModelling, dynamicModelling, functionalModelling]) .
compose(systemDesign, []) .
compose(objectDesign, []) .
```

```
task(identifyClass, perform(identifyClass), [problemStatement], [class, instance, object]) .
task(identifyAssociation, perform(identifyAssociation), [problemStatement, class], [association]) .
task(identifyAttribute, perform(identifyAttribute), [problemStatement, class], [attribute]) .
task(organiseInheritance, perform(organiseInheritance), [problemStatement, class], [generalisation]) .
task(testAccessPath, modify(objectModel), [objectModel], []) .
task(verifyObjectModel, perform(verifyObjectModel), [objectModel], []) .
task(groupClassIntoModule, insert(module), [objectModel], [module, sheet]) .
compose(objectModelling, [identifyClass, identifyAssociation, identifyAttribute, organiseInheritance,
testAccessPath, verifyObjectModel, groupClassIntoModule]) .
compose(dynamicModelling, []) .
compose(functionalModelling, []) .
```

```
task(insertClass, insert(class), [problemStatement], [class]) .
task(verifyClass, do(verifyClass), [problemStatement, class], []) .
task(specifyClass, specify(class, dataDictionary), [problemStatement, class], [dataDictionary]) .
compose(identifyClass, [insertClass, verifyClass, specifyClass]) .
```

```
task(deleteRedundantClass, delete(class), [class], []).
task(deleteIrrelevantClass, delete(class), [class], []).
task(deleteVagueClass, delete(class), [class], []).
task(retypeClassToAttribute, retype(class, attribute), [class], [attribute]).
task(retypeClassToOperation, retype(class, operation), [class], [operation]).
task(retypeClassToAssociation, retype(class, association), [class], [association]).
task(deleteImplementationConstruct, delete(class), [class], []).
refine(verifyClass, [deleteRedundantClass, deleteIrrelevantClass, deleteVagueClass,
retypeClassToAttribute, retypeClassToOperation, retypeClassToAssociation,
deleteImplementationConstruct]) .
```

```
task(insertAssociation, insert(association), [problemStatement, class], [association]) .
```

```

task(verifyAssociation, do(verifyAssociation), [association], []).
task(specifyAssociation, specify(association, dataDictionary), [association], [dataDictionary]).
compose(identifyAssociation, [insertAssociation, verifyAssociation, specifyAssociation]).

task(adjustAssociationBetweenEliminatedClass, adjust(association, class), [association, class], []).
task(deleteIrrelevantAssociation, delete(association), [association], []).
task(retypeAssociationToOperation, retype(association, operation), [association], [operation]).
task(modifyTernaryAssociation, modify(association), [association], []).
task(deleteRedundantAssociation, delete(association), [association], []).
task(retypeAssociationToDerivedAssociation, retype(association, derivedAssociation), [association],
[derivedAssociation]).
task(modifyMisnamedAssociation, modify(association), [association], []).
task(insertRoleName, insert(role), [association], [role]).
task(retypeAssociationToQualifiedAssociation, retype(association, qualifiedAssociation),
[association], [qualifiedAssociation]).
task(insertMultiplicity, insert(multiplicity), [association], [multiplicity]).
refine(verifyAssociation, [adjustAssociationBetweenEliminatedClass, deleteIrrelevantAssociation,
retypeAssociationToOperation, modifyTernaryAssociation, deleteRedundantAssociation,
retypeAssociationToDerivedAssociation, modifyMisnamedAssociation, insertRoleName,
retypeAssociationToQualifiedAssociation, insertMultiplicity]).

task(insertAttribute, insert(attribute), [problemStatement, class], [attribute]).
task(insertDerivedAttribute, insert(derivedAttribute), [problemStatement, class], [derivedAttribute]).
task(insertLinkAttribute, insert(linkAttribute), [problemStatement, class], [linkAttribute]).
task(verifyAttribute, do(verifyAttribute), [attribute], []).
task(specifyAssociation, specify(attribute, dataDictionary), [attribute], [dataDictionary]).
compose(identifyAttribute, [insertAttribute, insertDerivedAttribute, insertLinkAttribute, verifyAttribute,
specifyAttribute]).

task(retypeAttributeToObject, retype(attribute, object), [attribute], [object]).
task(retypeAttributeToQualifier, retype(attribute, qualifier), [attribute], [qualifier]).
task(retypeAttributeToOperation, retype(attribute, operation), [attribute], [operation]).
task(deleteAttributeAsIdentifier, delete(attribute), [attribute], []).
task(retypeAttributeToLinkAttribute, retype(attribute, linkAttribute), [attribute], [linkAttribute]).
task(deleteAttributeAsInternalValue, delete(attribute), [attribute], []).
task(deleteAttributeAsFineDetail, delete(attribute), [attribute], []).
task(adjustClassFromAttribute, adjust(class, attribute), [attribute], [class]).
refine(verifyAttribute, [retypeAttributeToObject, retypeAttributeToQualifier,
retypeAttributeToOperation, deleteAttributeAsIdentifier, retypeAttributeToLinkAttribute,
deleteAttributeAsInternalValue, deleteAttributeAsFineDetail, adjustClassFromAttribute]).

task(insertGeneralisation, perform(insertGeneralisation), [problemStatement], [generalisation]).
task(adjustAssociationByInheritance, adjust(association, generalisation), [association, generalisation],
[]).
task(adjustAttributeByInheritance, adjust(attribute, generalisation), [attribute, generalisation], []).
compose(organiseInheritance, [insertGeneralisation, adjustAssociationByInheritance,
adjustAttributeByInheritance]).

task(inheritanceFromGeneralisation, insert(generalisation), [problemStatement], [generalisation]).
task(inheritanceFromSpecialisation, insert(generalisation), [problemStatement], [generalisation]).
task(insertMultipleInheritance, insert(generalisation), [problemStatement], [generalisation]).
compose(insertGeneralisation, [inheritanceFromGeneralisation, inheritanceFromSpecialisation,
insertMultipleInheritance]).

task(checkClass, do(checkClass), [class, objectModel], []).
task(checkAssociation, do(checkAssociation), [association, objectModel], []).
task(checkAttribute, do(checkAttribute), [attribute, objectModel], []).
compose(verifyObjectModel, [checkClass, checkAssociation, checkAttribute]).

task(addClassInAsymmetryAssociation, adjust(class, association), [class, association], []).

```

```

task(splitClassInDisparateAttribute, adjust(class, attribute), [attribute, class], []) .
task(splitClassInGeneralisationDifficulty, adjust(class, generalisation), [class, generalisation], []) .
task(addMissingTargetClass, adjust(class, operation), [class, operation], []) .
task(addGeneralisedClass, adjust(class, generalisation), [class, generalisation], []) .
task(retypeAssociationToClass, retype(association, class), [association, class], [class]) .
task(deleteUnnecessaryClass, delete(class), [class], []) .
refine(checkClass, [addClassInAsymmetryAssociation, splitClassInDisparateAttribute,
splitClassInGeneralisationDifficulty, addMissingTargetClass, addGeneralisedClass,
retypeAssociationToClass, deleteUnnecessaryClass]) .

```

```

task(addAssociationInMissingPath, insert(association), [association], []) .
task(removeRedundantAssociation, delete(association), [association], []) .
task(adjustAssociationInHierarchy, adjust(association, generalisation), [association, generalisation],
[]) .
refine(checkAssociation, [addAssociationInMissingPath, removeRedundantAssociation,
adjustAssociationInHierarchy]) .

```

```

task(retypeAttributeToQualifiedAssociation, retype(attribute, qualifiedAssociation), [attribute],
[qualifiedAssociation]) .
refine(checkAttribute, [retypeAttributeToQualifiedAssociation]) .

```

%% HEURISTIC MODEL

%% CONCEPT HEURISTICS

```

heuristic(abstractClass, [abstractOperation, class, generalisation],
'Abstract class is a class that cannot have direct instances but whose descendants can have
instances.') .

```

```

heuristic(abstractOperation, [abstractClass, class, generalisation],
'Abstract operation is an operation defined but not implemented by an abstract class. The operation
must be implemented by all concrete descendent classes.') .

```

```

heuristic(action, [action, activity, entryAction, exitAction, internalAction, operation],
'An action is an instantaneous operation. Actions are associated with events and are usually formal in
nature.') .

```

```

heuristic(activity, [action, operation],
'An activity is an operation that takes time to complete. Activities are associated with states and
represent real-world accomplishments.') .

```

```

heuristic(actor, [dataFlowDiagram, object],
'Actor object is an active object that drives the data flow graph by producing or consuming values.') .

```

```

heuristic(aggregation, [fixedAggregation, resursiveAggregation, variableAggregation],
'Aggregation is a special form of association, between a whole and its parts, in which the whole is
composed of the parts.') .

```

```

heuristic(association, [associationConstraint, derivedAssociation, linkAttribute, multiplicity,
qualifiedAssociation, qualifier, role],
'Association is a relationship among instances of two or more classes describing a group of links with
common structure and common semantics.') .

```

```

heuristic(associationConstraint, [association, linkAttribute, multiplicity, qualifier],
'Association constraint is a functional relationship of association; a statement about some condition or
relationship that must be maintained as true.') .

```

```

heuristic(attribute, [class, object],
'An attribute is a named property of a class describing a data value held by each object of the class.')
.

```

heuristic(automaticTransition, [event, guardcondition, transition],
 'Automatic transition is an unlabelled transition that automatically fires when the activity associated with the source state is completed.') .

heuristic(class, [abstractClass, classConstraint, metaClass, object],
 'A class is a description of a group of objects with similar properties, common behaviour, common relationships and common semantics.') .

heuristic(classAttribute, [attribute, class],
 'Class attribute is an attribute whose value is common to a class of objects rather than a value peculiar to each instance.') .

heuristic(classConstraint, [class],
 'Class constraint is a functional relationship of class; a statement about some condition or relationship that must be maintained as true.') .

heuristic(classOperation, [class, operation],
 'Class operation is an operation on a class, rather than on instances of the class. An instance creation operation is a common example.') .

heuristic(concurrentSubdiagram, [state, stateDiagram],
 'Concurrency within the state of a single object arises when the object can be partitioned into subsets of attributes or links, each of which has its own subdiagram.') .

heuristic(controlFlow, [process],
 'Control flow is a boolean value that affects whether a process is executed.') .

heuristic(dataDictionary, [association, attribute, class, operation],
 'A data dictionary is a textual description of each class, its associations, attributes and operations.') .

heuristic(dataFlow, [actor, dataFlowDecomposition, dataFlowDiagram, dataStore, process],
 'Data flow is the connection between the output of one object or process and the input to another.') .

heuristic(dataFlowDecomposition, [dataFlow, dataFlowComposition],
 'Data flow decomposition split an aggregate data on a data flow into its components.') .

heuristic(dataFlowDiagram, [dataFlow, functionalModel, nestedDataFlowDiagram, process],
 'A data flow diagram is a graphical representation of the functional model, showing dependencies between values and the computation of output values from input values without regard for when or if the functions are executed.') .

heuristic(dataStore, [dataFlowDiagram, process],
 'A data store is a passive object that stores data for later access.') .

heuristic(delegation, [object, transition],
 'Delegation is an implementation mechanism in which an object, responding to an operation on itself, forwards the operation to another object.') .

heuristic(derivedAssociation, [association],
 'Derived association is an association that is defined in terms of other associations.') .

heuristic(derivedAttribute, [attribute],
 'Derived attribute is an attribute that is computed from other attributes.') .

heuristic(discriminator, [generalisation],
 'A discriminator is an attribute of enumeration type that indicates which property of a class is being abstracted by a particular generalisation.') .

heuristic(dynamicModel, [scenario, stateTransitionDiagram],

heuristic(dynamicModel, [action, exitAction],
'A dynamic model describes the aspects of a system concerned with control, including time, sequencing of operations and interaction of objects.') .

heuristic(entryAction, [action, exitAction],
'Entry action permits action to be associated with a state, to indicate all the transitions entering the state.') .

heuristic(event, [dynamicModel, eventGeneralisation, eventTrace, scenario, state, transition],
'An event is something that happens instantaneously at a point in time.') .

heuristic(eventGeneralisation, [event],
'Events can be organised into a generalisation hierarchy with inheritance of event attributes.') .

heuristic(eventTrace, [event, scenario],
'An event trace is a diagram that shows the sender and receiver of events and the sequence of events.') .

heuristic(exitAction, [action, entryAction],
'Exit action permits action to be associated with a state, to indicate all the transitions exiting the state.') .

heuristic(fixedAggregation, [aggregation, recursiveAggregation, variableAggregation],
'Fixed aggregation has an aggregate with a predefined number and types of components.') .

heuristic(functionalModel, [dataFlowDiagram],
'A functional model describes the aspects of a system that transform values using functions, mappings, constraints and functional dependencies.') .

heuristic(generalisation, [discriminator],
'Generalisation is the relationship between a class and one or more refined or specialised versions of it.') .

heuristic(guardCondition, [transition],
'Guard condition is a boolean expression that must be true in order for a transition to occur.') .

heuristic(instance, [class, instantiation],
'An instance is an object described by a class.') .

heuristic(instantiation, [class, instance, object],
'Instantiation is the process of creating instances from classes.') .

heuristic(linkAttribute, [association, attribute],
'A link attribute is a named data value held by each link in an association.') .

heuristic(mergingControl, [concurrentSubdiagram, splittingControl, transition],
'Concurrent subdiagrams in a composite state are automatically terminated when the merge transition fires. This is known as merging of control.') .

heuristic(metaClass, [class],
'A metaclass is a class describing other classes.') .

heuristic(module, [objectModel, sheet],
'Module is a coherent subset of a system containing a tightly bound group of classes and their relationship.') .

heuristic(multiplicity, [aggregation, association],
'Multiplicity is the number of instances of one class that may relate to a single instance of an associated class.') .

heuristic(nestedDataFlowDiagram, [DataFlowDiagram, process],

'A process can be expanded into a lower-level data flow diagram.') .

heuristic(nestedStateDiagram, [state, stateDiagram],

'States and events can both be expanded into nested state diagrams to show greater detail.') .

heuristic(object, [class, instance, objectDiagram],

'An object is a concept, abstraction or thing with crisp boundaries and meanings for the problem at hand. It is an instance of a class.') .

heuristic(objectDiagram, [object, objectModel, sheet],

'An object diagram is a graphical representation of the object model showing relationships, attributes and operations.') .

heuristic(objectModel, [module, objectDiagram],

'An object model describes the structure of the objects in a system including their identity, relationships to other objects, attributes and operations.') .

heuristic(objectModellingTechnique, [dynamicModel, functionalModel, objectModel],

'Object Modelling Technique is an object-oriented development methodology that uses object, dynamic and functional models throughout the development life cycle. Abbreviated OMT.') .

heuristic(operation, [action, activity, object, signature],

'An operation is a function or transformation that may be applied to objects in a class.') .

heuristic(operationPropagation, [aggregation, association],

'Operation propagation is the automatic application of an operation to selected objects in a network when the operation is applied to some starting object in the network.') .

heuristic(process, [controlFlow, dataFlow, dataFlowDiagram],

'Process is something that transforms data values.') .

heuristic(qualifiedAssociation, [association, qualifier],

'A qualified association is an association that relates two classes and a qualifier; a binary association in which the first part is a composite comprising a class and qualifier, and the second part is a class.') .

heuristic(qualifier, [qualifiedAssociation],

'A qualifier is an attribute of an object that distinguishes among the set of objects at the "many" end of an association.') .

heuristic(recursiveAggregation, [aggregation, fixedAggregation, variableAggregation],

'Recursive aggregation has an aggregate that contains, directly or indirectly, an instance of the same kind of aggregate.') .

heuristic(role, [association, object],

'A role is a direction across an association, which is particularly useful in dealing with association between objects of the same class.') .

heuristic(scenario, [dynamicModel, eventTrace],

'A scenario is a sequence of events that occur during one particular execution of a system.') .

heuristic(sheet, [module, objectDiagram],

'A sheet is the mechanism for breaking large object models into a series of pages.') .

heuristic(signature, [operation],

'A signature is the number and types of its arguments and the type of its result.') .

heuristic(splittingControl, [concurrentSubdiagram, mergingControl, transition],

'A transition on an event can split into concurrent parts, one to each concurrent subdiagram. This is known as splitting of control.') .

heuristic(state, [action, activity, event, nestedStateDiagram, stateDiagram, transition],
'A state is the values of the attributes and links of an object at a particular time.') .

heuristic(stateDiagram, [concurrentSubdiagram, dynamicModel, nestedStateDiagram],
'A state diagram is a directed graph in which nodes represent system states and arcs represent transitions between states.') .

heuristic(transition, [automaticTransition, delegation, event, guardCondition, mergingControl, splittingControl, state],
'A transition is a change of state caused by an event.') .

heuristic(variableAggregation, [aggregation, fixedAggregation, recursiveAggregation],
'Variable aggregation has an aggregate with a finite number of levels but a varying number of parts.')
.

%% TASK HEURISTICS

heuristic(analysis, [implementation, objectDesign, systemDesign],
'Analysis is a stage in the development cycle in which a real-world problem is examined to understand its requirements without planning the implementation.') .

heuristic(implementation, [analysis, objectDesign, systemDesign],
'A stage in the development cycle in which a design is realised in an executable form, such as a programming language or hardware.') .

heuristic(objectDesign, [analysis, implementation, systemDesign],
'An object design is a stage of the development cycle during which the implementation of each class, association, attribute and operation is determined.') .

heuristic(systemDesign, [analysis, implementation, objectDesign],
'System design is the first stage of design, during which high-level decisions are made about the overall structure of the system, its architecture and the strategies used to implement the system.') .

heuristic(identifyClass, [class, object],
'Identifying relevant classes from the application domain. Objects include physical entities as well as concepts; avoid computer implementation constructs.') .

heuristic(insertClass, [class, object],
'Listing candidate classes found in the written description of the problem. Classes often correspond to nouns. Don't worry much about inheritance or high-level classes; first get specific classes right so that you don't subconsciously suppress detail in an attempt to fit a preconceived structure.') .

heuristic(deleteRedundantClass, [class, object],
'If two classes express the same information, the most descriptive one should be kept.') .

heuristic(deleteIrrelevantClass, [class, object],
'If a class has little or nothing to do with the problem, it should be eliminated.') .

heuristic(deleteVagueClass, [class, object],
'A class should be specific. Some tentative classes may have ill-defined boundaries or be too broad in scope.') .

heuristic(retypeClassToAttribute, [class, object],
'Names that primarily describe individual objects should be restated as attributes.') .

heuristic(retypeClassToOperation, [class, object],
'If a name describes an operation that is applied to objects and not manipulated in its own right, then it is not a class.') .

heuristic(retypeClassToAssociation, [class, object],
'The name of a class should reflect its intrinsic nature and not a role that it plays in an association.') .

heuristic(deleteImplementationConstruct, [class, object],
'Constructs extraneous to the real world should be eliminated from the analysis model.') .

heuristic(specifyClass, [class, dataDictionary],
'Isolated words have too many interpretations, so prepare a data dictionary for all classes and objects.') .

```

heuristic(identifyAssociation, [association, class],
'Identify associations between classes. Any dependency between two or more classes is an
association; a reference from one class to another is an association.') .
heuristic(insertAssociation, [association, class],
'Extract all the candidates from the problem statement, refine and distinguish between association
and aggregation in a later stage.') .
heuristic(adjustAssociationBetweenEliminatedClass, [association, class],
'If one of the classes in the association has been eliminated then association must be eliminated or
restated in terms of other classes.') .
heuristic(deleteIrrelevantAssociation, [association, class],
'Eliminate any associations that are outside the problem domain or deal with implementation
constructs.') .
heuristic(retypeAssociationToOperation, [association, class, operation],
'An association should describe a structural property of the application domain, not a transient
event.') .
heuristic(modifyTernaryAssociation, [association, class],
'Most associations between three or more classes can be decomposed into binary associations or
phrased as qualified associations.') .
heuristic(deleteRedundantAssociation, [association, class, keepRedundantAssociation],
'Omit associations that can be defined in terms of other associations because they are redundant.') .
heuristic(keepRedundantAssociation, [association, class, deleteRedundantAssociation],
'The existence of an association can be derived from two or more primitive associations and the
multiplicity cannot. Retype the extra association if the additional multiplicity constraint is important.') .
heuristic(retypeAssociationToDerivedAssociation, [association, class, derivedAssociation],
'Retype redundant association to derived association if it is useful in the real world and in design.') .
heuristic(modifyMisnamedAssociation, [association, class],
'Names are important to understanding and should be chosen with great care. Don't say how or why a
situation came about, say what it is.') .
heuristic(insertRoleName, [association, class, role],
'Add role names where appropriate. The role name describes the role that a class in the association
plays from the point of view of the other class.') .
heuristic(retypeAssociationToQualifiedAssociation, [association, class, qualifiedAssociation,
qualifier],
'Most names are not globally unique, a qualifier distinguishes objects on the "many" side of an
association.') .
heuristic(insertMultiplicity, [association, multiplicity],
'Specify multiplicity, but don't put too much effort into getting it right, as multiplicity often changes
during analysis.') .
heuristic(specifyAssociation, [association, dataDictionary],
'Isolated words have too many interpretations, so prepare a data dictionary for all associations.') .

heuristic(identifyAttribute, [attribute, class],
'Identify attributes, which are properties of individual objects.') .
heuristic(insertAttribute, [attribute, class],
'Insert attributes that directly related to a particular application, but do not carry discovery of attributes
to excess.') .
heuristic(insertDerivedAttribute, [attribute, class, derivedAttribute],
'Derived attributes should be omitted or clearly labelled. They should not be expressed as operations,
although they may eventually be implemented as such.') .
heuristic(insertLinkAttribute, [attribute, class, linkAttribute],
'Identify link attributes, which are properties of the link between two objects, rather than being
properties of individual objects.') .
heuristic(retypeAttributeToObject, [attribute, object],
'If the independent existence of an entity is important, rather than just its value, then it is an object. If
an attribute appears to be unique, you may have missed the object class that is being qualified.') .
heuristic(retypeAttributeToQualifier, [attribute, qualifier],
'If the value of an attribute depends on a particular context, then consider restating the attribute as a
qualifier.') .
heuristic(retypeAttributeToAssociation, [attribute, association],

```

'If an attribute select among objects in a set, the attribute qualifies an association.') .
 heuristic(deleteAttributeAsIdentifier, [attribute],
 'An identifier is for unambiguously referencing an object in OO-languages, do not list implementation object identifiers in object model.') .
 heuristic(retypeAttributeToLinkAttribute, [attribute, linkAttribute],
 'If a property depends on the presence of a link, then the property is an attribute of the link and not of a related object.') .
 heuristic(deleteAttributeAsInternalValue, [attribute],
 'If an attribute describes the internal state of an object that is invisible outside the object, then eliminate it from the analysis.') .
 heuristic(deleteAttributeAsFineDetail, [attribute],
 'Omit minor attributes which are unlikely to affect most operations.') .
 heuristic(adjustClassFromAttribute, [attribute, class],
 'An attribute that seems completely different from and unrelated to all other attributes may indicate a class that should be split into two distinct classes.') .
 heuristic(specifyAttribute, [attribute, dataDictionary],
 'Isolated words have too many interpretations, so prepare a data dictionary for all attributes.') .

heuristic(organiseInheritance, [generalisation],
 'Organise classes by using inheritance to share common structure. Inheritance can be added in two directions: by generalising common aspects of existing classes into a superclass or by existing classes into specialised subclasses.') .
 heuristic(inheritanceFromGeneralisation, [generalisation],
 'Generalisation: search for classes with similar attributes, association or operation; define a superclass to share common features.') .
 heuristic(inheritanceFromSpecialisation, [generalisation],
 'Specialisation: look for noun phrases composed of various adjectives on the class name. Enumerated subcases in the application domain are the most frequent source of specialisation.') .
 heuristic(insertMultipleInheritance, [generalisation],
 'Multiple inheritance may be used to increase sharing, but only if necessary.') .
 heuristic(adjustAssociationByInheritance, [association, class, generalisation],
 'Associations must be assigned to specific classes in the class hierarchy. Each one should be assigned to the most general class for which it is appropriate.') .
 heuristic(adjustAttributeByInheritance, [attribute, class, generalisation],
 'Attributes must be assigned to specific classes in the class hierarchy. Each one should be assigned to the most general class for which it is appropriate.') .

heuristic(testAccessPath, [association, attribute, class, generalisation],
 'Trace access paths through the object model diagram to see if they yield sensible results. Where a unique value expected, there is a path yielding a unique result, especially for multiplicity "many". If something that seems simple in the real world appears complex in the model, you may have missed something.') .

heuristic(verifyObjectModel, [objectModel],
 'The entire software development process is one of continual iteration; different parts of a model are often at different stages of completion. If a deficiency is found, go back to an earlier stage if necessary to correct it. Some refinements can only come after the dynamic and functional models are completed.') .

heuristic(addClassInAsymmetryAssociation, [association, class, generalisation],
 'If there are asymmetries in associations and generalisations, add new classes by analogy.') .
 heuristic(splitClassInDisparateAttribute, [attribute, class, operation],
 'If there are disparate attributes and operations on a class, split a class so that each part is coherent.') .

heuristic(splitClassInGeneralisationDifficulty, [class, generalisation],
 'If there is difficulty in generalising cleanly, one class may be playing two roles. Split it up and one part may then fit in cleanly.') .
 heuristic(addMissingTargetClass, [class, operation],
 'If an operation has no good target class, add the missing target class.') .
 heuristic(addGeneralisedClass, [association, class, generalisation],

'If there are duplicate associations with the same name and purpose, generalise to create the missing superclass that unites them.') .

heuristic(retypeAssociationToClass, [association, class],

'If a role substantially shapes the semantics of a class, it may be a separate class, this often means converting an association into a class.') .

heuristic(deleteUnnecessaryClass, [association, attribute, class, operation],

'An unnecessary class lacks of attributes, operations and associations. Why is it needed?') .

heuristic(addAssociationInMissingPath, [association, operation],

'If there are missing access paths for operations, add new associations so that queries can be answered.') .

heuristic(removeRedundantAssociation, [association, class],

'If there are redundant information in associations, remove associations that do not add new information or mark them as derived. If no operations use a path, the information may not be needed. This test must wait until operations are specified.') .

heuristic(adjustAssociationInHierarchy, [association, class, generalisation],

'If role names are too broad or too narrow for their classes, move the association up or down in the class hierarchy.') .

heuristic(retypeAttributeToQualifiedAssociation, [class, qualifiedAssociation],

'If it is need to access an object by one of its attribute values, consider a qualified association.') .

heuristic(groupClassIntoModule, [class, module, sheet],

'The last step of object modelling is to group classes into sheets and modules. Diagrams may be divided into sheets of uniform size for convenience in drawing, printing and viewing. A module is a set of classes that captures some logical subset of the entire model. Each association should generally be shown on a single sheet, but some classes must be shown more than once to connect different sheets. Look for cut points among the classes: a class that is the sole connection between two otherwise disconnected parts of the object network. A star-pattern is frequently useful for organising nodules: a single core module contains the top-level structure of high-level classes. Reuse a module from a previous design if possible, but avoid forcing a fit.') .

%% HEURISTIC RULES

rule(insertClass, 'listing candidate classes found in the written description of the problem',
insert(class)).

rule(deleteRedundantClass, 'two classes express the same information', delete(class)).

rule(deleteIrrelevantClass, 'a class has little or nothing to do with the problem', delete(class)).

rule(deleteVagueClass, 'a class has ill-defined boundaries or be too broad in scope', delete(class)).

rule(retypeClassToAttribute, 'names that primarily describe individual objects',
retype(class,attribute)).

rule(retypeClassToOperation, 'a name describes an operation that is applied to objects',
retype(class, operation)).

rule(retypeClassToAssociation, 'name of a class should reflect its intrinsic nature and not a role',
retype(class,association)).

rule(deleteImplementationConstruct, 'constructs extraneous to the real world', delete(class)).

rule(insertAssociation, 'extract all associations from the problem statement', insert(association)).

rule(adjustAssociationBetweenEliminatedClass,

'one of the classes in the association has been eliminated', adjust(association, class)).

rule(deleteIrrelevantAssociation, 'any associations that are outside the problem domain or deal with
implementation constructs', delete(association) ;

rule(retypeAssociationToOperation, 'an association that describes a transient event',
retype(association, operation) ;

rule(modifyTernaryAssociation, 'associations between three or more classes that may be
decomposed into binary associations', modify(association)).

rule(deleteRedundantAssociation, 'associations that can be defined in terms of other associations',
delete(association)).

rule(retypeRedundantAssociationToDerivedAssociation, 'a redundant association that is useful in the
real world and in design', retype(association,derivedAssociation)).

rule(modifyMisnamedAssociation, 'rename association to say what it is', modify(association)).

rule(insertRoleName, 'add role names where appropriate', insert(role)).

rule(retypeAssociationToQualifiedAssociation, 'add qualifier to distinguish objects on the "many" side

```

    of an association', retype(association,qualifiedAssociation) ).
rule(insertMultiplicity, 'specify multiplicity of associations', insert(multiplicity) ).
rule(insertAttribute, 'insert attributes that directly related to a particular application', insert(attribute) ).
rule(insertDerivedAttribute, 'insert derived attribute but not expressed as operation',
    insert(derivedAttribute) ).
rule(insertLinkAttribute, 'identify attributes that are properties of the link between two objects',
    insert(linkAttribute) ).
rule(retypeAttributeToObject, 'an attribute appears to be an unique object', retype(attribute,object) ).
rule(retypeAttributeToQualifier, 'value of an attribute depends on a particular context'
    retype(attribute,qualifier) ).
rule(retypeAttributeToAssociation, 'an attribute select among objects in a set'
    retype(attribute,association) ).
rule(deleteAttributeAsIdentifier, 'an implementation object identifiers in object model'
    delete(attribute) ).
rule(retypeAttributeToLinkAttribute, 'a property depends on the presence of a link',
    retype(attribute,linkAttribute) ).
rule(deleteAttributeAsInternalValue, 'an attribute describes the internal state of an object'
    delete(attribute) ).
rule(deleteAttributeAsFineDetail, 'minor attributes which are unlikely to affect most operations',
    delete(attribute) ).
rule(adjustClassFromAttribute, 'attribute that seems completely different from and unrelated to all
    other attributes in the class', adjust(class, attribute) ).
rule(inheritanceFromGeneralisation, 'classes with similar attributes, association or operation'
    insert(generalisation) ).
rule(inheritanceFromSpecialisation, 'enumerated subcases in the application domain',
    insert(generalisation) ).
rule(insertMultipleInheritance, 'sharing from multiple classes', insert(generalisation) ).
rule(adjustAssociationByInheritance, 'associations that are not assigned to specific classes in the
    class hierarchy' => adjust(association,generalisation) ).
rule(adjustAttributeByInheritance, 'attributes that are not assigned to specific classes in the class
    hierarchy' => adjust(attribute,generalisation) ).
rule(addClassInAsymmetryAssociation, 'asymmetries in associations and generalisations',
    insert(class) ).
rule(splitClassInDisparateAttribute, 'disparate attributes and operations on a class'
    adjust(class,attribute) ).
rule(splitClassInGeneralisationDifficulty, 'difficulty in generalising cleanly and one class plays two
    roles' => adjust(class,generalisation) ).
rule(addMissingTargetClass, 'an operation has no good target class', insert(class) ).
rule(addGeneralisedClass, 'duplicate associations with the same name and purpose', insert(class) ).
rule(retypeAssociationToClass, 'a role substantially shapes the semantics of a class'
    retype(association,class) ).
rule(deleteUnnecessaryClass, 'a class lacks of attributes, operations and associations' delete(class) ).
rule(addAssociationInMissingPath, 'missing access paths for operations', insert(association) ).
rule(removeRedundantAssociation, 'redundant information in associations', delete(association) ).
rule(adjustAssociationInHierarchy, 'role names are too broad or too narrow for their classes'
    adjust(association,class) ).
rule(retypeAttributeToQualifiedAssociation, 'need to access an object by one of its attribute values'
    retype(attribute,qualifiedAssociation) ).
rule(groupClassIntoModule, 'group classes into sheets and modules' => insert(module) ).

```

BIBLIOGRAPHY

BASIC READING

- [Booch 91] Booch G.: *Object Oriented Design with Applications*. Redwood City CA: Benjamin/Cummings, 1991.
- [Gomaa 93] Gomaa H.: *Software Design Methods for Concurrent and Real-Time Systems*. Addison-Wesley, 1993.
- [IPSYS 92] *IPSYS Toolbuilder 1.3*, Manual, Issue 1.1. IPSYS Macclesfield, Cheshire, England, 1992.
- [Martin 92] Martin J. and Odell J.J.: *Object-Oriented Analysis and Design*. Englewood Cliffs NJ:Prentice-Hall, 1992.
- [Robinson 92] Robinson P.J.: *Hierarchical Object-Oriented Design*. Object-Oriented Series: Prentice Hall, 1992.
- [Rumbaugh 91] Rumbaugh J., Blaha M., Premerlani W., Eddy F. and Lorensen W.: *Object-Oriented Modelling and Design*. Prentice Hall, 1991.
- [Schreiber 93] Schreiber G., Wielinga B. and Breuker J.: *KADS - A Principled Approach to Knowledge-Based System Development*. Academic Press, 1993.

CHAPTER ONE : INTRODUCTION

- [Davis 83] Davis W.S.: *Systems Analysis and Design - A Structured Approach*. Addison-Wesley, 1983.
- [Freeman 92] Freeman P.A.: Needed Research in Systems Analysis and Design. Cotterman W.W. and Senn J.A. (Eds.): *Challenges and Strategies for Research in Systems Development*, chapter 3, pp 23-31. John Wiley & Sons, 1992.
- [Gaines 87] Gaines B.R.: An Overview of Knowledge-Acquisition and Transfer. *Internation Journal of Man-Machine Studies*, 26, pp 453-472. 1987.
- [Gillies 94] Gillies A.C. and Smith P.: *Managing Software Engineering: CASE Studies & Solutions*. Chapman & Hall, 1994.
- [Gilb 88] Gilb T.: *Principles of Software Engineering Management*. Addison-Wesley, 1988.

- [Hirschheim 92] Hirschheim R. and Klein H.K.: Future Information Systems Development Methodologies. Cotterman W.W. and Senn J.A. (Eds.): *Challenges and Strategies for Research in Systems Development*, chapter 14, pp 235-255. John Wiley & Sons, 1992.
- [Jackson 92] Jackson M.A.: The General and the Particular. Cotterman W.W. and Senn J.A. (Eds.): *Challenges and Strategies for Research in Systems Development*, chapter 4, pp 33-40. John Wiley & Sons, 1992.
- [Klir 76] Klir G.J. (1976). Identification of Generative Structures in Empirical Data. *International Journal of General Systems*, 3, pp 89-104, Gordon and Breach Science Publishers Ltd.
- [Madsen 95] Madsen J.: The Reptilian World of Software Team. Myers C. (Ed.): *Professional Awareness in Software Engineering*, chapter 7, pp 91-109. McGraw-Hill, 1995.
- [PACT 85] PA Computers & Telecommunications and Department of Trade & Industry: *Benefits of Software Engineering Methods and Tools*. 1985.
- [Popper 68] Popper K. R.. Epistemology without a knowledge subject. van Rootselaar B. (Ed.): *Logic, Methodology and Philosophy of Science III*, pp 333-373. Amsterdam, Holland: North-Holland Publishing Co., 1968.
- [Pressman 87] Pressman R.S.: *Software Engineering - A Practitioner's Approach*. 2nd Edn. McGraw-Hill, 1987.
- [Sommerville 89] Sommerville I.: *Software Engineering*. Addison-Wesley, 1989.
- [Tontsch 90] Tontsch F.: Methods and Tools. Mitchell R.J. (Ed.): *Managing Complexity in Software Engineering*, chapter 10, pp 181-199. Peter Peregrinus, 1990.
- [Ward 89] Ward P.T.: How to Integrate Object-Orientation with Structured Analysis and Design. *IEEE Software*, 6, pp 74-82, March 1989.
- [Wong 93] Wong C.Y.: *Transfer Report*. MPhil/PhD Research, Sheffield Hallam University, 1993.
- [Wynekoop 93] Wynekoop J.L. and Russo N.L.: System development methodologies: unanswered questions and the research-practice gap. *Proceeding 14th ICIS*, pp 181-190, 1993.
- [Yourdon 89] Yourdon E.: *Modern Structured Analysis*. Englewood-Cliffs, NJ: Prentice Hall, 1989.

CHAPTER TWO : INVESTIGATION OF SOFTWARE DEVELOPMENT METHODS

- [Blair 91] Blair G., Gallagher J., Hutchison D. and Shepherd D.: *Object-Oriented Languages, Systems and Applications*. Pitman, 1991.
- [Booch 86] Booch G.: *Object-Oriented Development*. IEEE Trans. on Software Eng., Vol SE-12(2), pp 211-221, 1986.
- [Booch 94] Booch G.: *Object-Oriented Analysis and Design with Applications*. Benjamin Cummings, 1994.
- [Champeaux 91] Champeaux D.: Object-Oriented Analysis and Top-Down Software Development. *ECOOP 91*, Proceedings of the European Conference on Object-Oriented Programming, pp 361-376, 1991.
- [Chen 76] Chen P.: The entity-relationship model: towards a unified view of data. *ACM Transaction on Database Systems*, 1(1), pp 9-36, 1976.
- [Coad 90] Coad P. and Yourdon E.: *Object-Oriented Analysis*. Englewood Cliffs, NJ: Yourdon Press/Prentice Hall, 1991.
- [Coad 91a] Coad P. and Yourdon E.: *Object-Oriented Analysis*, 2nd edn. Englewood Cliffs, NJ: Yourdon Press/Prentice Hall, 1991.
- [Coad 91b] Coad P. and Yourdon E.: *Object-Oriented Design*. Englewood Cliffs, NJ: Yourdon Press/Prentice Hall, 1991.
- [Coad 93] Coad P. and Nicola J.: *Object-Oriented Programming*. Englewood Cliffs, NJ: Yourdon Press/Prentice Hall, 1993.
- [Coleman 94] Coleman D., Arnold P., Bodoff S., Dollin C., Gilchrist H., Hayes F. and Jeremaes P.: *Object-Oriented Development - The Fusion Method*. Prentice Hall, 1994.
- [Cox 87] Cox B.J.: *Object-Oriented Programming - An Evolutionary Approach*. Addison-Wesley, 1987.
- [DeMarco 79] DeMarco T.: *Structured Analysis and System Specification*. Yourdon Press/Prentice Hall, 1979.
- [Downs 91] Downs E., Coe I. and Clare P.: *Structured Systems Analysis and Design Method: Application and Context*, 2nd edn. Prentice Hall, 1991.
- [Firesmith 93] Firesmith D.G.: *Object-Oriented Requirements Analysis and Design: A Software Engineering Approach*. Chichester: Wiley, 1993.
- [Graham 91] Graham I.: *Object Oriented Methods*. Addison-Wesley, 1991.
- [Graham 94] Graham I.: *Object Oriented Methods*, 2nd Edn. Addison-Wesley, 1994.

- [HOOD 91] HOOD Working Group: *HOOD Reference Manual*, Issue 3.1. European Space Agency, Noordwijk, Netherlands, 1991.
- [Jackson 75] Jackson M.A.: *Principles of Program Design*. Academic Press, 1975.
- [Jackson 83] Jackson M.A.: *System Development*. Prentice-Hall, 1983.
- [Jacobson 92] Jacobson I., Christerson M., Jonsson P. and Overgaard G.: *Object-Oriented Software Engineering: A Use Case Driven Approach*. Wokingham: Addison-Wesley, 1992.
- [Khoshafian 90] Khoshafian S. and Abnous R.: *Object Orientation - Concepts, Languages, Databases, User Interfaces*. John Wiley & Sons, 1990.
- [Luger 89] Luger G.F. and Stubblefield W.A.: *Artificial Intelligence and the Design of Expert Systems*. Redwood City CA: Benjamin/Cumming, 1989.
- [Martin 95] Martin J. and Odell J.J.: *Object-Oriented Methods*. Englewood Cliffs, NJ: Prentice Hall, 1995.
- [Masini 91] Masini G., Napoli A., Colnet D., Leonard D. and Tombre K.: *Object-Oriented Languages*. APIC Series volume 34, Academic Press, 1991.
- [Meyer 88] Meyer B.: *Object-Oriented Software Construction*. Englewood Cliffs NJ: Prentice-Hall, 1988.
- [Moss 94] Moss C.: *Prolog++: The Power of ObjectOriented and Logic Programming*. Addison-Wesley, 1994.
- [Nielsen 88] Nielsen K.W. and Shumate K.C.: *Designing Large Real-Time Systems with Ada*. McGraw-Hill, NewYork, 1988.
- [Nielsen 91] Nielsen K.W.: *Ada in Distributed Real-Time Systems*. McGraw-Hill, 1991.
- [Nielsen 92] Nielsen K.W.: *Object-Oriented Design with Ada: Maximizing Reusability for Real-Time Systems*. Bantam Books, 1992.
- [Perschke 93] Perschke S. and Liczbanski M.: *Access for Windows - Power Programming*, Que, 1993.
- [Plihon 95] Phlihon V. and Rolland C.: Modelling Ways-of-Working. Iivari J., Lyytinen K. and Rossi M. (Eds.): *Advanced Information Systems Engineering, 6th International Conference CAiSE '95 Proceedings*. LNCS 932, pp 126-139, Springer-Verlag, 1995.
- [Rich 91] Rich E. and Knight K.: *Artificial Intelligence*. McGraw-Hill, 1991.
- [Rossi 95] Rossi M. and Brinkkemper S.: Metrics in Method Engineering. Iivari J., Lyytinen K. and Rossi M. (Eds.): *Advanced Information Systems*

Engineering, 6th International Conference CAiSE '95 Proceedings. LNCS 932, pp 200-216, Springer-Verlag, 1995.

- [Shlaer 88] Shlaer S. and Mellor S.J.: *Object-Oriented Systems Analysis - Modelling the World in Data*. Englewood Cliffs, NJ: Yourdon Press, 1988.
- [Shlaer 91] Shlaer S. and Mellor S.J.: *Object-Lifecycles - Modelling the World in States*. Englewood Cliffs, NJ: Yourdon Press, 1991.
- [Sowa 84] Sowa J.F.: *Conceptual Structures*. Reading, MA: Addison-Wesley, 1984.
- [Stroustrup 86] Stroustrup B.: *The C++ Programming Language*. Reading, MA: Addison-Wesley, 1986.
- [Tansley 93] Tansley D.S.W. and Hayball C.C.: *Knowledge-Based Systems Analysis and Design: a KADS Developer's Handbook*. The BCS Practitioner Series, Prentice-Hall, 1993.
- [Taylor 92] Taylor D.A.: *Object-Oriented Technology: A manager's Guide*. Addison-Wesley, 1992.
- [Tello 89] Tello E.: *Object-Oriented Programming for Artificial Intelligence: A guide to Tools and System Design*. Addison-Wesley, 1989.
- [Ward 85] Ward P.T. and Mellor S.J.: *Structured Development for Real-Time Systems, Volumn 1: Introduction to Tools; Volume 2: Essential Modelling Techniques; Volume 3: Implementation Modelling Techniques*. New York, Yourdon Press, 1985.
- [Ward 89] Ward P.T.: How to Integrate Object-Orientation with Structured Analysis and Design. *IEEE Software*, 6, pp 74-82, March 1989.
- [Wasserman 90] Wasserman A.I., Pircher P.A. and Muller R.J.: The Object-Oriented Structured Design Notation for Software Design Representation. *IEEE Computer*, pp 50-62, March 1990.
- [Wirfs-Brock 90] Wirfs-Brock R., Wilkerson B. and Wiener L.: *Designing Object-Oriented Software*. Englewood Cliffs, NJ: Prentice Hall, 1990.
- [Yourdon 79] Yourdon E. and Constantine L.L.: *Modern Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Englewood-Cliffs, NJ: Prentice Hall, 1979.
- [Yourdon 89] Yourdon E.: *Modern Structured Analysis*. Englewood-Cliffs, NJ: Prentice Hall, 1989.

CHAPTER THREE : INVESTIGATION OF METHOD INTEGRATION, META MODELLING RESEARCHES AND METACASE TOOLS

- [Alderson 91] Alderson A.: Meta-CASE Technology. Endres A. and Weber H. (Eds.): *Software Development Environments and CASE Technology*, European Symposium Proceedings. LNCS 509, Springer-Verlag, pp 81-91, 1991.
- [Brinkkemper 90] Brinkkemper S. and ter Hofstede A.H.M.: The Conceptual Task Model: a Specification Technique between Requirements Engineering and Program Development. Steinholtz B., SØlvberg A. and Bergman L. (Eds.): *Advanced Information Systems Engineering, 2th International Conference CAiSE '90 Proceedings*. LNCS 436, Springer-Verlag, pp 228-250, 1990.
- [Brinkkemper 93] Brinkkemper S.: Integrating Diagram in CASE Tools through Modelling Transparency. *Information and Software Technology*. 35(2), pp 101-105, January, 1993.
- [Brough 92] Brough M.: Methods for CASE: a Generic Framework. Loucopoulos P. (Ed.): *Advanced Information Systems Engineering, 4th International Conference CAiSE '92 Proceedings*. LNCS 593, Springer-Verlag, pp 524-545, 1992.
- [Carmichael 94] Carmichael A.: Towards a Common Object-Oriented Meta-Model for Object Development. Carmichael A. (Ed.): *Object Development Methods*. pp 321-334, 1994.
- [Daley 93] Daley R.: Integration Technologies Available. Daley R. (Ed.): *Integration Technology for CASE*. Avebury Technical. pp 3-20, 1993.
- [Dewal 92] Dewal S.: A Methodology for Requirements Analysis and Evaluation of SDEs. Loucopoulos P. (Ed.): *Advanced Information Systems Engineering, 4th International Conference CAiSE '92 Proceedings*. LNCS 593, Springer-Verlag, pp 394-409, 1992.
- [ECMA 90] ECMA: Portable Common Tool Environment (PCTE), Abstract Specification. Standard ecma-149, European Computer Manufacturers Association, December 1990.
- [Finkelstein 92] Finkelstein A., Kramer J., Nuseibeh B., Finkelstein L. and Goedicke M.: Viewpoints: A Framework for Integrating Multiple Perspectives in System Development. *International Journal of Software Engineering and Knowledge Engineering*. 1(2), pp 31-58, World Scientific, 1992.

- [Grosz 91] Grosz G. and Rolland C.: Why and HOW should we hide Conceptual Models? *3rd International Conference of Software Engineering and Knowledge Engineering*. pp 28-33, 1991.
- [Gulla 91] Gulla J.A., Lindland O.I. and Willumsen G.: PPP - An Integrated CASE Environment. Andersen R., Bubenko J.A.jr. and Sølvsberg A. (Eds.): *Advanced Information Systems Engineering, Third International Conference CAiSE '91 Proceedings*. LNCS 498, pp 194-221, Springer-Verlag, 1991.
- [Harmsen 94] Harmsen F., Brinkkemper S., Oei H.: Situational method engineering for information system project approaches. *IFIP Transactions A [Computer Science and Technology]*, Vol A-55, pp 169-94, 1994.
- [Hofstede 92] ter Hofstede A.H.M., Verhoef T.F., Nieuwland E.R. and Wijers G.M.: Integrated Specification of Method and Graphic Knowledge. *Proceedings of 4th International Conference of Software Engineering and Knowledge Engineering*. pp 307-316, August, 1992.
- [Hofstede 93a] ter Hofstede A.H.M. and Nieuwland E.R.: Task Structure Semantics through Process Algebra. *Software Engineering Journal*. pp 14-20, January, 1993.
- [Hofstede 93b] ter Hofstede A.H.M. and van der Weide Th.P.: Expressiveness in Conceptual Data Modelling. *Data & Knowledge Engineering*. 10(1993), pp 65-100, 1993.
- [Imber 91] Imber M.: CASE Data Interchange Format standards. *Information and Software Technology*. 33(9), pp 647-706, November, 1991.
- [Kronlöf 93] Kronlöf K., Sheehan A. and Hallmann M.: The Concept of Method Integration. Kronlöf K. (Ed.): *Method Integration - Concepts and Case Studies*. John Wiley & Sons. 1993.
- [MarkV 93] Mark V Systems: *ObjectMaker Tutorial includes Installation Guide*. 1993.
- [Marttiin 94] Marttiin P.: Towards Flexible Process Support with a CASE Shell. Wijers G., Brinkkemper S. and Wasserman T. (Eds.): *Advanced Information Systems Engineering, 6th International Conference CAiSE '94 Proceedings*. LNCS 811, Springer-Verlag, pp 14-27, 1994.
- [Nilsen 92] Nilsen E., HeeSen Jong, Olson J.S. and Polson P.G.: Method engineering: from data to model to practice. Bauersveld P., Bennet J. and Lynch G.: *CHI '92 Conference Proceedings*. ACM Conference on Human Factors in Computing Systems, pp 313-320. ACM Press, 1992.

- [Oquendo 90] Oquendo F., Zucker J.-D. and Tassard G.: Support for Software Tool Integration & Process-centred Software Engineering Environments. *Toulouse '90, Third International Workshop, Software Engineering and its Applications Proceedings*. Vol. 1, EC2, pp 135-155, 1992.
- [Oquendo 91] Oquendo F., Boudier G., Gallo F., Minot R. and Thomas I.: The PCTE+'s OMS: A Software Engineering Distributed Database System for supporting Large-Scale Software Development Environments. Makinouchi A. (Ed.): *Database Systems for Advanced Applications '91*. World Scientific Publishing Co., pp 41-50, 1991.
- [Oquendo 92] Oquendo F., Zucker J.-D. and Griffiths P.: A Meta-CASE Environment for Software Process-centred CASE Environments. Loucopoulos P. (Ed.): *Advanced Information Systems Engineering, 4th International Conference CAiSE '92 Proceedings*. LNCS 593, Springer-Verlag, pp 568-588, 1992.
- [Oquendo 93] Oquendo F., Detienne F., Gallo N., Kästner A. and Martelli A.: SCALE: Building PCTE-based Process-centred Environments for Large and Fine Grain Reuse. *PCTE Technical Journal*, issue 1, pp 469-485. pp 469-485, 1993.
- [Potts 89] Potts C.: A Generic Model for Representing Design Methods. *Proceedings of 11th International Conference on Software Engineering*. IEEE Computer Society Press, pp 217-226, 1989.
- [Rossi 92] Rossi M., Gustafsson M., Smolander K., Johansson L.-Å. and Lyytinen K.: Metamodelling Editor as a Front End Tool for a CASE Shell. Loucopoulos P. (Ed.): *Advanced Information Systems Engineering, 4th International Conference CAiSE '92 Proceedings*. LNCS 593, Springer-Verlag, pp 546-567, 1992.
- [Rossi 95] Rossi M. and Brinkkemper S.: Metrics in Method Engineering. Iivari J., Lyytinen K. and Rossi M. (Eds.): *Advanced Information Systems Engineering, 6th International Conference CAiSE '95 Proceedings*. LNCS 932, pp 200-216, Springer-Verlag, 1995.
- [Saeki 93a] Saeki M., Iguchi K., Wen-yin K. and Shinohara M.: A Meta-Model for Representing Software Specification & Design Methods. Prakash N., Rolland C. and Pernici B. (Eds): *Information System Development Process (A-30)*. Elsevier Science Publishers, B.V. (North-Holland) IFIP, pp 149-166, 1993.

- [Saeki 93b] Saeki M., Iguchi K. and Shinohara M.: Supporting Tool for Cooperative Specification Processes. *International Conference on Software Engineering and Knowledge Engineering, SEKE'93 Proceeding*. pp 351-354, 1993.
- [Saeki 94a] Saeki M. and Wen-yin K.: PCTE based Tool for Supporting Collaborative Specification Development. *PCTE Technical Journal*, issue 2, pp 121-134, 1994.
- [Saeki 94b] Saeki M. and Wen-yin K.: Specifying Software Specification & Design Methods. Wijers G., Brinkkemper S. and Wasserman T. (Eds.): *Advanced Information Systems Engineering, 6th International Conference CAiSE '94 Proceedings*. LNCS 811, Springer-Verlag, pp 353-366, 1994.
- [Schefström 93] Schefström D. and van den Broek G.: *Tool Integration - Environment and Framework*. John Wiley & Sons. 1993.
- [Siau 92] Siau K.L., Chan H.C. and Tan K.P.: A CASE tool for Conceptual Database design. *Information and Software Technology*, Vol 34, No 12, December, 1992.
- [Simon 93] Simon A.R.: *The Integrated CASE Tools Handbook*. Van Nostrand Reinhold. 1993.
- [Slooten 93] Slooten van K. and Brinkkemper B.: A Method Engineering Approach to Information Systems Development. Prakash N., Rolland C. and Pernici B. (Eds.): *Information System Development Process*. A-30, Elsevier Science Publishers, North-Holland, IFIP, pp 167-186, 1993.
- [Smolander 91] Smolander K., Lyytinen K., Tahvanainen V.-P. and Marttiin P.: MetaEdit - A Flexible Graphical Environment for Methodology Modelling. Andersen R., Bubenko J.A.jr. and Sølvsberg A. (Eds.): *Advanced Information Systems Engineering, Third International Conference CAiSE '91 Proceedings*. LNCS 498, Springer-Verlag, pp 168-193, 1991.
- [Smolander 92] Smolander K.: OPRR - A Model for Modelling Systems Development Methods. Lyytinen K. and Tahvanainen V.-P. (Eds.): *Next Generation CASE Tools*. IOS Press, pp 224-239, 1992.
- [Sorenson 88] Sorenson P.G., Tremblay J.-P. and McAllister A.J.: The Metaview System for Many Specification Environments. *IEEE Software*. 2(5), pp 30-38, March, 1988.
- [Spurr 92] Spurr K. and Layzell P.: *CASE - Current Practice Future Prospects*. John Wiley & Sons, 1992.

- [Stobart 91] Stobart S.C., Thompson J.B. and Smith P.: An Examination of Computer Aided Software Engineering Tools as an Aid to Software Reliability. Matthews R.H.: *Reliability '91, International Conference on Reliability Techniques and Their Application*. Elsevier Applied Science, pp 152-161, 1991.
- [Thompson 93] Thompson K.: Data Interchange between CASE Tools: PCTE, IRDS or CDIF? Daley R. (Ed.): *Integration Technology for CASE*. Avebury Technical. pp 45-70, 1993.
- [Tolvanen 93] Tolvanen J.-P. and Lyytinen K.: Flexible Method Adaptation in CASE - The Metamodelling Approach. *Scandinavian Journal of Information Systems*. Vol 5, pp 51-77, 1993.
- [Verhoef 91] Verhoef T.F., Hofstede A.H.M. ter and Wijers G.M.: Structuring Modelling Knowledge for CASE shells. Andersen R., Bubenko J.A.jr. and Sølvsberg A. (Eds.): *Advanced Information Systems Engineering, Third International Conference CAiSE '91 Proceedings*. LNCS 498, Springer-Verlag, pp 502-524, 1991.
- [Wen-yin 92] Wen-yin K. and Saeki M.: Method Base: Database of Software Design Methods & Specifications. *Joint Conference on Software Engineering '92 (JCSE'92)*. 1992.
- [Wijers 90a] Wijers G.M. and van Dort H.E.: Experiences with the Use of CASE-tools in the Netherlands. Steinholtz B., Sølvsberg A. and Bergman L. (Eds.): *Advanced Information Systems Engineering, 2th International Conference CAiSE '90 Proceedings*. LNCS 436, Springer-Verlag, pp 5-20, 1990.
- [Wijers 90b] Wijers G.M. and Heijes H.: Automated Support of the Modelling Process - A view based on experiments with expert information engineers. Steinholtz B., Sølvsberg A. and Bergman L. (Eds.): *Advanced Information Systems Engineering, 2th International Conference CAiSE '90 Proceedings*. LNCS 436, Springer-Verlag, pp 88-108, 1990.
- [Wijers 92] Wijers G.M., Hofstede A.H.M. ter and van Oosterom, N.E.: Representation of Information Modelling Knowledge. Lyytinen K. and Tahvanainen V.-P. (Eds.): *Next Generation CASE Tools*. Studies in Computer and Communication Systems, IOS Press, pp 167-223, 1992.

CHAPTER FOUR : SEMANTIC KNOWLEDGE BASE

- [Boehm 86] Boehm B.: A Spiral Model of Software Development and Enhancement. *Software Engineering Notes*, vol 11(4), pp 22, August 1986.
- [Connell 95] Connell J. and Shafer L.: *Object-Oriented Rapid Prototyping*. Yourdon Press Computing Series, Prentice Hall, 1995.
- [Lyytinen 89] Lyytinen K., Smolander K. and Tahvanainen V.-P.: *Modelling CASE Environments in Systems Work*. CASE89 conference papers, Kista, Sweden. 1989.
- [Mullin 90] Mullin M.: *Rapid Prototyping in Object-Oriented Systems*. Addison-Wesley, 1990.
- [Prakash 94] Prakash N.: *A Process View of Methodologies*. Wijers G., Brinkkemper S. and Wasserman T. (Eds.): *Advanced Information Systems Engineering, 6th International Conference CAiSE '94 Proceedings*. LNCS 811, Springer-Verlag, pp 339-352, 1994.
- [Rossi 92] Rossi M., Gustafsson M., Smolander K., Johansson L.-Å. and Lyytinen K.: Metamodelling Editor as a Front End Tool for a CASE Shell. Loucopoulos P. (Ed.): *Advanced Information Systems Engineering, 4th International Conference CAiSE '92 Proceedings*. LNCS 593, Springer-Verlag, pp 546-567, 1992.
- [Seligmann 89] Seligmann P.S., Wijers G.M. and Sol H.G.: *Analyzing the structure of I.S. methodologies, an alternative approach*. Proceedings of the 1st Dutch Conference on Information Systems, Amesfoort, The Netherlands, 1989.
- [Steel 94] Steel B.D.: LPA Prolog for Windows Programming Guide, Logic Programming Associates Ltd, 1994.
- [Wynekoop 93] Wynekoop J.d. and Russo N.L.: System Development Methodologies: unanswered questions and the research-practice gap. *Proceedings of 14th ICIS*. Orlando, USA, pp 181-190, 1993.

CHAPTER FIVE : PRODUCT MODEL

- [Carmichael 94] Carmichael A.: Towards a Common Object-Oriented Meta-Model for Object Development. Carmichael A. (Ed.): *Object Development Methods*. pp 321-334, 1994.
- [Cribbs 92] Cribbs J., Moon S. and Roe C.: *An Evaluation of Object-Oriented Analysis and Design Methodologies*. Alcatel Network Systems, SIGS Books, 1992.

- [Davis 92] Davis A.M. and Jordan K.: *A Canonical Representation for Requirements*. 1992.
- [Imber 91] Imber M.: CASE Data Interchange Format Standards. *Information and Software Technology*, 33(9), pp 647-655, November 1991.
- [Kronlöf 93] Kronlöf K., Sheehan A. and Hallmann M.: The Concept of Method Integration. Kronlöf K. (Ed.): *Method Integration - Concepts and Case Studies*. John Wiley & Sons. 1993.
- [Mattos 91] Mattos N.M.: *An Approach to Knowledge Base Management*. Springer-Verlag, 1991.
- [Norcliffe 91] Norcliffe A. and Slater G.: *Mathematics of Software Construction*. Ellis Horwood, 1991.
- [Reichgelt 90] Reichgelt H.: *Knowledge Representation: An AI Perspective*. Ablex, 1990.
- [Short 91] Short K.W.: Methodology Integration: Evolution of Information Engineering. *Information and Software Technology*, 33(9), pp 720-731, November 1991.
- [Song 94] Song X. and Osterweil L.J.: Experience with an Approach to Comparing Software Design Methodologies. *IEEE Transactions on Software Engineering*, 20(5), pp 364-384, May 1994.
- [Sowa 91] Sowa J.F.: *Principles of Semantic Networks: Explorations in the Representation of Knowledge*. Morgan Kaufmann, 1991.
- [Wijers 92] Wijers G.M., Hofstede A.H.M. ter and van Oosterom, N.E.: Representation of Information Modelling Knowledge. Lyytinen K. and Tahvanainen V.-P. (Eds.): *Next Generation CASE Tools*. Studies in Computer and Communication Systems, IOS Press, pp 167-223, 1992.

CHAPTER SIX : PROCESS MODEL

- [Alderson 91] Alderson A.: Meta-CASE Technology. Endres A. and Weber H. (Eds.): *Software Development Environments and CASE Technology, European Symposium Proceedings*. LNCS 509, Springer-Verlag, pp 81-91, 1991.
- [Fisher 88] Fisher A.S.: *CASE: Using the Newest Tools in Software Developments*. John Wiley & Sons, 1988.
- [Oquendo 92] Oquendo F., Zucker J.D. and Griffiths P.: A Meta-CASE Environment for Software Process-centred CASE Environments. Loucopoulos P. (Ed.):

Advanced Information Systems Engineering, 4th International Conference CAiSE '92 Proceedings. LNCS 593, Springer-Verlag, pp 568-588, 1992.

- [Saeki 93a] Saeki M., Iguchi K., Wen-yin K. and Shinohara M.: A Meta-Model for Representing Software Specification & Design Methods. Prakash N., Rolland C. and Pernici B. (Eds.): *Information System Development Process (A-30)*. Elsevier Science Publishers, B.V. (North-Holland) IFIP, pp 149-166, 1993.
- [Smolander 91] Smolander K., Lyytinen K., Tahvanainen V.-P. and Marttiin P.: MetaEdit - A Flexible Graphical Environment for Methodology Modelling. Andersen R., Bubenko J.A.jr. and Sølvsberg A. (Eds.): *Advanced Information Systems Engineering*, Third International Conference CAiSE '91 Proceedings. LNCS 498, Springer-Verlag, pp 168-193, 1991.
- [Wasserman 90] Wasserman A.I., Pircher P.A. and Muller R.J.: The Object-Oriented Structured Design Notation for Software Design Representation. *IEEE Computer*, pp 50-62, March 1990.
- [Wijers 92] Wijers G.M., Hofstede A.H.M. ter and van Oosterom, N.E.: Representation of Information Modelling Knowledge. Lyytinen K. and Tahvanainen V.-P. (Eds.): *Next Generation CASE Tools*. Studies in Computer and Communication Systems, IOS Press, pp 167-223, 1992.

CHAPTER SEVEN : HEURISTIC MODEL

- [Causse 93] Causse K.: Heuristic Control Knowledge. Aussenac N., Boy G., Gaines B., Linster M., Ganascia J.-G. and Kodratoff Y. (Eds.): *Knowledge Acquisition for Knowledge-Based Systems*. Proceedings of 7th European Workshop, EKAW '93, LNCS 723, pp 1-22, Springer-Verlag, 1993.
- [Clancey 85] Clancey W.J.: Heuristic Classification. Bobrow D.G. and Hayes P.J. (Eds.): *Artificial Intelligence*, Elsevier Science Publishers B.V., 27, pp 289-350, 1985.

CHAPTER EIGHT : METHOD REPRESENTATION

- [Carmichael 94] Carmichael A.: Towards a Common Object-Oriented Meta-Model for Object Development. Carmichael A. (Ed.): *Object Development Methods*. pp 321-334, 1994.
- [Tello 89] Tello E.: *Object-Oriented Programming for Artificial Intelligence: A guide to Tools and System Design*. Addison-Wesley, 1989.

CHAPTER NINE : METHOD EVALUATION

- [Cribbs 92] Cribbs J., Moon S. and Roe C.: *An Evaluation of Object-Oriented Analysis and Design Methodologies*. Alcatel Network Systems, SIGS Books, 1992.
- [Graham 94] Graham I.: *Object Oriented Methods*. 2nd Edn., Addison-Wesley, 1994.
- [Jayaratna 94] Jayaratna N.: *Understanding and Evaluating Methodologies - NIMSAD: A Systemic Framework*. McGraw-Hill, 1994.
- [Law 88] Law D.: *Method for Comparing Methods: Techniques in Software Development*. NCC Publications, 1988.
- [Masini 91] Masini G., Napoli A., Colnet D., Leonard D. and Tombre K.: *Object-Oriented Languages*. APIC Series volume 34, Academic Press, 1991.
- [Shlaer 88] Shlaer S. and Mellor S.J.: *Object-Oriented Systems Analysis - Modelling the World in Data*. Englewood Cliffs, NJ: Yourdon Press, 1988.
- [Short 91] Short K.W.: Methodology Integration: Evolution of Information Engineering. *Information and Software Technology*, 33(9) November, 1991.
- [Wasserman 90] Wasserman A.I., Pircher P.A. and Muller R.J.: The Object-Oriented Structured Design Notation for Software Design Representation. *IEEE Computer*, pp 50-62, March 1990.

CHAPTER TEN : KNOWLEDGE ACQUISITION OF METHOD MODELS

- [Boose 90] Boose J.H. and Gaines B.R.: *The Foundations of Knowledge-Acquisition*. Academic Press, 1990.
- [Causse 93] Causse K.: Heuristic Control Knowledge. Aussenac N., Boy G., Gaines B., Linster M., Ganascia J.-G. and Kodratoff Y. (Eds.): Knowledge Acquisition for Knowledge-Based Systems. *Proceedings of 7th European Workshop, EKAW '93*, LNCS 723, pp 1-22, Springer-Verlag, 1993.
- [Clancey 85] Clancey W.J.: Heuristic Classification. Bobrow D.G. and Hayes P.J. (Eds.): *Artificial Intelligence*, Elsevier Science Publishers B.V., 27, pp 289-350, 1985.
- [Gaines 87] Gaines B.R.: An Overview of Knowledge-Acquisition and Transfer. *International Journal of Man-Machine Studies*, 26, pp 453-472, 1987.
- [Gaines 93] Gaines B.R.: Modelling and Extending Expertise. Aussenac N., Boy G., Gaines B., Linster M., Ganascia J.-G. and Kodratoff Y. (Eds.): *Knowledge*

Acquisition for Knowledge-Based Systems. Proceedings of 7th European Workshop, EKAW '93, LNCS 723, pp 1-22, Springer-Verlag, 1993.

- [Goodman 73] Goodman N.: *Fact, Fiction and Forecast*. Indianapolis: Bobbs-Merrill, 1973.
- [Hayward 87] Hayward S.A., Wielinga B.J. and Breuker J.A.: Structured Analysis of Knowledge. *International Journal of Man-Machine Studies*, 26, pp 487-498, 1987.
- [Hull 87] Hull R. and King R.: Semantic database modelling: Survey, application and research issues. *ACM Computing Surveys*, 19, pp 201-260, 1987.
- [Kelly 55] Kelly G.A.: *The Psychology of Personal Constructs*. New York: Plenum Press, 1955.
- [Neubert 93] Neubert S.: Model Construction in MIKE (Model Based and Incremental Knowledge Engineering). Aussenac N., Boy G., Gaines B., Linster M., Ganascia J.-G. and Kodratoff Y. (Eds.): *Knowledge Acquisition for Knowledge-Based Systems*. Proceedings of 7th European Workshop, EKAW '93, LNCS 723, pp 200-220, Springer-Verlag, 1993.
- [Shaw 93] Shaw M.L.G. and Gaines B.R.: Personal Construct Psychology Foundations for Knowledge Acquisition and Representation. Aussenac N., Boy G., Gaines B., Linster M., Ganascia J.-G. and Kodratoff Y. (Eds.): *Knowledge Acquisition for Knowledge-Based Systems*. Proceedings of 7th European Workshop, EKAW '93, LNCS 723, pp 256-276, Springer-Verlag, 1993.
- [Stich 84] Stich S.P. and Nisbett R.E.: Expertise, Justification and the Psychology of Inductive Reasoning. Haskell T.L. (Ed.): *The Authority of Experts*, pp 226-241. Bloomington, Indiana: Indiana University Press, 1984.
- [Tansley 93] Tansley D.S.W. and Hayball C.C.: *Knowledge-Based Systems Analysis and Design: a KADS Developer's Handbook*. The BCS Practitioner Series, Prentice-Hall, 1993.
- [Wielinga 93] Wielinga B.J., Bredeweg B. and Breuker J.A.: Knowledge Acquisition for Expert Systems. Nossum R.T. (Ed.): *Advanced Topics in Artificial Intelligence*. 2nd Advanced Course, ACAI '87, LNCS 345, pp 96-124, Springer-Verlag, 1993.

CHAPTER ELEVEN : MAPPING METHOD SEMANTICS TO METACASE TOOLS

- [Gillies 94] Gillies A.C. and Smith P.: *Managing Software Engineering: CASE Studies & Solutions*. Chapman & Hall, 1994.

[Goodman 73] Goodman N.: *Fact, Fiction and Forecast*. Indianapolis: Bobbs-Merrill, 1973.

CHAPTER TWELVE : CONCLUSION

[Boyce 92] Boyce J.: *Maximizing Windows 3.1*. New Riders Publishing, 1992.

[Norcliffe 91] Norcliffe A. and Slater G.: *Mathematics of Software Construction*. Ellis Horwood, 1991.