A Smart Meta-CASE.
Towards an Integrated Solution

by

VINCENT ENGLEBERT

Thesis submitted in conformity with the requirements
for the degree of Doctor of Phylosophy of the
Computer Science Department
University of Namur
Rue Grandgagnage 21, 5000 Namur
Belgium

January 1, 2000

Jury members

Prof. Jean Fichefet, University of Namur (President)
Prof. Jean-Luc Hainaut, University of Namur (Supervisor)
Prof. Kalle Lyytinen, University of Jyväskylä
Prof. Eric Dubois, University of Namur
Prof. Naji Habra, University of Namur

Copyright 2000, Vincent Englebert.
The original material of this dissertation cannot be used for commercial purposes without written permission of the author.

Abstract

For twenty years now, software engineers use CASE (Computer Aided Software Engineering) tools to design systems, to verify specifications, to generate programs, ... Unfortunately, their use is not as widespread as one could expect it. They are often both too limited and too coercive. But because computer projects become larger and more complex every year, such tools are now essential to ensure the correctness, the schedules, and the documentation of programs. For these reasons, researchers have proposed complementary (or alternative) approaches with meta-CASE tools. The latter are flexible frameworks that allow engineers to customize, to extend them, or even to define brand-new CASE tools in a dynamic way.

This thesis presents the results of a study about the implementation of a new meta-CASE. This research starts from a dozen case studies in this realm to draw up an assessment of the requirements that future meta-CASE tools should meet. This work has examined a subset of them to propose an integration of meta-modelling techniques that are generally both simpler and more expressive than the former approaches.

We have investigated the requirements about three major components that make up meta-CASEs: the repository, the representation of the information stored in this repository, and the processes that animate it. Our work has been guided by two key ideas: simplicity and expressivity. Although they are generally seen as contradictory requirements, we prove that it is possible to bring them together. Indeed, *1)* we have defined a repository with a very limited number of concepts coming from the object-oriented paradigm and we have demonstrated that complex methods can be represented with; *2)* a graphical symbolic language allows method engineers to define advanced representations of these methods; and finally *3)* a new programming language has provided the repository with an operational intelligence. Moreover, this architecture has been endowed with a *mirroring* service that blurs the distinction between the the meta-CASE (i.e., the meta-model level) and the CASE tools it describes (i.e., the specification level). This service made it possible to bootstrap some other major components[1] of the meta-CASE.

[1] A meta-model editor, a GraSyLa scripts editor, meta-extensions of Voyager 2^+, ...

Acknowledgements

Even though this dissertation may appear to be an individual achievement, there are a number of people to whom I am indebted. Acknowledgements are due first to my supervisor, Professor Jean-Luc Hainaut, for the trust he invested in this study and for his talented managerial guidance of the DB-MAIN research group.

I am most grateful to Professor Kalle Lyytinen. His work has been the lighthouse that has guided my research.

I thank Professor Eric Dubois for his encouragement and for his interest in my work. I will always remember the day when his smile convinced me that I would never be able to sing, *I am singing in the rain*! Then, I decided to begin a PhD Thesis ...

I thank Professor Naji Habra for his constructive comments and criticisms which helped me to improve this thesis.

I thank Professor Jean Fichefet for having kept the situation under control (as always).

I owe special thanks to the people that form the heart of the DB-MAIN research group: Didier Roland, Jean Henrard, and Jean-Marc Hick. Neither can I forget other colleagues who have helped me to improve the Voyager 2 programming language: Abdelmajid Chougrani, Alain Gofflot, Anne-France Brogneaux, Arnaud Deflorenne, Christine Delcroix, Denis Zampunièris, Philippe Thiran, Pierre Delvaux, Stéphane Bodart, Thierry Aerts, and Virginie Detienne.

My work would have been more cumbersome without the dubious jokes of Michal Petit (aka *le Docteur P.*), the juicy gossip of Patrick Heymans, and the other members of the *Bureau Albert II*. I am especially indebted to Jean-Franois Raskin for his encouragement when my thoughts were more black than white.

Christiane Leroy has a special place in my heart for her invaluable help. She dispensed encouragement, advice and great comfort whenever I needed them — and her sauerkraut is certainly the best in the world!

I also thank all the people who encouraged me in one way or another to pursue a scientific career, in particular Professor Jean-Paul Leclercq, Professor Baudouin Le Charlier, Professor Pascal Van Hentenrijk and Professor Maurice Bruynooghe.

I also received astonishing and unexpected help from Professor Marvin Minsky who helped me obtain the original extract of his book, *The Turing Option*.

If you see errors in this page, they should be limited to this sentence. Thanks a lot, Pat.

I gratefully recognize all of my many colleagues and friends: Anne-Marie Breny, Babette Di Guardia, Gyselle Henrard, Jean-Marie Jacquet, Laura Oger, Luc Goffinet, Philippe du Bois, Pierre-Yves Schobbens, Radu Cotet, Richard Mairesse and many others that I have certainly forgotten.

I gratefully acknowledge the patience and support of my parents. This work would have been impossible without their sacrifices.

Last but not least, Sophie, who has been the most enduring support I met during this gruelling period. I promise to reimburse her with all my love for this period of neglecting my family obligations.

Contents

I Overview 1

1 Why Do we Need Meta-CASEs? 3
1.1 CASE tools . . . 4
1.2 Meta-CASE tool . . . 5

2 Meta-CASE studies 7
2.1 Survey . . . 7
2.1.1 ConceptBase . . . 7
2.1.2 GraphTalk . . . 8
2.1.3 KOGGE . . . 8
2.1.4 MetaEdit+ . . . 11
2.1.5 MetaView . . . 13
2.1.6 ToolBuilder . . . 13
2.1.7 CDIF . . . 15
2.1.8 Universal Repository . . . 15
2.1.9 And Many Others . . . 16
2.2 Meta-CASE Comparison . . . 16
2.2.1 Meta-meta-model Semantics . . . 16
2.2.2 CASE Generation . . . 18
2.2.3 Specifications Representation . . . 18
2.2.4 User Interface . . . 19
2.2.5 Programming Language . . . 19
2.2.6 Methods Integration . . . 20
2.2.7 General Criticisms . . . 20
2.3 Conclusion . . . 20

II A New Meta-CASE Definition 23

3 What should a Meta-CASE be? 25
3.1 The Information Dimension . . . 25
3.2 The Representation Dimension . . . 27
3.3 The Intelligence Dimension . . . 27
3.4 The User Interface Dimension . . . 28
3.5 The Communication Dimension . . . 28
3.6 The Requirements . . . 29

4 Repository Axioms **33**
4.1 Introduction . . . 33
4.2 Overview . . . 33
4.3 Concepts . . . 37
4.3.1 Meta-Class . . . 37
4.3.2 Meta-Model . . . 38
4.3.3 Aggregation's Characteristics . . . 43
4.3.4 Meta-Class Inheritance . . . 43
4.3.5 Meta-Property . . . 45
4.3.6 Meta-Classes and Methods . . . 47
4.3.7 Meta-Relation . . . 48
4.3.8 The Ω's Axioms . . . 49
4.3.9 Class . . . 50
4.3.10 Specification . . . 51
4.3.11 Class Inheritance . . . 54
4.3.12 Property . . . 58
4.3.13 Classes and Methods . . . 60
4.3.14 Relation . . . 61
4.3.15 Identifiers . . . 63
4.4 Some Operations . . . 67
4.4.1 Deletion . . . 67
4.4.2 Other Kinds of Deletion . . . 72
4.4.3 Cut&Paste . . . 72
4.4.4 Remember&Paste . . . 73
4.4.5 Specialize . . . 75
4.5 The User Guide . . . 75
4.5.1 Specialization . . . 75
4.5.2 Inheritance with Exceptions . . . 77
4.5.3 Explosion . . . 77
4.5.4 Multiple Explosions . . . 78
4.5.5 Equivalences in Explosions . . . 79
4.5.6 Meta-Models about Meta-Models . . . 80
4.5.7 Meta-Relations and Meta-Models . . . 80
4.6 The Implementation of the Repository . . . 82
4.6.1 Presentation . . . 82
4.6.2 Details . . . 84
4.7 Case Studies . . . 88
4.7.1 a Statechart Meta-Model . . . 88
4.7.2 The Overview Meta-Model . . . 91
4.8 Summary . . . 91

5 The Mirroring Service **95**
5.1 Presentation . . . 95
5.2 The Mirroring Service . . . 97
5.3 Mirroring and Reflection . . . 104
5.4 Reflection: the First Benefits . . . 105
5.4.1 Fundamental Axioms and Reflection . . . 105

5.4.2 Mirroring and Reflection . . . 107
5.4.3 Reflection and Bootstrapping . . . 108
5.5 Prospect: Mirroring and Distributed CASEs . . . 108
5.6 Summary . . . 109

6 GraSyLa: a Graphical Symbolic Language 111
6.1 Overview . . . 111
6.2 The Challenges . . . 112
6.3 The GraSyLa Approach . . . 113
6.4 GraSyLa: its Syntax and its Semantics . . . 114
6.4.1 Lexical Conventions . . . 115
6.4.2 The GraSyLa Syntax . . . 115
6.4.3 The Directive Section . . . 115
6.4.4 The Main Section . . . 117
6.4.5 The Parameters of the Main Section . . . 132
6.4.6 The Connection Section . . . 132
6.4.7 The Display Processor . . . 134
6.5 GraSyLa and Inherited Definitions . . . 134
6.6 GraSyLa and Dedicated Display Processors . . . 136
6.7 Examples . . . 138
6.7.1 A Statechart Meta-Model . . . 138
6.7.2 Synchronized Textual and Graphical Views . . . 138
6.7.3 An Editor for GraSyLa Scripts . . . 139
6.7.4 The "Firework" Sample . . . 141
6.8 Summary . . . 142

7 Voyager 2$^+$ 153
7.1 Preliminaries . . . 153
7.2 Lexical Elements . . . 155
7.2.1 Comments . . . 155
7.2.2 Operators . . . 155
7.2.3 Identifiers . . . 155
7.2.4 Reserved Words . . . 156
7.2.5 Constants . . . 156
7.3 The Program's Skeleton . . . 157
7.4 Types . . . 158
7.4.1 Integers . . . 159
7.4.2 Characters . . . 159
7.4.3 Strings . . . 159
7.4.4 Lists . . . 160
7.4.5 Cursors . . . 161
7.4.6 Files . . . 162
7.4.7 Meta-Classes & Meta-Models . . . 162
7.4.8 Meta-Relation . . . 162
7.4.9 Meta-Property . . . 163
7.5 Block Clauses . . . 163
7.6 Expressions . . . 164

7.6.1 Precedence and associativity of operators . . . 164
7.6.2 Arithmetic Expressions . . . 166
7.6.3 The “dot” Notation . . . 166
7.6.4 `MetaClass` . . . 167
7.6.5 List Expressions . . . 167
7.7 Statements . . . 168
7.7.1 Assignment . . . 168
7.7.2 Selection Statement . . . 169
7.7.3 Iteration Statement . . . 171
7.8 Functions, Procedures and Methods . . . 175
7.9 Repository Access Methods . . . 180
7.9.1 Predicative Queries . . . 180
7.9.2 Class Creation . . . 187
7.9.3 Class Deletion . . . 190
7.9.4 Predefined Methods . . . 191
7.10 Modular Programming . . . 192
7.10.1 Presentation . . . 193
7.10.2 Voyager 2^+ Process . . . 194
7.10.3 Libraries . . . 198
7.10.4 Formal Definitions . . . 199
7.10.5 Literate Programming . . . 201
7.10.6 The Include Directive . . . 201
7.10.7 Processes and Packages . . . 203
7.11 Examples . . . 203
7.11.1 The Statechart Meta-Model . . . 203
7.11.2 An XML Generator . . . 206
7.12 Summary . . . 208

8 The Dialogues **209**
8.1 Representative Dialogues . . . 210
8.1.1 To Create a new Transition . . . 210
8.1.2 To Add a new Action . . . 210
8.1.3 The Meta-Properties in Dialogues . . . 213
8.2 Some Built-In Dialogue Boxes . . . 213

9 The Voyager Abstract Machine **219**

10 Conclusions **225**

III Appendices **231**

A The BNF Conventions **233**

B The DB-MAIN Conventions **235**

C The Voyager 2^+ Syntax **237**

D Listings **241**
D.1 The Voyager 2^+'s Constants . . . 241
D.1.1 The Builtin Types . . . 241
D.1.2 The Integer Constants . . . 243
D.2 The Voyager 2^+Predefined Operations . . . 243
D.2.1 Operations on Characters . . . 243
D.2.2 Operations on Strings . . . 244
D.2.3 Operations on Lists and Cursors . . . 247
D.2.4 Operations on Files . . . 248
D.2.5 Interface Operations . . . 254
D.2.6 Time Operations . . . 255
D.2.7 Flag Operations . . . 257
D.2.8 General Operations . . . 258
D.3 GraSyLa Scripts . . . 260
D.3.1 The GraSyLa Editor's Listing . . . 260
D.3.2 The Statechart Editor . . . 263
D.3.3 The Relational Editor . . . 264
D.4 Voyager 2^+ Programs . . . 266
D.4.1 The XML-Generator Program . . . 266
D.4.2 The "meta-class" Package . . . 269
D.4.3 Example . . . 270

List of Figures

1.1 CASE Tools Classification 5
1.2 Use of a Meta-CASE 6

2.1 The GraphTalk's Meta-Meta-Model 9
2.2 The KOGGE's Meta-Meta-Model (EER) 11
2.3 The MetaEdit+'s Meta-Meta-Model (GOPPR) 12
2.4 The EARA/GE Mapping 14
2.5 The CDIF's Meta-Meta-Model 15
2.6 The UREP's Meta-Meta-Model 16

3.1 Comparison of the Architectures Customizations 26
3.2 Comparison of the Architectures Expressivity 26
3.3 The Gap Between the Expressivity of the CASE and Meta-CASE tools 26
3.4 General Architecture 31

4.1 A Three-Layers Architecture of a Meta-CASE Tool 34
4.2 Software Engineering Projects and Hypergraphs 36
4.3 Meta-Modelling a Software Engineering Project 36
4.4 Inheritance with Exceptions 37
4.5 Graphical Representation of Meta-Classes and Meta-Models 38
4.6 Meta-Model Drawing (Compact View 1) 41
4.7 Meta-Model Drawing (Compact View 2) 42
4.8 Meta-Model Drawing (Extended View) 42
4.9 Inheritance and Meta-Model Definition 45
4.10 Drawings of Classes and Specifications 50
4.11 Classes and Specifications in Compact Views 51
4.12 Weak and Strong Meta-Models 53
4.13 Incorrect Inheritance 55
4.14 Set Inheritance 55
4.15 Set Inheritance (*bis*) 56
4.16 Specifications and Inherited Definitions 57
4.17 Inheritance: Several Leaves 58
4.18 Mandatory Role and $\Re^\star$ 63
4.19 Infinite identifier value 66
4.20 Cut&Paste Scenario 73
4.21 Remember&Paste Scenario 74
4.22 Inheritance with exceptions 77

4.23 Equivalence in statechart diagrams 81
4.24 How to Limit Meta-Relations to Meta-Models 82
4.25 Repository (logical level) 83
4.26 Statechart Meta-Model 88
4.27 Revised Statechart Meta-Model 89
4.28 The Normal Statechart Meta-Model 92
4.29 The Overview Meta-Model 93

5.1 CASE and Meta-CASE Architecture 95
5.2 The Mirroring and Dynamic Compartments 96
5.3 The "Guinea-Pig" Example 97
5.4 The Mirrored Sub-Model 104
5.5 The Reflective Meta-Model 106
5.6 Distributed Architecture & Integration 109
5.7 Distributed Architecture & Reflection 110

6.1 The GraSyLa Mechanism 114
6.2 The Normal Statechart Meta-Model 116
6.3 Restricted Scope 121
6.4 Scope of a Display Processor 121
6.5 The `boxH` Constructor 123
6.6 The `boxV` Constructor 124
6.7 The `circleH` Constructor 125
6.8 The `roundH` Constructor 126
6.9 The `bitmap` Constructor 130
6.10 The GraSyLa Architecture 135
6.11 The Display Processor 135
6.12 GraSyLa and Inheritance 137
6.13 The Component Architecture of GraSyLa 137
6.14 The Original "Phone" State Diagram 143
6.15 The GraSyLa Version of the "Phone" State Diagram 144
6.16 The GraSyLa Version of the "Phone" State Diagram (with explosion) 145
6.17 Textual and Graphical Synchronized Views 146
6.18 The `Script-Grasyla` Meta-Model 146
6.19 The `G-expression` Meta-Model 147
6.20 The `G-script-rel` Meta-Model 148
6.21 The `G-script-class` Meta-Model 149
6.22 The GraSyLa Script of the GraSyLa Editor 150
6.23 The Firework Sample 151

7.1 Some Patterns of Inheritance Graph 169
7.2 The Normal Statechart Meta-Model 182
7.3 Image and Stack in Voyager 2^+ 193
7.4 Literate Programming 202

8.1 Create a new Transition 211
8.2 The Transition's Dialogue Box 211
8.3 Edit the Origin of a Transition 211

8.4 Edit the Target of a Transition 212
8.5 Creation of a Transition 212
8.6 Add a new Action 212
8.7 Edit a new Action 212
8.8 Choose an Event 212
8.9 The "Edit Actions" Dialogue Box is Completed 212
8.10 The "Actions" Dialogue Box is Completed 214
8.11 The Result 214
8.12 A Dialogue Box with all the Kinds of Meta-Properties (Mono-Valued) 214
8.13 A Dialogue Box with all the Kinds of Meta-Properties (Multi-Valued) 214
8.14 Create a new Meta-Class 215
8.15 Create a new Meta-Property 215
8.16 Edit the Meta-Relations 215
8.17 Create a new Meta-Relation 216
8.18 Edit the Definition of a Meta-Model 216
8.19 Inspect a Specification 217

9.1 The Distinct Memory Areas 221
9.2 The Voyager 2^+ Process Statechart 222
9.3 Programs & Processes 223

D.1 A File Browsing Window 256
D.2 A Choice Dialog 256
D.3 A DialogBox Window 256
D.4 A Message Box 256
D.5 A Pseudo Relational Meta-Model 265

List of Tables

2.1 Meta-CASEs Conspectus 17

7.1 Operators and Separators 155
7.2 Reserved Keywords 156
7.3 Miscellaneous Constants 157
7.4 Conventions about Special Characters 160
7.5 Meta-Characters Used in String Constants 161
7.6 Operators: Precedence and Associativity rules 165

Notations

Mathematical Conventions

Logic

$P \Rightarrow Q$ **:** Proposition P implies Q.

$P \Leftrightarrow Q$ **:** Propositions P and Q are equivalent.

$\forall x \in S : \ P$ **:** The universal quantifier.

$\exists x \in S : \ P$ **:** The existential quantifier.

$\nexists x \in S : \ P$ **:** The negated existential quantifier, equivalent to $\forall x \in S : \ \neg P$.

$\exists ! u \in S : P$ **:** There exists one and only one value (denoted u) that satisfies the proposition P.

$F_1 \equiv F_2$ **:** Equivalence operator. The formula F_2 can be used in place of the formula F_1.

$A \wedge B$ **:** The **and** logical operator.

$A \vee B$ **:** The **or** logical operator.

$\neg A$ **:** the negation logical operator.

$A \otimes B$ **:** The **xor** logical operator ($A \otimes B \Leftrightarrow (A \wedge \neg B) \vee (\neg A \wedge B)$).

deduction :

$$\frac{P_1 \ \dots \ P_m}{Q_1 \ \dots \ Q_n}$$

means $\forall i, \ 1 \leq i \leq n : \ P_1 \wedge \cdots \wedge P_n \Rightarrow Q_i$. The free variables in P_j propositions are quantified with $\exists$. This notation is sometimes privilegied when propositions are too long.

Sets & Sequences

$A \cup B$ **:** The **union** operator for sets/sequences.

$A \dot{\cup} B$ **:** The **disjunctive union** operator for sets/sequences. Use this operator to emphasize that A has no common element with B ($A \cap B = \emptyset$).

$A \cap B$ **:** The **intersection** operator for sets/sequences.

$A \setminus B$ **:** The **difference** operator for sets/sequences.

$A \times B$ **:** A and B are sets or sequences. The product of A by B is $\{(x, y) \dagger x \in A \wedge y \in B\}$.

$n_A : A \times n_B : B$ **:** This denotes the $A \times B$ product where each component is named. If $x \in n_A : A \times n_B : B$ then one note $x.n_A$ (resp. $x.n_B$) the projection of x on its first (resp. second) component. The names n_A and n_B can also be interpreted as the projection mapping and therefore this notation is compatible with the dot notation of mappings (see further).

$\mathbb{Z}$ **:** The set of integers $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$.

$\mathbb{N}$ **:** The set of positive integers $\{0, 1, 2, \ldots\}$.

$\mathbb{R}$ **:** The set of real numbers.

$\{p \ldots q\}$ **:** If $p, q \in \mathbb{Z}$ then this denotes the set $\{i \in \mathbb{Z} \dagger p \leq i \wedge i \leq q\}$.

$[x_1, \ldots, x_n]$ **:** Ordered sequence of elements. If $n = 0$ the sequence is empty ($[\,]$).

sequence : One will confuse *sequences* and *sets* when the context makes it possible.

$S_1 + S_2$ **:** When S_1 and S_2 denote sequences, the $+$ operator denotes the concatenation of both sequences.

$\sharp S$ **:** If S is a set or a sequence, then $\sharp S$ denotes the number of elements in S. If S is sequence, repeated elements are counted several times ($\sharp[1, 2, 1] = 3$).

$S[i]$ **:** i^{th} element of the sequence s ($i \in \mathbb{N}$ and $1 \leq i \leq \sharp S$). If the sequence is empty, this notation has no sense.

$\mathsf{tail}(S)$ **:** If s is a sequence, then this denotes the last element of the sequence i.e., $S[\sharp S]$. If S is a set, then this denotes an arbitrary element of this set. If S is empty, this notation has no sense.

$E^{\star}$ **:** is the power set of E.

$$X \in E^{\star} \Leftrightarrow (X \text{ is a set } \wedge X \subseteq E)$$

$E^{[\star]}$ **:** is the power set of E where elements of the sets are ordered.

$$X \in E^{[\star]} \;\Leftrightarrow\; \Big(X \text{ is a sequence with no duplicates} \wedge \big(\forall i \in \{1 \ldots \sharp X\) : X[i] \in E\big)\Big)$$

$E^{\langle\star\rangle}$ **:** denotes the set of sequences that are subsets of E. Elements can be repeated several times.

$$X \in E^{\langle\star\rangle} \;\Leftrightarrow\; X \text{ is a sequence } \;\wedge\; \big(\forall i \in \{1 \ldots \sharp X) : \; X[i] \in E\big)$$

$\{x \in S \dagger \pi\}$ **:** denotes the largest subset of S such that every element of this subset verifies the boolean expression π. For instance, $\{x \in \mathbb{R} \dagger x/2 \in \mathbb{N}\}$ denotes the set of all the positive even numbers.

Mathematical Functions

$E \to F$ **:** denotes a function f from E to F.

$E \mapsto F$ **:** denotes a mapping f from E to F ($\text{dom}(f) = E$).

$x.f$ **:** When f is a mapping, this notation is sometimes preferred to denote $f(x)$.

$\text{dom}(f)$ **:** denotes the domain of a function $f : A \to B$, that is the greatest subset of A such that f is defined for each element.

$\text{codom}(f)$ **:** denotes the codomain[1] of a function $f : A \to B$, that is the greatest subset of A such that for each element y, there exists at least one $x \in A$ such that $y = f(x)$.

$f\langle\, a \mapsto b \,\rangle$ **:** is a function f' with the same signature as f and such that $\forall x \in \text{dom} f \setminus \{a\} : f'(x) = f(x)$ and $f'(a) = b$[2].

$f\langle\, a \mapsto \textsf{undef} \,\rangle$ **:** is a function f' with the same signature as f and such that $\forall x \in \text{dom}(f) \setminus \{a\} : f'(x) = f(x)$ and $f'(a) \nexists$.

f^{-1} **:** If f denotes a function $A \to B$, then f^{-1} is a function $B \to A^{\star}$ defined as

$$\forall b \in B : \; f^{-1}(b) = S \Leftrightarrow S = \{x \in A \dagger f(x) = b\}$$

$1/1$ **:** A relation $r \subseteq A \times B$ is 1/1 when it satisfies this property:

$$\forall x \in A, \forall y, z \in B : \; r(x, y) \wedge r(x, z) \Rightarrow y = z$$
$$\forall x, y \in A, \forall z \in B : \; r(x, z) \wedge r(y, z) \Rightarrow x = y$$

$f(x)\exists$ **:** If f denotes a partial function (e.g., $f : A \to B$), then we note $f(x)\exists$ if $x \in \text{dom}(f)$ and $f(x) \nexists$ otherwise.

Remark We will sometime define distinct functions or mappings with the same name. In such cases, these functons will be used in contexts allowing the reader to solve the ambiguity.

Miscellaneous Notations

Glossary

Words emphasized like this "example*" are defined in the glossary page 277.

Functions and Procedures

Operational functions[3] and procedures will be presented with respect to the following patterns.

[1] also called *range*
[2] This notation has been presented in [Ten81]
[3] Functions defined with a programming language instead of mathematical formulas.

`function` type : result `fct-name` ($\text{type}_1 : \text{arg}_1, \ldots, \text{type}_n : \text{arg}_n$)

Precondition. The condition the arguments and the context have to fulfill.

Postcondition. The condition that the completion of the function will ensure.
on error: The behaviour of the function if some error occurs during its computation (a precondition fails[1], the memory overflow, ...)

A procedure will be defined like this:

`procedure` `proc-name` ($\text{type}_1 : \text{arg}_1, \ldots, \text{type}_n : \text{arg}_n$)

Precondition. see above.

Postcondition. see above.
on error: see above.

These conventions do not follow the syntax of the Voyager 2^+ language and are mainly intuitive. The result of a function can be named and used in the postcondition. P_{old} denotes a proposition evaluated in the context of the precondition, that is before the call of the function/procedure and P_{new} denotes a proposition that will be evaluated after the completion of the function. For instance, the proposition $x_{\mathsf{new}} = x_{\mathsf{old}}$ means that the x variable is let unchanged by the function/procedure.

The DB-MAIN's Graphical Conventions

Extended Entity-Relationship schemas of this thesis will follow the conventions of the DB-MAIN CASE tool. They are summarized in appendix B page 235.

[1]Preconditions are not necessarily ensured by the system. Indeed, programmers can pass parameters that do not respect the preconditions.

Preface

When I began my studies in mathematical sciences (1986), a PC computer had a main memory of 512Kb, and privileged users enjoyed hard disks of 5Mb. The screen had a resolution of 320×200, and still again, only privileged users had colour screens. When I began my studies in computer sciences (1989), Turbo Pascal was supplied on a full 360Kb floppy disk and its reference manual had ±300 pages. Now, Visual C++ 6.0 requires some 200Mb and its documentation weights more than 3Kg.

Besides the anecdote, this short story should convince anybody that the tools of 1989 are no longer valid today. If the best companion of the software engineer* were a smart editor with a macro language and a good compiler, he now needs much more advanced techniques to overcome a new complexity: systems become larger and are intrinsically more complicated. *Programming in the small* is now confined to academical lessons and gives way to the *programming in the large*. Large systems cause interesting problems: they are developed by big teams (several tens), they involve heterogenous frameworks[1], and many modelling techniques[2] must be used to comprehend them.

A novice student could argue that architects and engineers conceive planes and bridges with the help of CAD/CAM modelling tools, and nevertheless, we are not aware of some engineering crisis in this realm. Hence, according to this student, if computer scientists defined precise modelling tools and if software engineers had corresponding tools at their disposal, then they could implement correctly software as large as a bridge or a plane without serious hitches.

Unfortunately, a program is not as concrete as a bridge or a plane. First and foremost, programs are ideas, or even dreams. Programs are dreams in the head of a CEO who expects to gain money with them or they are dreams in the head of PhD students who expect to have the Turing award. But bridges and planes are also dreams! Nevertheless, albeit engineers imagine their buildings with walls and roofs, software engineers have to imagine their systems with abstract concepts, i.e., nothing else than more or less organized ideas. And so far, there is no consensus on the hypothetical shape, smell, or colour of an idea. The changeability of the ideas makes their modelling little stable so that they cannot be easily reached by a general consensus. For the record, Newton's theory is now three centuries old and the Einstein's thesis is still not refuted. In computer science, engineers are less lucky: modelling techniques (i.e., methods*) exhibit a very high turnover[3]. Like architects and engineers, software engineers have also CASE* tools at their disposal which help them in their everyday tasks. But the

[1] For instance, COBOL & Java, SQL & OODBMS, Corba, NT / UNIX, ...

[2] For instance, dataflows, statecharts, entity relationship diagrams, sequence diagrams, organizational units, networks architectures, etc.

[3] Coad-Yourdon, MERISE, Booch, OMT, UML, SADT, Fusion, OML, SOMA, BON, MOSES, OOram, OPEN, Shlaer/Mellor, Martin/Odell, ...

lack of consensus on the engineering methods and the necessary incomplete nature of their dedicated tools make the CASE tools short-lived and quite expensive. Nevertheless, such tools are indispensable to manage the above-mentioned complexity.

Software engineers are not automata and love their freedom [JH98]. This observation is an important factor to explain the low adoption of CASE tools. Moreover, the introduction of CASE tools in organizations involves a considerable amount of investment. Scientists have thus to face an interesting challenge: making CASE tools both profitable and useful. On one side, CASE tools must support standard modelling techniques (the lingua franca of the method engineers*), and on the other side, they must match the expectations of every methodologist who has often needs that are out of the scope of the method. To answer this lack of flexibility, computer scientists have developed meta-CASEs*.

Meta-CASEs are high-level tools that generate (or emulate) CASE tools according to some specification. The added-value of those meta-CASEs (compared with hand coded CASE tools) is that the generation can be replayed as many times it is necessary to obtain the "perfect" CASE tool. Moreover, the specification language is generally simple enough to make it possible to define a brand-new product in a few weeks (and sometimes in a few days).

Despite the promising features of this technology, meta-CASEs are still a niche in the software engineering community [FPD+99]. Nevertheless, if software engineers continue to prefer methods dedicated CASE tools, it has to be admitted that more and more CASE tools are now implemented on top of a meta-CASE architecture. Hence, meta-CASEs have still to win over the final users: the software engineers.

This thesis will present the foundations of a new meta-CASE. This approach will be neither perfect nor exhaustive. Some problems will be left deliberately open and should be the subject of further investigations. It will not solve thorny problems of other meta-CASEs either. This research will rather bring a new reflection on the integration of several requirements to make meta-CASEs more suitable. For instance, its repository* will be drastically simpler (at least at the meta-model* level) with a similar expressiveness. The graphical representation of the specifications* stored in the repository will gain an unequalled power and combine remarkable qualities such as genericity, simplicity, and repository independence. The framework will be endowed with an integrated programming language that allow method engineers to access both the meta-model and the specification levels in the same way. Moreover, it has a syntax close to conventional languages. The integration of these components will be as seamless as possible.

The thesis is made up of three parts. Part I presents both an introduction and a summary of the state of the art. Part II includes chapters which describe every facet of our work. Chapter 3 lists the main guidelines that should be observed in the definition of new meta-CASEs. The next chapters propose answers to the requirements presented before. Chapter 4 defines the foundation of the new architecture: its *repository*. The *mirroring service* (*cfr.* chapter 5) defines both a mechanism and a principle that makes feasible to manage the meta-model level as a common specification, i.e., a first-class citizen. Chapter 6 presents a graphical symbolic language (GraSyLa). The programming language Voyager 2^+ is defined in chapter 7. The chapter 8 comments the dialogue between the software engineer and the tool. Finally, chapter 9 explains the architecture of the abstract machine (Voyager 1^+) which executes the Voyager 2^+ programs. The last part is just composed of appendices.

Part I

Overview

– What is the difference between a methodologist and a terrorist?
– You can negotiate with a terrorist.
MARTIN FOWLER. 1997.

Chapter 1

Why Do we Need Meta-CASEs?

While in their early stage (1980s), CASE tools were mainly computer-aided documentation and diagramming tools, they now tend to be a cyber guide dog for both analysts and the programmers. CASE tools are present from the requirement and elicitation stages until the implementation phases. They help software engineers to check the consistency of their analysis, to generate prototypes, to maintain the documentation and the programs, etc. Nevertheless, this description is somewhat optimistic and must be moderated. Many researchers and practitioners draw serious remarks. Marttiin claims that "*there is no clear evidence (either theoretical or empirical) that CASE leads to better productivity*" [Mar95b]; Jarzabek *et al.* explain that "*CASE tools need to become more useful if they are to be applied to the practice of software production*" and that "*CASE are dearly bought but sparsely used*" [JH98]. These observations are reinforced by Lending and Chevarny who write "*Few organizations use CASE tools [EG93, NTH89]; organizations abandon the use of the tools [EG93, SW95, Sum92]; and organizations that do use CASE tools contain many systems developers who do not actually use the tool [Mar95a]* ".

Nevertheless, CASE tools can really be helpful in cumbersome tasks such as the verification of the consistency, code generation from database schemas, transformation, reverse engineering tasks. Such tasks are *tedious*, *time-consuming*, and even *impractical*, if their use is not supported by automated tools [tHV96]. Our experience with the DB-MAIN CASE tool highlights that the benefits of a CASE tool to support a methodology is dependent on the capability to customize and to extend the tool. The 4th International Workshop on CASE, in December 1990, presented a set of consensus assertions on criticisms and expectations about the CASE technology [NSC⁺91]. The topic called "Use and Abuse of CASE" highlighted several remarkable points: WHAT IS NEEDED[1]

- CASE accommodation to user organization's evolution and maturity level.
- Ability to accommodate existing environments, data, tools.
- Customization without loss of functionality.
- Customizable user interfaces.

[1]This list is not exhaustive.

In the same time, Hardy *et al.* noticed (a survey made in UK, 1994) that 24% of the firms used in-house methods [HSTE95].

Customization is thus an essential key word to increase the use and the adequacy of CASE tools with the practice. Some research groups and software manufacturers have already looked closely at this new direction. So far, we observe two trends in the proposed architectures and we can classify them into two categories:

Customizable CASE tool: The first type offers a predefined set of supported methods and makes it possible to customize some aspects (functionalities, graphical representation, report generation, repository, ...). For instance, Toolframe [DK95], Paradigm+ [Pro94], DB-MAIN [Gro].

Meta-CASE tool: The second type stresses the customization functionality and pushes the support of predefined methods into the background.

Our experience with DB-MAIN tends to prove that both kinds of tools are useful. DB-MAIN proposes several kinds of customizations: ***a)*** the extension of its repository with new meta-properties; ***b)*** the definition of new functionalities with an integrated programming language (Voyager 2 [Eng99]); ***c)*** the customization of built-in functionalities[1] with overridden methods written in Voyager 2; and ***d)*** the definition of new process models [RH97].

Nevertheless, such approaches still lack important customization capabilities:

1. the set of graphical representations is often frozen;

2. the repository is too rigid;

3. their architecture is not flexible.

The main criticism is without contest their inability to extend their scope to exogenous models. And nevertheless, such models become more and more present and influence considerably the analysis stages: organizational units, workflows, distributed data, network architectures, user interfaces, WEB technologies, mobile computing, GIS, temporal data, and so on. Software engineers often have to manage themselves these artefacts although they sometimes imagine quite well the way to represent them and to model them. Moreover, since the recent Y2K problem, firms are now aware to control their information systems (where is stored this database, who is its manager, who developed this program, ...). They often try to represent them with repositories but such information is often very peculiar and needs thus ad-hoc tools.

Those signs indicate clearly that meta-CASE tools can have a crucial place in software development systems. But, defining a meta-CASE leads us to ask what is a CASE tool first!

1.1 CASE tools

One generally classifies a CASE tool according to two criteria: its abstraction level and its temporal scope. Upper-CASE tools are used in early stages of information system development for analysis, design and strategy planning purpose [McL89]. Lower-CASE tools are used during the implementation stage and consists of mainly programming tools (editors, compilers, version managers, ...). The second criterion analyzes the dependency between

[1] Analysis, transformations, generators, ...

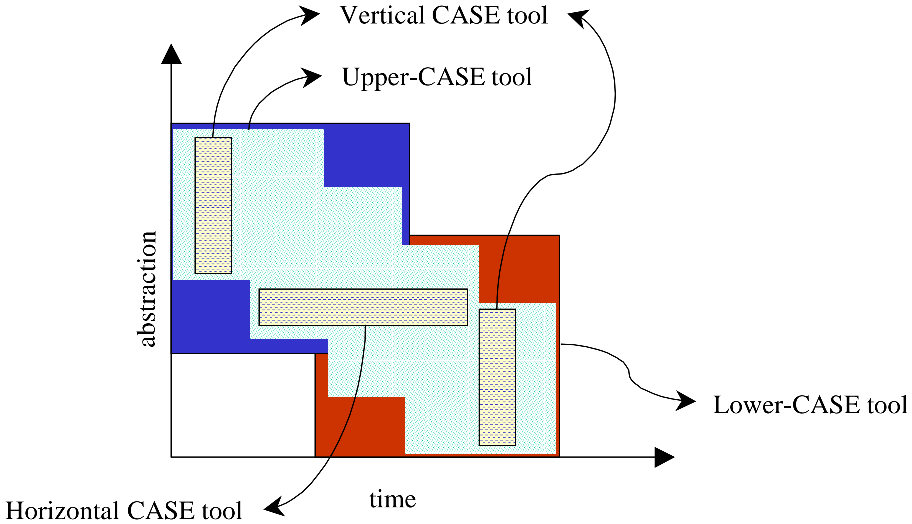


Figure 1.1: [**CASE Tools Classification**]

the CASE tool and its application scope during the methodological process. Vertical tools are limited to precise stages and horizontal tools are capable of supporting some aspect during the whole methodological process. Horizontal tools generally strech for several abstraction levels and are thus impossible to classify. Such tools are documentation tools, traceability tools, process management tools, defect tracking tools, etc. Figure 1.1 depicts the two criteria on a same diagram.

Beyond those two classic criteria, we could also classify CASE tools according to the ease of customizing them to the software engineer's expectations. This customization can itself be refined according to the services that CASE tools can propose. If we consider them as black boxes and if we observe them, we would identify a five-dimensions object described as follows:

1. CASE tools can store and manage **information**.
2. CASE tools can **display** information (graphs, browsers, tables, …).
3. CASE tools can **execute** tasks (analysis, generation, transformation, …).
4. CASE tools can **interact** with people (user interface, events, …).
5. CASE tools can **communicate** together (teamware, workgroup, concurrent engineering, …).

If we use the "Turing test" to evaluate how a meta-CASE succeeds in simulating hand coded CASE tools, then these five dimensions should be taken into consideration.

1.2 Meta-CASE tool

The aim of a meta-CASE is to build quickly and easily a CASE tool from some abstract definition — as depicted by the scenario in Fig. 1.2. Two architectures are possible: *a)* the

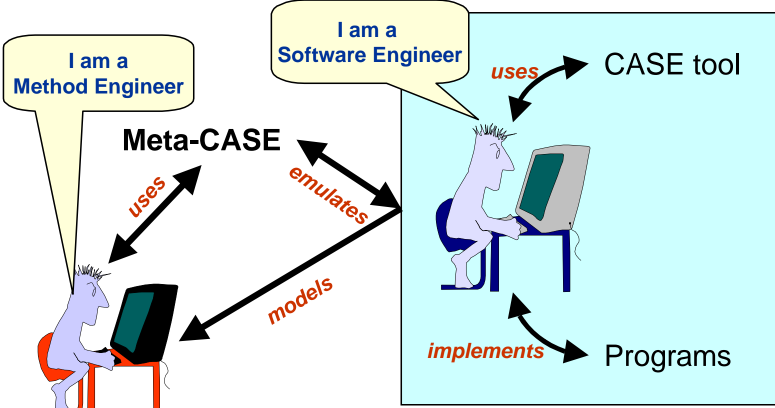


Figure 1.2: **[Use of a Meta-CASE]** This scenario illustrates the use of a meta-CASE by a method engineer who analyzes the working environnement (i.e., his tools, his representation models, his methods, his way of working, ...). Then, he models this framework with the help of a meta-CASE tool that will produce it. The method engineer will next be free to change the parameters of this "*factory*" to customize it to new requirements.

meta-CASE acts as a compiler and generates programs which are ready to compile (Phedias [WL95], ToolBuilder [ACE99], Kogge [ESU97]) and *b)* the meta-CASE interprets the specifications to emulate the CASE tool (MetaEdit+ [KLR96], MetaView [STM88], etc).

Lower-CASE tools are very often considered outside the meta-CASEs scope. They are generally dependent on some programming language and involve specific components (compilers, linkers, version managers, editors, ...). Modelling such heterogeneous tools is too difficult (or even impossible).

This restricts the use of the meta-CASEs to the elicitation, analysis and design phases. The information at these stages is mainly graphical and this aspect will thus play a crucial role. On the other hand, the information collected and managed in the first phases can be both ambiguous and complex. Moreover, this information influences all the further steps. This will lead the meta-CASEs to deal with very complex information systems. The distance between the elicitation phase and the design (or even sometimes the implementation) stage is generally fulfilled with the execution of tasks such as analysis, verification, transformation, and generation processes. Of course, other cycles of development can be considered for, e.g., reverse engineering and re-engineering processes.

> **Remark** We deliberately omitted the cooperative work in the sketch. This dimension is still widely unsupported by current CASE tools.

Hence, the execution of a meta-CASE could be described as a set of **processes** that manage **complex information** which has a **graphical representation**. To conclude this brief presentation, we cite Rossi [RK99], who identifies 9 requirements about Meta-CASEs: *1)* The simplicity of the system architecture and associated meta-model and the ease of use. *2)* The expressive power of the meta-model should be maximized and kept simple at the same time. The language definition should be efficient. *3)* The identification of fine grained units or models. *4)* The ability to model complex objects. *5)* The possibility to use partially implemented meta-models and models. *6)* The support for data continuity. *7)* The dynamism of the type and instance levels. *8)* Several levels of modifiability. *9)* Multi-user support.

“*The human mind treats a new idea in the same way the body treats a strange protein; it rejects it.*”
Peter Medawar

Chapter 2

Meta-CASE studies

This chapter will study several frameworks that are either meta-CASE tools or related works. They all constitute the state of the work in this realm and they place, all together, the milestones of an hypothetical leading meta-CASE. A section will be devoted to each tool and the last section will present a comparison of their features.

2.1 Survey

2.1.1 ConceptBase

References [JJNS98, Tea98, JJQ98, JGJ$^+$95, EJ91, Myl92]

ConceptBase has been developed at the RWTH Aachen (Germany) and is available since 1988. Although ConceptBase is not really a meta-CASE, the requirements it answers to, overlap those of meta-CASEs. ConceptBase is a deductive meta data management system. The product supports the O-Telos language — a Datalog-like language with object facilities. As for the other deductive database systems, ConceptBase has an *extensional* component and an *intentional* database. The knowledge base of the system is stored with a unique predicate $P/4$ whose semantics depends on the signature. The first argument always denotes an automatic index that identifies every fact of the predicate P. The interpretation of the other arguments differs:

$P(o, o, l, o)$ **:** This denotes an individual named l.

$P(o, x, \texttt{in}, c)$ **:** This fact denotes that object x is an instance of the c class.

$P(o, c_1, \texttt{isa}, c_2)$ **:** This fact denotes that class c_1 is a subtype of class c_2.

$P(o, x, l, y)$ **:** This means that object x has an attribute named l and its valued is y.

The database can be distributed[1] and be shared by several users. The system is noteworthy for several reasons:

1. The logic model proposed by O-Telos is very simple and can be considered as the essence of more complex models.
2. The system supports potentially infinite hierarchy of classifications (object, class, meta-class, meta-meta-class, meta-meta-meta-class, ...).

[1] Client/Server architecture.

3. Queries are themselves first-class objects.

4. The O-Telos language is simple and easy to learn.

Unfortunately, although the product shows some similarities with a meta-CASE architecture (a meta-repository, a graph browser, ...), it proposes a too raw approach of the problem. Indeed, it suggests no way to structure the knowledge, no aggregation, and no integrated programming language for instance.

2.1.2 GraphTalk

References [Ranb, Rand, Ranc, Rana, Par, Par94, Ran92, Ran91, BCPC+97]

The GraphTalk meta-CASE was produced by a research team of the Rank Xerox company. It was then sold by Parallax and is now the property of the Computer Sciences Corporation company. Our description of GraphTalk is based on the versions 2.3 and 2.4 that are now 6 or 7 years old.

The product has a four layers architecture (*cfr.* Fig. 2.1). The top level is composed of the *meta-object*, *meta-link* and *meta-property* meta-classes. Instances of these meta-classes describe the second level. The instances of *meta-object* are: *G-link*, *G-object*, *G-entity*, *G-property*, *G-dispatcher*, *G-method*, *G-hypergraph*, *G-graph*, *G-action*, and *G-shape*. The instances of *meta-link* are: *G-composition*, *G-inheritance*, *G-assignment*, *G-connection*, *G-exclusion*, and *G-inclusion*. The instances of the *meta-property* meta-class describe the properties of the instances of the *meta-object* meta-class. *G-graph* instances are composed of *G-object* and *G-link* instances and can be themselves be components of *G-hypergraph* instances. The *meta-link* instances draw links between the instances of the *meta-object* meta-class. For instance, *G-connection* instances will attach *G-link* instances to *G-object* instances; *G-assignment* will attach *G-property* instances to *G-object*/*G-link*/*G-graph* instances; and *G-inheritance* instances will define the inheritance graph between the instances of *G-object* (but also *G-link*, *G-graph*, ...). Methods[1] and daemons[2] can be attached to the instances of the *meta-object* meta-class. GraphTalk has a script language (GQL) that is close to SQL.

Meta-objects that inherit from *G-shape* instances are displayed in accordance with the characteristics of the inherited shape. The shape of these instances is defined in a graphical editor.

Ledit is a distinct meta-tool that allows the method engineer to define the syntax of a foreign language and to produce reports (or code) from the information stored in the meta-CASE. The coupling between the output text and the information is strong enough to make possible the "reverse-maintenance" of this information when the text is edited.

2.1.3 KOGGE

References [EWD+96, ESU97, EC94, ESU98, Fra97]

KOGGE[3] is a meta-CASE developed by the University of Koblenz (Germany). The meta-meta-model is a conceptual interpretation of raw-level entities of the graph theory. On one side a conceptual language EER/GRAL[4] is proposed: EER diagrams (*cfr.* Fig. 2.2) can be

[1] Methods are routines written in C,C++or LISP.

[2] Daemons are routines that are triggered before/after some events like modification, edition, creation, ...

[3] KOGGE: KOblenz Generator for Graphical design Environments.

[4] Extended Entity Relationship/GRAph specification Language [EWD+96].

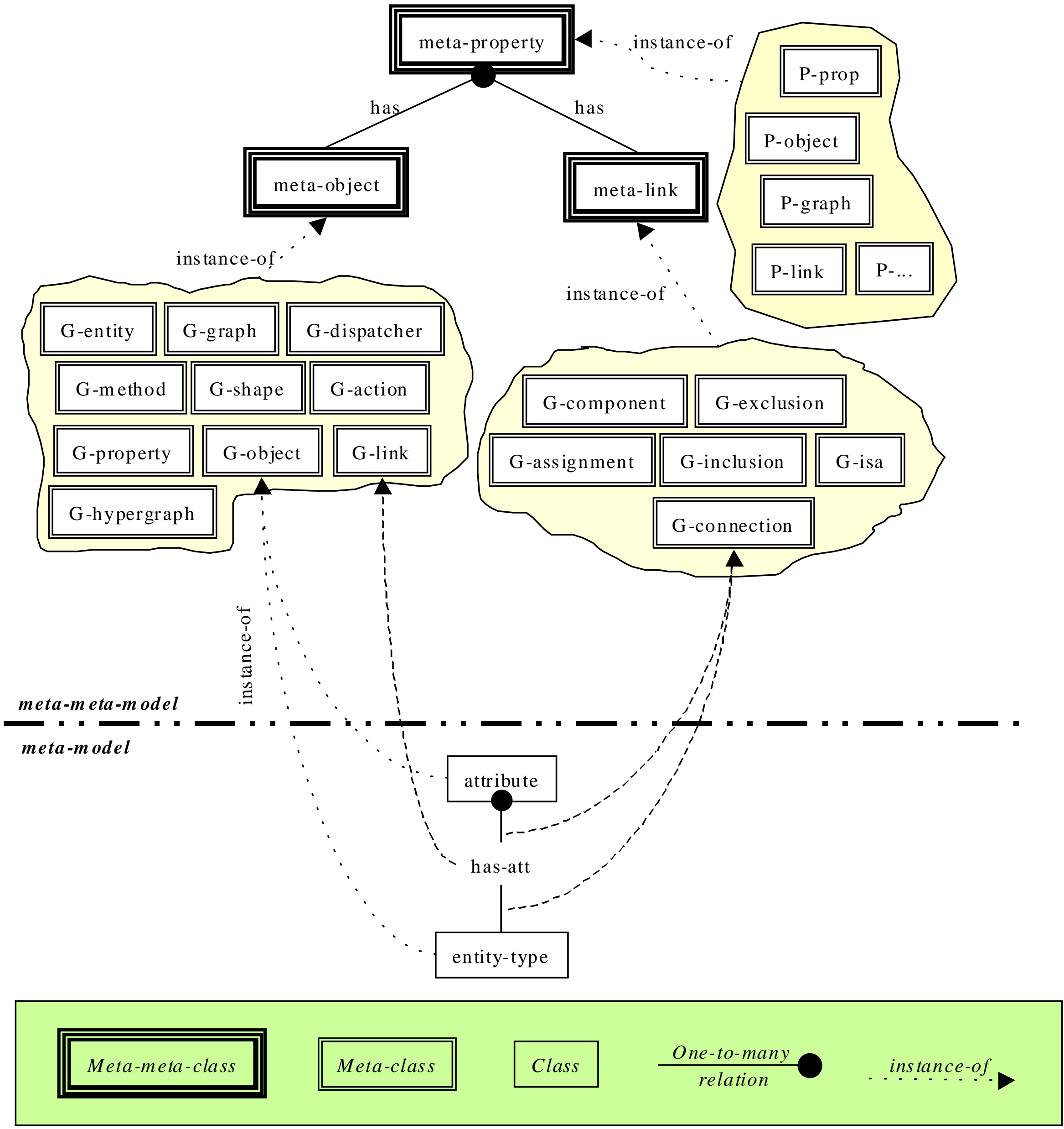


Figure 2.1: [**The GraphTalk's Meta-Meta-Model**] This meta-meta-model has been recovered from the reference manuals. For this reason, this meta-model is not exhaustive. Only the three top levels of the architecture are illustrated. The instance level is not present.

described with *entity types*, *relationship types*, *generalizations*, *attributes*, and *aggregations*. The GRAL language[1] allows the method engineer to add additional constraints to enforce integrity rules. On the other side, the authors have defined TGraphs which are a generalization of graphs. In this approach, graphs are directed, typed, attributed, and ordered[2]. TGraphs are defined as mathematical objects (*vertices* and *edges*) with the Z-notation [Spi92].

The mapping between the conceptual level and the TGraph formalism is defined in a straightforward manner [EWD+96]:

- entity types denote vertex types,
- relationship types denote edge types,
- generalizations describe a vertex type hierarchy,
- attributes describe the attribute structure of vertices and edges, depending on their type, and
- higher-level modelling constructs (like aggregation) add additional structural information.

KOGGE is a "*3=2+2*" levels architecture (see below). The development of a CASE tool in the KOGGE's methodology follows two stages. The first step consists in defining three major components:

1. the EER model, and
2. the definition of the menus (modelled as directed acyclic graphs), and
3. the interaction of the CASE tool with the software engineer. This is modelled with a Statechart diagram. The CASE tool is always in a certain state, and events can cause state changes which trigger actions defined in KOGGE-Modula (see below).

All of these components have been bootstrapped in KOGGE, and hence, have a dedicated graphical editor. The method engineer can enrich these components with procedures written in KOGGE-Modula[3]. This environment (urKOGGE) has thus been developed with KOGGE itself. Once this step is completed, the next step is simply the generation of the CASE tool. This package (i.e. the CASE tool) is composed of three interpreters[4] and a graphical module (the kernel is developed with VarioCAD, which is originally a CAD system). The tool produced in this stage is named a "*KOGGE tool*". For instance, urKOGGE is a KOGGE tool. This two step methodology explains the equation "*3=2+2*". The first stage (urKOGGE) utilizes a meta-meta-model to define a meta-model, and the second stage (the KOGGE tool) drives specifications with respect to some meta-model produced in the first stage.

Recent works [ESU98] in this research group have endowed KOGGE (renamed JKOGGE for this purpose) with hyper-references across the WEB. The new system uses *distributed graphs* in place of TGraphs. They have been enriched with the concepts of *ProxyVertices* and *ProxyEdges* which reference vertices and edges in other graphs distributed on the WEB.

[1] GRAL is a Z-like language [Fra97].

[2] i.e., the edges incident with a particular vertex have a persistent ordering.

[3] The KOGGE-Modula macro language is a extension of Modula 2 with GRAL statements.

[4] One interpreter per component.

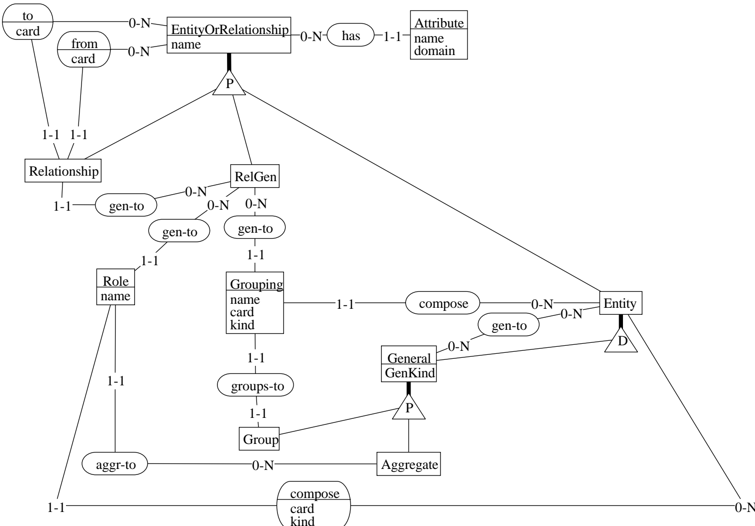


Figure 2.2: **[The KOGGE's Meta-Meta-Model (EER)]** This meta-meta-model has been defined in [EWD+96] on page 21.

2.1.4 MetaEdit+

References [Kai97, MHR96, RK99, Met99, Con99, KLR96, Met96, Mar95b, Mar98, Kel97]

MetaEdit+ is a multi-user, multi tool, multi-method, multi-form, multi-level meta-CASE developed by the University of Jyväskylä (Finland) and is distributed by the MetaCase Consulting company. MetaEdit+ is a three-levels architecture[1] and uses the GOPRR[2] conceptual data model to define the methods (*cfr.* Fig. 2.3). This model is structured with the following concepts: Graphs, Objects, Properties, n-ary Relationships, and Roles. Properties describe the objects, the relationships, the graphs, and the roles. Roles can have cardinalities and can be played by several objects. A graph is composed of objects and relationships (with specific roles). Moreover, graphs can share components. Objects can have decomposition graphs and several explosion graphs[3]. Specialization/generalization between objects is possible via simple inheritance. Properties can be constrained by rules which can be overridden in the subtype properties.

The graphical representation of specifications is defined in a quite direct way. A graphical editor allows the method engineer to describe each object with a shape composed of its attributes. This shape is automatically used when the objects are displayed. MetaEdit+ also proposes other kinds of representations like matrices or tables. Nevertheless, objects always have the same representation in the diagrams (graphs).

The product has no integrated programming language. Nevertheless, it is possible to define customizable reports with a script language (ReportDesigner). Recent works have added hypertext facilities [Kai97] and process control [Mar98] to the tool.

[1] The information systems development level, the methodology engineering level, and the meta-meta-level.

[2] GOPRR: Graph Objects Properties Relationships Roles.

[3] Although the decomposition can describe many objects at the same time, explosion is a mechanism which is attached to a single object.

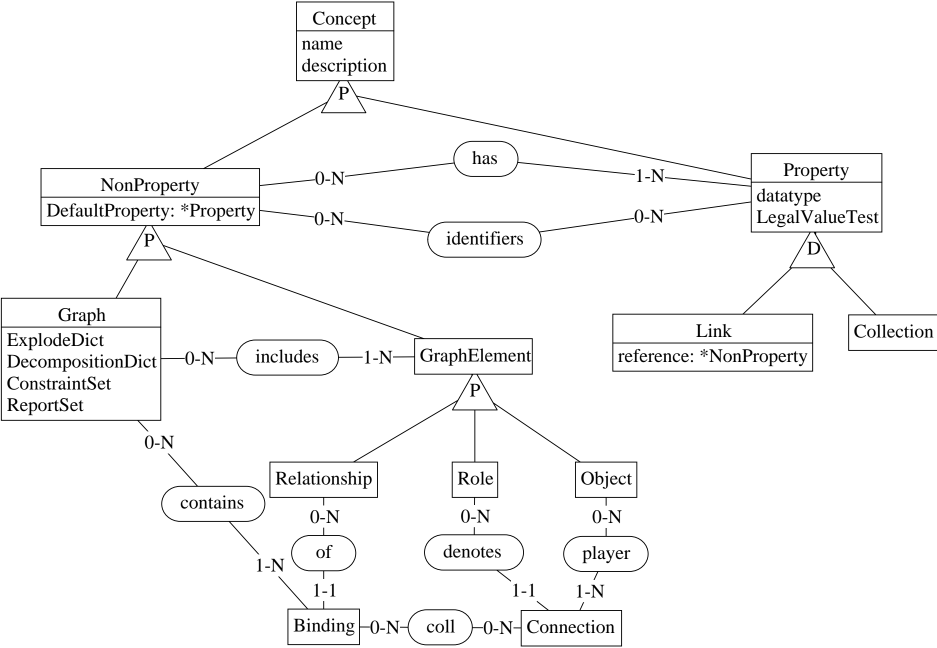


Figure 2.3: [**The MetaEdit+'s Meta-Meta-Model (GOPPR)**] This meta-meta-model has been recovered from the Steven Kelly's PhD Thesis[4] [Kel97] and [LMT+98]. Nevertheless, this meta-model is incomplete. Indeed, the MetaEdit+'s meta-meta-model is a Smalltalk schema of which the meta-models are derived by specialization. Hence, an ERA or Statechart meta-model would be specified as a Smalltalk class which inherits from the `Graph` class (and so on for the other concepts — meta-class, meta-property, meta-relation, ...). For this reason, MetaEdit+'s meta-models are implicitly endowed with the Smalltalk's inheritance, and by way of consequence, this facility is not depicted in our schema. This meta-meta-model is used for modelling both the meta-models and their instances (i.e., the specifications). Nevertheless, for convenience sake, this feature is not illustrated in our schema.

2.1.5 MetaView

References [FTS95, SFT96, ZFS95, Fin94a, Fin94d, Fin94b, GFS+94, Fin93b, Fin93a, Fin92a, Fin92b, STM91, Fin94c]

MetaView [Fin93a] is a meta-CASE under development at the Universities of Alberta and Saskatchewan (Canada). It is based on a three-layers architecture: the meta-level, the environment level, and the user level. The meta-level uses the EARA/GE[1] [Fin94b] model to describe the environment level. This model uses the artifacts of *entity*, *relationship*, *role*, *aggregation*, and *attribute*. Entities, relationships and aggregations can be described with attributes. The meta-level's model contains advanced modelling techniques like aggregation and generalization. The aggregation combines a set of entities and relationships to form simple, high-level aggregate that can, in turn, be used in other aggregations. Generalization makes possible the inheritance of properties (like aggregation's components, roles, and attributes) between aggregation, relationships, and entities. This relation defines a hierarchy between the artifacts. Nevertheless, the inheritance is simple. Relationships can have multiple roles and have several "schemas", that is, a relationship can have several signatures. For instance, the `transition` relationship in a Statechart meta-model could be defined like this:

```
transition(init,state,event)
transition(state,state,event)
transition(state,final,event)
```

This definition prevents the user to define transitions from *final* states to the *init* state[2]. Moreover, this makes possible the definition of dedicated graphical representations for every pattern.

The graphical representation of the specifications is defined with the GE extension of the EARA model [Fin92b, GFS+94, GLM94]. The GE language consists of graphical objects (diagrams, icons, edges, labels, subdiagrams, and adornments) that the method engineer can associate with the conceptual elements of the EARA model. Figure 2.4 shows the mapping between the conceptual and the graphical objects.

The tool includes a powerful constraint language ECL[3] [Fin94c]. Authors have succeeded in defining a simple declarative language to express constraints to be satisfied by the specifications. Besides the consistency and completeness constraints, ECL makes it possible to express graphical constraints[4] and error messages.

2.1.6 ToolBuilder

References [ACE99, IPS98, Lin98]

Last but not least, this product is also the oldest one. The genesis of ToolBuilder can be traced back to 1972. ToolBuilder is now a mature product (distributed by Lincoln Software Ltd[5].) which has supported very large development projects. ToolBuilder is bootstrapped in itself. This feature makes its evolution much simpler than other rigid implementations.

[1] EARA/GE: Entity-Aggregate-Relationship-Attribute with Graphical Extension.

[2] As well as reflexive transitions from those states to themselves.

[3] ECL: Environment Constraint Language

[4] For instance: a subtype must be placed below its supertypes.

[5] Despite our many E-mails, the Lincoln's representatives have never communicated any information on their product.

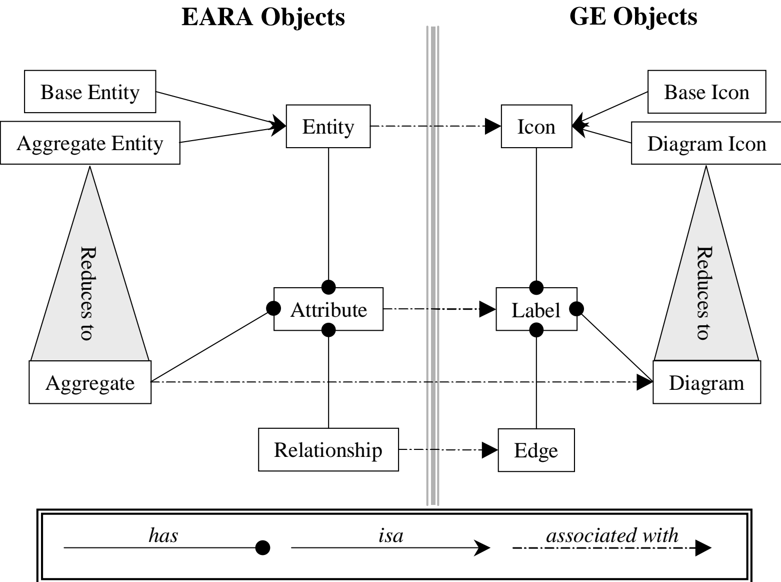


Figure 2.4: [**The EARA/GE Mapping**]

The experience of ToolBuilder also proves that the meta-CASE approach of the method engineering task is credible. Alderson [ACE99] cites a quite impressive list of success stories:

- The specification and the generation of the code and relational database definitions for an operational telephone mail-order system using 2.000 Sun workstations 20 Sun servers and an IBM mainframe, linking the world's largest Oracle database (1999).

- The Lincoln Software's Engineer Tool has also been created using ToolBuilder. Its data model consists of 510 entity types, 567 attributes and 1.339 relationships.

The development of a CASE tool with ToolBuilder follows two stages: *a.* the specification of the tool and *b.* the generation of the run-time.

During the first stage, the method engineer uses the METHS tools (that was bootstrapped with ToolBuilder) to specify the requirements: *a.* the *data model*, *b.* the *frame model*, *c.* the *diagrammatic notation*, and *d.* the *textual representation*.

The data model's ontology consists of entity types, attributes, and relationships. Entity types can be classified with an inheritance relationship. Relationships can be composed together to form aggregations, paths, or even recursive definitions. The data model is active in that triggers can be associated with events (attribute modification, entity type deletion, ...).

The frame model relates to the user view of the underlying data model. Frames are structured collections of either graphical objects or text objects. The method engineer can either associate new actions with events (cut-and-paste, navigation, checking, ...) or even override predefined operations.

The meta-CASE has its own integrated programming language (easel) that can be either interpreted or compiled to C++. Moreover, a range of API functions allow access to the underlying PCTE* database, user interface, etc.

The second stage generates the compiled specifications that will specialize the DEASEL generic CASE tool.

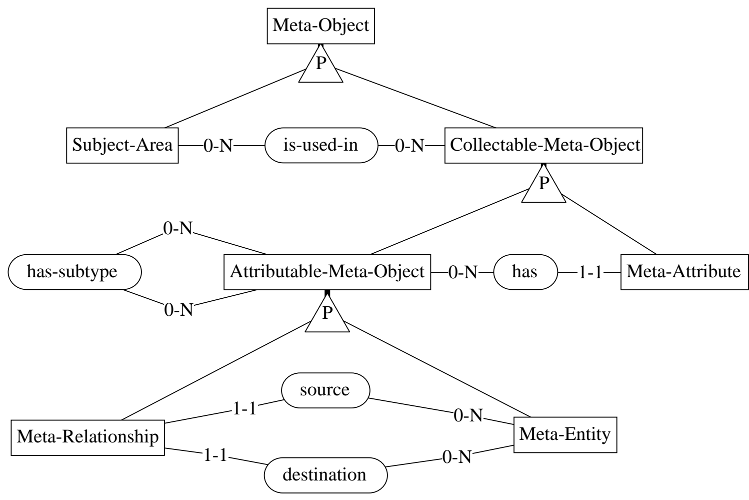


Figure 2.5: [**The CDIF's Meta-Meta-Model**]

2.1.7 CDIF

References [Ern97, Ele94b] (and also [Ele97, Pid96, Ele96a, Ele96b, Ele96c, Ele95a, Ele95b, Ele94a, Ele94d, Ele94c]).

CDIF[1] is a collection of standards and defines the interface of an architecture to exchange data between modelling tools and repositories. It has been defined by the CDIF Division of the EIA[2], an industry standards committee. Hence, CDIF can serve as a bridge with a well-defined interface to import or export data between tools. So far, the committee has defined about ten integrated meta-models of various methods such as "data flow model", "state event model", "object-oriented analysis and design model", ... Each method has been modelled by the EIA in a "*subject area*" definition in using the meta-meta-model defined in [Ele94b].

CDIF has been used in many situations either as an interface of an integrated repository for CASE tools or as an exchange format between tools. Moreover, the large variety of subject areas it supports demonstrates the genericity of the EIA's approach. This should suffice to demonstrate the expressiveness of its meta-meta-model (depicted in Fig. 2.5). It provides thus an interesting benchmark for other meta-meta-models.

2.1.8 Universal Repository

References [Uni96a, Uni96b]

Unisys began to create its universal repository (UREP) in 1995. It was later proposed to the Object Management Group (OMG) and has evolved into the OMG's Meta Object Facility (MOF*) specification. Its recent adoption by the OMG makes it a technical leader in this domain and the success story of UML justifies our interest in this product. The UREP's meta-meta-model is depicted in Fig. 2.6.

[1] CDIF: CASE Data Interchange Format.
[2] EIA: Electronic Industries Association.

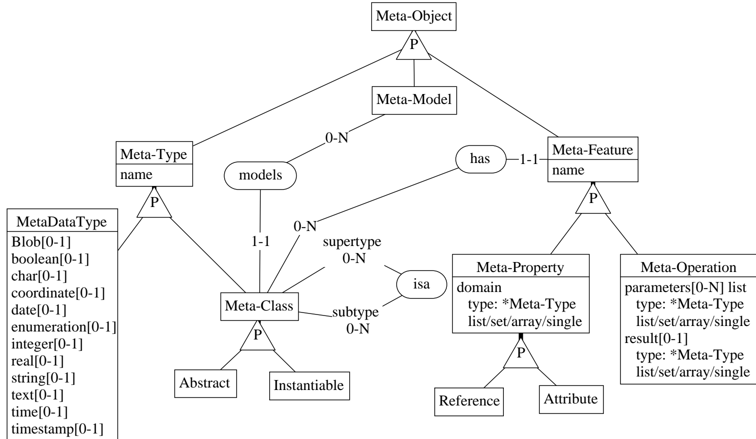


Figure 2.6: [**The UREP's Meta-Meta-Model**]

2.1.9 And Many Others

The previous sections have presented some remarkable references in the meta-modelling realm. Nevertheless, this list is not exhaustive and many other products exist: Phedias [WL95], DOME[1] [Hon99a, Hon99b, Hon99c], MetaMOOSE [FPD+99],MViews [GH93], RAMATIC, [BBD+89], Hardy [SR96], Vampire [McI95] or CASE shells [ZK96, ZK95] ... It is impossible to review all these products in this thesis. Nevertheless, the previous sections have presented a large sample of the state of the art.

2.2 Meta-CASE Comparison

This section summarizes the previous sections in order to derive general observations about the meta-CASE realm. About ten operational meta-CASEs have been investigated. Although some of them follow orthogonal directions when modelling the problem, they present common qualities or weaknesses. Table 2.1 lists some key features.

2.2.1 Meta-meta-model Semantics

The meta-meta-models presented so far show two trends. On the one hande, architectures[2] that aim at implementing complete meta-CASEs have quite complex[3] meta-meta-models. On the other hand, systems[4] which did not integrate the interaction[5] with the software engineer in their requirements, have quite simple meta-meta-models (i.e., meta-class, meta-property, meta-relation and some way to gather them together), these can be called "repository" approaches.

This dichotomy can be explained easily. The first tools are intended to produce complete CASE tools with a minimum of efforts while this challenge does not fall within the concern of

[1]DOME: DOmain Modelling Environment, Copyright Honeywell, inc.
[2]GraphTalk, KOGGE, MetaEdit+, MetaView, and ToolBuilder
[3]Complex means here: a high number of concepts and a non trivial semantics.
[4]CDIF, UREP, and ConceptBase
[5]I.e., the graphical representation, the dialogues, the process modelling, ...

	CDIF/UREP	ConceptBase	GraphTalk	KOGGE	MetaEdit+	MetaView	ToolBuilder
Generation	∅	interp.	interp.	C++	interp.	interp.	C++
Implementation	∅	Prolog&C++	?	C++(β)	Smalltalk	Prolog&C	?(β)
Multi-user	∅	yes	yes	no	yes	yes	yes
Meta-meta-model	CDIF	O-Telos		EER/GRAL	GOPRR	EARA	?
Meta-model editor	∅	yes	yes	yes	browser	no	yes
Aggregation	no	no	yes	yes	yes	yes	no
Inheritance	multiple	multiple	simple	simple	simple	simple†	yes
Prog. lang.	∅	CBQL	GQL	KOGGE-Modula	report script	rules	easel/C++

† Although the tool certainly supports generalization, we found no trace of it in its meta-model (see Fig. 2.2).

Generation : "interp." means that the CASE tool is a component that is interpreted in the meta-CASE itself. Hence, the meta-CASE and the CASE tool are one alone program. The "C++" item means that the meta-CASE will generate C++code from the CASE tool specification.

Implementation : The native language of the meta-CASE. The "β" character means that a part of the meta-CASE has been bootstrapped.

Multi-user : Will the CASE tool support concurrent accesses from many users (i.e., software engineers).

Meta-meta-model : The name of the meta-meta-model.

Meta-model editor: Does the meta-CASE support an integrated editor for the meta-models.

Aggregation : Does the meta-meta-model support aggregation?

Inheritance : Does the meta-meta-model support classification of meta-classes?

Prog. lang. : Does the meta-CASE have an integrated programming language?

Table 2.1: [**Meta-CASEs Conspectus**]

the second school. "Minimal efforts" often means (at least for those tools) producing in a quite automatic way both a graphical representation and a user interface. So far, the best way to achieve these goals consists in defining meta-meta-models that integrate those requirements. For instance, the "polymorphic relationships" of MetaView makes it possible to define peculiar graphical representation of roles depending on the type of the participating classes (i.e., the extremities). In MetaEdit+, S. Kelly explains that "*An example of a complex property can be found from object type Class in object-oriented methods because it has collections of attributes and of methods as properties, and each attribute or method itself is an object, and as such can have one or more properties. In principle these new properties could be again complex ones and therefore they too could lead to more properties and so on. GOPRR does not limit the number of complex properties or their depth on any way: for instance, cyclic structures are allowed*" (Appendix 1 of [Kel97]). This excerpt explains how the complexity of the GOPRR meta-meta-model has to match/represent the complexity of the studied method. Although "repository" approaches would model this case with meta-relationships between *classes* and *attributes* (resp. *methods*), MetaEdit+ prefers to trace the complexity out of the method in their meta-meta-model. And indeed, the way this tool generates and manages the dialogue boxes is impressive. Nevertheless, the complexity of these meta-meta-models exceeds largely that of, say, ERA, UML, and CDIF. Let us note that few meta-CASEs have integrated the multiple inheritance in their semantics.

2.2.2 CASE Generation

Some tools prefer to generate raw code rather than interpreting the CASE tool. Both approaches have their strengths and weaknesses. The first approach permits the meta-CASE architecture to benefit from classical toolkits (like C++ with X-libraries, relational or object-oriented databases, ...). In the meantime, these toolkits do not generally permit reification*. For this reason, other meta-CASEs have to choose more advanced toolkits (like Prolog, Lisp or Smalltalk). Hence, they have to redefine and implement their own modelling of the distribution, the concurrent accesses, the security, the graphical user interface, ... This leads to additional constraints. Nevertheless, they are difficult to circumvent when we wish to allow the method engineer to modify the methods. A two-stages generation breaks the link between the CASE tool and its specification, repository updates must be made either with batch scripts or with distant accesses. That is no the easiest way to proceed. On the other side Alderson [ACE99] explains that frequent modifications of the methods must be discouraged. The best solution is certainly a compromise.

2.2.3 Specifications Representation

All the generated CASE tools propose graphical representations of their information system. If the way to proceed differs from a meta-CASE to another (MetaEdit+ and GraphTalk use a graphical editor to define this component; MetaView prefers textual scripts with numerical positions; GraphTalk models the graphical representation in its repository and make possible to derive meta-classes from such "modelled shapes"; ...), all of them have adopted the same approach, that is, mapping meta-classes to shapes and meta-relations to edges — the shapes being defined from the meta-properties of the meta-class. Few of them (MetaEdit+) propose multiple graphical representations, but this freedom is limited. For instance, here is a list of requirements that were not fulfilled:

- Textual views[1] are never supported. MViews [GH93] claims to implement them, but, since this framework must be completed by hand, this makes it an outsider in our comparison.
- They do not propose several graphical views of a same specification such as extended and simple views, nor views with respect to distinct criteria[2].
- Contextual display is often needed to represent evolving situations or transient properties. For instance, the animation of a Statechart could require that the appearance of a state varies according to its condition (active/passive); selected or marked objects can change their colour; ...

2.2.4 User Interface

Three cases were observed:

1. The user interface's skeleton is automatically deduced from the meta-model's definition (its static part and its constraints). The method engineer is then free to customize this interface (moving/removing edit fields, changing fonts and colours, ...).
2. The user interface is limited to a graphical display with a controlled interaction with the software engineer. The latter can add/move/delete objects, edit its fields one by one, and drag edges between objects with respect to the signatures of the relationships the objects belong to.
3. The dialogues are partially modelled and can be customized with a classical graphical toolkit (for instance, the X library). This approach has no real limitations, except that it does not much help the method engineer.

2.2.5 Programming Language

The spectrum is quite large. Although some systems have no programming language, no scripting language, and no rules (active databases), others have all of them. We can derive three cases:

1. The system has its own programming language. It can be interpreted (ConceptBase, DOME), or compiled with the CASE tool (KOGGE).
2. The system relies on its "active repository" to enforce complex constraints. Nevertheless, this approach makes it impossible to write advanced applications like report generator, metrics computation, parsers, ...
3. Some systems have no programming languages (MetaEdit+, GraphTalk) but have sophisticated ways to define ASCII generators.

Finally, some architectures make the interface of their repository public. This makes it possible to add new functionalities (written in C or C++ for instance).

[1] We do not consider textual reports as views

[2] For instance, OO diagrams with(out) attributes, with(out) methods, with(out) constraints, ...

2.2.6 Methods Integration

Method integration is an important issue for meta-CASE architectures since their flexibility places them naturally at the intersection of methodologies that were not intented to coexist. Hence, meta-CASEs should propose a methodology along with mechanisms to facilitate this task. Nevertheless, few meta-meta-models surveyed really assist the method engineer to achieve the integration. As far as we know, the only mechanisms we have observed are proposed by MetaEdit+. Meta-properties (and their instances) can be shared by meta-classes and meta-classes (resp. classes) can be shared by meta-models (resp. specifications).

2.2.7 General Criticisms

This comparison has pointed out some weaknesses which are generally partially fulfilled requirements. Nevertheless, some important questions are still left unanswered. We can cite for instance:

long transactions: Transactions that stretch out as far as several weeks.

versioning: To remember the successive states of an object.

large databases: Meta-CASEs often address the problem of complex databases with small extensions. Reverse engineering and reengineering tasks often need to manipulate huge collections of programs, data, ...

cooperative work: The interaction between software engineers (messaging, events, warnings, agenda, etc).

support of foreign tools: compilers, version managers[1].

process modelling: the process modelling is rarely taken into account. It seems that first priority in the design of a CASE tool still consists in generating rapidly an operational prototype. The integration of this component in the conception would slow down the "machinery". Moreover, present meta-CASEs manage with satisfaction the problem of the evolution of their meta-models (and their specifications). The enforcement of processes would make this task still harder.

2.3 Conclusion

1. Contrary to general belief, quite simple meta-meta-models seem sufficient to model methods (*cfr.* CDIF, UREP, [HSB98, ES97, SE97]). The extra semantics found in complex meta-meta-models if we compare them with UREP or CDIF is mainly used to derive the user interface or to enrich the graphical representation of the specifications.

2. If the evolution and the a posteriori customization of the CASE tool is of the utmost importance for the method engineer, then the generation by interpretation is the more appropriate (but also the more complicated to implement). Let us remember that this criterium is generally the foremost motivation to utilize a meta-CASE!

[1] For instance, PVCS, RCS.

3. Whatever are the statements, no one has suqcceeded in defining a meta-meta-model which is independent of the way the method engineer wishes to visualize it. All of them use the same approach (one meta-class $\leftrightarrow$ one shape). Hence, aggregated shapes oblige him to define the meta-class as an aggregation when its representation goes beyond the intrinsic semantics of this meta-class.

4. The user interface is always "repository oriented" (as for the representation). Other approaches should be examined. Meta-CASEs should also support the modelling of task oriented interfaces [Joh99] and, hence, offer a bridge between the *scientist representation* and the *real world*.

5. Programming languages are generally too weak (scripts for generating reports) or inadequate (too slow, too complicated, badly integrated).

6. There is no systematic reflection on the crucial problem of methods integration.

Part II

A New Meta-CASE Definition

Chapter 3

"*If you can dream it,*
you can do it."
WALT DISNEY

What should a Meta-CASE be?

We have explained in section 1.1 that a meta-CASE should compete against conventional CASE tools in 5 directions to succeed the "Turing test". This 5-dimensions space[1] can be summarized as follows:

Information : How complex can be the information modelled in the tool?

Representation : How easy is it to display complex representations of this information?

Intelligence : Can the tool execute complex tasks easily?

User Interface : Can people communicate easily with the tool?

Communication : Is the tool open (with other tools, communication protocols, ...)?

We will represent this space according to two criteria: *1)* to which extent can CASE tools and actual meta-CASE tools be customized (Fig. 3.1) and *2)* how rich is each dimension for conventional CASE tools and meta-CASE tools (Fig. 3.2). Such diagrams are of course quite subjective since they tend to model a large diversity of products with only one number for each dimension and the number itself is assigned in a subjective way. Nevertheless, they should match reasonably the reality. It is obvious that more the meta-CASE tools will be customizable, more the weaknesses observed in Fig. 3.1 will dwindle. For this reason, we have reproduced the gap between the expressivity of both architectures in Fig. 3.3. This new diagram puts thus forward the directions that we should follow to improve the meta-CASE tools in a drastic way.

Each dimension will be discussed in the next sections.

3.1 The Information Dimension

We have not observed important lacuna in the information model of the current meta-CASEs. Their meta-meta-models are quite expressive from the information point of view. Nevertheless, they generally use brand-new concepts that do not belong to the common qualifications of a software engineer[2]. Moreover, those concepts are rarely easy to learn (or to teach). Hence,

[1]Marttiin *et al.* [MRTL93] had already identified four components: database, interface, extension kit, and knowledge.

[2]We should use the "method engineer" term here. Nevertheless, as it will be explained later, this profession is also concerned by this sentence.

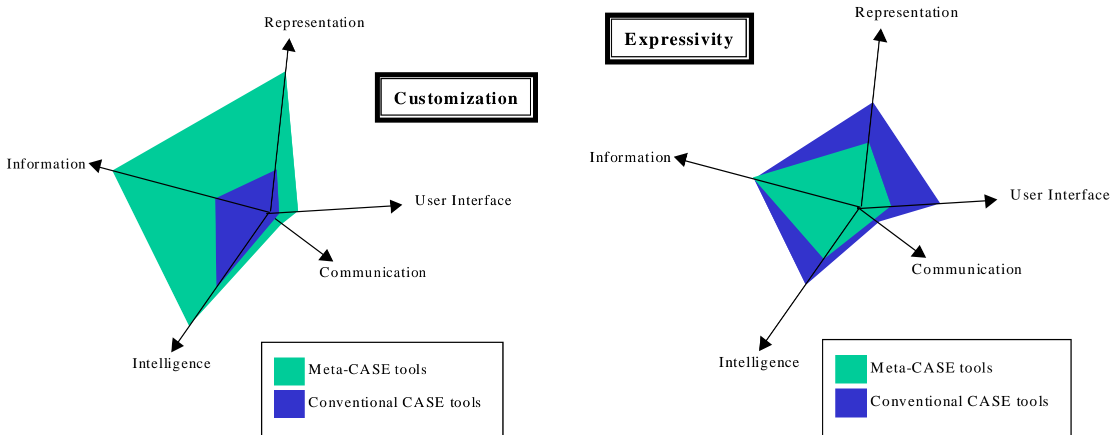


Figure 3.1: [**Comparison of the Architectures Customizations**] Meta-CASE tools surpass obviously the CASE tools architectures — by definition. Nevertheless, the *User Interface* and *Communication* themes are a bit neglected.

Figure 3.2: [**Comparison of the Architectures Expressivity**] This diagram stresses the *Intelligence*, *Representation* and *User Interface* axes.

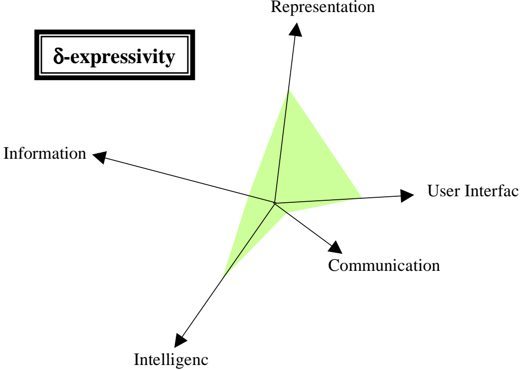


Figure 3.3: [**The Gap Between the Expressivity of the CASE and Meta-CASE tools**] Each dimension is the difference between the respective dimensions of CASE and meta-CASE architectures in Fig. 3.2.

it would be pertinent to define a "simpler" meta-meta-model[1]. This motivation is mainly enforced by a lack of encapsulation of meta-meta-models: software engineers have generally to be aware of the meta-meta-model's semantics to use the meta-models. We do not criticize this, since this terminology can serve as a franca lingua between software engineers to understand, to compare, to stand back and look objectively at their work. Nevertheless, too complex meta-meta-models do not facilitate this "introspection".

3.2 The Representation Dimension

So far, the graphical[2] representation has always been modelled in the same way: one concept ↔ one shape. If this restriction has sometimes been nicely skirted by someones (e.g., MetaEdit+), it also has serious effects on the semantics of their meta-meta-models. This drawback can make more difficult the understanding of the meta-models semantics[3]. Moreover, this principle (i.e., one concept ↔ one shape) contradicts with the wish to have simpler meta-meta-models.

3.3 The Intelligence Dimension

The "*intelligence*" is certainly the main value added of a CASE tool. The *intelligence* has three aims:

1. To make possible the definition of operational processes that will exploit the specifications.
2. To be the bridge between the various dimensions.
3. To overcome the intrinsic limits of the other dimensions.

The current meta-CASEs propose a quite broad spectrum of modelling languages about this aspect. Nevertheless, their approach is either limited to *1)* a restricted scope (languages dedicated to queries, constraints, or report generation), *2)* an API* of their repository, or *3)* a bootstrapped component in a meta-language (e.g., Prolog, Lisp, SmallTalk). Apart from the obvious limitations of the two first techniques, the last one introduces high-level languages that software engineers have difficulty to learn. Moreover, their semantics is well-defined and could thus not match the semantics of the information model. For instance, those languages do not support features like aggregations, multiple inheritance, or queries in the information model's paradigm.

Software engineers can be "large consumers" of the information model as the DB-MAIN project[4] tends to prove it, and they naturally wish to exploit it by adding new processes: parsers, text generators, analysis and metrics are very common applications.

[1] We mean an intuitive semantics from the method engineer's point of view.

[2] The textual representation is rarely taken into account. Apart from MViews and GraphTalk, no tools propose (strongly or weakly) synchronized textual and graphical views.

[3] This mainly occurs when a specification must be displayed according to rules that were not taken into consideration when the meta-model has been defined.

[4] First, the DB-MAIN CASE tool already supports an embryo of meta-repository. This allows the method engineer to add new meta-properties to existing meta-classes. This feature is heavily used by both our joint projects (distributed applications, temporal databases, reverse engineering, statistics, ...) and our private partners. Secondly, the Voyager 2 programming language is the best means to exploit those meta-properties. It has been used by graduate and master students, private companies, and researchers.

Remark DB-MAIN is a CASE tool developed at the University of Namur under the leadership of Prof. Jean-Luc Hainaut. Altough this tool is mainly dedicated to database modelling, it can be customized and extended to other domains. The tool and its documentation are available at the following address: `http://www.info.fundp.ac.be/~dbm`.

Hence, this language must be defined with great care. It must be simple enough to be appealing to method engineers, and in the same time, it must incorporate all the peculiar features of the information model (i.e., the repository) in a very natural way. For instance, the repository should be a seamless extension of the language's data model and conversely. In the same way, the language should offer syntactic constructs to help the programmer to manage and to map his ideas to the meta-CASE's idioms.

3.4 The User Interface Dimension

The *user interface* dimension is probably the major factor (with the *representation* dimension) which will convince or sicken a decision-maker. Unfortunately, our experience with the DB-MAIN tool tends to prove that the user interface can show a very large spectrum of facets. This complexity makes very difficult the modelling of this dimension. Besides some well-known needs (like property boxes and contextual pop-up menus), many other utilizations exist: analysis tools, editing parameters of processes, modal/not modal dialog boxes, chaining of dialog boxes, contextual help, error messages, text editing facilities, wizards, etc. They all require an expressivity that can only be found in graphical toolkits (e.g., MFC, Tcl/Tk, Motif, ...).

This dimension would require too much efforts and would thus exceed the limits of this Phd Thesis. Nevertheless, we will try to make our contribution to this edifice. The main[1] requirement is obviously the creation of the dialog boxes to create and modify the objects of the information model. We will present an automatic process that makes it possible to create complex dialog boxes to implement those two goals. Although our proposal is the exact opposite of what has just been commented (i.e., this component can not be defined in an automatic way), this has the merit of proposing an operational prototype that can immediately be validated. Indeed, this "swiftness" is a common characteristics of the interpretative framework that makes them very valuable. This functionality should obviously be completed by a more powerful toolkit.

3.5 The Communication Dimension

The *communication* dimension is rarely taken into consideration in the current meta-CASE tools. The realm has still to face important problems to compete with the hand-coded CASE tools category. Moreover, the latters do not often exploit this capability (except in some dedicated tools). Some works have investigated this direction (Kogge [UES98]) but do not propose any technique to help, e.g., cooperative work. Such a work should cover these topics:

- Cooperative work and teamwork.
- Documents and messages exchange.

[1] This functionality is the more important because it appears first.

- Versioning and integration of specifications.
- Customizable and distributed event manager.
- Support of open standards like ODBC, CORBA, ...
- etc.

This non-exhaustive list suffices to demonstrate the complexity of this last dimension. This component (once modelled) could be partially bootstrapped or emulated on top of the four previous ones.

3.6 The Requirements

This section will present the challenges we set ourselves in defining our meta-CASE.

The previous observations indicated clearly that some directions still needed to be investigated. We decided to consider the problem in its entirety. This leads us to propose a brand-new architecture where the integration of the various components should be emphasized. The two key words that dictated our work have been: expressivity and simplicity. For this reason, we estimated that the product should:

1. be able to take into consideration all the information that goes beyond the scope of classical CASE tools. This information is often complex and stretches over several ontologies or domains[1]. This complexity must then be hidden and the meta-CASE must be able to project it on pertinent subject areas according to specific graphical conventions;
2. enable the engineers to process this information easily. They should have a simple and expressive programming language at their disposal;
3. minimize the investments of both the software engineer and the method engineer. Its learning curve should be as short as possible;
4. be user-friendly from the point of view of both the method engineer and the software engineer;
5. be open to other CASE tools.

The repository should be defined with as few concepts as possible to speed up its understanding and to reduce the risk of error. Indeed, we observed that concurrent repositories were sometimes unnecessarily complex — generally, this complexity did not aim at helping the method engineer in defining his meta-model. It should be based on common concepts and common composition rules.

The complexity that has been removed from this hypothetical repository was, of course, motivated. Meta-CASE architects need it to improve the "comfort" of both the dialogue boxes and the graphical representation of the specifications. The other meta-CASE's components will thus have to apprehend and to make vanish this stumbling block although they should remain simple.

[1] Organization modelling, traceability, requirements, security model, functional analysis, network configuration, user interface, distributed computing, workflows, planning, ...

The symbolic language GraSyLa should fill the gap between the *hypothetical* austerity of this repository and the advanced challenges the method engineer will entrust to the meta-CASE. Structural mappings established too rigid links between the meta-model definitions and their representations. Hence, GraSyLa will play the role of an elastic hinge between the repository and the graphical requirements. When the latter will change, GraSyLa scripts will be modified and the meta-models (and their instances) will be left unchanged.

The user interface is certainly the more active and more visible component that interacts with the software engineer. Nevertheless, proposing a solution in this thesis would irremediably leads us to propose a user interface model with the well-known limitations of this approach in this domain. Nevertheless, it would be possible to offer a presentation layer of the repository in using the semantics of the meta-models.

The keystone of this architecture will be played by the programming language (Voyager 2^+). The Voyager 2 programming language that has been proposed in the DB-MAIN CASE tool has proved its usefulness: it is efficient, simple to use, reliable, and is now a valuable component. This work should translate this success story in the context of a full meta-CASE. The best recipe to preserve the intrinsic qualities of Voyager 2 in this new context is to consider it as an instance of another programming language. The latter would be aware of the meta-layer, but it would propose mapping rules to build a programming environment similar to the one of DB-MAIN.

The meta-CASE should have an architecture open to other implementations such as repositories, databases, CASE tools, ... Nevertheless, the features of those frameworks are generally very peculiar and cannot easily be integrated whithout multiplying the number of the interfaces to them. But nevertheless, a solution exists and consists in wrapping every exogeneous component into an endogenous widget.

Those requirements will influence every component that composes our architecture (described in Fig. 3.4) as it is depicted in this table:

Requirement	**Component** (♯chapter)
1	repository (Chap. 4), GraSyLa (Chap. 6)
2	Voyager 2^+(Chap. 7)
3	Voyager 2^+(Chap. 7), GraSyLa (Chap. 6)
4	mirroring (Chap. 5)
5	mirroring (Chap. 5), Voyager 2^+(Chap. 7)

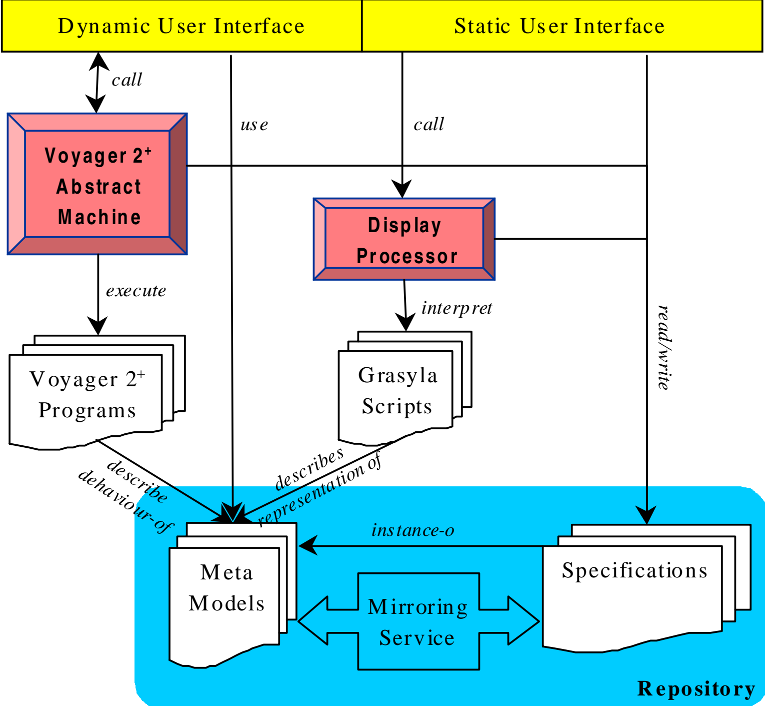


Figure 3.4: [**General Architecture**] The repository stores both the meta-model definitions and their instances (i.e., the specifications). Besides predefined functions, the user-interface is partally influenced by the repository's content. The static user interface concerns mainly administration tools (e.g., dialogue boxes to manage the meta-meta-model). The dynamic user interface depends on the meta-model definitions and is specific to a method.

repository: [...] The repository of all important knowledge in a small town was the chief barman of the local pub.
Collins Cobuild, dictionary.

Chapter 4

Repository Axioms

4.1 Introduction

CASE tools modelled in meta-CASEs are described in terms of concepts that will structure the mental representation of the "CASE world". The elementary block in this mental building will be the *meta-classes*. Meta-classes will be grouped together to form more or less complicated buildings called *meta-models*. Meta-models can themselves be used like blocks to build larger buildings. Since meta-models can be used as meta-classes, the method engineer can use a top-down approach in the CASE tool modelling, exactly like programmers use procedures as primitive statements. Here is summarized the general principles that will underlie our way to model the "semantics" of a CASE tool.

Any program can be decomposed in two levels: *1)* the *instance level*, i.e., all the data built, stored and manipulated by programs and *2)* the *type level*, i.e., the knowledge model of these data. Because CASE and meta-CASE tools are themselves programs, each one can be decomposed in this way. However let us remark that we can identify the instance level of the meta-CASE with the two levels of the modelled CASE tool since it is the subject of the meta-CASE tool. Therefore, we can refine the meta-CASE into three levels. We will call them respectively *meta-meta-model*, *meta-model* and *specification* levels. This three-levels architecture is depicted in Fig. 4.1. The database that stores these data (whatever the abstraction level) will be called the *repository*.

Concepts will be interconnected together with more or less complex relationships. To make the understanding of these graphs easier, we will use two visualization techniques. The *compact view* will focus the interest of the method engineer on the main concepts and their dependencies. The *extended view* will be used like a magnifying glass to explore subgraphs in detail. The DB-MAIN graphical conventions will be used for this last view (*cfr.* Appendix B).

4.2 Overview

Although we argue that this meta-meta-model is simple, it introduces new concepts that must be explained and motivated before being formally defined. We will not discuss here the concepts of meta-classes, meta-relations, and meta-properties since they are close to the terms

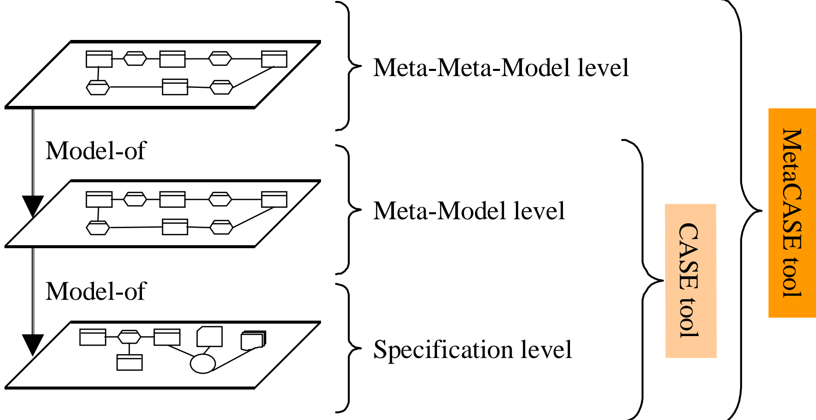


Figure 4.1: [**A Three-Layers Architecture of a Meta-CASE Tool**]

found in other models like ERA or UML. We will concentrate our efforts on the presentation of the more advanced concepts like the inheritance and the meta-models.

Our understanding of a software engineering project is a large graph of interconnected concepts that can be themselves be "exploded" to get other graphs. For instance, a node can denote an "entity-type" in a data model, a "state" in a statechart, a "statement" in a program, ... Graphs will denote aggregated information like an entity-relationship diagram, a statechart diagram, or a program. As we explained above, the distinction between atomic concepts and aggregated information is not as strict as we could expect it. Indeed, states can denote compound state, entity-types can denote business objects with a much more complicated definition than an *usual* entity-types, etc. Moreover, we observe that all those graphs can be laid down flat and that their nodes can still be connected together whatever the graph they belong to. Hence, the global graph that encompass this information can be very complex (*cfr.* Fig. 4.2).

Our will to define a repository as simple as possible leads us to propose an ontology that matches the complexity that has just been described. In other words, method engineers must have at their disposal the right vocabulary to describe their problems. For instance, the software engineering project described in Fig. 4.2 can be modelled by the diagram depicted in Fig. 4.3. The reader can observe that the definition of the meta-models is quite intuitive and the correspondance between this schema and the project is quite straightforward.

Meta-Models

Our perception of a software engineering project is a large graph (or semantic network), where nodes denote key information and edges denote relationships between those concepts. Moreover every node can be described by a set of values.

So far, this description does not stray from usual modelling approaches like ERA or UML. And in such approaches, the elements of those graphs are the subject of a taxonomy that divide the nodes and edges in several categories (classes,attributes,relationship, ...). Because we are modelling concepts that are generally one step higher (i.e., more abstract), we call those categories meta-classes, meta-relations, meta-properties, ...

Because the size of these graphs is often very large, engineers usually group nodes together

to propose simplified views. The latter can be disjoined, or can overlap other ones (and thus share nodes and edges). They act as windows on a complex world. As for nodes, those views are also classified into meta-models. In other words, although meta-classes describe the world, meta-models structure it. But depending on the level of abstraction, it is often judicious to hide the complexity of a meta-model and to use it as a meta-class to describe more abstract concepts. This leads us to consider the graph as an hyper-graph. And hence, we consider meta-models as specializations of meta-classes. This entails that meta-models can be used anywhere meta-classes are expected. Figure 4.2 shows how meta-models can structure the graph that represents a software engineering project.

The classification of graphs into meta-models entails that they denote also constraints on their components. For instance, meta-models could define constraints such as:

- to restrict its meta-classes;
- to restrict its meta-relations;
- to restrict the meta-properties of its meta-classes;
- to limit its size;
- to restrict its access (can be added but not deleted);
- or more complex constraints (such as predicates).

We have decided in this thesis to define a meta-model in terms of the type of its nodes. Hence, graphs will be constrained to contains nodes that are instances of types allowed by this meta-model.

Inheritance

We present in this thesis a new semantics for the inheritance relationship that is wider than the usual relationships in programming languages and other models. Our aim is dual: *1)* to propose a definition as expressive as in other OO models, and *2)* this relationship should use a minimum of hypotheses. We examine in the next paragraphs the requirements a proper inheritance mechanism should propose.

Although multiple inheritance is rarely proposed in OO languages, we encountered severeral places in our meta-modelling tasks where this was helpful[1]. But multiple inheritance can cause problems. We will propose restrictive constraints on its use that will make this kind of inheritance as “harmless” as simple inheritance.

Another quality we are expecting from this relationship is the possibility to derive several subtypes from one common supertype at the instance level. For instance, if `researcher` and `student` are both specializations of `person`, then “John” (a PhD researcher) is a researcher, a student and a person at the same time. This property is helpful when meta-models share common meta-classes in their definitions but with specific information each one in its turn. For instance, a same concept could denote a process in a dataflow diagram and a procedure in a Pascal program. Although its name is free in the dataflow (“check in” for instance), it must be a valid identifier in Pascal (“checkin”), be refined with parameters, a body, and so on. This pattern is well modelled with multiple specialization.

[1] The DB-MAIN’s repository and the MOF’s meta-model [Obj97b] are such examples.

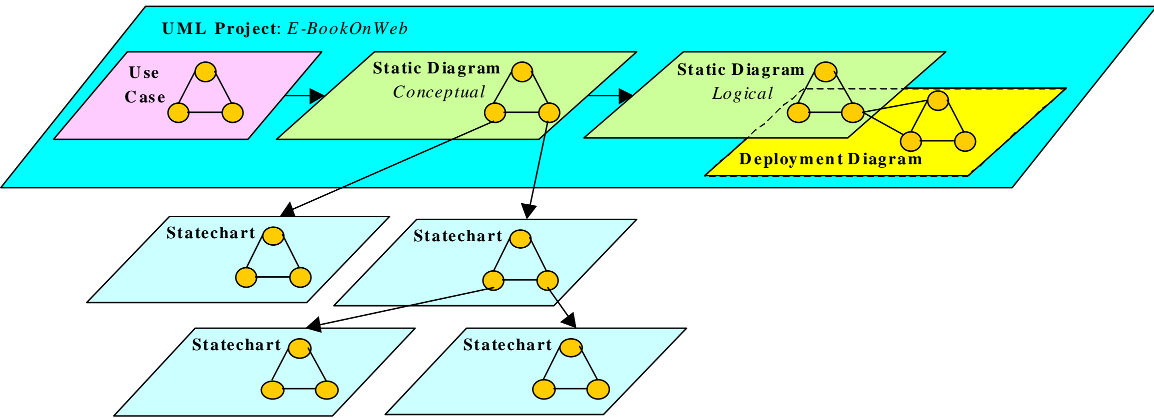


Figure 4.2: [**Software Engineering Projects and Hypergraphs**] This graph illustrates a software engineering project developed with the UML methodology. Static diagrams have been derived from a use case. Classes have been refined with statecharts and the conceptual schema has then been transformed to a logical schema where some classes have been deployed in the deployment diagram. This project shows that nodes can be shared between graphs (static and deployment diagrams), that nodes can be linked to graphs (a class is described by a statechart, i.e., a graph), and finally, that a node can itself denote a graph (a compound state in a statechart).

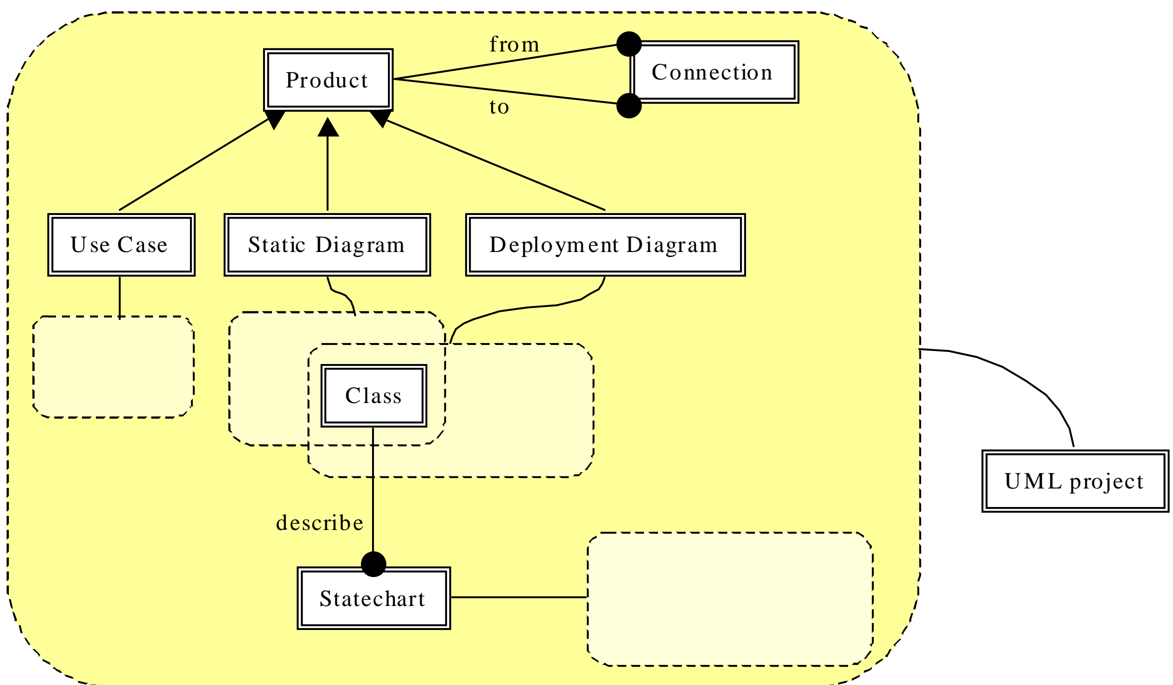


Figure 4.3: [**Meta-Modelling a Software Engineering Project**] This schema shows how the software engineering project depicted in Fig. 4.2 can be modelled with the concepts that will be presented and defined in this chapter. `Class` and `Connection` are meta-classes; `Use Case`, `Static Diagram`, `Deployment Diagram`, `Statechart` and `UML project` are meta-models. The dashed frames denote the meta-model definitions. Lines with bullets represent one-to-many relationships and arrows are inheritance links.

Dynamic inheritance is the possibility to change the type of an instance. This mechanism will often be used in our approach (along with the multiple specialization) to reuse concepts between methods in a posteriori decisions. For instance, "entity type E in the ERA schema S is modelled by class C in the static diagram D", where "class" inherits from "entity type".

The last property that interests us could be called "*can be a*". When a meta-class inherits from another meta-class, the instances of the subtype usually inherit the properties of their supertypes. But some exceptions exist. One of them is well illustrated with this well-known example: `birds` are `flying-animals`, but penguins are birds, and nevertheless, they do not fly! This has been observed in [Tou86]. The latter proposed to extend the semantics of the inheritance relationship in order to take into account those exceptions because "*Mandatory inheritance of properties is too inflexible for representing real-world knowledge*". He also proposes to exploit this facility (the exceptions) to solve dilemna in complex modelling problems (*cfr.* Fig. 4.4).

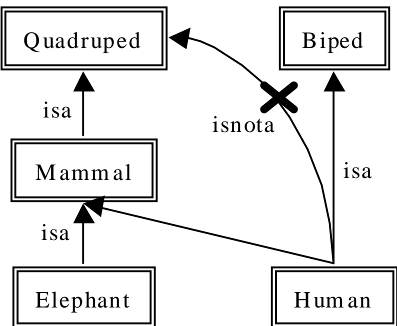


Figure 4.4: [**Inheritance with Exceptions**] This diagram is an extract of [Tou86]. This shows one way to represent mamal, elephant, and human types in NETL (a parallel semantic network language created by Scott Fahlman).

Although we think this example a little questionable, it is easy to find other examples in the realm of meta-modelling. For instance, this characteristic is convenient to integrate meta-models and their specifications a posteriori, to represent exceptions (*cfr.* the `Fork`, `Join`, `Branch` states in the diagram depicted in Fig. 4.28 on page 92), or to extend existing meta-models with new concepts that do not match the existing modelling decisions.

In order to propose all these properties, we decided to represent explicitly the instances of the meta-classes (even when they belong to a same inheritance graph) as well as the inheritance link between them. This makes it possible to edit the inheritance links at the specification level and to represent all the features that were presented so far (multiple inheritance, inheritance with exceptions, dynamic inheritance, multiple specializations, ...). Although this is not usual in programming languages, semantic networks sometimes use these conventions.

4.3 Concepts

4.3.1 Meta-Class

We call MetaClass the set of all the meta-classes used by the modelled CASE-tools. In ERA-based CASE tools, we can state that MetaClass $\supseteq$ {`entity-type`, `relation-type`, `attribute`,

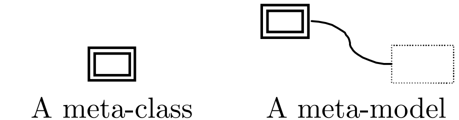


Figure 4.5: [**Graphical Representation of meta-classes and meta-models**]

`role`}.

Definition 1 (MetaClass – **Meta-class name**)
MetaClass *is a set and its elements are called meta-classes. We define the mapping* name : MetaClass $\mapsto$ String *that gives a name to each meta-class.*

Axiom 1 (Meta-class name is unique)
The name of a meta-class is unique.

$$\forall X, Y \in \mathsf{MetaClass} : X.\text{name} = Y.\text{name} \Rightarrow X = Y$$

Definition 2 (Virtual meta-class)
We define the mapping virtual : MetaClass $\mapsto \mathbb{B}$. *A meta-class is virtual when it depends on the existence of one or more subtypes. Its meaning will be clarified with axiom 27*[1] *that comes further.*

Graphical Conventions In the compact view, meta-classes will be represented by rectangles with a double frame. The meta-class name will sometimes be written inside the box. In the extended view, a meta-class will be represented like an entity-type *à la* DB-MAIN. More details on the graphical conventions of this view can be found in [Gro]. The reader can read the appendix B[2] to have an overview of those conventions. We will just give the mapping rules between the DB-MAIN's terminology and our concepts.

4.3.2 Meta-Model

We call MetaModel the set of all the meta-models that are being modelled. A meta-model structures the mental representation of one "CASE tool's aspect". An aspect is a coherent and elementary group of concepts proposed by a CASE tool to a software engineer for his visualizing/editing/computing/*etc* needs. Those components (i.e., the concepts that made up a CASE tool) are considered atomic even if their definition can be complex.

Definition 3 (MetaModel)
MetaModel *is a set and its elements are called meta-models.*

Axiom 2 (Meta-models are meta-classes)
*All meta-models are themselves meta-classes (*MetaModel $\subseteq$ MetaClass*).*

[1]See page 58.
[2]See page 235

Axiom 2 will allow us to describe the meta-models with the same expressiveness than for meta-classes (i.e., with meta-relations, meta-properties, ...).

Although people will often associate the notion of diagram with a meta-model, this is just one possible interpretation of the semantics of a meta-model but this is not the only one. A meta-model can model information that a particular diagram does not show (technical details/external references/multimedia data). A meta-model can also be a viewpoint of some information found in a repository. Hence, one deduces that a meta-model is somewhat complex[1] and that the relationship between a meta-model and its components can be relatively "strong" or "weak". To illustrate this relationship, just consider the relation between a class (in a school), a family and one child. Although every child has a family (biological rule) and must assist lessons in a class (legal rule), the link of composition is not identical: an official could suppress a class without having to "kill" each schoolboy. Same remarks can be observed in the computer science realm, a strong analogy exists between respectively the {class, family, child} and the {procedure, statement, dependency graph} concepts. A procedure removal will delete its statements although that the deletion of a dependency graph between statements will not necessarily suppress its statements. Now, a software engineer could wish to keep statements alive as long as a dependency graph uses them even if its parent procedure has disappeared. Many other kinds of composition rule can be found in other realms like: CAD/CAM, Artificial Intelligence, Semantic Networks, etc. A meta-CASE must not be aware of the domain it is modelling, so the modelling of the composition rule has to be as general as possible. Indeed, Magan *et al.* [MO97] and Winston *et al.* [WCH87] have identified 7 kinds of composition[2]. Moreover, other compositions such that the *inclusion relation* (e.g., birds are flying animals) and the *topological inclusion* (e.g., a bird is in a cage) were not listed. This large variety of interpretation of the composition rule suggests having a representation as generic as possible of this rule.

The composition relationship is modelled by the Mod function in the meta-meta-model.

Definition 4 (Mod **Mapping)**
We define a mapping $\mathrm{Mod} : \mathsf{MetaModel} \mapsto \mathsf{MetaClass}^{\star}$.

Definition 4 expresses that meta-models are composed of meta-classes. We will call this composition the definition of the meta-model. For instance, an elementary ERA meta-model can be defined as Mod(`Entity-Relationship Model`)={ `entity type`, `relation type`, `attribute`, `role`}.

Definition 5 (Aggreg**)**
The Aggreg *mapping defines the kind of aggregation the meta-model will represent. Three aggregation types are possible:* `weak`*,* `strong` *and* `delayed`.

$$\mathrm{Aggreg} : \mathsf{MetaModel} \mapsto \{\texttt{weak}, \texttt{strong}, \texttt{delayed}\}.$$

This function expresses how a meta-model gathers its components. Components of strong aggregation depend on the existence of their container, its deletion entails the deletion of its components. Weak aggregation means that a component exists as long as it exists a container

[1] "complex" means here: defined in terms of some components, composite.

[2] Composant/Object (wheel/car), Element/Collection (tree/forest), Portion/Mass (part/cake), Substance/Object (steel/bicycle), Step/Event (introduction/presentation), Stage/Activity (payment/purchase), Place/Space (Namur/Wallonie).

(i.e., a specification) that contains it. The delayed aggregation is either a strong or a weak aggregation but whose the kind is determined later (when the container is deleted). In other words, this function expresses how a meta-model gathers its components and answers this question: *Is the meta-model acting as a "class(school)" or as a "family" ?* These aggregation types will be detailed in section 4.3.10[1].

Axiom 3
There is at least one "special" element in $\mathsf{MetaModel}$ *named* Ω *(*$\Omega \in \mathsf{MetaModel}$*).*

This element is the meta-model composed of all the meta-classes that do not belong to another meta-model. The Ω meta-model will act as a root and therefore will be the first meta-model a software engineer will have to face. Meta-models that would not belong to the Ω's definition will be considered as "sub" meta-models that depend on other meta-models. We could illustrate this paragraph with this example:

- $\mathrm{Mod}(\Omega) = \{\texttt{Merise}, \texttt{UML}\}$
- $\mathrm{Mod}(\texttt{Merise}) = \{\texttt{Entity-Relationship Model}, \dots\}$
- $\mathrm{Mod}(\texttt{UML}) = \{\texttt{Use Case}, \dots\}$

A software engineer can edit a "Use Case" diagram only when he is using a "UML" diagram, and he can use a "UML" diagram, once he has opened a Ω-specification. The latter is often called "project" in CASE tools (DB-MAIN, MetaEdit+). Let us remark that contrary to what is suggested by this example, the definition of a meta-model into its components is not necessarily a tree, indeed, this is more often a graph.

Axiom 4
The Ω *meta-model does not participate in the definition/composition of other meta-models.*

$$\mathrm{Mod}^{-1}(\Omega) = \emptyset$$

From axiom 4, we infer that $\Omega \notin \mathrm{Mod}(\Omega)$. Nevertheless, no axiom prevents another meta-model to belong to its own definition.

The following axiom ensures that any meta-model is either Ω or is in the definition of a meta-model that is in the definition of a meta-model that ... that is in the definition of Ω. This ensures that all the meta-models can be retrieved from the Ω definition.

Definition 6 ($\ll$**)**
$\ll \in \mathsf{MetaClass} \times \mathsf{MetaModel}$ *is an infix relation and is defined as:*

$$X \ll Y \Leftrightarrow X \in \mathrm{Mod}(Y) \ \vee \ \exists Z \in \mathsf{MetaModel}: \ X \in \mathrm{Mod}(Z) \wedge Z \ll Y$$

Axiom 5 (Ω **is a root)**

$$\forall X \in \mathsf{MetaModel}: \ (X = \Omega) \otimes (X \ll \Omega)$$

Axiom 5 ensures that the Ω meta-model is the root of all the other meta-models. The following diagram shows a hypothetical scenario that does not respect it.

[1]See page 51.

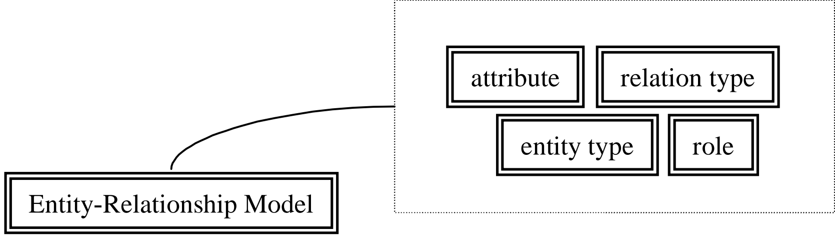
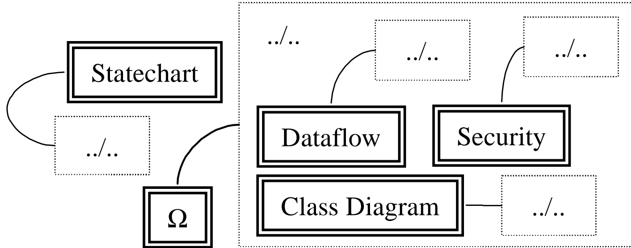


Figure 4.6: [**Meta-Model Drawing (Compact View 1)**] A possible "Entity-Relationship Model" meta-model and its definition



Theorem 1
All the meta-classes distinct from Ω *belong to at least one meta-model.*

$$\forall X \in \mathsf{MetaClass} : X \neq \Omega \Rightarrow \mathrm{Mod}^{-1}(X) \neq \emptyset$$

Proof Let us suppose that $\exists X \in \mathsf{MetaClass}$ such that $X \neq \Omega$ and $\mathrm{Mod}^{-1}(X) = \emptyset$. This entails $\neg(X \ll \Omega)$, which is contrary to axiom 5. □

Although this theorem seems trivial, this means that meta-classes have no sense outside a meta-model. Although meta-classes can belong to the definition of Ω, we discourage this practice and encourage the method engineers to define the meta-class inside the most accurate meta-model. For instance, in the UML method, the right place of the `state` meta-class is neither in Ω nor in the `UML` meta-model but should be in the `Statechart` meta-model.

Graphical Conventions In the compact view, a meta-model will be drawn like a meta-class[1] with a dashed frame attached with a "rope" between them (*cfr.* Fig. 4.5). The dashed frame will contain the meta-classes which define the meta-model. An empty frame may denote an incomplete frame as well as an empty frame ($\mathrm{Mod}(m) = \emptyset$) upon the context. A "../.." inside a frame stresses the incomplete nature of the frame. Figures 4.6 and 4.7 depict some possible cases.

In the extended view, a meta-model will be displayed like a meta-class with a bold frame. Moreover, the definition of a meta-model will be shown inside a cylinder. The name on the top of the cylinder is the name of the meta-model, and the contents of the cylinder denote the meta-model definition. An example of these graphical conventions is depicted in Fig. 4.8.

Compact and extended views may describe several meta-models with their meta-classes as well as other meta-classes that would not belong to the definitions of the meta-models depicted in the diagram. They must be considered as partial views of the Ω meta-model.

[1] A meta-model is a meta-class.

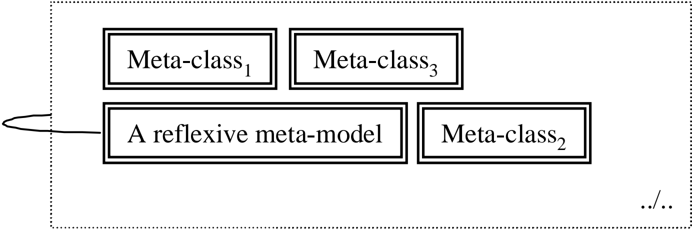


Figure 4.7: [**Meta-Model Drawing (Compact View 2)**] A reflexive meta-model

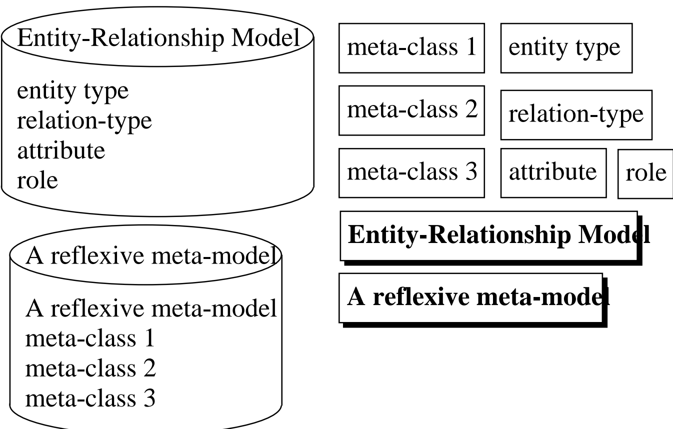


Figure 4.8: [**Meta-Model Drawing (Extended View)**] This extended view covers the views depicted in figures 4.6 and 4.7. Cylinders depict the definition of both meta-models. Meta-models are drawn like meta-classes with a bold frame around

4.3.3 Aggregation's Characteristics

Magnan *et al.* [MO97] presents four criteria to describe aggregation relationships. They can be *1)* exclusive or sharing, *2)* dependent or independent, *3)* predominant, and *4)* have a cardinality. Each property is summarized below and will be compared with our proposition of aggregation.

1. **Exclusivity or Sharing :** An aggregation is exclusive when their compositions are disjoint. Example: two bicycles cannot share the same wheel.

   Comparison: Sharing is possible and the exclusivity can be obtained with the CanCreate[1] method. This method acts as a predicate which makes it possible to control the creation of a class. The method engineer can overload it to enforce a property (such as the exclusivity for instance). Indeed, he can check that the class does belong to no more than one specification when it will be created. In the same way, the ToDelete could check that specifications

2. **Dependency or Independency :** Can an element survive outside its composite? Example: a capital disappears once its country no longer exists.

   Comparison: These properties are respectively equivalent to our strong and weak aggregation.

3. **Predominance** Can a composite live without one of its component? Example: a human without a heart is not a human.

   Comparison: This property can be achieved easily with the ToDelete[2] method. This method acts as a trigger and computes a list of classes to delete if the current class is deleted. Hence, a class can entail the deletion of the specifications (one or more) that contain it.

4. **Cardinality :** How much components does a composite need? Example: a bicycle needs two wheels.

   Comparison: This property cannot been achieved with our meta-meta-model.

The way we model the composition rule meets most of these requirements.

4.3.4 Meta-Class Inheritance

Meta-CASEs have to encompass a majority of OO paradigms in order to be able to easily represent meta-models. Hence, the inheritance relationship between meta-classes is essential. However, we can point out from [Bra83] that "*[...] there are almost as many meanings for the IS-A link as there are knowledge-representation systems [...]*". Brachman identified ten distinct meanings and since that time (1983) other ones have surfaced!

We propose a lax approach that is both simple and generic: an IS-A relationship between two meta-classes just suggests the possibility to establish some "connection" between the instances of both meta-classes so that the *subtype* class inherits the properties of the *supertype* class.

[1] *Cfr.* section 4.3.6 page 47.
[2] *Cfr.* section 4.3.6 page 48.

Definition 7 (Inheritance)
We define a mapping from each meta-class to the set of its supertypes.

$$\text{super} : \mathsf{MetaClass} \mapsto \mathsf{MetaClass}^\star$$

For convenience sake, we define the transitive closure of the super *relation:*

$$\text{super}^\star : \mathsf{MetaClass} \mapsto \mathsf{MetaClass}^\star$$

where

$$\text{super}^\star(X) = \text{super}(X) \cup \bigcup_{Y \in \text{super}(X)} \text{super}^\star(Y)$$

Definition 8 (isa and isa* operators)
We define the infix ***isa*** *and* ***isa**** *operators as:*

$$X \textit{\textbf{isa}}\, Y \equiv X, Y \in \mathsf{MetaClass} \wedge X \in \text{super}(Y)$$

$$X \textit{\textbf{isa}}^\star\, Y \equiv X, Y \in \mathsf{MetaClass} \wedge X \in \text{super}^\star(Y)$$

Notation 1 (ρ)
The ρ relation is defined as

$$\rho(A, B) \;\equiv\; A \textit{\textbf{ isa }} B \;\vee\; B \textit{\textbf{ isa }} A$$

The **isa** and **isa*** relationships will have to obey to some axioms that will confer a minimal semantics to the inheritance concept.

Axiom 6
A meta-class cannot be a direct or indirect supertype of itself.

$$\forall X \in \mathsf{MetaClass} : X \notin \text{super}^\star(X)$$

Axiom 7
If a meta-class has a supertype which is a meta-model, then this meta-class is a meta-model.

$$\forall X \in \mathsf{MetaModel}, \forall Y \in \mathsf{MetaClass} : Y \textit{\textbf{ isa }} X \Rightarrow Y \in \mathsf{MetaModel}$$

Axiom 8
A supertype belongs to the same meta-model as its subtypes.

$$\forall X \in \mathsf{MetaClass} : \forall M \in \text{Mod}^{-1}(X) : \text{super}^\star(X) \subseteq \text{Mod}(M)$$

Axiom 8 ensures that the supertypes have a meaning in the meta-models its subtypes belong to. Because the supertype's characteristics will be inherited, it would be incoherent to omit a supertype in a meta-model definition although its characteristics would be visible. One will find the same axiom at the “specification” level.

The possibility to define an inheritance link between two meta-models brings us to investigate the semantics of the Mod( ) function. Three strategies are possible:

1. There is no link between the Mod( ) definition and the inheritance principle.

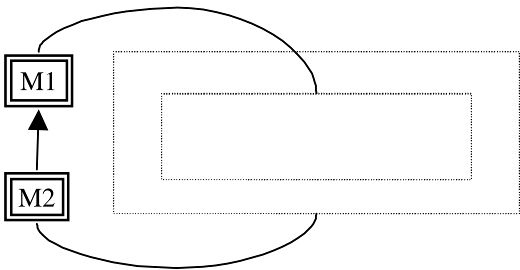


Figure 4.9: [**Inheritance and Meta-Model Definition**] Subtype meta-models have wider definitions that their supertypes. The definition of the M_1 meta-model is a subset of the definition of the M_2 meta-model

2. We adopt the "set" semantics. A specification s that would be an instance of a meta-model M which is at its turn a subtype of another meta-model M' would also be an instance of M'. Hence, the meta-model's definition should necessarily a subset of its supertype's definition. In other words, a meta-model that would be a subtype of another meta-model would be *less expressive* since its definition is necessarily a subset ($\leq$) of the supertype's definition.

3. The "reuse" semantics, i.e., the supertype acts as a pattern that can be extended with new characteristics (meta-properties, meta-relations, ...).

The pragmatism of the last option incited us to choose it. Moreover, this choice is compliant with our original definition of the inheritance link between the meta-classes. The next axiom formalizes these ideas.

Axiom 9

$$\forall M_1, M_2 \in \mathsf{MetaModel} : \; M_1 \, \textit{\textbf{isa}} \, M_2 \Rightarrow \mathrm{Mod}(M_2) \subseteq \mathrm{Mod}(M_1)$$

Graphical Conventions In the compact view, the **isa** relationship is represented by an arrow from the subtype to the supertype. In the extended view, the **isa** relationship is represented with the DB-MAIN's conventions: a triangle with the apex towards the supertype. A virtual meta-class will be represented with the "total" flag in the inheritance properties.

4.3.5 Meta-Property

Meta-properties are identifiers (i.e., names) that attach information to meta-classes. When the method engineer defines a meta-property, he creates a function that will associate some information to the instances of the meta-class. This information is constrained by some axioms defined below. Briefly, this information is typed (integer, string, boolean, video, ...) and can be either a single data (monovalued), or a sequence of data (multivalued).

We note Θ the information types we can associate with properties.

Definition 9 (Elementary types)
$\Theta = \{\mathbb{Z}, \mathbb{R}, \mathbb{B}, \mathsf{Char}, \mathsf{String}, \mathsf{Document}, \mathsf{Video}, \mathsf{Sound}, \mathsf{Picture}\}$

Definition 10 (Value Universe – ValUniv)
We define the value universe (noted ValUniv*) as the set of all the possible values of a property. The value universe is defined on the basis of* Θ *and comprises elementary values as well as sequences of values.*

$$\mathsf{ValUniv} = \bigcup_{T \in \Theta} (T \cup T^{\langle\star\rangle})$$

Hence, an element of ValUniv *is either an element of* T *or* $T^{\langle\star\rangle}$ *for some* $T \in \Theta$*. This element is thus either an elementary value or a sequence of elementary values of the same type.*

Definition 11 (MetaProperty– name – type – multi – prop)
We call MetaProperty *the set of meta-properties. We define* $\mathrm{Prop} : \mathsf{MetaClass} \mapsto \mathsf{MetaProperty}^{[\star]}$ *a mapping that associates properties (or none) to a meta-class. We define several functions that will characterize the elements of* MetaProperty*:*

$\mathrm{name} : \mathsf{MetaProperty} \mapsto \mathsf{String}$ *is the identifier associated with the meta-property.*

$\mathrm{type} : \mathsf{MetaProperty} \mapsto \Theta$ *defines the type of the meta-property. Only primitive types are allowed.*

$\mathrm{multi} : \mathsf{MetaProperty} \mapsto \mathbb{B}$ *defines if the meta-property is mono or multivalued.*

Axiom 10 (One meta-class per meta-property)
A meta-property describes only one meta-class.

$$\forall P \in \mathsf{MetaProperty} : \ \sharp\mathrm{Prop}^{-1}(P) = 1$$

Axiom 11 (Local name uniqueness)
The meta-properties of a meta-class have distinct names.

$$\forall X \in \mathsf{MetaClass} : \forall p_1, p_2 \in \mathrm{Prop}(X) : p_1 \neq p_2 \Rightarrow p_1.\mathrm{name} \neq p_2.\mathrm{name}$$

Axiom 12 (Global name uniqueness)
The meta-properties that describe meta-classes of a same inheritance graph, must have distinct names. $\forall X \in \mathsf{MetaClass} : \forall Y \in \mathrm{super}^\star(X) : p_X \in \mathrm{Prop}(X) \wedge p_Y \in \mathrm{Prop}(Y) \Rightarrow p_X.\mathrm{name} \neq p_Y.\mathrm{name}$

Axioms 11 and 12 ensures that a meta-property is identified by its name in the whole inheritance graph whatever its type and its multiplicity. This property will also be used at the "specification" level to avoid well-known problems[1] that come with multiple inheritance.

The Ω meta-model has one meta-property defined as: $P.\mathrm{name} =$ "name", $P.\mathrm{type} = \mathsf{String}$, $P.\mathrm{multi} =$ **false**.

Graphical Conventions Meta-properties are represented by attributes in the extended view and have no counterparts in the compact view.

[1] Conflicting names.

4.3.6 Meta-Classes and Methods

Methods denote the user-defined behaviour of meta-classes. That is, how a meta-class can interact with its environment, the software engineer, ... We will give here a simplified description of the methods. Indeed, we will content ourself with considering a method as an address to some instructions to execute. The methods will be defined with the Voyager 2^+ programming language (*cfr.* chapter 7). For the same reason, the methods will be defined with a high-level pseudo-language that does not always match Voyager 2^+ statements. Nevertheless, the semantics of this “high level language”is intuitive enough to avoid extensive definitions that will be postponed to the chapter dedicated to Voyager 2^+.

Definition 12 (Methods – Meth **)**
We define Method *as a set of addresses where begins a list of statements that it is possible to execute. A name is associated with each method. This mapping is defined formally as* name : $\mathsf{Method} \mapsto \mathsf{String}$. *We define a mapping* Meth : $\mathsf{MetaClass} \mapsto \mathsf{Method}^\star$.

Axiom 13 (Method uniqueness)
Methods associated with a meta-class have distinct names.

$$\forall C \in \mathsf{MetaClass},\ \forall M_1, M_2 \in \mathrm{Meth}(C) :\ M_1.\mathrm{name} = M_2.\mathrm{name}\ \Rightarrow\ M_1 = M_2$$

Let us remark that meta-classes can have methods with the same name even if they belong to the same inheritance graph.

Notation 2 (C::M())
If C is a meta-class and if M is a method of Meth(C)*, then we note C::M this method. A call will be noted C::$M(\alpha_1, \ldots, \alpha_n)$.*

Methods can be procedures or functions, they can accept arguments and return a value. Some methods are predefined with a default behaviour. We give here the list of these methods:

function X::`CanCreate(list:`l`)` $\rightarrow \mathbb{B}$ This method is called every time an instance of X will be created. The creation will fail if the method returns **false**. The list l denotes the future state of this class (its properties, relations, supertypes and specifications). By default, this function returns **true**.

Example

> In ERA diagrams, attributes and roles can have minimum and maximum cardinalities. Although the minimum should inferior to the maximum, it is impossible to express this constraint in the meta-model. Hence, the method engineer can check it with this predicate.

---

function X::`Constructor(list:`l`)` $\rightarrow X$ This method creates one instance of the meta-class X. The definition of this method is builtin and cannot be modified/edited.

procedure X::`OnCreate(`X`:`x`)` This procedure is executed after a class has been created. Its execution can be delayed and other classes can have been created meanwhile. This restriction ensures that method engineers will not exploit this facility to define intricate triggers. Clearly, we will see in section 7.9.2 that the creation of a class is not atomic. Hence, without this restriction, the method engineer could imagine to influence the creation process with side effects.

function X::CanDelete($X : x$) $\rightarrow \mathbb{B}$ Tells if one can delete the class (default=**true**).

procedure X::Delete($X : x$) This procedure is called each time a class is deleted. Nevertheless, the method engineer cannot redefine it. (*Cfr.* section 5.2[1]).

function X::ToDelete($X : x$) $\rightarrow$ Class* A class can ask to delete other classes when it will be deleted (default=$\emptyset$). When this method is being executed, the repository is frozen, that is, it is impossible to edit the repository. Such attempts would fail.

> **Remark** Contrary to the creation process, there is no `OnDelete` method. There are two possible implementations; which are both unsatisfying for our purpose:
>
> 1. This method is called everytime a class is deleted, and it can access the state of the deleted class. Then the state must be preserved. Nevertheless, this state can be very complex (its properties, its relations, its supertypes, its specifications, ...). Moreover, contrary to the ToDelete method, this method should access the repository and modify it to be useful. It could then delete other classes, and call other ToDelete and OnDelete methods. We observe that the behaviour and the semantics of such triggers are very complicated both to implement and to understand – from the method engineer's point of view.
> 2. This method can not access the state of the deleted class. But then, this method is not very useful.

Other methods are defined in section 7.9.4[2].

Remark We have intentionally simplified the Voyager 2^+ language. In some words, Voyager 2^+ allows the method engineer to define packages. A package is a collection of functions, procedures, methods and global variables that are gathered together in a file that describes the methods of a meta-class. In this chapter, we will make no distinction between procedures, functions and methods.

4.3.7 Meta-Relation

Meta-relations express relationships between concepts/meta-classes. The method engineer can define such relationships between meta-classes and, hence, expect that the meta-CASE will support them (to display, to manage, to navigate, ...). If the ERA notation accustomed us to very rich types of relationships, we prefer here to limit their semantics to binary one-to-many relationships. This limitation is not disturbing since rich relationships can generally be transformed into equivalent constructs with only one-to-many relationships [Hai91]. Other researches [Fin94b] have chosen a more elaborated semantics. Nevertheless, it is generally possible to transform these constructs in simpler ones with the same expressivity.

Definition 13 (Meta-relation)
We define MetaRelation *as the set of binary, one-to-many relationships between meta-classes and we define a mapping that associates a name with each meta-relation.*

$$\text{name} : \mathsf{MetaRelation} \mapsto \mathsf{String}$$

[1]See page 97.
[2]See page 191

Axiom 14 (Name uniqueness)
The name of a meta-relation identifies it.

$$\forall R_1, R_2 \in \mathsf{MetaRelation} : R_1.\text{name} = R_2.\text{name} \Rightarrow R_1 = R_2$$

Definition 14 (Owner – Member – Mandatory)
We define two mappings that will explicit which meta-class will respectively play the owner/member role in a relationship.

$$\text{owner} : \mathsf{MetaRelation} \mapsto \mathsf{MetaClass}$$
$$\text{member} : \mathsf{MetaRelation} \mapsto \mathsf{MetaClass}$$

If $R \in \mathsf{MetaRelation}$, then the $R.\text{owner}$ meta-class will play the owner role. Hence, we can associate a sequence[1] *of instances of the $R.\text{member}$ meta-class to each instance of $R.\text{owner}$. This sentence will be formally defined further. We also define another mapping that will explicit if an instance of $R.\text{member}$ can exist without playing a role in a relation.*

$$\text{mandatory} : \mathsf{MetaRelation} \mapsto \mathbb{B}$$

This mapping will have a formal definition once the notion of class will be defined.

Graphical Conventions Meta-relations are represented as relationships in the extended view. In the compact view, they are drawn as edges (lines) between both meta-classes; a bullet is placed on the member side. The line is sometimes labelled with the name of the meta-relation.

4.3.8 The Ω's Axioms

The Ω meta-model plays in our approach a special role and the method engineer is not allowed to modify its semantics. For this reason, we define the next axioms:

Axiom 15 (Ω has no meta-relations)
The Ω meta-model cannot participate in meta-relations.

$$\forall R \in \mathsf{MetaRelation} : R.\text{owner} \neq \Omega \wedge R.\text{member} \neq \Omega$$

Axiom 16 (Ω's meta-properties)
The Ω meta-model has one meta-property named "name".

$$\exists P \in \text{Prop}(\Omega) : \; P.\text{name} = \text{"}\textit{name}\text{"} \wedge P.\text{type} = \mathsf{String} \wedge P.\text{multi} = \textbf{\textit{false}}$$

Axiom 17 (Ω has no inheritance)
The Ω meta-model cannot have subtypes of supertypes.

$$\text{super}(\Omega) = [\,]$$

$$\forall C \in \mathsf{MetaClass} : \Omega \notin \text{super}(C)$$

[1] The members are ordered. From an operational point of view, hiding the order with a set semantics would require addition efforts since this order is naturally present in most programming languages and DBMS. Moreover, the order is often useful.

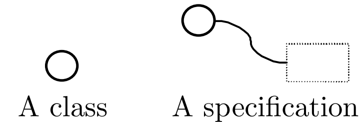


Figure 4.10: **[Drawings of Classes and Specifications]**

4.3.9 Class

The Class set represents the extension of the meta-model described in terms of the concepts defined above. The elements of this set are called *classes*. Each class is typed and belongs to the extension of exactly one meta-class.

Definition 15 (Type of – Instance of – Γ)
We define one mapping from Class *to* MetaClass *that represents the type of every class. In the same way, its inverse will represent the extension of the meta-classes.*

$$\Gamma : \mathsf{Class} \mapsto \mathsf{MetaClass}$$

We suppose here that the intersection of the Class set with the MetaClass set is empty. Some approaches generalize the "type of" relation in order to get reflective meta-meta-models, i.e., a meta-class is itself a class and can thus be manipulated like a class. These approaches add the $\mathsf{MetaClass} \subseteq \mathsf{Class}$ proposition in their postulates. This generalization leads to non-trivial problems. First, since every class x is usually typed and has thus a meta-class y, that is itself a class that has a meta-class z and so on ..., computer scientists have to face this interesting challenge. One approach supports infinite graphs where only the minimal part is built [WF86]. Forman *et al.* [FD98] avoids infinite graphs by introducing a circular link inside the "type-of" graph. In avoiding this reflective semantics, we wish to keep our meta-meta-model as close as possible to the "average programmer" knowledge level. However, the reflexive extension of our meta-meta-model will be discussed in chapter 5.

Definition 16 (cid)
We define a mapping cid : $\mathsf{Class} \mapsto \mathbb{N}$. *Function* cid *is an isomorphism and we have:*

$$\forall x, y \in \mathsf{Class} : \ \mathrm{cid}(x) = \mathrm{cid}(y) \Leftrightarrow x = y$$

cid *stands for "**c**lass **id**entifier".*

Graphical Conventions In the compact views, a class is drawn as a simple circle (*cfr.* Fig. 4.10). Classes are not drawn in the extended view. Let us remark that the compact view can represent both meta-classes and classes. When the author wants to represent the link [1] between a class and its meta-class, a dashed arrow can be drawn from the class to its meta-class. When a meta-class has too many classes in its extension, the author can draw a dashed arrow between the meta-class and a form[2] encompassing the instances. An example is illustrated in Fig. 4.11.

[1] i.e., the *instance-of* relationship.
[2] This form can be a square, a rectangle, ...

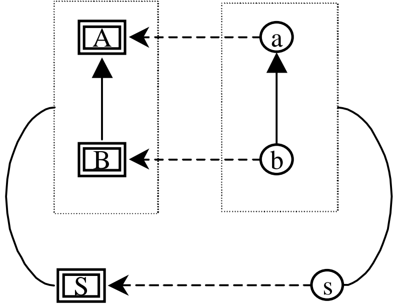


Figure 4.11: [**Classes and Specifications in Compact Views**] This diagram illustrates two classes a and b. Dashed arrows endow the $\Gamma(a) = A$, $\Gamma(b) = B$ and $\Gamma(s) = S$ propositions. Plain arrows denote inheritance between meta-classes and their instances. With respect to our conventions, $a, b \in \mathsf{Class}$, $s \in \mathsf{Specification}$, $A, B \in \mathsf{MetaClass}$ and $S \in \mathsf{MetaModel}$. Moreover, $A, B \in \mathrm{Mod}(S)$ and $a, b \in \mathrm{Def}(s)$

4.3.10 Specification

Specification is a subset of Class and comprises classes denoting specifications. A specification is the materialization of a meta-model according to all the constraints the method engineer has defined. A specification is edited by a software engineer and is composed of (not necessarily) interconnected concepts having some sense for him. He will use specifications like sentences to build complex assertions from simple concepts like words or classes. Since specifications are themselves classes, a specification can talk about other specifications, even indeed itself.

Definition 17 (Specification)
Specification *is a set of classes which denotes specifications. A specification is a grouping of classes that has some meaning for the software engineer.*

$$\mathsf{Specification} \subseteq \mathsf{Class}$$

A specification can be an ERA schema, a statechart, an organisational diagram, a C++ program, the history of a method, a call-graph, a dependency graph, but also an excerpt[1] of these examples.

Axiom 18
The type of a specification is necessarily a meta-model and the instances of a meta-model are necessarily specifications.

$$\forall x \in \mathsf{Specification} : \Gamma(x) \in \mathsf{MetaModel}$$

$$\forall M \in \mathsf{MetaModel} : \Gamma^{-1}(M) \subseteq \mathsf{Specification}$$

Definition 18 (Def**)**
A specification is defined in terms of classes that compose it. We name Def *the mapping that associates classes to the specifications in which they participate.*

$$\mathrm{Def} : \mathsf{Specification} \mapsto \mathsf{Class}^{\star}$$

[1] We mean here a view like in the relational theory, and not a copy of a part of another specification.

Axiom 19 (Well-formed specifications)
Each instance occurring in the definition of a specification must have a type compatible with the definition of the corresponding meta-model.

$$\forall s \in \mathsf{Specification} : \forall x \in \mathrm{Def}(s) : \Gamma(x) \in \mathrm{Mod}(\Gamma(s))$$

This axiom may seem like a gratuitous constraint. Why such a constraint when it is still possible to specify the definition of a meta-model as the Class set? Definition 4[1] and axiom 19 must be considered as a guide amongst a plethora of concepts of which only a small part is pertinent. For instance, the ERA vocabulary is useless while defining a Statechart diagram and reciprocally.

Let us remark that neither this axiom nor the following axioms will prevent software engineers to define "connections" like ISA-link or relations through the specifications borders.

Definition 5[2] defines the behavioural rules of the specifications components (their classes). A specification denotes an aggregation of concepts and is characterized by three possible modes depending on the meta-model it belongs to. We will examine these three modes below:

`strong`: When a specification is described as `strong`, its destruction entails the destruction of all its components.

`weak`: When a specification is described as `weak`, its destruction is minimal. Its components are destroyed if and only if they will transgress one or several repository's axioms. The main property this kind of deletion will infringe is explained in theorem 3[3]. This "violation" just means that the class has become useless.

`delayed`: An aggregation is delayed if the method engineer does not wish to specify the aggregation type. The responsibility to define this mode is left to the software engineer. As it was explained for the first two modes, the aggregation type is only pertinent at the destruction time. A "delayed" aggregation defers the destruction mode at this moment. Hence, this mode is "fictitious" and only the first two modes must be taken into account from a semantics or mathematical point of view. Nevertheless, software engineers will sometimes find this capability useful.

Figure 4.12 shows the different strategies presented so far.

Axiom 20
There is a special element in Specification *named ω which is the only class in the extension of the Ω meta-model.*

$$\omega \in \mathsf{Specification} \wedge \Gamma^{-1}(\Omega) = \{\omega\}$$

From axiom 4, we can assert the following theorem:

Theorem 2
ω does not participate in the definition of other specifications.

$$\forall s \in \mathsf{Specification} : \omega \notin \mathrm{Def}(s)$$

[1]See page 39.
[2]See page 39.
[3]See page 53.

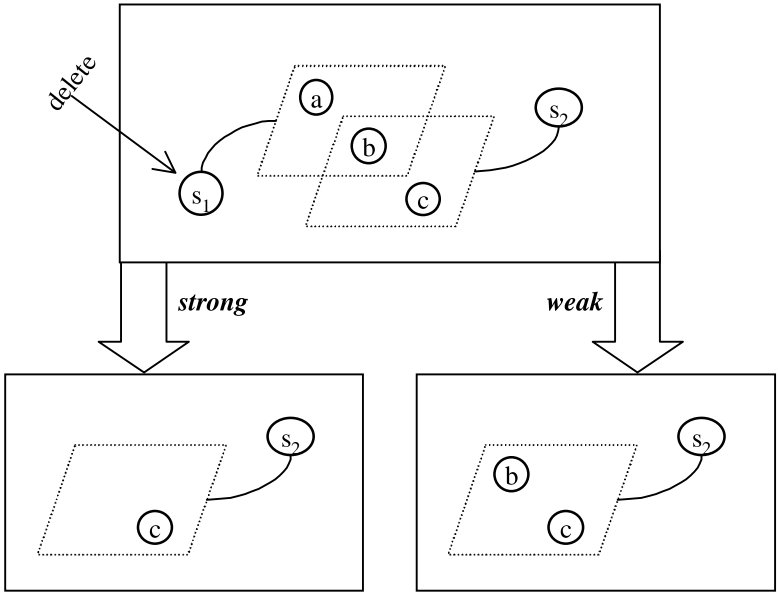


Figure 4.12: [**Weak and Strong Meta-Models**] This diagram shows on one example the effect of deleting a specification (s_1) depending on its aggregation type. When the aggregation is weak, only the transgressing classes are removed. In this example: a is removed because it does not belong anymore to a specification, this class violates the theorem 3, but b is preserved since it still belongs to s_2. When the aggregation is strong, all the components are deleted whatever is their types (a meta-class or a meta-model). In this example, meta-classes a and b are deleted although b participates in the definition of s_2

Proof If $\exists s \in \mathsf{Specification} : \omega \in \mathrm{Def}(s)$, then $\Omega \in \mathrm{Mod}(\Gamma(s))$. But from axiom 4, $\mathrm{Mod}^{-1}(\Omega) = \emptyset$. □

And the "root" function of the ω specification will be reinforced by the following axiom.

Definition 19 ($\lll$)
$\lll \subseteq \mathsf{Class} \times \mathsf{Specification}$ *is an infix relation defined as*

$$\forall x \in \mathsf{Class}, y \in \mathsf{Specification} : x \lll y \Leftrightarrow x \in \mathrm{Def}(y) \\ \vee \ \exists z \in \mathsf{Specification} : x \in \mathrm{Def}(z) \wedge z \lll y$$

Axiom 21 (**ω is a root**)

$$\forall x \in \mathsf{Specification} : (x = \omega) \ \otimes \ (x \lll \omega)$$

Theorem 3
Every class (distinct from ω) must belong to at least one specification definition.

$$\forall x \in \mathsf{Class} : x \neq \omega \Rightarrow \exists s \in \mathsf{Specification} : x \in \mathrm{Def}(s)$$

Proof The proof ensues from axiom 21 □

Specifications have thus a crucial role in the repository since all the classes will be structured by them.

Graphical Conventions In the compact view, a specification is drawn as a circle with a dashed box attached with a "rope". The dashed box denotes the definition of the specification and recommendations[1] about the meta-models are also applicable. Specifications are not drawn in the extended views.

4.3.11 Class Inheritance

A class that inherits from another class can borrow its characterics (roles, properties, ...). Although we could theoretically define inheritance links between arbitrary classes, the presence/absence of inheritance link between the respective meta-classes will restrict this mechanism to only pertinent classes. This principle is explained in this section.

Definition 20 (Class Inheritance – $\mathfrak{isa}$)
We define a mapping $\mathfrak{isa}$ defined as

$$\mathfrak{isa} : \mathsf{MetaClass} \times \mathsf{MetaClass} \mapsto (\mathsf{Class} \times \mathsf{Class})^\star$$

that associates to each couple $(A, B) \in \mathsf{MetaClass} \times \mathsf{MetaClass}$ a set of couples from $\mathsf{Class} \times \mathsf{Class}$. If $B \in \mathrm{super}(A)$, then $\mathfrak{isa}(A, B)$ will be the set of all the instances $\{(a, b) \dagger \Gamma(a) = A \wedge \Gamma(b) = B \wedge$ a is a subtype of b$\}$. This will be explained formally in the following axioms.

Notation 3 ($\mathfrak{isa}$ – $\mathfrak{isa}^\star$)
The $\mathfrak{isa}$ operator will be used in an infix way.

$$a\,\mathfrak{isa}\,b \equiv a, b \in \mathsf{Class} \wedge \Gamma(a)\,\textbf{\textit{isa}}\,\Gamma(b) \wedge (a, b) \in \mathfrak{isa}(\Gamma(a), \Gamma(b))$$

We define the infix $\mathfrak{isa}^\star$ operator as the transitive closure of the $\mathfrak{isa}$ operator.

$$a\,\mathfrak{isa}^\star\,b \equiv a\,\mathfrak{isa}\,b \vee \left(\exists c \in \mathsf{Class} : a\,\mathfrak{isa}\,c \wedge c\,\mathfrak{isa}^\star\,b\right)$$

Axiom 22 (Well-formed inheritance)
Two classes can inherit iff their respective meta-classes also inherit.

$$\forall X, Y \in \mathsf{MetaClass} : \neg(X\,\textbf{\textit{isa}}\,Y) \Rightarrow \mathfrak{isa}(X, Y) = \emptyset$$

$$\forall X, Y \in \mathsf{MetaClass} : (x, y) \in \mathfrak{isa}(X, Y) \Rightarrow \Gamma(x) = X \wedge \Gamma(y) = Y$$

The definition of the $\mathfrak{isa}$ relationship makes it possible to "draw" inheritance links between any classes whatever their meta-classes. Axiom 22 restricts its use to pertinent classes. Figure 4.13 shows two inheritance links that are prohibited by this axiom.

Notation 4 (ϱ)
The ϱ relation is defined as:

$$\varrho(x, y) \equiv (x\ \mathfrak{isa}\ y) \vee (y\ \mathfrak{isa}\ x)$$

Axiom 23 (Set inheritance)
A meta-class can not have more than one instance in the same inheritance graph. Let x and y be two classes, then if it exists $z_1, \ldots, z_n \in \mathsf{Class}$ $(n \geq 1)$ such that $\varrho(x, z_1)$ and $\varrho(z_n, y)$ and $\forall i,\ 1 \leq i \leq n-1 :\ \varrho(z_i, z_{i+1})$, hence we have $\Gamma(x) = \Gamma(y) \Rightarrow x = y$.

[1]See 4.3.2, page 41.

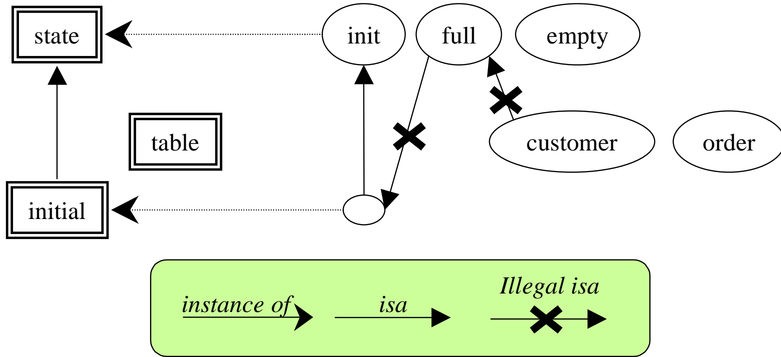
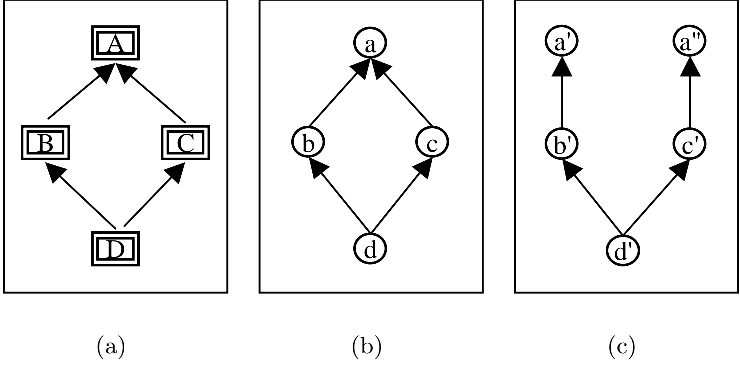


Figure 4.13: [**Incorrect Inheritance**] State `full` cannot inherit from an initial state, and table `customer` cannot inherit from a state (`full`).



(a) (b) (c)

Figure 4.14: [**Set inheritance**] Diagrams b and c are two possible instantiations of the type diagram a. Diagram b is compliant with axiom 23 although the c diagram is not. Class d' has two distinct supertypes (a' and a'') of the same type (A).

Axiom 23 explains, for instance, how the inheritance relation must behave in the "diamond structure". This problem is well-known when multiple inheritance is allowed. This axiom adopts a "set" approach of this problem that is: if a meta-class B inherits from another meta-class A, then we have $\Gamma^{-1}(B) \subseteq \Gamma^{-1}(A)$. The axiom reaches a similar effect to this semantics in our approach. This axiom also prohibits multiple supertypes to share one subtype if they have the same meta-class as type. These cases are illustrated in Fig. 4.14 and 4.15.

This axiom can seem very restrictive for a framework who pretends to be as generic as possible. However this will bring very convenient properties. For instance, the meta-properties will be attached to classes in a deterministic way, and type casting of subtypes to supertypes will be very simple.

Definition 21 (Type Casting – $(c \downarrow D)$)
*Let c be a class and C be its meta-class, if D is a supertype of C (C **isa** D), we note $(c \downarrow D)$ the class d such that $\Gamma(d) = D$ and c $\mathtt{isa}^\star$ d. By extension, we define $\forall c \in \mathsf{Class} : (c \downarrow \Gamma(c)) = c$.*

Axiom 23 ensures us that the $(\circ \downarrow \circ)$ operator is deterministic, i.e., there is at most one class responding to the criteria of the operator definition.

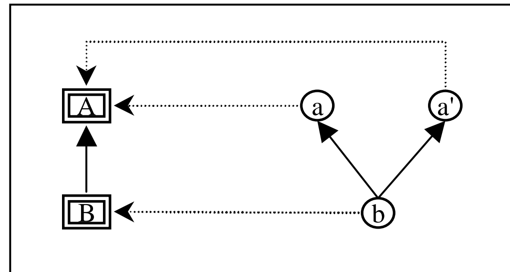


Figure 4.15: **[Set inheritance (*bis*)]** Two classes cannot share one subtype if they have the same type. Hence, this diagram violates axiom 23.

Axiom 24
A supertype class belongs to the specifications its subtypes belong to.

$$\forall x, y \in \mathsf{Class},\ \forall s \in \mathsf{Specification} : x \in \mathrm{Def}(s)\ \wedge\ y\,\mathfrak{isa}\,x \Rightarrow y \in \mathrm{Def}(s)$$

From axiom 8, we observe that axiom 24 is consistent with the types/meta-classes allowed in the composition of the respective meta-model of the s specifications.

Definition 22 (Leaf)
A class c is a leaf in a specification s iff c belongs to the definition of s and has no subtypes belonging to this specification.

$$\forall c \in \mathsf{Class}, \forall s \in \mathsf{Specification} : c \textit{ is a leaf in } s \Leftrightarrow c \in \mathrm{Def}(s) \wedge \left(\nexists x \in \mathrm{Def}(s) : x\,\mathfrak{isa}\,c\right)$$

By extension, when we do not specify any specification, we say that c is a leaf iff c has no subtypes at all.

Axiom 25 (Specification's definition cannot be overriden)
In the inheritance graph of a specification, there is only one definition which is pertinent. In other words, if a specification has a definition, then all the other specifications in its inheritance graph must have an empty definition (i.e., no definition). Moreover, only leaf specifications can have a definition.

$$\forall s, t \in \mathsf{Specification} :\ s\ \textbf{\textit{isa}}^{\star}\, t \Rightarrow \mathrm{Def}(t) = \emptyset$$

The previous axiom is certainly a debatable subject. Although its absence could lead us to judicious uses, it would also be possible to misunderstand its intuitive semantics. The inheritance between meta-models has been defined with the "reuse" capabilitiy in mind, i.e., a meta-model extends (and does not restrict) the semantics of its supertypes. The possibility to have several definitions in a same inheritance graph could make the software engineer (or even the method engineer) believe that the semantics is defined in terms of restriction. We can illustrate this argumentation with two examples (pros and cons):

- **cons** In a ERA methodology, it is common to derive views from database schemas. Views and database schemas share obviously a large portion of their semantics, and modelling a view as a sub-type of the ERA-schema's meta-model is certainly a good idea. Nevertheless, at the specification level, it would be dangerous to derive a view (e.g., *loan-service*) from an ERA schema (e.g., *bank-company*) as a subtype of this specification. Indeed, a view is a derived product (either manually or automatically) from a schema and it is certainly not a refinement or an extension of it.

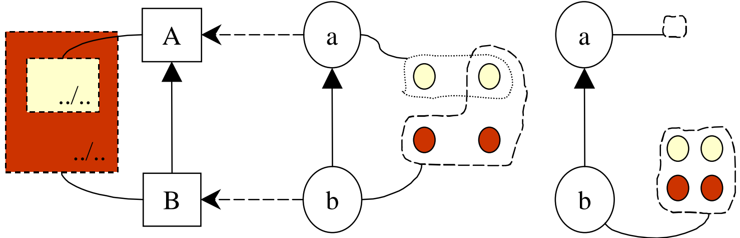


Figure 4.16: [**Specifications and Inherited Definitions**] In the left configuration, specification a owns its own definition. This infringes the axiom 25. The right graph depicts a correct configuration.

- **pros** Another example is the object-oriented extension of the ERA method. Such methods add OO-concepts to the ERA model such as: inheritance, methods, etc. Let us call this extension OO-ERA. Still again, it is judicious to define it as a subtype of the ERA meta-model. But contrary to the previous example, the best way to define an OO-ERA specification consists in specializing a ERA specification into a OO-ERA subtype. This new specification would naturally extend the previous schema with methods for instance. Unfortunately, axiom 25 prevents us to proceed so.

Nevertheless, the definition of a supertype specification can be automatically computed as follows: let s be a specification which owns one or more subtypes, then we define its *large definition* as

$$D = \bigcup_{C \in \mathrm{Mod}(\Gamma(s))} \Gamma^{-1}(C)$$

where D is the largest possible definition of s, and

$$S = \bigcup_{t \in \mathsf{Specification}:\, t\ \mathfrak{isa}^\star\ s} \mathrm{Def}(t)$$

where S denotes the union of all the subtypes definitions. Finally, the "*virtual*" definition of s can be obtained as:

$$\text{definition of } s = D \cap S$$

We observe that, according to this definition, s acts as a view on its subtypes but does not really own its definition. Hence, if we use the second example, visualizing the ERA specification that is a supertype of a OO-ERA specification entails a projection of the OO-ERA specification to the concepts of the ERA meta-model. Figure 4.16 shows two graphical representations of configurations which respectively infringe/respect this axiom.

Axiom 26 (Uniqueness of leaves)
To each class of a specification s corresponds one and only one subtype that is a leaf in s.

$$\forall s \in \mathsf{Specification}, \forall x \in \mathrm{Def}(s) : \ x \textit{ is a leaf in } s \ \vee \ \big(\exists y \in \mathrm{Def}(s) : y\ \mathfrak{isa}^\star\ x \wedge y \textit{ is a leaf in } s\big)$$

We call this subtype the leaf *of the class c in s.*

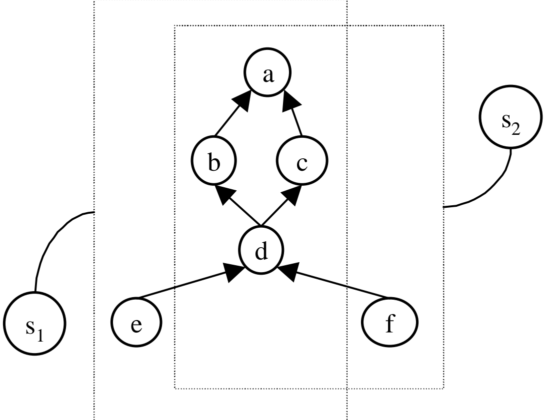


Figure 4.17: [**Inheritance: Several Leaves**] Class a has two leaves e and f. However these leaves belong to distinct specifications s_1 and s_2. If the class f moved to the definition of specification s_1, then this diagram would infringe the axiom 26

In the context of a specification, we will sometimes confuse the "type" of a class and the meta-class of its leaf in this specification. Axiom 26 also makes our approach closer to the conventional ISA semantics as defined in most programming languages. From the software engineer's point of view, his work will very often be limited to the edition of specifications. Hence, in this context, a class will have only one leaf/type that will be his object of concern. The possible subtypes that would be outside this specification would be out of his concerns. Specifications will then act as windows on complex inheritance graphs to only show a "normal inheritance" (i.e., as encountered in OODBMS and programming languages). But beyond the scope of specifications, classes may have various forms of inheritance trees that are compliant with the former axioms as depicted in the Fig. 4.17.

Axiom 27
The instances of a virtual meta-class must necessarily have at least one subtype.

$$\forall C \in \mathsf{MetaClass} : \; C.\text{virtual} = \boldsymbol{true} \; \Rightarrow \; \big(\forall c \in \mathsf{Class} : \; \Gamma(c) = C \; \Rightarrow \; (\exists y \in \mathsf{Class} : \; y \,\mathfrak{isa}\, x)\big)$$

Unlike many object-oriented languages or DBMS, a virtual meta-class may have no subtype. The restriction is stated on the instances only. Hence, a virtual meta-class with no subtype has an empty extension.

Graphical Conventions Inheritance between classes is represented in the compact view by a arrow from the subtype class to its supertype class. An example is illustrated in Fig. 4.11.

4.3.12 Property

Meta-properties define the signature of named functions that will map some classes to values. That is the spirit of the meta-properties. A *property* is the value a meta-property associate with a class. This section defines formally this mapping.

Definition 23 (ξ Valuation – $\xi_P(x)$)
The valuation *is a mapping from meta-properties to a function from classes to values.*

$$\xi : \mathsf{MetaProperty} \mapsto (\mathsf{Class} \to \mathsf{ValUniv})$$

To make the use of the ξ function easier, we will use the following notation $\xi_P(x) \equiv \big(\xi(P)\big)(x)$.

The valuation function is too lax regarding the semantics of meta-properties (*cfr.* section 4.3.5). So far, the valuation could return values that are not consonant with the type or the cardinality (multi) of the meta-property. The next axioms will give a restrictive formal scope to the ξ valuation.

Axiom 28 (Well-typed properties)
A class receives some values from the valuation with respect to some meta-property if and only if this meta-property belongs to the meta-class.

$$\forall P \in \mathsf{MetaProperty}, \forall c \in \mathsf{Class} : \ \xi_P(c)\exists \ \Leftrightarrow \ P \in \mathrm{Prop}(\Gamma(c))$$

Axiom 29 (Well-formed properties)
We will use a class c that is an instance of a meta-class C. This last one has a meta-property called P.

1. *If P is multivalued, then its valuation will only return sequences.*

$$P.\mathrm{multi} = \textbf{\textit{true}} \ \Leftrightarrow \ \xi_P(c) \in P.\mathrm{type}^{\langle\star\rangle}$$

2. *The valuation returns values of the right type*

$$\xi_P(c) \in P.\mathrm{type} \cup P.\mathrm{type}^{\langle\star\rangle}$$

Definition 10[1] *guarantees that sequences consist only of one component type.*

The meta-model axioms do not allow optional meta-properties (*cfr.* axiom 28). Every property is necessarily valuated. For our convenience, the software engineer is free to let some property values unspecified. In such cases, default values will be supposed. The next table gives the default value for each possible elementary type:

Type	Default value
$\mathbb{Z}$	0
$\mathbb{R}$	0.0
$\mathbb{B}$	**false**
Char	'␣'
String	“ ”

Type	Default value
Document	“ ”
Video	“ ”
Sound	“ ”
Picture	“ ”
$X^{\langle\star\rangle}$	[]

and DV_T will denote the default value corresponding to each type.

Optional meta-properties bring new problems. Indeed, their semantics become complicated. This problem is crucial once declarative query languages have to be defined. This problem is known in both SQL[2] and OQL[3]. This leads us to the theory of three-valued

[1] See page 46.
[2] *cfr.* [Dat95] for more information on null values in OQL.
[3] *cfr.* [CG99] for more information.

logic [Dat95] or even four-valued logic [Cod90] and the technology to support them is necessary less efficient. On an other side, optional mono-valued meta-properties can easily be emulated by the empty and one-element sequences. This reinforces our idea that optional meta-properties are not necessary in our approach.

We can now define the semantics of inherited meta-properties. That is, what is the value of a property if it is attached to a supertype. Our peculiar definition of the inheritance graph makes this aspect non-trivial. Let us build a new valuation $\xi^\star$ that associate values to both direct and inherited properties.

Definition 24 **($\xi^\star$ – $\xi^\star_P(x)$)**
$\xi^\star : \mathsf{MetaProperty} \mapsto (\mathsf{Class} \rightarrow \mathsf{ValUniv})$ *is a new valuation function defined as:*

$$\forall P \in \mathsf{MetaProperty}, \forall x \in \mathsf{Class}:$$

$$\big(\xi^\star(P)\big)(x) = \begin{cases} \textit{\textbf{if}}\ P \in \mathrm{Prop}(\Gamma(x))\ \textit{\textbf{then}}\ \xi_P(x) \\ \textit{\textbf{if}}\ \exists Y \in \mathsf{MetaClass},\ \exists y \in \Gamma^{-1}(Y):\ \Gamma(x)\ \textit{\textbf{isa}}^\star\, Y \wedge P \in \mathrm{Prop}(Y) \\ \quad \wedge\, x\, \mathfrak{isa}^\star\, y\ \textit{\textbf{then}}\ \xi_P(y) \\ \textit{\textbf{if}}\ \exists Y \in \mathsf{MetaClass}:\ \Gamma(x)\ \textit{\textbf{isa}}^\star\, Y \wedge P \in \mathrm{Prop}(Y) \\ \quad \wedge \left(\nexists y \in \Gamma^{-1}(Y):\ x\, \mathfrak{isa}^\star\, y\right)\ \textit{\textbf{then}}\ DV_{P.type} \end{cases}$$

If no condition succeeds, then $\big(\xi^\star(P)\big)(x)$ *is left undefined (i.e.,* $\big(\xi^\star(P)\big)(x)\ \not\exists$*). As for definition 23, we use the same notation that is* $\xi^\star_P(x) \equiv \big(\xi^\star(P)\big)(x)$

From axiom 23[1] we deduce that the definition of $\xi^\star$ is deterministic and from axioms 11[2] and 12[3] we observe that the main argument of the $\xi^\star$ function could be replaced by its name. Either the meta-property can be retrieved without ambiguity from this name and a class, or the result is undefined. This feature will be convenient to make easier the use of meta-properties in the Voyager 2^+ language.

For instance, the Ω meta-model has one meta-property (see section 4.3.5) named "name" and its only instance ω will have one property defined as: $\xi_P(\omega) =$ "omega".

4.3.13 Classes and Methods

While the definition of the methods is associated with the meta-classes, their execution sometimes depends on a class in the extension of the corresponding meta-class. Let C be a meta-class, and $c \in \Gamma^{-1}(C)$ be a class. Then $\forall M \in \mathrm{Meth}^{-1}(C)$, we note $c \rightarrow M(\text{args})$ the execution of this method. If the method is a function, then this denotes an expression. Since the name identifies a method amongst all the methods of a meta-class, we sometimes note $c \rightarrow \text{foo(args)}$ the call of a method named here "foo".

Method semantics is very simple. First, let us note that contrarily to usual object-oriented programming languages, meta-classes do not inherit the methods of their supertypes. Indeed, the absence of an axiom similar to axiom 12 makes impossible the identification of a method in the inheritance graph from its name. In case of multiple inheritance, this would lead to an arbitrary choice. Moreover, it is not possible to define polymorphic or overloading methods.

Those deficiencies should not trouble the method engineer. Indeed, the repository and Voyager 2^+ are defined in the spirit of an "open architecture". That is, we tried to define

[1] See page 54.
[2] See page 46.
[3] See page 46.

Program 4.1 [The `Polymorph` function.] This Voyager 2^+ procedure is a possible[1] implementation of a polymorphic call to a method named N on a class x of a meta-class MC.

```
function integer Polymorph(string: N, meta_class: MC, MC: x)
  meta_class: sub;
{ for sub in meta_class{rel_subtype_of:meta_isa{@rel_supertype_of:[MC]}} do {
    // for each subtype of 'MC' do:
    if IsNoVoid(sub %% x) then {
      // 'x' can be type casted
      if Polymorph(N,sub,sub %% x) then {
        // the polymorphic call has succeeded.
        return TRUE;
      }
    }
  }
  // there is no valid redefinition of 'N' in the subtypes
  // then try the local definition
  if meta_method{@rel_methods:[MC] with self."name"=N}=[] then {
    // there is no method named 'N' in 'MC'
    return FALSE;
  } else {
    // call the local definition
    x->N();
    return TRUE;
  }
}
```

a minimum semantics that can more or less easily encompass other semantics. For instance, the method engineer could define a procedure Polymorph($M \in$ Method, $x \in$ Class) that implements his own semantics of the polymorphism.

Example

The function defined in program 4.1 implements a a polymorphic call of procedures. Altough this function is dedicated to polymorphic procedures without arguments, it would be possible to define a generic function for procedures or functions with an arbitrary number of arguments. Moreover, this function makes no hypothesis on the nature of the inheritance graph.

4.3.14 Relation

Each element r of MetaRelation allows to represent a "link" between one class of r.owner and a sequence of classes of r.member. We will define formally this "link" as a mapping from meta-relations to functions that associates sequences of member classes to owner classes of right types.

Definition 25 $(\Re_R(x) - \Re_R^{-1}(x))$
$\Re$ is a mapping defined as MetaRelation $\mapsto$ (Class $\to$ Class$^{[\star]}$). *We define this notation* $\Re_R(x) \equiv \big(\Re(R)\big)(x)$ *for a convenient use. For the same reason, we still abuse the notation to define* $\Re^{-1}$ *as*

$$\Re_R^{-1}(x) = y \;\equiv\; x \in \Re_R(y)$$

Let us remark that this function can be left undefined for some classes and that it is not the exact mathematical inverse function of $\Re_R$.

These functions will memorise the connections between the owner classes and their member classes. Some axioms will be necessary to insure that this "memory" stores information like the meta-model stipulates it.

Axiom 30 (Well-typed relations)

1. *Each class of the owner meta-class has a well-defined sequence of member classes wrt. the meta-relations the owner meta-class owns.*

$$\forall R \in \mathsf{MetaRelation} : \ \Gamma^{-1}(R.\mathrm{owner}) = \mathrm{dom}\big(\Re(R)\big)$$

2. *The* $\Re$ *application draws links between classes with the right type.*

$$\forall R \in \mathsf{MetaRelation}, \forall x \in \mathrm{dom}\big(\Re(R)\big) : \ \Gamma(x) = R.\mathrm{owner}$$

Each meta-relation defines a function from Class to Class. We represent it by its inverse in order to add an order between the elements of the domain. However this notation could lead to bad use and represent "functions" that are no longer mathematical functions but "many-to-many" relationships. This axiom will preserve us of this risk.

Axiom 31 (Functional relationships)
$\forall R \in \mathsf{MetaRelation} : \forall x, y \in \Gamma^{-1}(r.\mathrm{owner}) : x \neq y \Rightarrow \Re_R(x) \cap \Re_R(y) = \emptyset$

Axiom 32 (Mandatory)
When a meta-relation is declared "mandatory", the classes of the member meta-class have no sense outside their role of member in some relation.

$$\forall R \in \mathsf{MetaRelation} : \ R.\mathrm{mandatory} = \textbf{\textit{true}} \Rightarrow \bigcup_{o \in R.\mathrm{owner}} \Re_R(o) = \Gamma^{-1}(r.\mathrm{member})$$

Definition 26 ($\Re^\star$)
We define the mapping $\Re^\star : \mathsf{MetaRelation} \to (\mathsf{Class} \to \mathsf{Class}^{[\star]})$ *as a generalisation of the* $\Re$ *mapping. This function is defined as follows:*

$$\Re^\star_R(c) = \begin{cases} \texttt{if} \ \ R.owner = \Gamma(c) \\ \texttt{then} \ \Re_R(c) \\ \texttt{else} \ \begin{cases} \texttt{if} \ \Gamma(c) \ \textbf{\textit{isa}}^\star \ R.owner \\ \texttt{then} \ \begin{cases} \texttt{if} \ (c \downarrow R.owner)\exists \\ \texttt{then} \ \Re_R\big((c \downarrow R.owner)\big) \\ \texttt{else} \ [\,] \end{cases} \end{cases} \end{cases}$$

Let us make two remarks: *1)* if R is a meta-relation, then we have $\mathrm{dom}(\Re_R) \subseteq \mathrm{dom}(\Re^\star_R)$. Hence, this function encompasses $\Re$ and extends it on the subtypes. *2)* When a meta-relation is declared as mandatory, this property is not preserved by the semantics of the $\Re^\star$ function. Indeed, because a subtype may have no supertype at the instance level, it is possible to have a subtype on the member side of a meta-relation that is not linked to a supertype and thus, does not participate in any relations although the meta-relation could be mandatory. Figure 4.18 depicts an instance schema with this kind of problem.

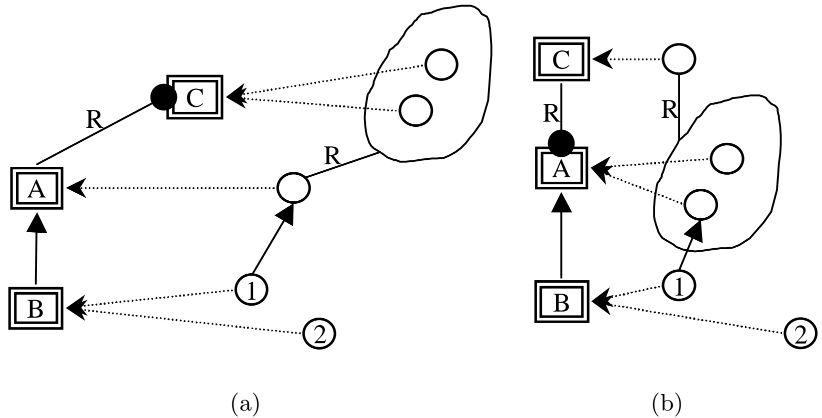


Figure 4.18: **[Mandatory Role and $\Re^\star$]** These diagrams illustrate the semantics of the $\Re^\star$ function. Schema 4.18(a) shows that a class that has no members, has an empty list of members. Schema 4.18(b) shows that although a meta-relation is mandatory, some subtypes on its member side may not participate in any relation (here class 2)

Notation 5 ($\Re_R^{\star -1}$)
As for $\Re_R^{-1}$, we will abuse the inverse notation to define $\Re_R^{\star -1}$ as

$$\Re_R^{\star -1}(x) = y \equiv y \in \Re_R^\star(x)$$

Graphical Conventions In the compact views, a relation is illustrated as a lasso. Its rope is labelled with the name of the meta-relation.

4.3.15 Identifiers

As in any object-oriented data model, each class/object owns a value that identifies it amongst the other classes. We could content us with this characteristics. Nevertheless, this identifier is mainly used at a technical level and has no special meaning for the software engineer, except maybe to "debug" some specifications. In many occasions, classes need to export themselves in a user-friendly way, that is a human readable form. This could be a picture, a sound, a number, or more often a sequence of characters. The identifier we will associate with meta-classes will serve this purpose.

An identifier is defined as a sequence[1] of meta-properties and roles that identify the instances of meta-classes. The roles are either the owner part or the member part.

Definition 27 (Meta-class identifier – local & global identifier – KindId**)**
We define

$$\text{Ident} : \mathsf{MetaClass} \rightarrow \Big(\mathsf{MetaProperty} \cup \big(\mathsf{MetaRelation} \times \{\mathtt{owner}, \mathtt{member}\}\big)\Big)^{[\star]}$$

a function that associates an identifier with some meta-classes. In the same way, this identifier will be qualified as `local` *or* `global` *by this function:* KindId : $\mathsf{MetaClass} \rightarrow \{\mathtt{local}, \mathtt{global}\}$.

[1]This identifier will be use to name the classes, hence, the order is meaningful.

Axiom 33 (Well-formed identifier)
The following propositions must hold for every identifier associated with a meta-class C:

1. $\mathrm{Ident}(C) \cap \mathsf{MetaProperty} \subseteq \mathrm{Prop}(C)$.
2. $(R, \texttt{owner}) \in \mathrm{Ident}(C) \;\Rightarrow\; R.\mathrm{member} = C \wedge R.\mathrm{mandatory} = \textbf{\textit{true}}$
3. $(R, \texttt{member}) \in \mathrm{Ident}(C) \;\Rightarrow\; R.\mathrm{owner} = C$.

and each identifier must be qualified: $\mathrm{dom}(\mathrm{Ident}) = \mathrm{dom}(\mathrm{KindId})$

Axiom 33 expresses that inheritance is not taken into account to build the identifier of a meta-class. This restriction has two main reasons: *1)* the validation of the identifier will be easier and thus will take less time, *2)* this will facilitate the edition of the inheritance graph both at the meta-model and specification levels. Indeed, the identifier does not depend on the subtypes nor on the supertypes.

Let us note that identifiers can be empty. This definition is not in keeping with the common sense but, nevertheless, it is sound and it just forbids to have more than one instance of a meta-class whatever its roles and its properties. This situation is sometimes encountered (the initial state in Statechart diagrams, the main function in programs, a director in organizational diagrams, ...).

Definition 28 (Id value − idval)
Let C be a meta-class such that $\mathrm{Ident}(C)\exists$ and we note $\sigma \equiv \mathrm{Ident}(C)$. We will associate to each class c of C a value corresponding to the identifier of its type. We compute this value as follows: build a sequence τ such that $\sharp\tau = \sharp\sigma$. For each index $i \in \{1 \ldots \sharp\sigma\}$:

a) if $\sigma[i] \in \mathrm{Prop}(C)$ then $\tau[i]$ is $\xi_{\sigma[i]}(c)$. From axiom 28[1]*, $\xi_{\sigma[i]}(c)$ is well-defined.*

b) if $\sigma[i] = (R, \texttt{owner})$ then $\tau[i]$ is $\Re_R^{-1}(c)$. From proposition 2 in axiom 33, $\Re_R^{-1}(C)\exists$.

c) if $\sigma[i] = (R, \texttt{member})$ then $\tau[i]$ is $\Re_R(c)$.

We call this value $\mathrm{idval}(c)$*.*

Theorem 4
If C is a meta-class such that $\mathrm{Ident}(C)\exists$, then $\forall x \in \Gamma^{-1}(C) : \; \mathrm{idval}(x)\exists$.

> ***Proof*** *The elements of this proof have been indicated in the definition of* idval. □

Axiom 34 (Id value uniqueness)
Let C be a meta-class such that $\mathrm{Ident}(C)\exists$. If this identifier is global ($\mathrm{KindId}(C) = \texttt{global}$), then every class in C must have a distinct identifier value.

$$\forall c, d \in \Gamma^{-1}(C) : \; \mathrm{idval}(c) = \mathrm{idval}(d) \;\Rightarrow\; c = d$$

If this identifier is local, then two classes in a specification must have distinct identifier values whatever this specification.

$$\forall c, d \in \Gamma^{-1}(C) : \; \Big(\mathrm{idval}(c) = \mathrm{idval}(d)\Big) \;\wedge\; \Big(\mathrm{Def}^{-1}(c) \cap \mathrm{Def}^{-1}(d) \neq \emptyset\Big) \;\Rightarrow\; c = d$$

[1]See page 59.

Axiom 35 (An identifier for each class)
In a specification, all the classes must have a valid identifier.
$\forall c \in \mathsf{Class}$, $\forall s \in \mathrm{Def}^{-1}(c)$: *it exists one and only one class l that is a leaf of c in s (axiom 26). Hence, it must exist a class d such that $l \,\mathfrak{isa}^\star\, d$ or $l = d$, and* $\mathrm{Ident}(\Gamma(d))\exists$.

This axiom ensures us that it will always be possible to associate a "value" to a class that identifies it in the context of its specification. Let us remember that a specification is very often a context, a surroundings in which the software engineer will work. For this reason, it is important that identifiers have a meaning in this context. For instance, a car could be identified by its number plate, its motor's serial number and its assurance policy's number in respectively a car park, a car factory and an insurance company. In computer science, the same situation can occur: a "task" could be identified by its name in some high level analysis; by its name, its arguments and a package in a JAVA program or, by an address in the object file. So, it is important to choose the best identifier depending on the context (i.e., the specification). Let us remark that a class can have several valid identifiers. In such cases, one recommends to choose the identifier that is the closest to the leaf.

Theorem 5
Let C be a meta-class. If neither C nor its supertypes have an identifier, then C should be a virtual meta-class, i.e., if $\forall X \in \mathrm{super}^\star(C) \cup \{C\}$: $\mathrm{Ident}(X) \nexists$ then the model engineer can specify $\mathrm{virtual}\langle\, C \mapsto \textbf{\textit{true}}\,\rangle$ *without changing the meta-model semantics.*

> **Proof** Let us suppose that $C.\mathrm{virtual} = \textbf{false}$, then it would be possible to find a $c \in \Gamma^{-1}(C)$ such that c has no subtypes at all. However, the hypothesis states that C and its supertypes have no identifier. Hence, it is not possible to find an identifier for c unless its has one subtype "somewhere" that has an identifier. □

Some important remarks must be made about the identifiers.

1. The previous properties (theorem 4 and axiom 35) ensure us that it will always be possible to associate an identifier along with a class whatever is its type and the specification it occurs in.

2. Identifiers allow to partially control the form of the inheritance graph. Let us imagine one meta-class that has an identifier and another meta-class that inherits from the previous one but that has no identifier. So, we have this situation

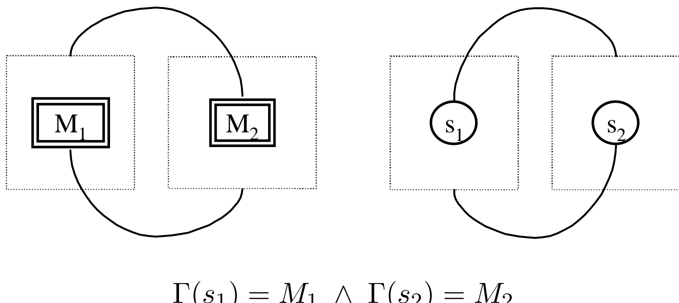
$$A, B \in \mathsf{MetaClass} \;\wedge\; A \,\mathfrak{isa}\, B \;\wedge\; \mathrm{Ident}(A) \nexists \;\wedge\; \mathrm{Ident}(B)\exists$$

   It becomes obvious that all the instances of meta-class A will must inherit from a B class to respect the axiom 35.

3. When an identifier is local, then one can use this identifier with the identifier of a meta-model that contains the corresponding meta-class to obtain a global identifier. This principle is recursive and could be applied to the meta-model.

4. An identifier value can have an infinite size. Two reasons explain this infinite expansion.

   (a) The identifier of a meta-class A can require a meta-class B (via a meta-relation), which is itself identified in a direct or indirect way by A. This cycle could be reproduced at the specification level, and we could necessarily obtain an infinite loop.



$$\Gamma(s_1) = M_1 \ \wedge \ \Gamma(s_2) = M_2$$

Figure 4.19: **[Infinite identifier value]** The identifier value of the specification s_1 depends on s_2, and the identifier value of s_2 depends on s_1.

(b) The next case can occur when a mutual dependency appears in the local identifier of a meta-model. Let us study this case: $M_1, M_2 \in \mathsf{MetaModel}$ are two meta-models. $M_1 \in \mathrm{Mod}(M_2)$ and $M_2 \in \mathrm{Mod}(M_1)$. The meta-model M_1 (resp. M_2) has a meta-property P_1 (resp. P_2). Now, let us define the identifiers as: $\mathrm{Ident}(M_1) = P_1$ and $\mathrm{Ident}(M_2) = P_2$ with $\mathrm{KindId}(M_1) = \mathrm{KindId}(M_2) = \texttt{local}$. Then we can reproduce this pattern at the specification level i.e., $s_1 \in \Gamma^{-1}(M_1)$, $s_2 \in \Gamma^{-1}(M_2)$, $s_1 \in \mathrm{Def}(s_2)$, and $s_2 \in \mathrm{Def}(s_1)$. Then we deduce easily that the identifier value of s_1 will be $\xi_{P_1}(s_1)$. However, if we wish to have an unambiguous representation of s_1, we will need to use both the $\xi_{P_1}(s_1)$ and the identifier value of s_2. But the same pattern will appear for s_2 and we will need the identifier value of s_1. The cycle is closed. This example is depicted in Fig. 4.19

Infinite identifier values are not a problem from a mathematical point of view. However, this will certainly annoy the method engineer who cannot wait an infinite time for a result! Hopefully, the infinite structures that a program could produce from such identifiers would be rational trees. That is, trees where the infinite branches can be cut and replaced by references to some node near the root. Hence, producing a finite representation of these trees is easy[1] and possible in a finite time. Let us note that cyclic identifier definitions are not so frequent and even when this pattern occurs, the identifier value is not necessarily a cycle.

Axiom 36 (Ω's identifier)
Ω has one global identifier.

$$\mathrm{Ident}(\Omega) = [\mathit{name}] \ \wedge \ \mathrm{KindId}(\Omega) = \texttt{global}$$

Graphical Conventions In compact views, identifiers are not visible since meta-properties are not illustrated. In extended views, global identifiers are shown as *primary identifiers* and local identifiers as *secondary identifiers*.

[1]The tree traversal has just to remember the nodes that have been visited. Once a node (i.e., a class) has already been visited, the branch that leads to it is replaced.

4.4 Some Operations

The meta-CASE contains many operations that can have an effect on the repository described in the previous sections. But, it is impossible to describe all of them in this thesis. Section 4.4.1 presents the deletion of a class (or a specification). The repository contains enough constraints to make the deletion of a class quite simple from the software engineer's point of view (and also for the method engineer). Nevertheless, these constraints makes the deletion a hard piece to implement (from the meta-CASE architect's point of view). Sections 4.4.3 and 4.4.4 introduce two common operations in editing environments: the "Cut&Paste" and its counter-part "Remember&Paste". Finally, the last section gives the specifications operations that update the meta-models.

4.4.1 Deletion

The problem of deleting a class from the repository is quite complex. Indeed, a class can be connected with a complex environment: it can play roles in mandatory meta-relations, it can have subtypes or virtual supertypes, it can be a specification, and a specification can have a strong or weak connection with its components ... We have illustrated here some difficulties that the deletion procedure will have to face with.

The "DeleteClass" procedure will be in charge of removing a class from the repository in such a way that *1)* the axioms will be respected before and after the call and *2)* only the minimal information will be withdrawn from the repository. Its implementation is described below as well as a proof of its correctness.

Here follow some preliminary definitions to make the algorithm more concise.

Definition 29 ($\lll_S$)
If S denotes a subset of Class *(i.e., $S \subseteq \mathsf{Class}$), then $\lll_S$ denotes an infix relation defined on* $\mathsf{Class} \times \mathsf{Specification}$ *as:*

$$\forall x \in \mathsf{Class}, \forall y \in \mathsf{Specification} : \ x \lll_S y \Leftrightarrow \Big(\exists z_1, \ldots, z_m \ (m \geq 0) \ \in \mathsf{Specification} \setminus S : \\ x \in \mathrm{Def}(z_1) \wedge z_1 \in \mathrm{Def}(z_2) \wedge \cdots \wedge z_{m-1} \in \mathrm{Def}(z_m) \wedge z_m \in \mathrm{Def}(y)\Big)$$

If the property $x \lll_S s$ is true for some set S, then this simply means that it possible to find a path that goes from class x to specification s without visiting the elements of S. This relation expresses how a class depends on a specification.

> ***Remark*** *Since the supertypes are systematically in the definitions of their subtypes (axiom 24[1]), the path can use the inheritance graph.*

Definition 30 (subtypes — subtypes*)
We define the functions subtypes *and* subtypes* : $\mathsf{Class} \mapsto \mathsf{Class}^\star$ *as* $\text{subtypes}(x) = \{y \in \mathsf{Class} \dagger y \,\mathfrak{isa}\, x\}$ *and* $\text{subtypes}(x) = \{y \in \mathsf{Class} \dagger y \,\mathfrak{isa}^\star\, x\}$.

The pseudo-code of the `DeleteClass` procedure and of its functions is listed separated programs:

[1]See page 55.

Program 4.2 [**The** `DeleteClass` **Procedure**]

procedure `DeleteClass`($x \in \mathsf{Class}$)
Precondition $x \neq \omega$ and all the axioms are verified.

```
begin
      S := DeleteClass0(x)
      if ∃y ∈ S : y → CanDelete() = false then abort
    ♣ for z ∈ S do
            if ∃s ∈ Def⁻¹(z) :  z isa y1 ∧ z isa y2 ∧ y1 ≠ y2
                ∧ ((subtypes*(y1) ∩ subtypes*(y2) ∩ Def(s)) \ S) = ∅
            then abort
    ♠ for C ∈ MetaClass :  Ident(C)∃ do
            for R ∈ MetaRelation :  (R, member) ∈ Ident(C) do
                  if ∃o ∈ Γ⁻¹(R.owner) : ℜ_R(o) ⊆ S
                  then check the identifier of o and abort otherwise
      % remove all the elements of S from Class
      % and unlink them (ξ, ℜ, isa , . . . ).
      Class := Class \ S
      Specification := Specification
      for s ∈ Specification \ S do
            Def := Def⟨ s ↦ Def(s) \ S ⟩
      Clean ℜ, ξ, and  isa

end
```

Program 4.3 [**The** `DeleteClass0` **Function**]

function `DeleteClass0`($x \in \mathsf{Class}$)$\rightarrow \mathsf{Class}^\star$
% 'ToCheck' is a global variable.

```
begin
   ToCheck := ∅
   Δ0 := DeleteClass1(∅, x)
   i := 0
   repeat S := {y ∈ ToCheck \ Δi  †  ¬(y ⋘_Δi ω)}
            Δ(i+1) := Δi
            for y ∈ S do
                  Δ(i+1) := DeleteClass1(Δ(i+1), y)
            i := i + 1
            until Δi = Δ(i−1)
   return Δi
end
```

Program 4.4 [**The** `DeleteClass1` **Function**]

```
function DeleteClass1(W ∈ Class⋆, x ∈ Class) → S ∈ Class⋆
% 'ToCheck' is a global variable.
begin

     if x ∈ W then return W
     C := Γ(x)
     S := W ∪ {x}
   ♢ S := S ∪ (DeleteClass2(x → ToDelete( ) ∩ Class) \ {ω})
   ♠ for R ∈ MetaRelation : R.mandatory ∧ R.owner = C do
          S := DeleteClass2(S, ℜ_R(x))
     for D ∈ super(C) : D.virtual = true do
          y := (x ↓ D)
   ♣      if subtypes(y) ⊆ S then
               S := DeleteClass1(S, y)
     if x ∈ Specification then
          if x.mode = strong then
               S := DeleteClass2(S, Def(x))
          if x.mode = weak then
               ToCheck := ToCheck ∪ Def(x)

end
```

$\diamondsuit$: The `ToDelete` method returns a list composed of arbitrary elements. If the method engineer defines this method in a wrong way, some elements may not necessarily denote classes. Hence, the intersection with the Class set ensures us that this statement is well-typed.

Program 4.5 [The "DeleteClass2" Function]

```
function DeleteClass2(W ∈ Class⋆, S ∈ Class⋆) → Class⋆
begin
   if S = ∅ then return W
   else let x ∈ S
        return DeleteClass2(DeleteClass1(W, x), S \ {x})
end
```

Function	Program n°
`DeleteClass`	prog. 4.2 page 68
`DeleteClass0`	prog. 4.3 page 68
`DeleteClass1`	prog. 4.4 page 69
`DeleteClass2`	prog. 4.5 page 69

One will enforce to prove the correctness of these algorithms. Theorem 7 proves that the deletion mechanism halts, theorem 6 shows that the procedure respects the axioms, and theorem 8 shows that the `DeleteClass` procedure will only withdraw the minimal information in order to delete its argument and to respect the axioms.

Lemma 1
If $R = \texttt{DeleteClass1}(W, x)$ *(*$W \subseteq \mathsf{Class}$ *and* $x \in \mathsf{Class}$*) then* $W \cup \{x\} \subseteq R$*.*

> **Proof** The statement $S := W \cup \{x\}$ is a mandatory step in the body of this function. The other statements do not remove elements from S. □

Lemma 2
If $R = \texttt{DeleteClass2}(W, S)$ *(*$W, S \subseteq \mathsf{Class}$*) then* $W \cup S \subseteq R$*.*

> **Proof** The proof can easily be made by induction with the $\subseteq$ relation on the second argument of `DeleteClass2`. If $S = \emptyset$, then $R = W$. Otherwise, by induction, we know that $\texttt{DeleteClass1}(W, X) \cup (S \setminus \{x\}) \subseteq R$, and from lemma 1, we deduce $W \cup \{x\} \cup (S \setminus \{x\}) \subseteq R$, that is, $W \cup S \subseteq R$. □

Theorem 6
The `DeleteClass` *procedure can be completed in a finite time.*

> **Proof** Let x be a class. Then $\texttt{DeleteClass}(x)$ will terminate iff DeleteClass0($\emptyset$, x) will terminate. In `DeleteClass0`, several cases can occur: *1)* the "repeat" statement is infinite. But we observe that $\Delta_i \subseteq \mathsf{Class}$, then, this means that $\sharp\mathsf{Class} = \infty$; this hypothesis can be dismissed. *2)* $\sharp S = \infty$, but then $\sharp\text{ToCheck} = \infty$. Since ToCheck $\subseteq \mathsf{Class}$, this is impossible. *3)* $\exists W \in \mathsf{Class}^\star, y \in \mathsf{Class}$ such that $\texttt{DeleteClass1}(W, y)$ does not halt. But all the loops in the body of this function are finite. Hence, the only possibility is to have an infinite chain of calls[1] $\texttt{DeleteClass1}(W_1, y_1)$, $\texttt{DeleteClass1}(W_2, y_2)$, ... , $\texttt{DeleteClass1}(W_i, y_i)$, ... where $(W, y) = (W_1, y_1)$. But, from lemmas 1 and 2, we know that $\forall i \geq 1 : W_i \subseteq W_{i+1}$. From the first statement in the body of `DeleteClass1`, we observe that if $y_i \in W_i$ then the chain is finite and if $y_i \notin W_i$, then $\sharp W_{i+1} > \sharp W_i$, but $W_i \subseteq \mathsf{Class}$, $\forall i \geq 1$ and $\sharp\mathsf{Class} \neq \infty$.
>
> The calls to the `CanDelete` and `ToDelete` functions are the only dark points of this proof. Indeed, these functions are defined by the method engineer. □

Theorem 7
The `DeleteClass` *verifies the axioms of the repository.*

> **Proof** Axioms 18, 19, 21, 22, 23, 28, 29, 30 and 31 have trivial demonstrations.

[1] We consider that `DeleteClass2` is an hidden call to `DeleteClass1`.

Axiom 20 page 52: From its precondition, the procedure cannot delete directly the ω specification. The only place where the axiom could be broken is in the `DeleteClass1` function. But from axioms 15 and 17, we see that this function cannot delete ω as a mandatory member of some meta-relation or as a subtype/supertype of some class. Moreover, from theorem 2, the ω class cannot be deleted as a component of a strong specification. The only statement that could infringe the axiom is the call to the user-defined methods, but ω is explicitly removed from that union.

Axiom 24 page 55: Let us suppose that $\exists x, y \in \mathsf{Class},\ \exists s \in \mathsf{Specification}:\ x \,\mathtt{isa}\, y \wedge x \in \mathrm{Def}(s) \wedge y \notin \mathrm{Def}(s)$. The precondition ensures us that $y \in \mathrm{Def}(s)$ before the call of the procedure. Since, x, y still belong to Class after the call, they have not been deleted, and the only way to reach this result is to edit the Def function. The only place where this function is modified is in the `DeleteClass` procedure: $\mathrm{Def} := \mathrm{Def}\langle\, s \mapsto \mathrm{Def}(s) \setminus S\,\rangle$. But this would mean that $y \in S$ and this is impossible since y has not been deleted.

Axiom 26 page 57: This axiom is explicitly checked at line ♣ of `DeleteClass`. If the withdraw of a class breaks this axiom, then the deletion operation is aborted. The reason is quite simple. Due to the complex semantics of the multiple inheritance, there would be multiple ways to restore this axiom and it would be impossible to choose the best one a priori. For this reason, the management of the deletion of such classes is let to the method engineer when he defines the X::`ToDelete` method of meta-classes with multiple supertypes.

Axiom 27 page 58: Statements ♣ and ♡ in `DeleteClass1` insures us that such virtual classes would also be deleted.

Axiom 32 page 62: All the members of mandatory meta-relations whose the owner will disappear are deleted at line ♠ of the `DeleteClass1` function.

Axiom 34 page 64: Let us suppose that meta-class C has an identifier defined as follows:

$$\mathrm{Ident}(C) = [P, (R_o, \mathtt{owner}), (R_m, \mathtt{member})]$$

One will examine the effect of the `DeleteClass` procedure on each component of this identifier.

The meta-property: The procedure does not change the properties, and hence, this component will be preserved.

The $(R_o, \mathtt{owner})$ component : let us take this scenario into consideration,

$$c \in \Gamma^{-1}(C) \text{ and } \exists x \in \Gamma^{-1}(R_o.\mathrm{owner}):\ c \in \Re_{R_o}(x)$$

and x is a class that will be deleted by the procedure. Then the value of the c's identifier will change and this should need a new validation. But from axioms 32[1] and 33.2[2], we know that the deletion of x would lead the procedure to delete c as well.

[1]See page 62.
[2]See page 64.

The (R_m, `member`) **component:** From axiom 31[1], we know that this component is quite weak in the identifier value, i.e., the value of this component does not matter in the identifier unless its value is the empty sequence. The statement at line ♠ in `DeleteClass` verifies this property and aborts if it is not satisfied. The method engineer can override the `ToDelete` method to avoid this abortion.

□

Theorem 8
The `DeleteClass`*(x) procedure only deletes the minimal information in order to delete the class x and to respect the repository axioms.*

Proof *a)* The `DeleteClass` procedure does not delete any class but it just calls the `DeleteClass0` function with the class to delete as argument. *b)* The classes the `DeleteClass0` function will delete are stored in the Δ_i set. The initialization of Δ_0 and the assignments of $\Delta_i\,(i \geq 1)$ in the loop are correct if the `DeleteClass1` function deletes a minimal information. *c)* The `DeleteClass1` function received two parameters: W is a set of classes that will be deleted, and x is a class that the function should delete. The result consists in the set of all the classes we shall have to delete if we delete x. At line ♢, we delete the classes as asked by the `ToDelete` method. At line ♠, we only delete the mandatory members of relations whose the owner is x. At line ♣, we delete the classes that violates the axiom 27. The body of the last condition removes the components of the deleted specifications with respect to the semantics defined in section 4.3.10.

This function makes several call to the `DeleteClass2` function. But this one is just a wrapping around the `DeleteClass1` function to generalize it when the second parameter is a set. □

4.4.2 Other Kinds of Deletion

The minimal approach[2] of the `DeleteClass` procedure may not suit the method engineer's requirements. Nevertheless, the `DeleteClass` function can be refined in several ways to propose other semantics and meet other requirements: *a)* it could delete the whole inheritance graph; *b)* it could cut all the branches under its argument and only preserve the supertypes; or *c)* it could delete all the supertypes of its parameter. This list of examples is not exhaustive. Moreover, the method engineer can refine itself the `DeleteClass` procedure in overriding the `ToDelete` method. Hence, he could define himself the three behaviours presented as examples.

4.4.3 Cut&Paste

The *Cut & Paste* procedure will move a class from the definition of a specification (s) to another specification (t).

procedure Cut&Paste (Class : c, Specification : s, Specification : t)

Precondition.

[1] See page 62.
[2] *Cfr.* theorem 8.

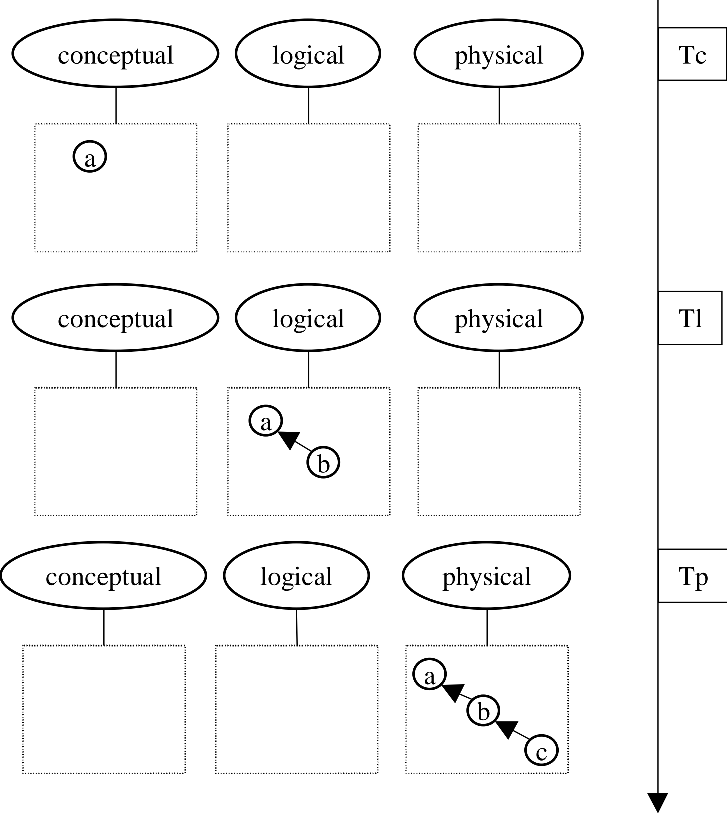


Figure 4.20: [**Cut&Paste Scenario**] The product is moved from one step to another. The product is refined at each step with a subtype that contains the added value of the transformation.

- $c \in \mathrm{Def}(s) \ \wedge \ c \notin \mathrm{Def}(t)$
- $\Gamma(c) \in \mathrm{Mod}(\Gamma(t))$
- c is a leaf of s.

Postcondition.

- $c \notin \mathrm{Def}(s) \ \wedge \ c \in \mathrm{Def}(t)$
- $\forall x \in \mathsf{Class} : \ c \ \mathfrak{isa}^{\star} \ x \ \Rightarrow \ \big(x \in \mathrm{Def}(t) \ \wedge \ x \notin \mathrm{Def}(s)\big)$

This procedure allows the software engineer to migrate one class from one specification to another. One possible use would be to pass one "product" throughout several steps/stages. The latter would be represented by specifications of distinct meta-models. Figure 4.20 shows a scenario which illustrates this example. Nevertheless, this scenario does not preserve the history of the process and is valid only for simple processes which involve only refinements and no complex transformations.

4.4.4 Remember&Paste

The *Remember&Paste* procedure will add a class in the definition of another specification.

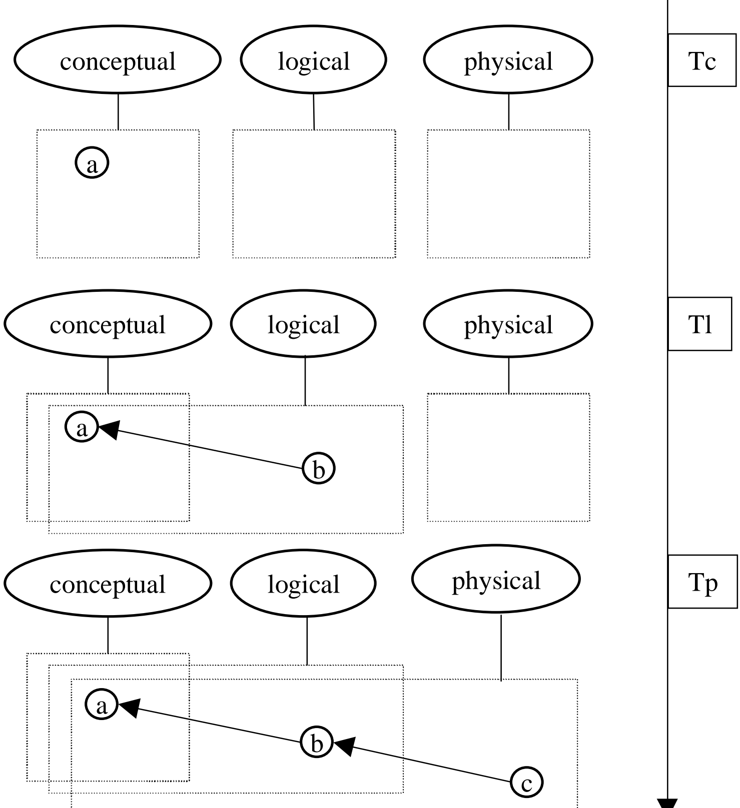


Figure 4.21: [**Remember&Paste Scenario**] The product is moved from one step to another. The product is refined at each step with a subtype that contains the added value of the transformation. Nevertheless, the product is not removed from the previous specifications.

`procedure Remember&Paste` (Class : c, Specification : s)

Precondition.

- $c \in \mathsf{Class} \setminus \mathrm{Def}(s)$
- $\Gamma(c) \in \mathrm{Mod}(\Gamma(s))$

Postcondition.

- $c \in \mathrm{Def}(s)$
- $\forall x \in \mathsf{Class} : \ c \ \mathfrak{isa}^{\star} \ x \Rightarrow x \in \mathrm{Def}(s)$

This procedure allows the software engineer to force two distinct specifications to share classes — even if these specifications have distinct meta-models.

The scenario of section 4.4.3 can be reused here to illustrate a process which will preserve its history (*cfr.* Fig. 4.21). The product is still moved from one specification to another one but it is not removed from the previous one. Hence, once the process is completed, every modification at the conceptual level on class a will be reflected in the further steps.

4.4.5 Specialize

The *Specialize* function derives a new subtype from an existing class. This subtype can be created in any valid specification. As for the previous operations, this one is supported by a dialogue box with which the software engineer will precise the information needed by the function. This information is stored in the list l of the function.

`function` $\Phi : \phi$ `Specialize` (Class : c, Specification : s, MetaClass : Φ)

Precondition.

- $c \notin$ Specification. Indeed, the content of the c's definition should be moved to the derived subtype (i.e., ϕ). Nevertheless, this operation is not yet supported by `Specialize`. (*cfr.* axiom 25 page 56)
- Φ **isa** $\Gamma(c)$
- $\Phi \in \text{Mod}(\Gamma(s))$
- Φ::`CanCreate`(l)=**true**
- $\neg\Phi$.virtual

Postcondition.

- $\phi \notin \Gamma^{-1}(\Phi)_{\mathsf{old}} \ \wedge \ \phi \in \Gamma^{-1}(\Phi)_{\mathsf{new}}$
- ϕ $\mathfrak{isa}$ c
- $\{\phi\} \cup \text{super}^\star(\phi) \subseteq \text{Def}(s)$

The `Specialize` operation has already been encountered in figures 4.20 and 4.21. This operation was utilized jointly with the previous ones to derive the classes b and c from the class a. In the scenario depicted in Fig. 4.21, the `Remember&Paste` operation could completely be supplanted by `Specialize`.

4.5 The User Guide

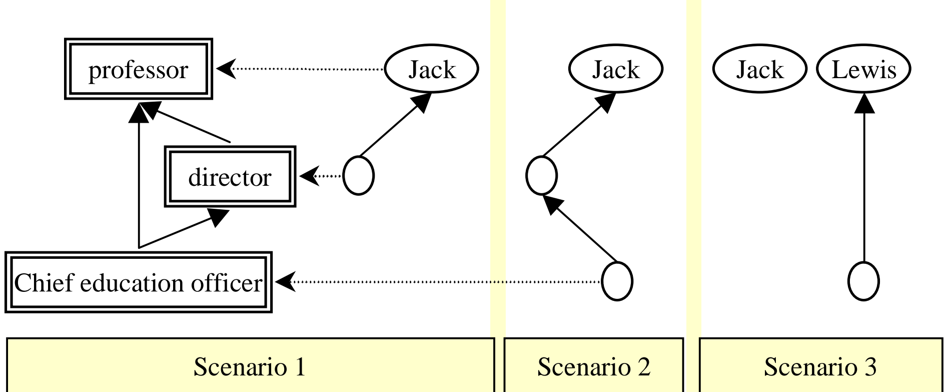
The previous sections were a quite rough presentation of the repository. Hence these sections did not give any guidelines to define the software engineer's needs although they are precise, complete and formal. We will enforce us to give some rules on how to use the previous concepts to gather and organize the semantics of the problems to be analysed. Although there are very few concepts (meta-classes and meta-models), one will see that the repository can reach the same expressiveness than other repositories.

4.5.1 Specialization

The meta-CASE exploits the dynamic inheritance semantics to offer a continuum during the mutation of classes. One will use here a very simple example to show that this mechanism can be used in very common situations like "promotions".

To make our explanation easier, we will use a simple case study where `professor`, `director` and `CEO` denote meta-classes and people (`Jack` and `Lewis`) are classes. Let us suppose that a professor can become a director, a director can become in his turn chief education officer (CEO) and in some special cases, when no directors wish to be promoted,

then the council can choose a professor as CEO. If "*Jack*" is a professor, and if he becomes a director, then usual databases will require to delete professor Jack so that he can come to life again as a director. Our repository supports the mutation principle via the dynamic inheritance. This mutation scheme can be modelled by an inheritance graph that shows the possible promotions: `director` **isa** `professor`, `CEO` **isa** `director` and `CEO` **isa** `professor`. Now, let us imagine a possible scenario: *Jack* is a professor and is just being promoted to the *director* status. This promotion/mutation can be represented by a specialization of the `Jack` class into a `director` class (Scenario 1). Now, let us imagine that he is being promoted to the *CEO* status in accordance with the system's rules (Scenario 2). Unfortunately, he made a serious error and he is moved down to its original status, i.e., *professor*. But no other directors accept this charge, and the council has to choose a simple *professor* (Lewis) as "victim". This mutation is illustrated in scenario 3 of our story. The diagram clearly represents the situation, i.e., Lewis is now a CEO but he is not a director.



Now, let us imagine that Jack let some unfinished job on his desk (modelled with meta-relations or meta-properties at the `CEO` meta-class level). It would also be possible to preserve these tasks (i.e. the `CEO` class) and to attach this class to the `Lewis` class. In the above interpretation, Lewis becomes a fresh CEO and is in the same situation than Jack at his beginning. With this mechanism (one preserves the `CEO` class), Lewis will inherit both the CEO status and the Jack's duties. Indeed, the CEO of scenario 2 is preserved in scenario 3.

Specialization can be used with profit in software engineering methods to transform products between stages. The next table shows several examples of base meta-classes with its possible subtypes which could be used as specializing meta-classes in some methodologies:

Base Class	Possible Specializations
`Person`	`Manager`, `Programmer`, `Analyst`, `Responsible`, `Consultant`
`State`	`CompositeState`, `StateMachine`
`EntityType`	`Class`, `Template`, `DataStore`, `Persistent`, `Transient`
`Package`	`Unit`, `Library`, `COM`, `DLL`
`Process`	`Completed`, `Active`, `Sleeping`

Specialization often results from a posteriori decisions to reuse some concept and to extend it with the ad-hoc information.

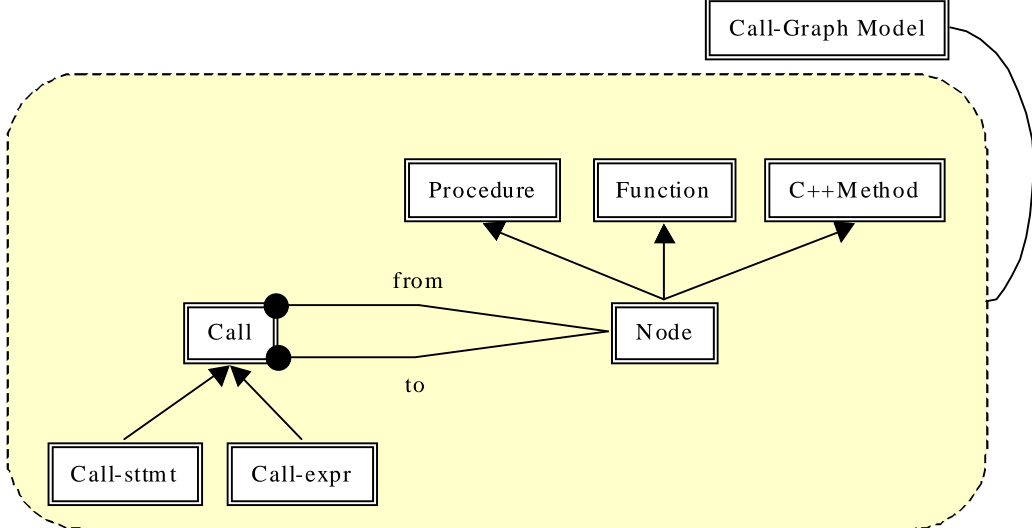


Figure 4.22: [**Inheritance with exceptions**] Call-graph diagrams denote calls between nodes which denote functions, procedures or methods but never all of them at the same time. Because the semantics of the functions (resp. procedures, methods) does not depend on that of the nodes, the inheritance relationship goes naturally from the `node` subtype to the `function` supertype. Because the inheritance takes the exceptions into consideration, it is possible to derive the `node` meta-class as a subtype of the `function`.

4.5.2 Inheritance with Exceptions

The method engineer can exploit the semantics of the inheritance with exceptions to model complicated situations quite easily. For instance, call-graph diagrams represent calls between nodes. The latter denote functions, procedures, methods, programs, etc Nevertheless, a node will never be a function and a procedure at the same time. The explanation of the call-graph's semantics suggests the obvious presence of an inheritance relationship between those terms but this would be impossible to model without the exceptions as this has been defined and proposed in our repository. Figure 4.22 depicts the most natural way to model this semantics with the exceptions.

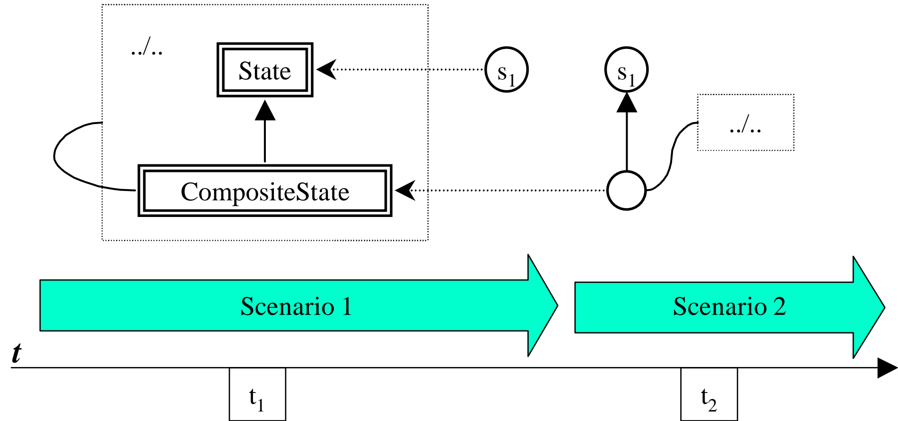
4.5.3 Explosion

An explosion consists in exploding an atomic concept into several others that are in some relationship with the first one. This relationship could have the following semantics: *defined-by*, *composed-of*, *related-to*, *used-by*, ... The explosion mechanism is a very frequent mechanism in CASE tools. Explosion is deeply linked to the aggregation composition. In our meta-meta-model, meta-models are themselves meta-classes. Hence, a specification can either be considered as an atomic concept or as a composite object. The explosion consists in replacing the composite object with its components. The meta-CASE supports the explosion mechanism in many ways: *1)* software engineers can request to view the composition of a composite object presented as an atomic object; *2)* the meta-CASE is able to visualize specifications inside specifications and so on; and *3)* the method engineer can permit the software engineer to explode classes in a arbitrary way (i.e., to decide what is composite or not a posteriori).

The first two mechanisms will be presented in the chapter devoted to GraSyLa (*cfr.* chap-

ter 6). The last one is still again linked to the dynamic inheritance. The top-down analysis is a great consumer of this principle. Indeed, top-down analyses often confuse composite objects with atomic concept during the top-down decomposition. For instance, atomic statements become procedures or functions, elementary types become compound types, states become compound states, and so on. Let us inspect the last example. At some stage of an analysis, a software engineer defines states in a Statechart Diagram. And at this abstraction level, he decides to omit some technical details in the states definition. This refinement will occur in the next stage where he will redefine some states with finer concepts (substates). The software engineer will have to decide which states he will refine, that is, which atomic states will become compound states.

A posteriori decisions are possible by using the technique described in section 4.5.1. The next schema shows both how a method engineer should model this situation and how a software engineer could use the meta-CASE to exploit a posteriori explosions.

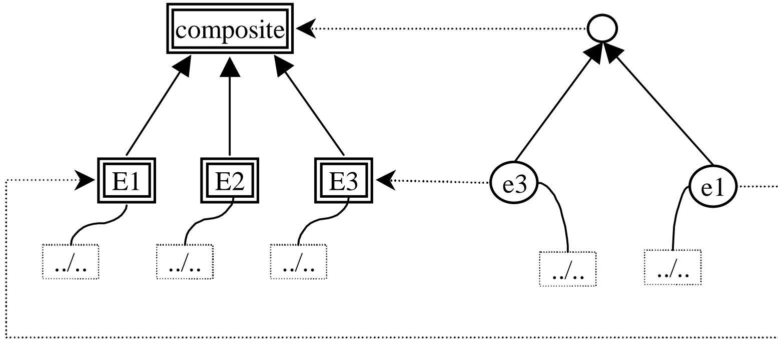


In scenario 1, the software engineer has defined one atomic state s_1. In scenario 2, he decides to refine s_1 and to explode it. Hence, he specializes the class into a subtype (i.e., a specification) and can continue his job in defining the states describing s_1.

4.5.4 Multiple Explosions

The problem to explode a concept depending on several criteria is well-known [GFS$^+$94, Met99]. MetaEdit+ proposes two techniques (explosion and decomposition) although other projects do not support any explosion mechanism.

Contrary to the current researches, the multiple explosion mechanism is not a special technique that would require a dedicated semantics. Indeed, it is supported by the shared dynamic inheritance. The next diagrams shows on one example how it is possible to explode one concept into several definitions.



In this diagram, the `composite` meta-class can be decomposed depending on three possible criteria denoted by three subtypes (meta-models E_1, E_2 and E_3). At the specification level, the software engineer decided to explode the concept by using only two criteria ($\rightarrow\ e_1, e_3$). Let us remark that the meta-meta-model distinguishes a concept having no explosion from those having an empty explosion. We can also observe that an exploded class can still appear in the definition of its explosions.

Remark

1. In order to repect axiom 26[1], the e_1 and e_3 specifications of the diagram must belong to distinct specifications. From the software engineer's point of view, this means that he cannot visualize both explosions in the same display (e.g., a window). This restriction can be expressed in english like this: in one ontology, a class can be exploded only once. This axiom will thus limit the interest of this technique.
2. Axiom 23[2] limits the number of *explosions* but let the number of *kind of explosion* free! That is, E_1, E_2 and E_3 are three valid subtypes, and each one denotes a possible explosion. The axiom limits only the number of materialisations to 0 or 1.

4.5.5 Equivalences in Explosions

In the explosion process, it would be illusory to imagine that all the decompositions of a concept (whatever the criterion) are independent of the composite's neighbourhood. The most famous example is without contest the Statechart Diagram and we will inspect its meta-model depicted in Fig. 4.23(a). We will use peculiar graphical conventions for the sake of improving the readibility of the speciication: states and transitions will be represented respectively by ▭ and ◪. The diagram depicted in Fig. 4.23(b) shows two specifications. A state (2) is exploded into its specification (states 2_a, 2_b, 2_c and 2_d). With respect to the Statechart's semantics, one observes that transitions x and x' (and also y and y') express the same information and should be merged. Some approaches define a special mapping to denote an equivalence between classes in distinct specifications [Fin94b]. The diagram depicted in Fig. 4.23(c) shows another way to represent the original neighbourhood in the decomposition: the transitions have been merged and are now shared in both specifications. However, we observe that the `to` and `from` meta-relations between the `state` and `transition`

[1]See page 57.
[2]See page 54.

meta-classes become many-to-many. This semantic modification can be fixed in replacing these meta-relations with the composition of two one-to-many meta-relations.

Remark A many-to-many meta-relation can disturb the software engineer. Indeed, he is often working inside only one ontology, where a many-to-many meta-relation is not pertinent. For this purpose, the method engineer is free to define an explicit equivalence meta-relation between both concepts (i.e., the transitions).

4.5.6 Meta-Models about Meta-Models

One of the repository's strengths is the ability to speak about meta-models in the same way as about meta-classes. This principle has already been commented a lot in previous sections. We propose here some practices that fall exactly in our scheme.

A typical use of this would be a process documentation where method engineers would wish to maintain information about the products consumed/produced by the modelling activity. Such an information could serve to answer questions such as:

- Which requirements does a process use to produce this kind of diagram?
- Who created this diagram?
- Which are the diagrams of this modelling process?
- Which department required this element in this diagram?
- Which department is in charge of this task?
- Which use-cases define this task?

All these questions concern either meta-models and meta-classes or even meta-models and other meta-models.

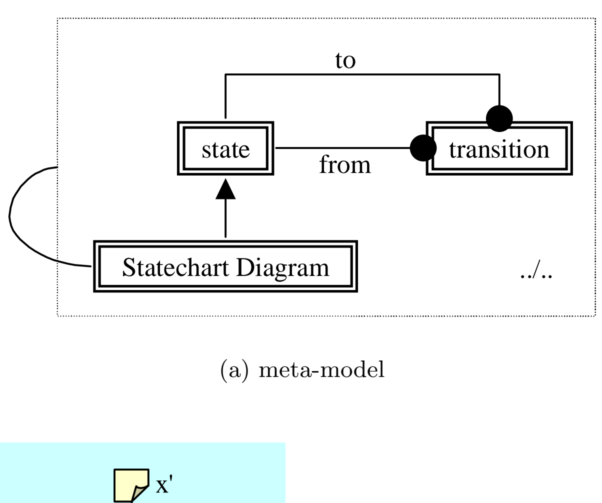
4.5.7 Meta-Relations and Meta-Models

In some projects (*cfr.* [KLR96] for instance), the method engineer can prevent the software engineer to use some meta-relations depending on the specification in use.

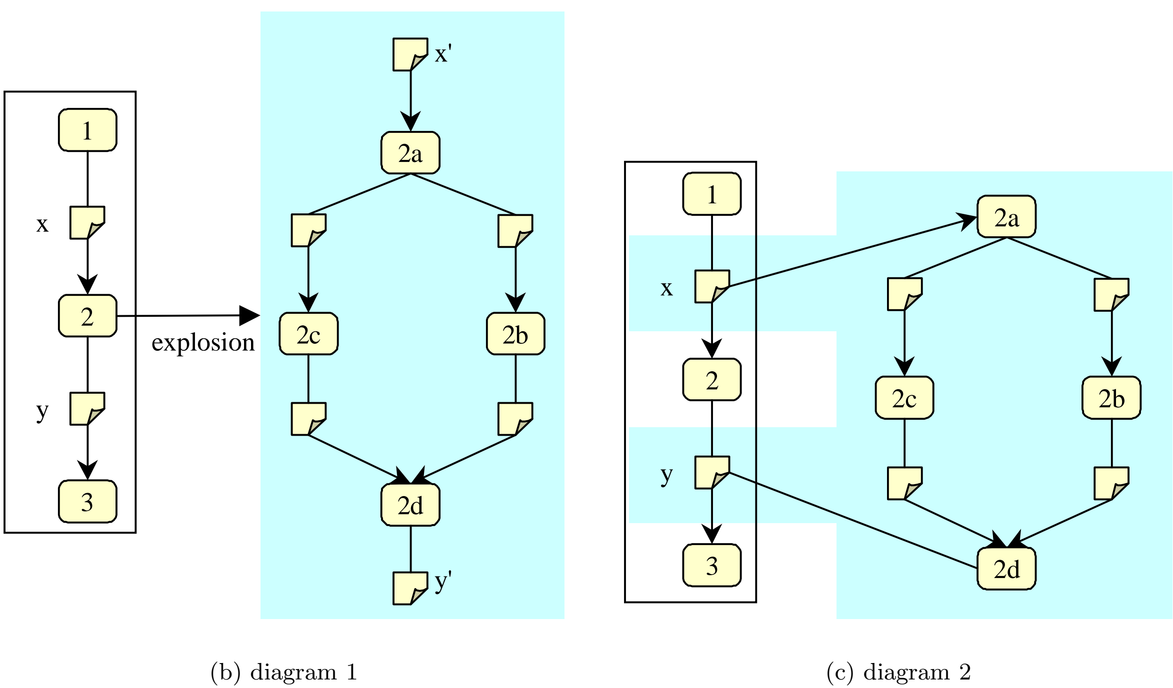
For instance, we can imagine two meta-models that represent respectively the conceptual and the logical analyses of a database. In both meta-models, the method engineer decides to represent the “entity types” and the “tables” concepts by the “generic” meta-class[1]. Two reflexive meta-relations are defined on the “generic” meta-class: *1)* “references” that means that one table has a foreign key wich references another table, and *2)* “inherits” that means that one entity-type inherits from other entity-types. Of course, “references” is meaningful only at the logical level although the “inherits” meta-relation is pertinent only at the conceptual level.

A way to take this way of modelling into consideration consists in adding the meta-relations in the definition of the meta-models. Hence, a sofware engineer can use a meta-relation only if it appears in the definition of the meta-model that is being utilized. But this adds necessarily additional constraints (i.e., axioms) to the repository definition.

[1] Conceptual, logical and physical concepts are often mapped on the same meta-classes. Only their interpretation varies depending on the context.



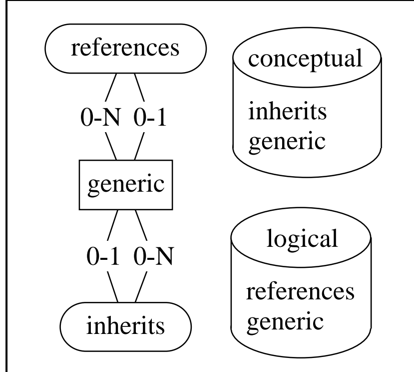
(a) meta-model



(b) diagram 1

(c) diagram 2

Figure 4.23: [**Equivalence in statechart diagrams**]



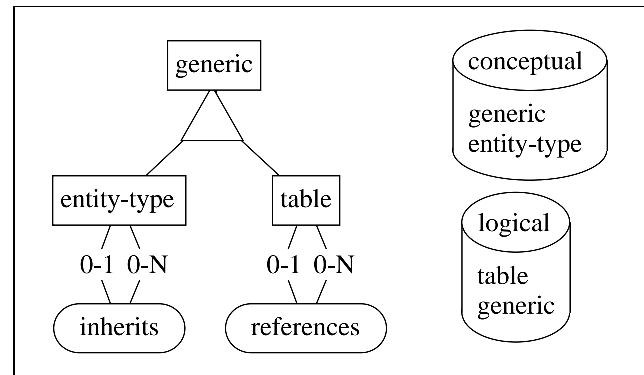


Figure 4.24: [**How to Limit Meta-Relations to Meta-Models**] The left example has been defined with a meta-meta-model that allows the method engineer to specify meta-relations in the meta-model definition. The latter has been transformed in order to remove the meta-relations from the meta-model definitions.

A way to by-pass this limit consists in deriving subtypes for every meta-model which needs specific add-ons. For instance, "generic" can be specialized into two subtypes which are specific to each meta-model. The roles of the meta-relation are then moved to the ad-hoc subtypes. This transformation is depicted in Fig. 4.24.

4.6 The Implementation of the Repository

4.6.1 Presentation

The main concepts introduced so far have been presented with a mathematical formalism. They describe a repository that will have to store not only meta-models descriptions but also their extensions. That is an extended class diagram that models both the meta-models and their specifications. This section will describe a possible implementation of this repository and we will use the extended entity-relationship paradigm for this issue. We will call the first description the *conceptual level* and the latter the *logical level*.

In order to avoid conflicts in the nomenclatures, each entity-type name at the logical level will be prefixed by the letter '*e*'. Sets MetaClass, MetaRelation, MetaProperty, MetaModel, Class, Specification and Method are represented by the `eMetaClass`, `eMetaRelation`, `eMetaProperty`, `eMetaModel`, `eClass`, `eSpecification` and `eMethod` entity-types. The "super " function is denoted by the composition of the `supertype_of` and `subtype_of` relationships. Functions "owner", "member", "Prop ", and "Meth " are represented respectively by the `owner`, `member`, `prop` and `methods` relationships. The Mod function is represented by the composition of the `belongs_to` and `contains` relationships. The "Def" function is represented by the composition of the `def_class` and `def_spec` relationships.

Since the repository is defined with more or less the same concepts as a meta-model, one will still abuse the notation defined in the beginning of this chapter. Nevertheless, some minor differences exist between both interpretations:

1. In the `eClass` entity-type, the types of the attributes are no longer in the ValUniv set only and can denote classes. Nevertheless, this detail is not disturbing.

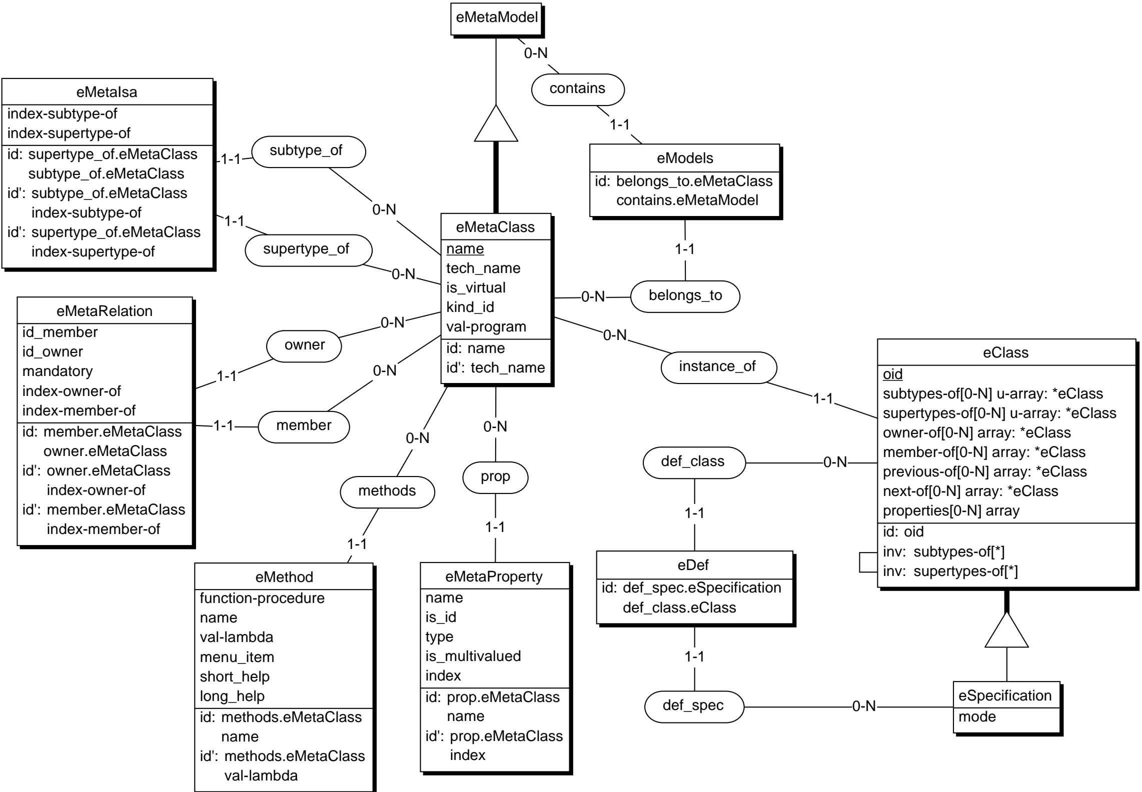


Figure 4.25: **[Repository (logical level)]** The graphical conventions follow the DB-MAIN's rules: `id` and `id'` denote identifiers (and no longer local or global identifiers); cartouches denote relationships and the cardinatilties of the roles follow the ERA conventions; attributes `A[0-N] array: *T` denote arrays of references to a type T; `inv` constraint denotes an inverse constraint between references (as in some OODBMS); and `u-array` means that the array has no duplicates.

2. The extension of each entity-type contains a special value named `void` which is distinct from all the other values.

3. Arrays will be represented by sequences. The empty slots of the array will be represented by the `void` value in the sequence.

Henceforth, we will describe the schema in figure 4.25 in terms of meta-classes, meta-relations, meta-properties, ... as in the beginning of this chapter.

4.6.2 Details

Let us examine how this meta-model stores the specifications.

Relations

Meta-properties `owner-of`, `member-of`, `previous-of` and `next-of` of the `eClass` meta-class implement the $\Re$ function. The `index-owner-of` and `index-member-of` meta-properties of the `eMetaRelation` meta-class denote indices that will be used in the `eClass` meta-class to store the $\Re$ function. Let us inspect one well-known proposition borrowed from the beginning of the chapter:

$$\Re_R(o) = [m_1, \ldots, m_n]$$

R will be represented by a class of the `eMetaRelation` meta-class and classes o, m_1, ... ,m_n will be represented by instances of the `eClass` meta-class. The next paragraph will describe the translation of this proposition in terms of our logical meta-model.

Let $i_o = \xi_{\texttt{index-owner-of}}(R)$ and $i_m = \xi_{\texttt{index-member-of}}(R)$ two indices of R. Then the proposition will be stored as follows:

$$\begin{aligned}
\forall j \in \{1 \ldots n\} &: \xi_{\texttt{member-of}}(m_j)[i_m] = o \\
\forall j \in \{1 \ldots n-1\} &: \xi_{\texttt{previous-of}}(m_j)[i_m] = m_{j+1} \\
\forall j \in \{2 \ldots n\} &: \xi_{\texttt{next-of}}(m_j)[i_m] = m_{j-1} \\
& n = 0 \;\otimes\; \xi_{\texttt{next-of}}(m_1)[i_m] = \texttt{void} \\
& n = 0 \;\otimes\; \xi_{\texttt{previous-of}}(m_n)[i_m] = \texttt{void} \\
& \big(n > 0 \;\wedge\; \xi_{\texttt{owner-of}}(o)[i_o] = m_1\big) \;\otimes\; \big(\xi_{\texttt{owner-of}}(o)[i_o] = \texttt{void}\big)
\end{aligned}$$

The size of the arrays `previous-of`, `next-of` and `member-of` (resp. `owner-of`) of meta-class `eClass` is computed as the greatest value of the `index-member-of` (resp. `index-owner-of`) properties of the corresponding `eMetaRelation` classes.

Inheritance Graph

The $\mathfrak{isa}$ relation is implemented with the `subtype-of` and `supertype-of` meta-properties of the `eClass` meta-class. The elements of these arrays denote respectively the supertypes and subtypes corresponding to the index of the `eMetaIsa` class which represents the inheritance between both corresponding meta-classes. The size of these arrays is computed on basis of the greatest index of the instances of `eMetaIsa` for which the meta-class plays respectively the role of subtype/supertype.

Properties

The ξ function is implemented with the `properties` meta-property of `eClass`. A proposition like $\xi_P(x) = v$ will be stored as $\xi_{\texttt{properties}}(x)[i] = v$ where $i = \xi_{\texttt{index}}(P)$.

Methods

Meta-class `eClass` is endowed with methods that allow the meta-CASE architect to navigate throughout the repository. In this section, one will abuse the similarity between the instances of the conceptual and the logical level to confuse them. Hence, in some places, we will not distinguish elements from MetaRelation and `eMetaRelation` for instance. The adoption of such a strictness would lead us to too complicated propositions without any interest.

`function eClass`: r eClass::`GetFirstRelation` (`eClass`: c, `eMetaRelation`: R)

Precondition. $c \neq \texttt{void} \ \wedge \ R \neq \texttt{void} \ \wedge \ \Gamma(c) = R.\text{owner}$

Postcondition. $r = \texttt{void} \ \otimes \ \Re_R(c) = [r, \ldots]$

`function eClass`: r eClass::`GetNextRelation` (`eClass`: c, `eClass`: p, `eMetaRelation`: R)

Precondition.

- $c \neq \texttt{void} \ \wedge \ p \neq \texttt{void} \ \wedge \ R \neq \texttt{void}$
- $\Gamma(c) = R.\text{owner}$
- $\Re_R(c) = [\ldots, p, \ldots]$

Postcondition. $r = \texttt{void} \ \otimes \ \Re_R(c) = [\ldots, p, r, \ldots]$

`function eClass`: r eClass::`GetOwnerRelation` (`eClass`: c, `eMetaRelation`: R)

Precondition. $c \neq \texttt{void} \ \wedge \ R \neq \texttt{void} \ \wedge \ \Gamma(c) = R.\text{member}$

Postcondition. $r = \texttt{void} \ \otimes \ \Re_R(r) = [\ldots, c, \ldots]$

`procedure` eClass::`AddFirstRelation` (`eClass`: c, `eClass`: m, `eMetaRelation`: R)

Precondition.

- $c \neq \texttt{void} \ \wedge \ m \neq \texttt{void} \ \wedge \ R \neq \texttt{void}$
- $\Gamma(c) = R.\text{owner}$
- $\Gamma(m) = R.\text{member}$
- $\forall x \in \Gamma^{-1}(R.\text{owner}) : m \notin \Re_R(x)$

Postcondition. $\Re_{\mathsf{new}\,R}(c) = [m] + \Re_{\mathsf{old}\,R}(c)$

`procedure` eClass::`AddNextRelation` (`eClass`: c, `eClass`: p, `eClass`: m, `eMetaRelation`: R)

Precondition.

- $c, p, m \neq \texttt{void} \;\wedge\; R \neq \texttt{void}$
- $\Gamma(c) = R.\text{owner}$
- $\Gamma(m) = \Gamma(p) = R.\text{member}$
- $\Re_R(c) = [x_1, \ldots, x_\alpha, p, y_1, \ldots, y_\beta]$
- $\forall x \in \Gamma^{-1}(R.\text{owner}) : m \notin \Re_R(x)$

Postcondition. $\Re_{\mathsf{new}\,R}(c) = [x_1, \ldots, x_\alpha, p, m, y_1, \ldots, y_\beta]$

`procedure` eClass::`RemoveRelation` (`eClass`: m, `eMetaRelation`: R)

Precondition.

- $m \neq \texttt{void} \;\wedge\; R \neq \texttt{void}$
- $\Gamma(m) = \Gamma(p) = R.\text{member}$

Postcondition. $\Re_R^{-1}(m) = \emptyset$

`function eClass`: r eClass::`GetSupertype` (`eClass`: c, `eMetaIsa`: I)

Precondition.

- $c \neq \texttt{void} \;\wedge\; I \neq \texttt{void}$
- $I \in \Re_{\texttt{subtype-of}}(\Gamma(c))$

Postcondition. $r = \xi_{\texttt{subtype-of}}(c)[\xi_{\texttt{index-subtype-of}}(I)]$

`function eClass`: r eClass::`GetSubtype` (`eClass`: c, `eMetaIsa`: I)

Precondition.

- $c \neq \texttt{void} \;\wedge\; I \neq \texttt{void}$
- $I \in \Re_{\texttt{supertype-of}}(\Gamma(c))$

Postcondition. $r = \xi_{\texttt{supertype-of}}(c)[\xi_{\texttt{index-supertype-of}}(I)]$

`procedure` eClass::`UpdateISA` (`eClass`: c, `eClass`: s, `eMetaIsa`: I)

Precondition.

- $I \neq \texttt{void}$
- $c \neq \texttt{void} \;\vee\; s \neq \texttt{void}$
- $c \neq \texttt{void} \;\Rightarrow\; I \in \Re_{\texttt{subtype-of}}(\Gamma(c))$
- $s \neq \texttt{void} \;\Rightarrow\; I \in \Re_{\texttt{supertype-of}}(\Gamma(s))$

Postcondition.

- $c \neq \texttt{void} \;\Rightarrow\; \xi_{\texttt{subtype-of}\,\mathsf{new}}(c)[\xi_{\texttt{index-subtype-of}}(I)] = s$

- $s \neq \texttt{void} \Rightarrow \xi_{\texttt{supertype-of}_{\mathsf{new}}}(s)[\xi_{\texttt{index-supertype-of}}(I)] = c$

function eSpecification: r eClass::GetFirstDef (eClass: c)

Precondition. $c \neq \texttt{void}$

Postcondition. $r = \texttt{void} \otimes r \in \mathrm{Def}^{-1}(c)$

function eSpecification: r eClass::GetNextDef (eClass: c, eSpecification: p)

Precondition. $c, p \neq \texttt{void} \wedge p \in \mathrm{Def}^{-1}(c)$

Postcondition. $\big(\mathrm{Def}^{-1}(c) = [\ldots, p] \wedge r = \texttt{void}\big) \otimes \big(\mathrm{Def}^{-1}(c) = [\ldots, p, r, \ldots]\big)$

Remark The result of the $\mathrm{Def}^{-1}(.)$ function is obviously a set and not a sequence. The sequence that is suggested in this proposition, results from the operational order.

function ValUniv : r eClass::GetProperty (eClass: c, eMetaProperty: p))

Precondition. $p \neq \texttt{void} \wedge c \neq \texttt{void} \wedge p \in \mathrm{Prop}(\Gamma(c))$

Postcondition. $r = \xi_p(c)$

procedure eClass::UpdateProperty (eClass: c, eMetaProperty: p, ValUniv: v)

Precondition.

- $c \neq \texttt{void} \wedge p \neq \texttt{void}$
- $p \in \mathrm{Prop}(\Gamma(c))$
- the type of $v = p.\texttt{type}$
- the multiplicity of $v = p.\texttt{multi}$

Postcondition. $\xi_{p_{\mathsf{new}}} = \xi_{p_{\mathsf{old}}}\langle c \mapsto v \rangle$

The eSpecification meta-class owns its own methods defined as:

function eClass: r eSpecification::GetFirstComponent (eSpecification: s)

Precondition. $s \neq \texttt{void}$

Postcondition. $r \in \mathrm{Def}(s)$

function eClass: r eSpecification::GetNextComponent (eSpecification: s, eClass: p)

Precondition. $s, p \neq \texttt{void} \wedge p \in \mathrm{Def}(s)$

Postcondition. $\mathrm{Def}(s) = [\ldots, p, r, \ldots] \otimes \big(\mathrm{Def}(s) = [\ldots, p] \wedge r = \texttt{void}\big)$

Remark Same remark as for the eClass::GetNextDef method.

procedure eSpecification::AddComponent (eSpecification: s, eClass: c)

Precondition. $c, s \neq \texttt{void} \wedge \Gamma(c) \in \mathrm{Mod}(\Gamma(s))$

Postcondition. $c \in \mathrm{Def}(s)$

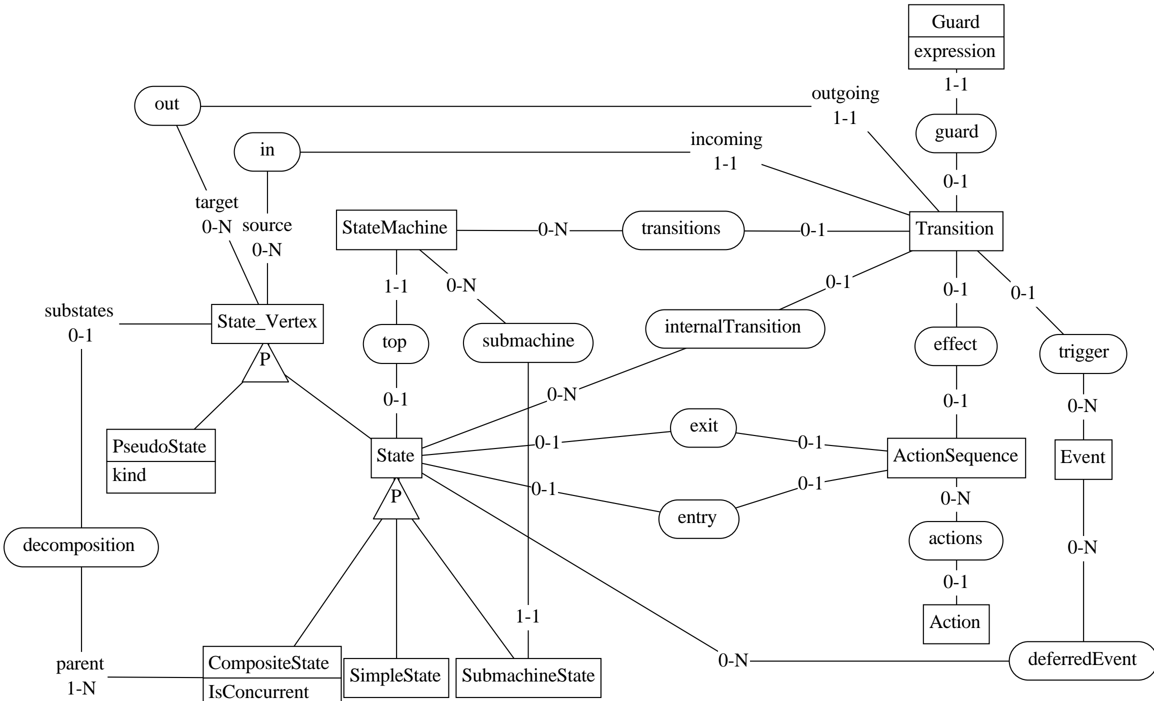


Figure 4.26: **[Statechart Meta-Model]** This schema denotes the Statechart meta-model described in [Rat97b]

4.7 Case Studies

4.7.1 a Statechart Meta-Model

This section presents a Statechart meta-model. This study is directly inspired from the UML reference manual [Rat97b] and should convince the reader that even industrial approaches can be modelled with our concepts. The original meta-model definition is presented in Fig. 4.26. We will transform it in order to exploit the semantics of our repository. Each step of this process is described in the next paragraphs and the resulting schema is depicted in Fig. 4.27

1. The `deferredEvent` many-to-many meta-relation is replaced with two one-to-many meta-relations. The `deferredEvent` meta-relation becomes a meta-class.

2. > "*top designates the top level* State *directly owned by the* StateMachine*. Other* States *are owned by the parent composite states. The multiplicity is 1, there must be one* State *designated as the top* State*. The rest of the* StateMachine *is an expansion of this* CompositeState*. [. . .] The top state is always a composite.*"

   The `top` association is specialized with respect to its semantics: one role is moved from `State` to `CompositeState`.

3. The `PseudoState` can be specialized into several subtypes that will explicit the criterion `kind`. The `kind` meta-property is enumerated and can be set to `Initial`, `DeepHistory`, `ShallowHistory`, `Join`, `Fork`, `Branch` or `Final`. These values will provide as much subtype meta-classes as there are values.

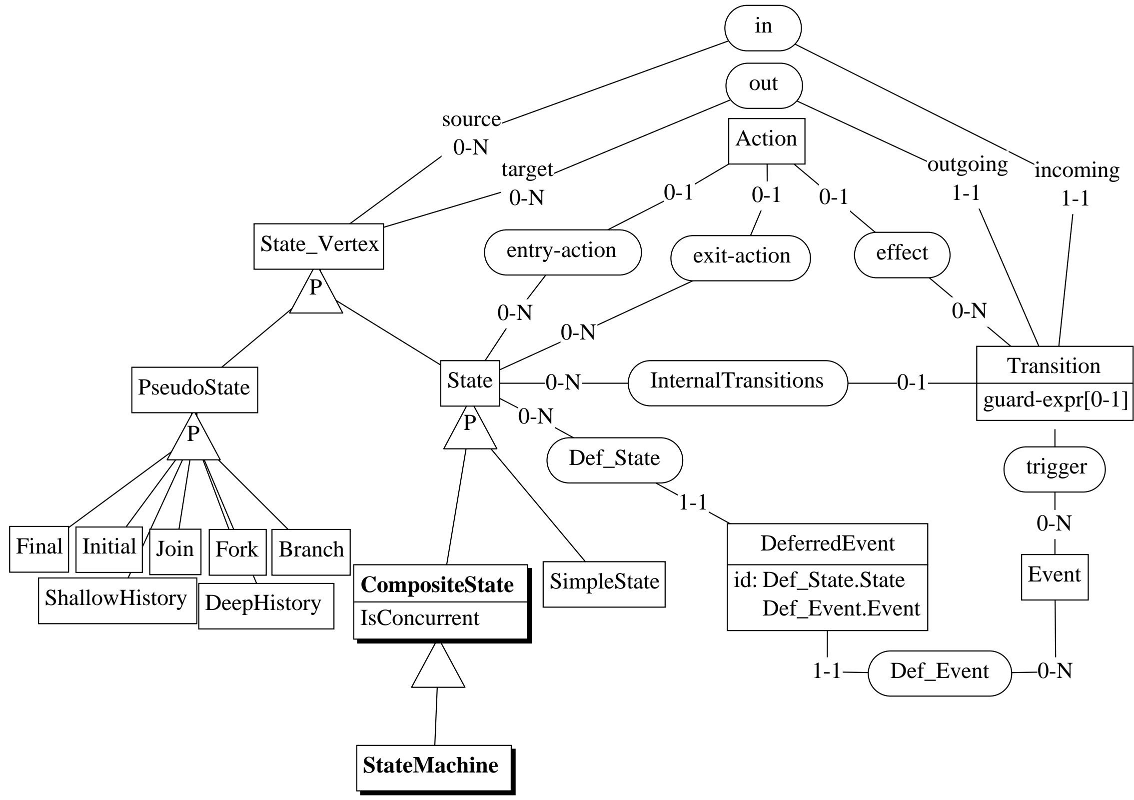


Figure 4.27: [**Revised Statechart Meta-Model**] This schema denotes the Statechart meta-model after transformation. It takes advantage of our meta-meta-model's expressiveness to simplify its complexity. The `CompositeState` and `StateMachine` meta-models are defined in terms of all the illustrated meta-classes except the `StateMachine` meta-class

4. *"A composite state is a state that consists of substates. In the metamodel a CompositeState is a subclass of State that contains one or more substates that are subtypes of StateVertex."*

   We can transform the `CompositeState` into a meta-model with the `StateVertex` meta-class in its definition. Since `StateVertex` is a virtual meta-class, we easily deduces from the described semantics that its subtypes can be added in the meta-model's definition as well as the `Transition` meta-class. The `decomposition` meta-relation disappears and becomes implicit.

5. *"The StateMachine has a composition aggregation to a State that represents the top state and a set of Transitions. As a consequence the StateMachine owns its Transitions and its top State, but nested states are transitively owned through their parent States."*

   The `top` state is necessarily composite and each `StateMachine` must own a top state. Hence, we prefers to replace the `top` association between `StateMachine` and `CompositeState` with a `isa` relationship such that a state machine inherits the characteristics from a composite state. In the same way, this definition suggests that a state machine behaves like a specification whose components would be transitions. This is clarified in our repository in transforming `StateMachine` into a meta-model whose the definition is composed of `Transition` meta-classes. The `transitions` meta-relation disappears and becomes implicit.

6. *"A SubmachineState represents a nested state machine. A nested state machine is semantically equivalent to a composite state, but facilitates reuse and modularity in the form of an independent nested state machine. [...] The semantics of a submachine state is equivalent to the semantics of replacing the submachine state with the state machine related by the submachine association, where the top state of the submachine merges with the submachine state, resulting in a composite state."*

   The merging process of the `SubmachineState` meta-class, the `top` association and the `StateMachine` encourages our transformation approach of this UML meta-model into the new one described so far. The `submachine` association denotes an equivalence relationship between a state machine and its sub-machines. This equivalence is a good argument to merge them into a single meta-class: `StateMachine`. Because `StateMachine` is already a subtype of `State`, we have just to remove the `SubmachineState` meta-class from our schema!

7. The `Guard` meta-class satisfies the preconditions to become a direct meta-property of the `Transition` meta-class. Nevertheless, the `ModelElement` supertype[1] of both `Transition` and `Guard` could prevent us to apply this transformation. But this is not bothering since the aim of this meta-class is to endow its subtypes of meta-facilities. Because this mechanism is implicit in our repository, this facility does not have to be clarified.

8. The `ActionSequence` meta-class is removed from the schema and its associations are linked together by composition. Hence, we define `exit-actions` as `exit` $\circ$ `actions` and `entry-actions` as `entry` $\circ$ `actions`. The `effect` meta-relation is now linked directly to the `action` meta-class and is now one-to-many. Although there is no clear evidence of

[1] Its is not illustrated in our schema. The reader can refer to [Rat97b] for more information.

this constraint in UML 1.1, we suppose that actions are not shared between transitions and states (this is clearly indicated in UML 1.3). This makes this transformation possible. Of course, the `ModelElement` could add some semantics to an action of sequence, and reenforces its semantics autonomy but this add-on could also be added indirectly to `State`.

The result is depicted in Fig. 4.27. If we omit the seven new meta-classes created at step 3 (this transformation is independent of our approach), we have "lost" 3 meta-classes and 5 meta-relations. Nevertheless, the meta-model has no identifier and it should be very difficult to define them since most of the meta-classes have no meta-properties. We note that the main concepts inherit from the `ModelElement` meta-class that can have several `TaggedValue` meta-classes which act like hidden meta-properties. Moreover, we have observed several divergences between the presented meta-model and its current use in the Rational Rose CASE tool:

- States have a unique name. This property is not mentioned in the documentation.
- The `event` meta-class is implemented like a meta-property attached to the `Transition` meta-class.
- The `deferredEvent` and `internalTransition` meta-relations are not implemented.
- The `Action` meta-class is used in a different way depending of whether it is used in a transition or in a state.

We have showed in this section how the reengineering of existing meta-models can be done with respect to the constraints and the expressiveness of our repository. Moreover, some semantics constraints could be expressed directly via local identifiers. For instance, the documentation mentions that the initial state should be unique, this constraint can be expressed with an empty identifier attached to the `Initial` meta-class.

As we showed in the previous paragraphs, the studied meta-model is sometimes too complete concerning its use and sometimes not enough described (some meta-properties were lacking). For these reasons, we propose another meta-model that should suit the common needs. Moreover, this meta-model encompass a large part of the current meta-models [HSB98]. Its schema is depicted in Fig. 4.28. One will call it the *normal statechart meta-model*.

4.7.2 The Overview Meta-Model

This section emphasizes the power of meta-models for modelling software engineering methodologies made up of several abstraction levels. The meta-model depicted in Fig. 4.29 describes an hypothetical methodology that would be the integration of statecharts, organizational diagrams, database schemas, traceability, use cases, ...

4.8 Summary

So far we have presented a modelling language[1] (i.e., the meta-meta-model) to define meta-models. As for any new language, we have introduced new words and new grammatical rules. Our will to keep the language as simple as possible leads us to limit its vocabulary to

[1] Its syntax was out of the scope of this thesis and we have presented only its graphical equivalent.

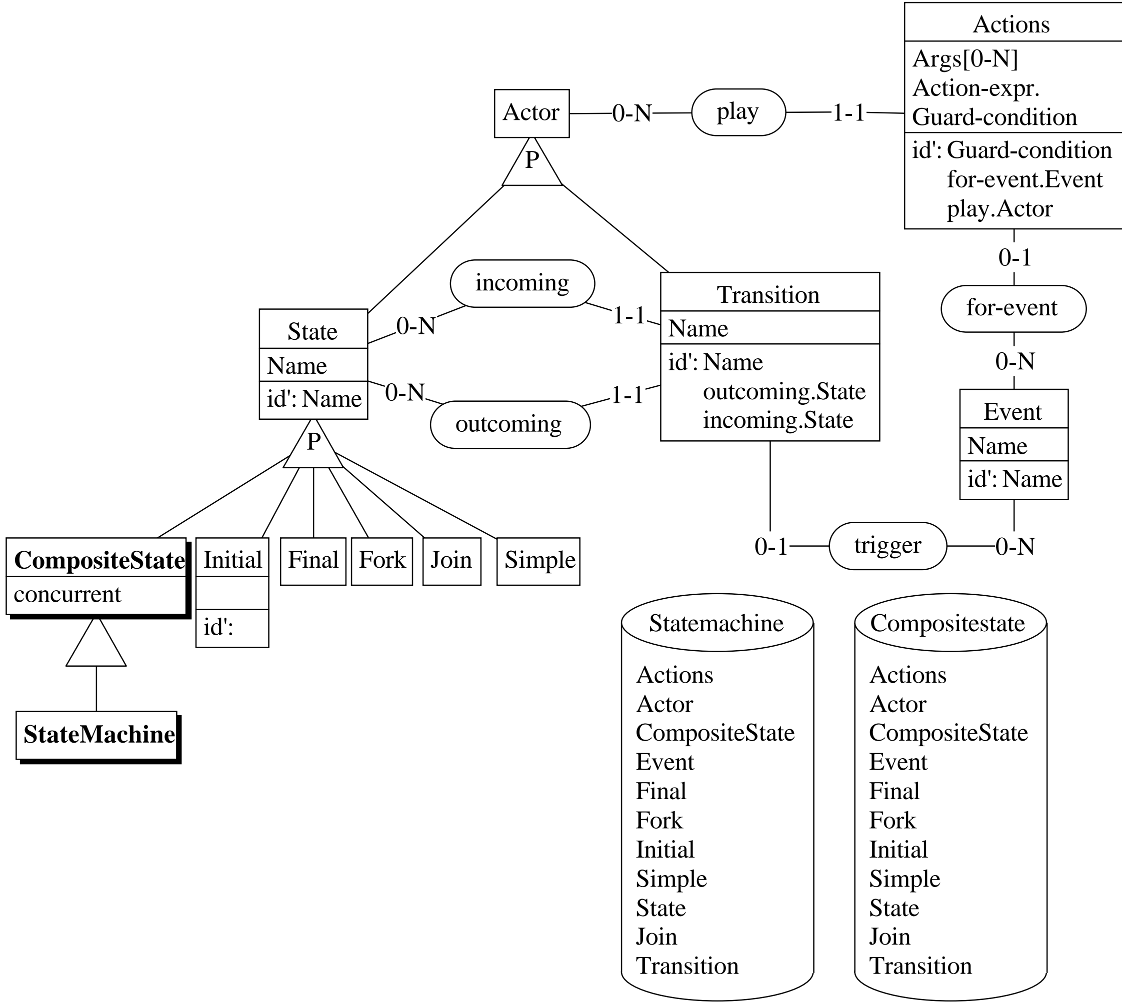


Figure 4.28: **[The Normal Statechart Meta-Model]** This meta-model is inspired by the UML reference documents [Rat97a, Rat97b] and the use of Rational Rose. A *StateMachine* is a *CompositeState*. They are both meta-models defined in terms of *State*, *Transition*, *Event*, *Actions* and other related meta-classes. A concurrent (*CompositeState.concurrent*=true) *CompositeState* should be composed of only *CompositeStates*. The *name* of a transition identifies it between two states. Transitions and states may have actions that should be attached to an *event*. To avoid misspelled names, we define the *Event* meta-class as a directory. Let us note that a state can be composite, and hence, this meta-model supports composite states

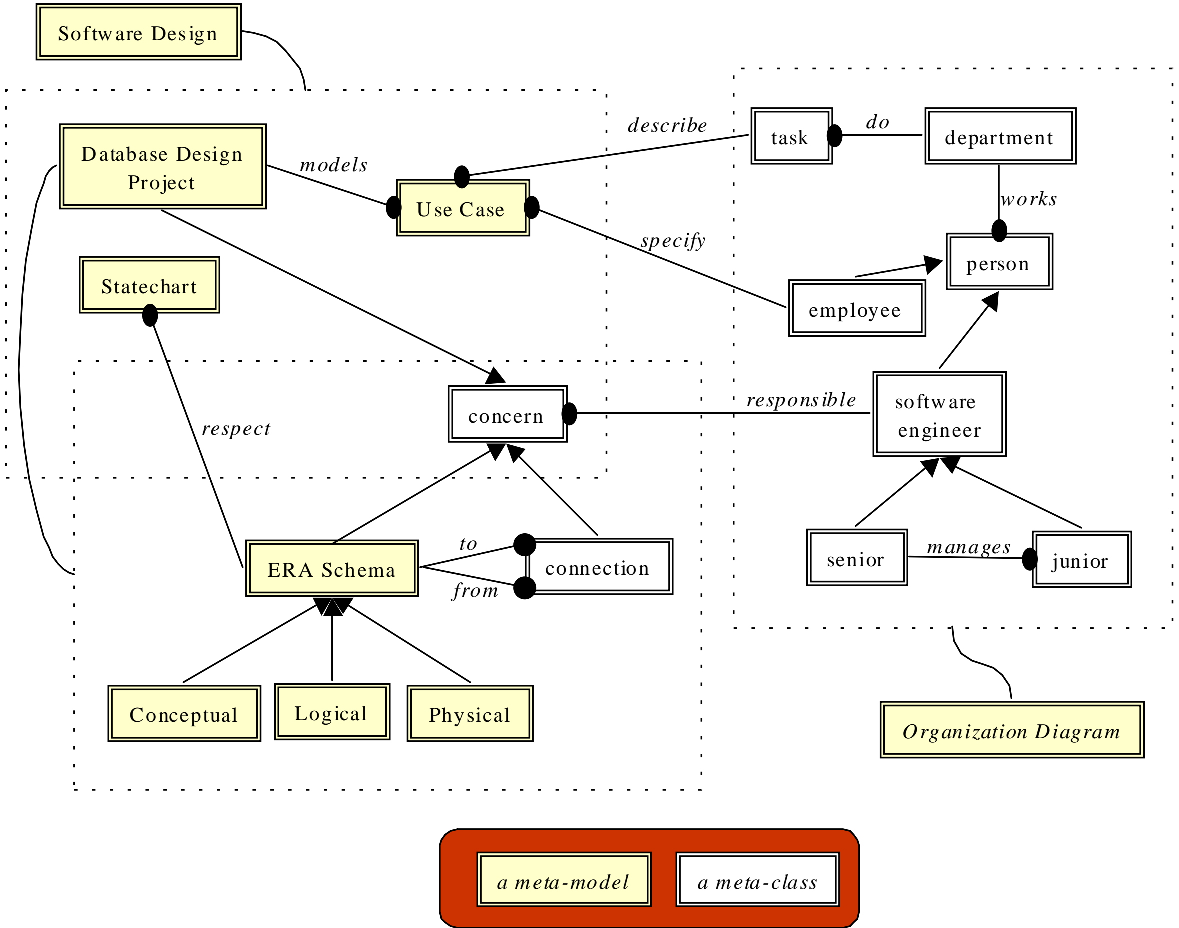


Figure 4.29: [**The Overview Meta-Model**]

5 words: meta-class, meta-relation, meta-property, method, and meta-model. The latter is the only word that makes this language specific to meta-modelling techniques. Meta-models make it possible to talk about specifications as complex objects that can denote aggregated information. In spite of an axiomatic definition, the graphical conventions for representing the meta-models with this language make this job very easy.

The definition of all the language's features has been strongly guided by the the observation of the intrinsic nature of a software engineering project. Let us recall here some important correspondences between the characteristics of this language and the requirements (i.e., observations) that have motivated their existence:

Meta-models are meta-classes: A software engineering project can be depicted as a hypergraph (often due to top-down analyses).

Sharing between meta-models : Specifications often overlap and share common information.

Local/global complex identifiers : Identifiers can be made of meta-properties and roles. Moreover, the meta-model can be added to its definition (local identifier).

Dynamic inheritance : Refinement processes are very fond of this mechanism: they change the type of a concept as they make progress.

Multiple specialization : A concept can denote two orthogonal information in two meta-models. Each information is specific to its meta-model (e.g., a `data` can be both an `entity-type` in an ERA schema and a `datastore` in a data flow diagram).

Multiple inheritance : This feature has been observed in several places (e.g., MOF, DB-MAIN, Martin/Odell, ...).

Inheritance with exceptions : Exceptions generalize the inheritance semantics to model intricate situations. Moreover, this makes it possible to add meta-classes as new supertypes in an inheritance graph although the subtype has already an extension.

The meta-CASE exploits all these features to interpret and to bring life to the models defined by those meta-models. For instance, sharing, multiple specialization and dynamic inheritance will be used to integrate specifications (in either a posteriori or a priori decisions) and identifiers will make it possible to give a name to concepts.

Chapter 5

"Mirrors should think longer before they reflect."
JEAN COCTEAU

The Mirroring Service

5.1 Presentation

This presentation will introduce the *Mirroring Service* step by step. It will present the general idea, bring some problems up and state the solutions.

The general idea of the *Mirroring Service* is to duplicate the repository of a CASE tool in the meta-CASE and to synchronize their extensions. The synchronization makes it possible to make any operation on any of them, whatever the repository, and to keep them consistent. But the mirroring should make it also possible to extend the semantics of the duplicated meta-model with concepts that the original CASE tool ignores.

To illustrate this mechanism, we will use one CASE tool called "*Guinea-Pig*" for this purpose. This tool will have a two layers architecture (as many other CASE tools), namely, a meta-model C_2 and the specification layer C_1. Now as a style exercise, let us reproduce C_2 in the meta-CASE. We call it M_2. Figure 5.1 shows the different planes (models) of both tools.

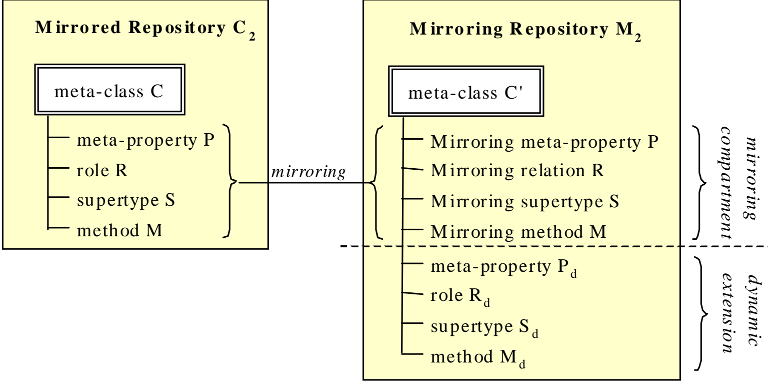


Figure 5.1: [**CASE and Meta-CASE Architecture**] The left column represents a CASE tool architecture and the right column the meta-CASE tool architecture. Each plane denotes a model, i.e., an abstraction level.

We have now to face an interesting problem. Let us inspect a specification s which is an instance of either C_2 or M_2. s could be stored *1)* in Guinea-Pig *2)* in the meta-CASE or *3)* in both! Only the last case interests us. Indeed, although the first cases bring no new advantages, the last one will allow us to use all the benefits of the meta-CASE on a specification that is stored in both places. Nevertheless, the last hypothesis poses a problem:

how can we garantee that these specifications will be synchronous?

To achieve this synchronisation, we have endowed the meta-classes with the capability to *mirror* meta-classes from other systems (Guinea-Pig for instance). Mirroring meta-classes will not only emulate the mirrored meta-classes but will also let the possibility to extend their semantics with new meta-relations, new meta-properties, and new supertypes as for any other meta-classes! Mirroring meta-classes will encapsulate mirrored meta-classes and inherit their semantics. Hence, if the C meta-class belongs to the Guinea-Pig meta-model C_1, another meta-class C' will belong to the meta-model M_2 and C' will mirror C. In order to maintain their extension synchronous, the meta-class C will remain the only effective manager of its intrinsic semantics (meta-properties, meta-relations and supertypes) and these characteristics will be visible in the mirroring meta-class as *virtual*[1]. Mirrored meta-classes serve their characteristics to the mirroring meta-class as in a Client/Server architecture.

Instances of the C meta-class will only exist on the Guinea-Pig side. Nevertheless, on the meta-CASE side, meta-class C' will have as many elements in its extension as C, and every class will be strongly connected with the instances of C, and vice-versa. This bi-directional link will allow the system to maintain both tools synchronously.

Mirroring meta-classes will then have two "semantic compartments". The *mirroring compartment* will be a collection of *virtual* meta-properties, meta-relations, supertypes and methods. They represent the characteristics/semantics that exist on the other side of the "mirror" (i.e., on the Guinea-Pig side). The *dynamic compartment* is composed of characteristics that only exist on the meta-CASE side and that enrich the meta-class: new meta-properties, new meta-relations, new supertypes and new methods. This schema is illustrated in Fig. 5.2.

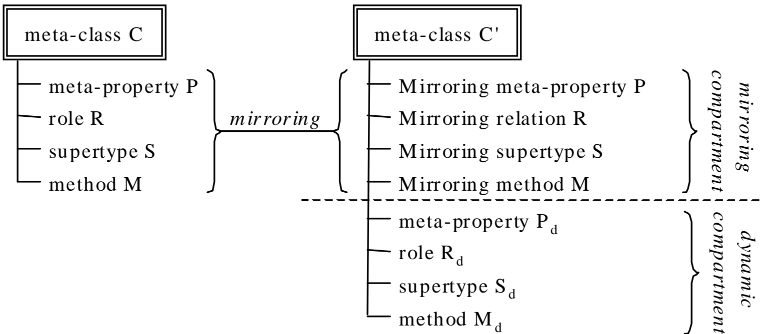


Figure 5.2: **[The Mirroring and Dynamic Compartments]** This schema shows two meta-classes. C is a meta-class from the mirrored CASE tool and C' is the corresponding mirroring meta-class. P, R, S, and M are here four possible characteristics of C that are mirrored in C'. P_d, R_d, S_d, and M_d are four new characteristics that were added to C'. They do not exist at all in C and, hence, are not visible in the mirrored CASE tool.

Example Let us suppose that Guinea-Pig is modelling computer networks. Let us suppose that `computer` is a meta-class of Guinea-Pig's meta-model, with `name` as meta-property, and with a role in a meta-relation `connected-to` with `network`. Such a meta-class could be mirrored in the meta-CASE with all these characteristics. Moreover, the method engineer could reuse this mirroring meta-class to enrich his meta-models. Hence, he could *a)* add a dynamic meta-relation `stores` between `computer` and the

[1]Virtual means here: something that does not really exist.

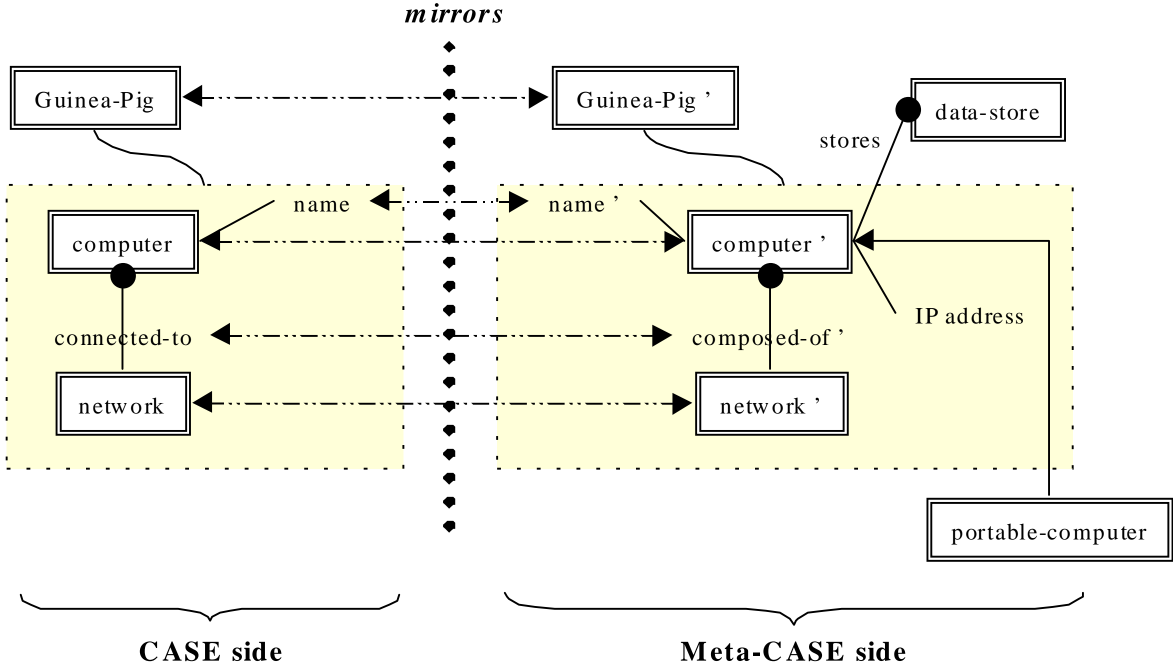


Figure 5.3: [**The "Guinea-Pig" Example**]

`data-store` meta-class, *b)* add a dynamic meta-property `IP address` to `computer`, and even *c)* derive a new subtype `portable-computer` from `computer`. This situation is described in Fig. 5.3.

5.2 The Mirroring Service

The mirroring service will thus marry two distinct systems (i.e., repositories) and synchronize their respective extensions. This section defines the axioms that will guarantee that *a)* both systems are synchronous and *b)* the repository's axioms are preserved on the meta-CASE side.

We will thus discuss about two systems: a CASE tool and the meta-CASE. We make the hypothesis that the CASE tool can be described as a set of meta-classes that belong to the Ω meta-model definition. Moreover, there is no other meta-model. The CASE tool's repository is thus flat: $\mathsf{Specification} = \{\omega\}$. In other words, we suppose that the CASE tool can be described in terms of meta-classes, meta-relations, meta-properties and inheritance links.

Since these two systems are distinct, we have to consider two repositories and that means: two Ω meta-models, two ξ functions, two Class sets and so on. We call the repository of the meta-CASE "META" and the other one "CASE". In order to avoid any ambiguity when we will write the propositions relative to the mirroring service, we introduce a new notation that will precise the context of a proposition.

Notation 6 ($\left(P\right)_R$)
If R denotes a repository, then the notation $\left(P\right)_R$ means that proposition P is relative to

repository R.

Example

1. $A \neq B \Rightarrow \forall x \in \left(\mathsf{Class}\right)_A : x \notin \left(\mathsf{Class}\right)_B$ *and vice versa.*
2. $A \neq B \Rightarrow \left(\mathsf{Class}\right)_A \cap \left(\mathsf{Class}\right)_B = \emptyset$.

These two propositions express the same idea: if two repositories[1] *are distinct, then they do not share classes.*

Definition 31 (mirrors)
We say that a concept X in a system A mirrors another concept X' in a system A', when the first one emulates the second one. The problem that interests us will concern meta-classes, meta-properties and, meta-relations between the CASE *and the* META *systems. We will thus define the infix "mirrors" relation as:*

$$mirrors \subseteq \left(\mathsf{MetaClass}\right)_{\text{META}} \times \left(\mathsf{MetaClass}\right)_{\text{CASE}} \cup \left(\mathsf{MetaProperty}\right)_{\text{META}} \times \left(\mathsf{MetaProperty}\right)_{\text{CASE}} \cup \left(\mathsf{MetaRelation}\right)_{\text{META}} \times \left(\mathsf{MetaRelation}\right)_{\text{CASE}}$$

mirrors is a 1/1-relation. A meta-class can mirror only one other meta-class and vice versa.

The "mirrors" relation will establish a strong link between two meta-classes. Nevertheless, this relation would be useless without the existence of another relation within their respective instances that represent the materialization of the mirroring principle.

Definition 32 (The $\rightsquigarrow$ relation — $\tilde{x}$)
The infix $\rightsquigarrow$ relation is defined between the classes of two systems ($\rightsquigarrow \subseteq \left(\mathsf{Class}\right)_A \times \left(\mathsf{Class}\right)_B$). When two classes a and b are in relation ($a \rightsquigarrow b$), this denotes that b will be the representation of the class a in the system B. $\rightsquigarrow$ is a 1/1-relation. We note $\tilde{x}$ the element that satisfies either the proposition $x \rightsquigarrow \tilde{x}$ or $\tilde{x} \rightsquigarrow x$ depending on the context. $\tilde{x}$ denotes the mirror of x in the other system.

Axiom 37 (The mirroring meta-model exists)
There exists a meta-model M (in the meta-CASE) which represents the modelled CASE tool. We say that M mirrors CASE *and we call this meta-model* CASE′. *Its aggregation type is* `strong`.

$$\exists M \in \left(\mathsf{MetaModel}\right)_{\text{META}},\ \forall C \in \left(\mathsf{MetaClass}\right)_{\text{CASE}},\ \forall C' \in \left(\mathsf{MetaClass}\right)_{\text{META}} : C' \text{ mirrors } C \ \Rightarrow\ C' \in \left(\mathrm{Mod}(M)\right)_{\text{META}}$$

Remark The axiom does not prevent other meta-models to contain these mirroring meta-classes

[1] Those repositories are denoted by A and B in the propositions.

Axiom 38 (The mirroring specification exists)
The extension of the $\textsc{case}'$ *meta-model has at least one element* case'

$$\text{case}' \in \Gamma^{-1}(\textsc{case}')$$

$\textsc{case}'$ (resp. case') is the meta-model (resp. the specification) which represents the CASE tool (resp. its state) in the meta-CASE. We now have all the elements to build the mirroring service. The next axioms will ensure us that the mirroring service will do its job: no less and no more.

The mirroring mechanism implements mirroring meta-concepts (i.e., meta-classes, meta-relations, …) in such a way that all their instances are only proxies which mirror distant objects. Mixing "real classes" with such proxies is impossible.

Axiom 39 (Synchronous classes)
If a meta-class mirrors another meta-class, then there must be a one-to-one correspondence between their respective instances.

$$\frac{C \in \Big(\mathsf{MetaClass}\Big)_{\text{CASE}} \textit{ and } C' \in \Big(\mathsf{MetaClass}\Big)_{\text{META}} \textit{ and } C' \textit{ mirrors } C}{\begin{array}{l}\forall x \in \Big(\Gamma^{-1}(C)\Big)_{\text{CASE}}, \exists x' \in \Big(\Gamma^{-1}(C')\Big)_{\text{META}} : x \rightsquigarrow x' \\ \forall x' \in \Big(\Gamma^{-1}(C')\Big)_{\text{META}}, \exists x \in \Big(\Gamma^{-1}(C)\Big)_{\text{CASE}} : x \rightsquigarrow x'\end{array}}$$

Axiom 40 (Mirroring meta-property)
If a meta-property mirrors another meta-property, then they must be similar.

$$\frac{P' \in \Big(\mathsf{MetaProperty}\Big)_{\text{META}} \wedge P \in \Big(\mathsf{MetaProperty}\Big)_{\text{CASE}} \wedge P' \textit{ mirrors } P}{\begin{array}{l}\Big(\text{Prop}^{-1}(P')\Big)_{\text{META}} \textit{ mirrors } \Big(\text{Prop}^{-1}(P)\Big)_{\text{CASE}} \\ P.\text{multi} = P'.\text{multi} \\ P.\text{type} = P'.\text{type} \\ \forall x \in \Big(\Gamma^{-1}(C)\Big)_{\text{CASE}} : \Big(\xi_P(x)\Big)_{\text{CASE}} = \Big(\xi_{P'}(\tilde{x})\Big)_{\text{META}}\end{array}}$$

As for mirrored meta-classes, we do not enforce mirroring concepts to share the same name as their twin. For instance, meta-class could be renamed "méta-classe" (in French) in the mirroring system. Mirroring meta-relations also follow this rule.

Axiom 41 (Mirroring meta-relation)
If a meta-relation mirrors another meta-relation, then they must be similar.

$$\frac{R \in \Big(\mathsf{MetaRelation}\Big)_{\text{CASE}} \textit{ and } R' \in \Big(\mathsf{MetaRelation}\Big)_{\text{META}} \textit{ and } R' \textit{ mirrors } R}{\begin{array}{l}R'.\text{member } \textit{mirrors } R.\text{member} \\ R'.\text{owner } \textit{mirrors } R.\text{owner} \\ R'.\text{mandatory} = R.\text{mandatory}\end{array}}$$

$$\forall x \in \Big(\Gamma^{-1}(R.\text{owner})\Big)_{\text{CASE}} : \begin{array}{l} \Big(\Re_R(x)\Big)_{\text{CASE}} = [y_1, \ldots, y_n] \wedge \\ \Big(\Re_{R'}(\tilde{x})\Big)_{\text{META}} = [\tilde{y}_1, \ldots, \tilde{y}_n] \end{array}$$

Axiom 42 (Mirroring inheritance)

$$A, B_1, \ldots, B_n \in \Big(\mathsf{MetaClass}\Big)_{\text{CASE}}$$
$$\Big(\text{super}(A)\Big)_{\text{CASE}} = [B_1, \ldots, B_n]$$
$$A' \in \Big(\mathsf{MetaClass}\Big)_{\text{META}} \wedge A' \textit{ mirrors } A$$

$$\exists B'_1, \ldots, B'_n \in \Big(\mathsf{MetaClass}\Big)_{\text{META}} : \Big(\text{super}(A')\Big)_{\text{META}} = [B'_1, \ldots, B'_n] \wedge B'_i \textit{ mirrors } B_i \ (\forall i)$$
$$\forall a \in \Big(\Gamma^{-1}(A)\Big)_{\text{CASE}}, \forall b \in B_i \, (1 \leq i \leq n) : \Big(a \ \mathfrak{isa} \ b\Big)_{\text{CASE}} \Rightarrow \Big(\tilde{a} \ \mathfrak{isa} \ \tilde{b}\Big)_{\text{META}}$$
$$\forall a \in \Big(\Gamma^{-1}(A')\Big)_{\text{META}}, \forall b \in B'_i \, (1 \leq i \leq n) : \Big(a \ \mathfrak{isa} \ b\Big)_{\text{META}} \Rightarrow \Big(\tilde{a} \ \mathfrak{isa} \ \tilde{b}\Big)_{\text{CASE}}$$

Axiom 42 prevents the method engineer to edit the inheritance tree of a mirrored meta-class. Supertypes can not be removed or added to a mirrored meta-class. Moreover, all the supertypes of a mirrored meta-class (direct or indirect) must also be mirrored. This restriction can easily be understood: programming languages do not generally allow ones to control their inheritance mechanism in a dynamic way. Hence, it is impossible to "wrap" this mechanism in order to control it and to take into account a new meta-class that would be inserted in a list of supertypes on the mirroring side. Nevertheless, none of the axioms prevent us to derive subtypes from mirroring meta-classes.

Axiom 43 (case′ is exhaustive)
The case′ specification will group together all the mirroring classes.

$$C \in \Big(\mathsf{MetaClass}\Big)_{\text{CASE}} \textit{ and } C' \in \Big(\mathsf{MetaClass}\Big)_{\text{META}} \textit{ and } C' \textit{ mirrors } C$$

$$\forall x \in \Big(\Gamma^{-1}(C)\Big)_{\text{CASE}} : \tilde{x} \in \Big(\text{Def}(\text{case}')\Big)_{\text{META}}$$

Remark This axiom does not prevent other specifications to contain mirroring classes in their definition.

Axiom 44
A virtual meta-class on the CASE *side should also be virtual on the meta-CASE side.*

$$C \in \Big(\mathsf{MetaClass}\Big)_{\text{CASE}} \textit{ and } C' \in \Big(\mathsf{MetaClass}\Big)_{\text{META}} \textit{ and } C' \textit{ mirrors } C$$

$$C.\text{virtual} = C'.\text{virtual}$$

Axiom 45 (Semantics compatibility)
The mirrored system's semantics must be compatible with the meta-CASE's semantics.

The axioms that govern the mirrored system must be coherent with the axioms of the meta-CASE. Although we tried to keep our semantics as general as possible, some difficulties may arise. Some examples are:

The inheritance semantics: A base meta-class could be instantiated several times in a inheritance graph. This is the default behaviour in C++(see [Str97] pages 399–401). Axiom 23[1] forbids this kind of semantics.

Object references: Although object references[2] are quite usual in other programming languages, our axioms limit the properties to "printable" values. Nevertheless, this case can be considered as an implementation detail of a one-to-one meta-relation (*cfr.* the next item).

One-to-one meta-relations A meta-relation has necessarily one-to-many cardinalities. The one-to-one constraint must be explicitly managed by the method engineer in the MetaRelation::`CanInsert` predicate (*cfr.* section 7.9.4 page 191).

Identifiers The meta-model on the CASE side could have too complex identifiers. For instance, an identifier could be composed of inherited meta-properties.

We have listed some limitations of the mirroring service. Nevertheless, the limitations can be circumvented in some cases. For instance, the object references and the one-to-one meta-relations cases can be emulated with ad-hoc predicates (*cfr.* section 7.9.4).

Axiom 46
If C' mirrors C, then C must have these methods: C::`CanCreate`, C::`Constructor`, C::`CanDelete`, and C::`Delete`. Although these methods are implicitly defined in the repository, the mirrored system could follow other rules and, hence, this condition must be explicitly checked.

In order to maintain both extensions synchronous, some fundamental methods must be modified. The maintenance of the $\rightsquigarrow$ relation will be managed at the level of the constructors and destructors of the corresponding meta-classes in both systems. We will thus modify their definition in order to manage this link. The relation will be implemented with a new meta-property that is added to the meta-classes on both sides. This meta-property (called "mirror") will store the reference of $\tilde{x}$ in the class x and vice versa. We give here the pattern of the transformations to apply on these methods. Let us suppose that C' mirrors C where C' and C are two meta-classes. Then, we replace the method "C::`CanCreate`(α)" with "C::`CanCreate`(α, $\mathbb{B} : \beta$)" which is defined as follows:

> **Remark** In the following sentences, the α symbol is a kind of "meta-variable" which denotes the group of the former parameters.

function C::`CanCreate`$_{\mathsf{new}}(\alpha, \mathbb{B} : \beta)$:$\mathbb{B}$

```
begin
   if β = true
   then return C::CanCreate_old(α) ∧ C'::CanCreate_new(α, false)
   else return C::CanCreate_old(α)

end
```

[1]See page 54.
[2]I.e., pointers

And next, the definition of the method "C::Constructor(α)" is replaced with "C::Constructor(α, Class : β)" which is defined as:

function C::Constructor$_{\text{new}}$(α, $\left(\text{Class}\right)_{\text{META}}$: β) : C
var C: this;

```
begin
   if C::CanCreate(α, true)=false then abort
   this := C::Constructor_old(α)
   if β = void then β := C'::Constructor(α, this)
   mirror := β
   return this
end
```

Remark Although the type of a meta-property must be printable, meta-property "mirror" is defined as a class value. This exception will be tolerated since this meta-model is just descriptive and will not be stored in the repository.

The same kind of transformation will be applied to the DELETE and CANDELETE methods.

function C::CanDelete$_{\text{new}}$($\mathbb{B}$: β) :$\mathbb{B}$

```
begin
   if β = true
   then return C::CanDelete_old( ) ∧ C'::CanDelete_new(false)
   else return C::CanDelete_old( )

end
```

procedure C::Delete$_{\text{new}}$($\left(\text{Class}\right)_{\text{CASE}}$: c, $\mathbb{B}$: β)

```
begin
   if C::CanDelete(c, true)=false then abort
   if β = TRUE then C'::Delete_new(c.mirror, false)
   C'::Delete_old(c)
end
```

Finally all the calls to these methods must also be modified in order to reflect these transformations:

Old pattern	New pattern
C::Constructor$_{\text{old}}$(α)	C::Constructor$_{\text{new}}$(α, void)
C::CanCreate$_{\text{old}}$(α)	C::CanCreate$_{\text{new}}$(α, **true**)
C::Delete$_{\text{old}}$($C : x$)	C::Delete$_{\text{new}}$($C : x$, **true**)
C::CanDelete$_{\text{old}}$()	C::CanDelete$_{\text{new}}$(**true**)

The modification operated so far aimed to maintain the mirroring service on the CASE tool side. Similar transformations must be applied on the meta-CASE side in order to respect the reflection in the two directions.

function C'::CanCreate$_{\mathsf{new}}$(α, $\mathbb{B} : \beta$) :$\mathbb{B}$

```
begin
   if β = true
   then return C'::CanCreate_old(α) ∧ C::CanCreate_new(α, false)
   else return C'::CanCreate_old(α)
end
```

function C'::Constructor$_{\mathsf{new}}$(α, $\left(\mathsf{Class}\right)_{\text{CASE}} : \beta$) : C'

var C': this;

```
begin
   if C'::CanCreate(α, true)=false then  abort
   this := C'::Constructor_old(α)
   if β = void then β := C::Constructor(α, this)
   mirror := β
   return this
end
```

function C'::CanDelete$_{\mathsf{new}}$($\mathbb{B} : \beta$) :$\mathbb{B}$

```
begin
   if β = true
   then return C'::CanDelete_old( ) ∧ C::CanDelete_new(false)
   else return C'::CanDelete_old( )

end
```

procedure C'::Delete$_{\mathsf{new}}$($\left(\mathsf{Class}\right)_{\text{CASE}} : c$, $\mathbb{B} : \beta$)

```
begin
   if C'::CanDelete(c, true)=false then abort
   if β = TRUE then C::Delete_new(c.mirror, false)
   C::Delete_old(c)
end
```

Transformations on the meta-CASE side are made at a low level. That means that only the meta-CASE architect* must be aware of the transformations. The method engineer can continue to use these fundamental methods as they were presented and defined in section 4.3.6[1]. For instance, the method engineer will continue to write the calls to the CanCreate method like this

EntityType::CanCreate("customer")

and the system will convert it into

EntityType::CanCreate("customer",**true**)

In the same way, he could define this method like this:

function EntityType::CanCreate(String : s) :$\mathbb{B}$

```
begin
   return true ⇔ "there is not other entity-type s"
end
```

[1]See page 47.

The transformation that will take into account the precondition of the mirrored system will be made dynamically by the meta-CASE.

5.3 Mirroring and Reflection

The mirroring service will play a crucial role in our architecture since it will be the keystone of our architecture. It will allow us to bootstrap major components of the meta-CASE and will homogenize the Voyager 2^+ language. Indeed, the mirroring service will be the foundation of a very expressive reflective* system.

So far, we have supposed that the CASE and META repositories were distinct. We will now suppose that CASE=META. This exercise is just a particular case of the mirroring method described in the previous sections.

For this purpose, we will use the repository described in fig 4.25[1] as a starting point of the mirroring process that we will describe in the following stages:

Step I. To define the mirrored sub-model The mirroring service allows us to mirror only a submodel of a meta-model. Indeed none of the axioms obliges us to mirror a repository in its entirety. That is, we can let some meta-classes in the mirrored meta-model "unmirrored". We will concentrate our efforts on the submodel corresponding to the meta-model level of the 3-layers architecture. Hence, we will omit all the meta-classes relative to the specification level: `eClass`, `eDef`, and `eSpecification`. In the same way, we will omit all the "technical" meta-properties (for instance, the indices). This submodel is illustrated in Fig. 5.4.

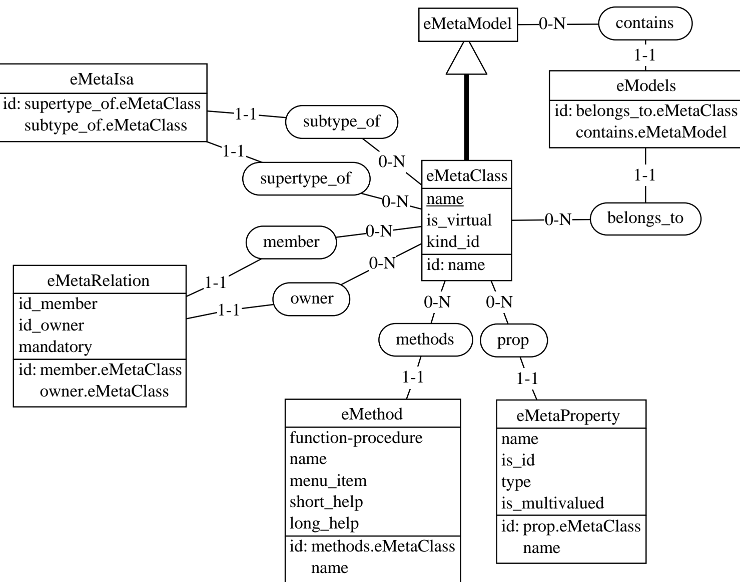


Figure 5.4: **[The Mirrored Sub-Model]** This ERA schema illustrates the sub-model that will be mirrored in the meta-CASE.

[1]See page 83.

Step II. To check the semantics compatibility We can observe that this sub-model has a semantics that is compatible with our axioms: no object references, no one-to-one meta-relations, no optional meta-property, ... Moreover, every meta-class owns an identifier (sometimes inherited). The translation of this schema into our formalism is thus trivial. Axiom 45[1] is thus verified.

Step III. To define the mirroring relation We define and create one mirroring concept in the meta-CASE per mirrored concept.

1. We create one meta-model named `MetaMetaModel` (prev. CASE′).

2. We create all the necessary mirroring meta-classes:

$$\{\texttt{_MetaClass}, \texttt{_MetaIsa}, \texttt{_MetaModel}, \texttt{_MetaProperty}, \texttt{_MetaRelation}, \texttt{_Method_Models}\} \subseteq \Big(\mathsf{MetaClass}\Big)_{\text{META}}$$

3. We proceed in the same way for the meta-relations:

$$\{\texttt{_belongs_to}, \texttt{_contains}, \texttt{_member}, \texttt{_methods}, \texttt{_owner}, \texttt{_prop}, \texttt{_subtype_of}, \texttt{_supertype_of}, \} \subseteq \Big(\mathsf{MetaRelation}\Big)_{\text{META}}$$

4. And we continue with the meta-properties. We do not reproduce the list, that would be too long.

Each concept will obviously mirror its twin (`_MetaClass` mirrors `eMetaClass`, ... , `_belongs_to` mirrors `belongs_to`, ... , and `_MetaClass._name` mirrors `eMetaClass.name`). Moreover, it will be defined in such a way that its semantics will be similar to the mirrored concept.

The repository endowed with its reflective definition is depicted in Fig. 5.5

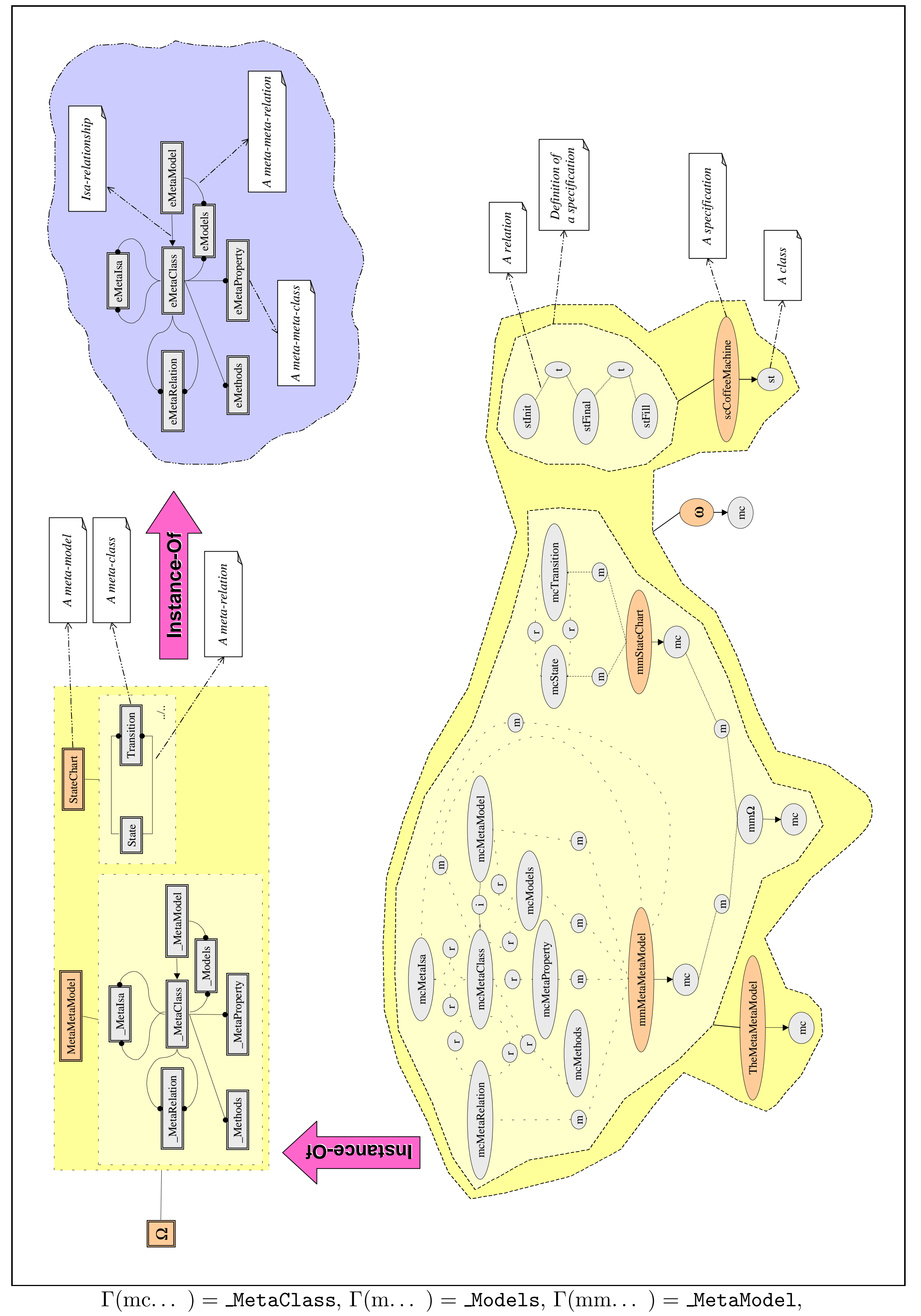
5.4 Reflection: the First Benefits

This section presents the main advantages of the reflective system. Both the software engineer and the meta-CASE architect can benefit from this characteristic to either limit the expressiveness of the repository or to endow it with new possibilities.

5.4.1 Fundamental Axioms and Reflection

The meta-CASE architect can use the reflective system to define some axioms that would have been hard-coded otherwise. This leaves the architecture much more open and easier to maintain. Moreover, the method engineer can refine these methods to debug and improve its knowledge of the repository. However, this feature permits engineers to access the heart of the repository and thus to corrupt its behaviour. The method engineer plays here the same role as a database administrator and shares thus the same responsibilities.

[1]See page 100.



$\Gamma(\text{mc}\ldots) =$ `_MetaClass`, $\Gamma(\text{m}\ldots) =$ `_Models`, $\Gamma(\text{mm}\ldots) =$ `_MetaModel`,
$\Gamma(\text{r}\ldots) =$ `_MetaRelation`, $\Gamma(\text{st}\ldots) =$ `State`, $\Gamma(\text{t}\ldots) =$ `Transition`

Figure 5.5: **[The Reflective Meta-Model]**

Example

This method ensures that axiom 7[1] is verified.

> *"If a meta-class has a supertype which is a meta-model, then this meta-class is a meta-model."*

```
function _MetaIsa::CanCreate(_MetaClass: sub, _MetaClass: super)
var _MetaModel : msub, msuper;
begin
   msub := sub;
   msuper := super;
   return ¬ ( msub=void ∧ msuper≠void)
end
```

5.4.2 Mirroring and Reflection

The meta-meta-model is now available to the method engineer in the same way than any other user-defined meta-model. Consequently, he can refine the fundamental methods of the meta-meta-model's meta-classes.

So much so that some constraints relative to the mirroring service can be directly expressed in these methods. For instance, axiom 42[2] forbids to add new supertype to a mirroring meta-class. This constraint can be ensured by redefining this method:

```
function _MetaIsa::CanCreate(_MetaClass: sub, _MetaClass: super)
begin
   return sub ∉ { _MetaClass, _MetaIsa,
                  _MetaModel, _MetaProperty,
                  _MetaRelation, _Method
                  _Models }
end
```

In the same way, if the method engineer[3] estimates that deriving new subtypes from a mirroring meta-class cannot be allowed, he could rewrite the method as:

```
function _MetaIsa::CanCreate(_MetaClass: sub, _MetaClass: super)
begin
   return {sub,super} ∩ { _MetaClass, _MetaIsa,
                          _MetaModel, _MetaProperty,
                          _MetaRelation, _Method
                          _Models } = ∅
end
```

[1]See page 44.
[2]See page 100.
[3]Although this job corresponds to the skill of the method engineer, this person will be more probably the meta-CASE architect.

5.4.3 Reflection and Bootstrapping

Last but not least, bootstrapping* is certainly the major contribution of this reflective architecture. As it will be explained later (chapter 6), GraSyLa will use the reflective definition of the repository to propose a visualization of the meta-CASE architecture to the method engineer to allow him to edit both meta-model's definitions and GraSyLa scripts. Moreover, the Voyager 2^+ programming language (chapter 7) will be able to manipulate types as ordinary values and will confer it the title of meta-language.

But this reflection will allow us to introduce new methods to increase the control of the method engineer on its meta-models (and the specifications). Let us remember that the reflection mechanism has added the counterparts of the meta-meta-model's meta-classes in a meta-model named `_MetaMetaModel`. Consequently, these meta-classes can have their own methods, and the method engineer can define/edit them.

We have added some methods that will correspond to the main events that can cause a change of the repository. These methods will be taken into account by the meta-CASE to control the behaviour of the events and their consequences. For instance, the event "to change the name of an attribute in a ERA schema" will be controlled by a predicate that will forbid or permit this change.

These methods[1] are defined below:

_MetaRelation::`CanInsert` : Can a class be inserted in a relation?

_MetaRelation::`CanRemove` : Can a class be removed from a relation?

_MetaProperty::`CanUpdate` : Can the value of a property be edited?

5.5 Prospect: Mirroring and Distributed CASEs

So far, the mirroring mechanism has been implemented on the meta-CASE itself to obtain a reflective architecture. Nevertheless, it has been described at the beginning as a technique to pilot and extend "distant" CASE tools. This goal is of course compatible with the procedure described in this chapter.

Distributed architectures (like CORBA* or DCOM*) are possible technologies to manage this "distant mirroring". Moreover, many repositories are already CORBA compliant (UREP, MOF) and would be de-facto candidates for the mirroring service. Moreover, this puts forward new qualities that were not perceptible before.

Once integrated in a distributed architecture, the meta-CASE can coexist with several other distant CASE tools (not necessarily identical) and acts as an integration platform, which mirrors specifications from other CASE tools, carries out their integration (under the control of software engineers), and finally propagates the result of the integration task to its partners. Indeed, the meta-CASE's repository can hold several mirroring meta-models, and each one can have several mirroring specifications, that can be extended and enriched with dynamic properties (i.e., meta-relations, meta-properties, subtypes, ...) to support the integration task. Figure 5.6 depicts a scenario of integration with two CASE tools and one meta-CASE. Other scenarios are of course possible.

[1]These methods are formally defined in section 7.9.4 page 191.

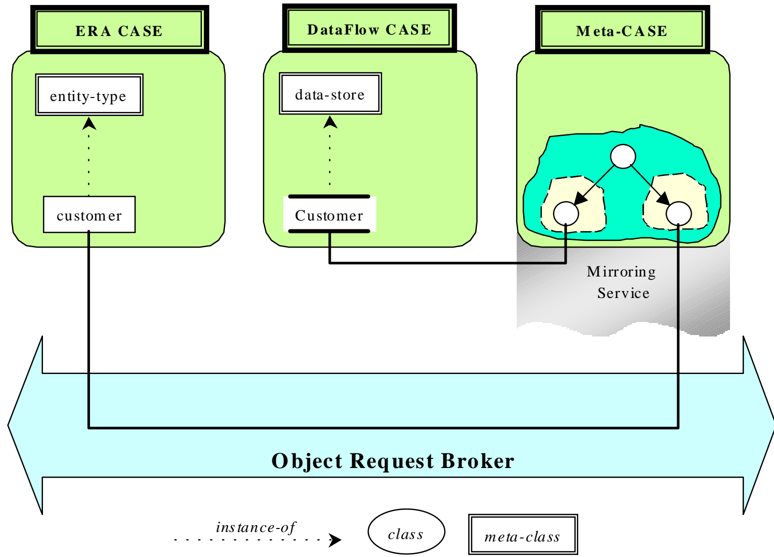


Figure 5.6: [**Distributed Architecture & Integration**] In this scenario, the meta-CASE is mirroring two distinct CASE tools (i.e., that do not have the same meta-model). Both of them export their objects on a distributed architecture (here the CORBA's ORB). This feature is exploited by the meta-CASE to implement its mirroring service. In this example, the meta-CASE is mirroring ERA and Dataflow specifications. Their meta-models can been specialized and extended in the meta-CASE to make possible the integration of the similar concepts (for instance, the top instance inherits from the mirroring classes to denote their integration).

Finally, as any other CASE tool, the meta-CASE can export itself on a distributed architecture and, hence, it can also take advantage of this system. This is depicted in Fig. 5.7.

Although this distributed mirroring architecture has not yet been tested, other products rely on a similar architecture. The problem is of course much broader, we have just given some hints of a possible implementation of a distributed mirroring service. Problems like the synchronization, the security, the transactions, the cooperation, etc have not been considered and should be the subject of further research

5.6 Summary

The mirroring service makes it possible to manage replicated repositories. The repository of a CASE tool is duplicated in the meta-CASE's repository, and the synchronization between the two *databases* is ensured by the mirroring service. The principle consists in copying the classes with a minimum of information. As far as possible, the information about classes is left only on the CASE tool side, and the latter acts as a server to answer queries that come from the meta-CASE's repository.

The meta-CASE's repository is defined in such a way that the meta-model which represents the CASE tool can be enriched (under some restrictions) with new meta-properties, new meta-relations, new subtypes, new meta-classes, etc. This semantic extension will provide copied classes with an extra information which denotes the added value of the enriched meta-model.

This chapter has also explained how it has been possible to apply the mirroring service

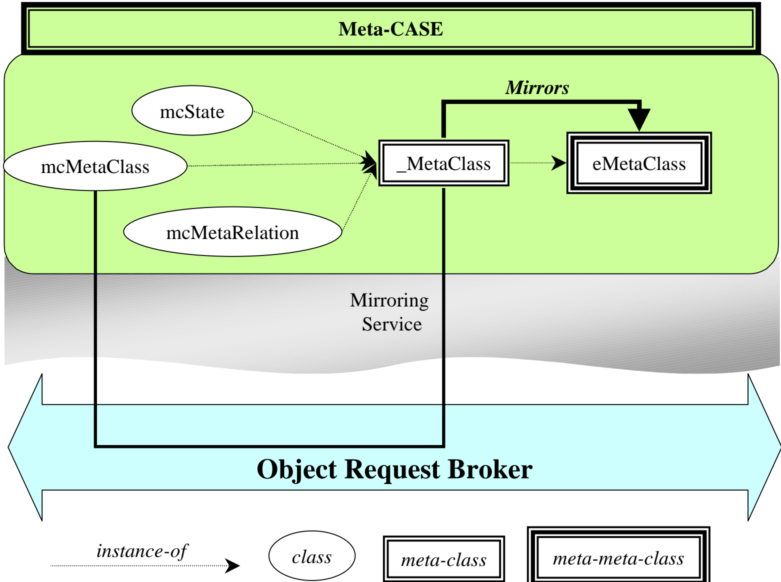


Figure 5.7: [**Distributed Architecture & Reflection**] This diagram represents the meta-CASE's three layers from left to right: the specification level, the meta-model level, and the meta-meta-model. The mirroring is applied to the meta-CASE and CORBA is used to provide the "bus" between the distinct levels (i.e., specification & meta-model).

on the meta-CASE to fold it back over itself. In other words, its meta-meta-model and meta-model layers can be shifted one step lower. Hence, the meta-meta-model has a synchronized copy at the meta-model level and the meta-models have a synchronized copy at the specification level. Those shifts endow the meta-CASE's components with a reflective architecture. Moreover, besides the introspection possibilities, it becomes possible to reify the kernel of the meta-CASE. Those features will be expoited by other components such as the Voyager 2^+ programming language and the GraSyLa language for instance.

"It was a subject of psychology, concerned basically with the question why people begin to use diagrams whenever verbal explanations get too complicated. This was because language is still fundamentally serial and one- dimensional. We can say 'former' or 'latter'–but there is NO easy way to refer to four or five things at the same time."
H. Harrison and M. Minsky.
in *The Turing Option*

Chapter 6

GraSyLa : a Graphical Symbolic Language

6.1 Overview

An important difference between a repository (ConceptBase[1], PCTE [LM93], UREP) and a meta-CASE consists in the capability to display the content of the repository according to some requirements peculiar to a methodology, software engineers, or method engineers. Visualizing this information is one of the major goals of a meta-CASE! The "visualizing task" permits a user to define, to edit and to validate specifications. If one uses the analogy with a CAD/CAM system, one observes that a plane's wing can be displayed as an electric circuit, as a general plan, as a 3D wired view, etc. In computer science, specifications should be displayed in many ways, depending on the culture of the method engineer, on the context, etc. On the other side, one also observes that the graphical conventions can evolve for several reasons: a methodology change[2], new needs, or conventions modifications. All the current researches in this area have so far adopted a simple philosophy:

classes are displayed as graphical objects constituted of the property values and relationships are displayed as edges between those graphical objects. Unfortunately, this approach is oversimplified. The display of the concept much often depends either on its context or on its content. The context of a concept is here a display process that requires the drawing of this concept. The context of a concept may have an influence on its form. For instance, in ERA diagrams, attributes could[3] have different representations in the entity-types, relation-types and compound attribute contexts. On the other hand, the look of a concept can also depend on its content/semantics. In UML, for instance, methods have distinct representa-

[1] Although ConceptBase has a graphical editor, we estimate that this one is insufficient for facing software engineers needs.

[2] The recent shift from OMT and Booch notations to UML is a good example.

[3] DB-MAIN's extended view has distinct representations of attributes depending on their contexts.

tions depending on their privacy[1]. Moreover, when the concept's appearance can depend on its semantics, this allows the method engineer to define, for instance, its own select/mark[2] strategies. In some ERA graphical conventions, attributes taking part in the identifier are underlined. If this participation is stored as a meta-property of the attribute meta-class, it becomes obvious that the graphical display process can be influenced by the semantics of a concept/meta-class. This dependency has already been observed by the researchers of the MetaView project who have proposed to adopt a conditional visualization in further meta-systems: "*Need to have the graphical representation of conceptual objects dependent on its attribute values*" in [GFS+94, ZFS95].

The graphical visualization process must also take in consideration the rich semantics of the inheritance principle proposed in the meta-meta-model. For instance, a method engineer can expect specific representations for a supertype and its subtypes. This expectation can be fulfilled by a display mechanism with a semantics close to the "polymorphism" principle of the object-oriented languages. This mechanism will choose the best representation of a concept depending on its place in the inheritance graph. Let us note that this is very close to the conditional display[3].

6.2 The Challenges

The graphical representation of specifications is a crucial issue when constructing CASE tools. Indeed, they cannot be imposed, on the contrary, they have to attract reluctant engineers [LC98]. As for many programs, the graphical interface will be the major criterion (both subjective and objective) in a CASE tool evaluation. Norman *et al* [NSC+91] argues that the "*customization without loss of functionality* " and "*customizable user interfaces*" were needed in CASE tools and should be investigated in the 1990's. Moreover, recent studies [JH98, LKNF95] explain that this gap is not yet filled "*A well-designed user interface should create a metaphor that bridges the conceptual gap between a computer system and human thinking. Such a metaphor is not the strength of current CASE tools* ".

From our experience in the building process of the DB-MAIN tool [HEH+96], we observed these criticisms and have deduced some important rules concerning the graphical user interface, and more specially the graphical representation:

1) People need both textual and graphical views. Our experience leads us to consider two kinds of specification visualization: graphical diagrams (graph, tree, table, matrix, ...) and textual views (report, code, hypertext, ...). Software engineers often require several views of a same specification depending on the process to complete. For instance, we have oberserved that graphical views are preferred for teaching or for validation although textual views[4] are preferred to edit huge specifications. Moreover, Jeusfeld *et al.* [JJNS98] explain that "*conceptual modeling has different goals (e.g., system analysis, system specification, documentation, training, decision support); heterogeneous goals lead to heterogenous representation languages, and to heterogeneous ways-of-working even with given languages*". Meta-CASE tools have thus to offer a large variety of views of its information system.

[1] i.e., private/protected/public. We suppose here that the privacy is modelled as a meta-property.

[2] See DB-MAIN for examples and uses of *marking planes*.

[3] A meta-class with a finite enumerated meta-property can be transformed into as much subtypes as there are enumerated values. This reenforces the idea that the graphical representation of the concept does not depend on the way it is meta-modelled.

[4] Textual views can have facilities like: sort algorithms, cross-referencing, ...

2) The graphical representation of a concept is contextual. Its representation depends on its use. For instance, in ERA diagrams, attributes can be indented (resp. underlined) depending on their belonging in compound attributes (resp. in identifiers).

3) The shape of a concept is polymorphic. In OO meta-models, concepts are modelled in using inheritance trees/graphs. Although this graph denotes an atomic concept, its representation will depend on degree of specialization of this graph. For instance, if one encounters this case: "`weak-entity-type isa entity-type`" in a meta-model, software engineers may wish to have distinct shapes for them.

4) The graphical representations must be evolutive. Scientists have certainly not yet discovered the ultimate methodology; new ones appear every year. However, they all share common ontologies like statecharts, static model (ERA/NIAM/class/ ...), use cases, ... Semantics changes but the structural meta-model varies very little. The main changes we observed is about the graphical representation[1]. Moreover, in meta-CASEs, method engineers can specialize and edit the meta-models to adapt them to their company's requirements. These modifications must often be reflected in the graphical representation.

5) CASE tools must support multimedia data. CASE tools must be *softer* [JH98] and must accompany the software engineer in its rigorous steps as well as in softer aspects of software development. Multimedia data can allow to enrich his specifications with non formal information like interviews, requirement documents, imported diagrams, ... and bridges a gap between CASE tools and less usual tools in the software engineer environment.

6) The graphical requirements should not affect the meta-model definition. Repositories often offer richer abstractions than usual DBMS in order to meet the peculiar needs of software meta-models. However, this quality would be useless if we had to change the meta-model definition each time we modify or add a new visualization. This leads to an interesting challenge, since the "main entities" in a visualization may differ from the entities (i.e., the meta-classes). For instance, in ERA-like diagrams, attributes are dependent graphical objects although in NIAM-like diagrams, attributes become autonomous.

This list exposes the main challenges the engineers should investigate when defining the GUI requirements of a meta-CASE.

6.3 The GraSyLa Approach

We have privileged a general approach of the problem (see the previous challenges). Distinct practices were observed and have allowed us to draw the following conclusions. First, we identified several situations that need to display specifications: screen visualization, printing, diagram interchanges, ... We have modelled them with the concept of *display*. A display is any device that can require the display of a specification. We leave its definition open in order to encompass as many requirements as possible. For instance a Postscript file could be such a display!

Next we observed that, behind the display, computers needed some algorithm to produce the representation of the specifications. These algorithms can often be considered as autonomous agents who interact with their environment, i.e., events, the software engineer, or the system. We have modelled the generic idea of these algorithms as an abstract machine named the *display processor* (DP).

[1]For instance: the static diagrams in the Booch, OMT, UML and OML methodologies.

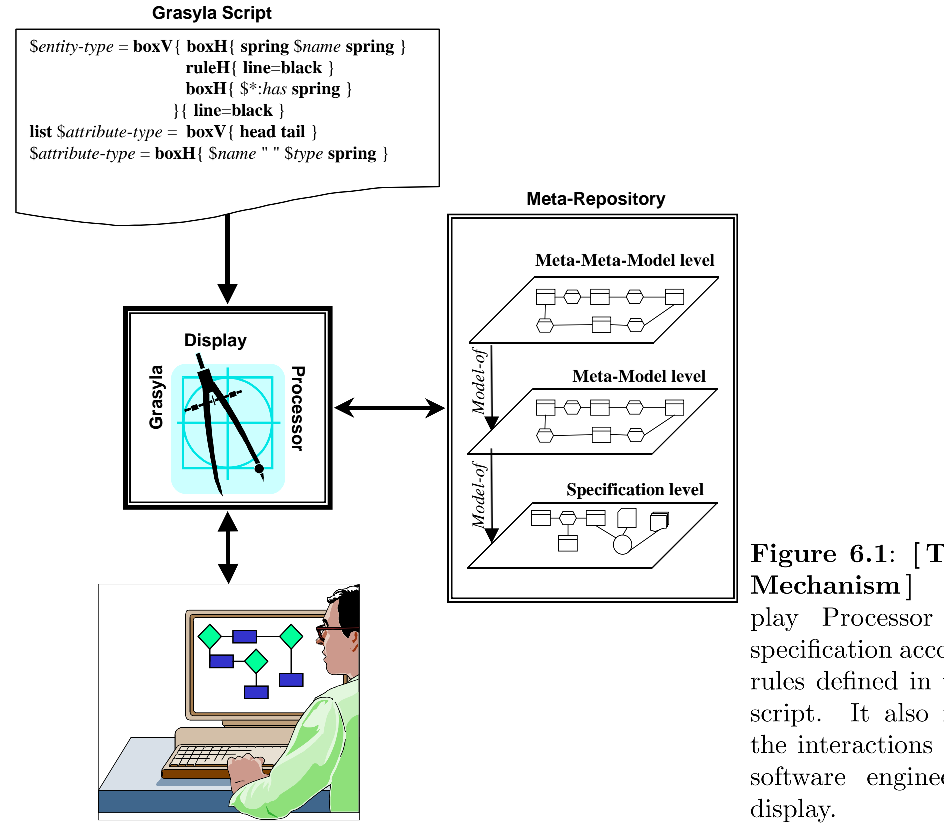


Figure 6.1: [**The** GraSyLa **Mechanism**] The Display Processor displays a specification according to the rules defined in the GraSyLa script. It also manages all the interactions between the software engineer and the display.

Thirdly, the abstract machine would be useless without a script to control its execution. Indeed, the DP is just able to manage shapes but has absolutely no idea of the content of this shape. This is left as a free parameter that is defined with the GraSyLa language.

So, we can sum up our ideas as follows. Each display will be controlled by a display processor that will display the information stored in the repository (i.e., a specification) and manage all the interactions with the software engineer. This schema is depicted in Fig. 6.1.

6.4 GraSyLa : its Syntax and its Semantics

GraSyLa is a graphical symbolic language that permits the method engineer to define in a script the appearance of a specification. GraSyLa proposes high level abstractions for display algorithms that are very common in CASE tools. It will relieves the method engineer of awkward technical concerns like windows, pens, positions computation, the mouse management, geometric shapes, ... Of course, as any abstraction, GraSyLa made some simplifying hypotheses.

Each meta-model can have several named GraSyLa scripts; each one will correspond to a possible visualization of its specifications. Scripts have three sections. The method engineer defines in the *directive section* the meta-classes to display. Next comes the *main section* that associates to each meta-class an expression that explains how to display it. Finally, the *connection section* will describe which meta-relations must be displayed.

6.4.1 Lexical Conventions

Identifiers will be either strings that respect the regular expression `[a-zA-Z][a-zA-Z0-9_]*` or any sequence of characters enclosed between `<` and `>`. In the last case, the `>` character can be escaped with the backslash character "\". Hence, `Foo_123` and <the \> operator> are two valid identifiers named "Foo_123" and "the > operator".

The { (resp. }) and `begin` (resp. `end`) words denote the same token and can be exchanged anywhere at anytime. Comments must be placed either between the `/*` and `*/` words or between the `//` word and the end of line (as in C++).

6.4.2 The GraSyLa Syntax

A script will follow this syntax:

- ◇ ***script*** ← ***header*** ⟨***engine-clause***⟩ ***use-clauses*** `begin` ***body*** `end`
- ◇ ***header*** ← `view` "ident" ";"
- ◇ ***engine-clause*** ← *see section 6.6*[1] *for more information.*
- ◇ ***use-clauses*** ← *see section 6.5*[2] *for more information.*
- ◇ ***body*** ← ***directive-section main-section connection-section***

A meta-CASE has to manage several kinds of *display*. It has to display specifications in windows on screen, to support printing functions or exchange format to clipboard. More advanced functions like browsers, navigators will also be considered as display devices. Nevertheless, the technology that underlies a display (i.e., a printer or a screen for instance) does not matter in the GraSyLa approach. Every display is described in the same way and is managed by a *display processor* (DP). The display processor acts as a controller that is responsible for displaying specifications and to manage all the possible interactions between the software engineer and the display. The DP will use a GraSyLa script as a recipe to present classes and their characteristics.

Through this section, we will illustrate the syntax and the underlying ideas of GraSyLa with the "Normal Statechart Meta-Model" that has been presented in section 4.7.1. For convenience sake, we have reproduce this meta-model in Fig. 6.2. We will abbreviate this meta-model as M and we will take a specification $s \in \Gamma^{-1}(M)$ that will be used as a guinea-pig for illustrating the concepts of GraSyLa in the next sections.

6.4.3 The Directive Section

The *directive section* informs the DP of the meta-classes to display. We will call them the *root* meta-classes and note this set: $\mathsf{Root} \in \mathsf{MetaClass}^{[\star]}$. Nevertheless, we observed that some diagrams demand more elaborated strategies. For this reason, GraSyLa accepts another set of meta-classes called $\mathsf{Junk} \in \mathsf{MetaClass}^{[\star]}$. Junk classes would normally not be displayed. They are neither root nor used while the processing of the root classes. Therefore, the method engineer can indicate that once the DP has finished its job, it should investigate the instances of the junk meta-classes to look for classes that were not displayed.

[1] Page 136.
[2] Page 134.

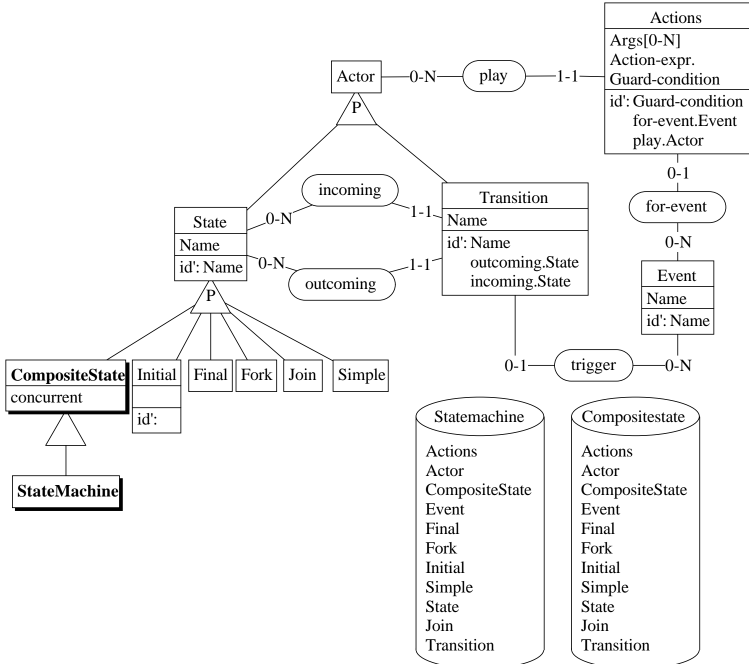


Figure 6.2: [**The Normal Statechart Meta-Model**] This meta-model has been defined and commented in section 4.7.1

Example

Let us suppose that one wishes to display the eletronic devices of a park (computers, printers, scanners, ...). Of course, these devices can themselves be composed of other elements (chips, hard disks, processors, ...). Hence, to visualize this park, we would have to display every computer, printer, or scanner that makes up this park. The visualization of those complex devices will entail the visualization of their contents (chips, hard disks, ...). The computers, printers, and scanners will be the "roots" in our terminology. Let us suppose now, that some hard disks are stored in a cupboard to replace defective hardware. They will be skipped from the inventory since they are not installed. To have an exhaustive inventory, we can specify that hard disks (but also chips, processors, ...) are "junk" meta-classes. Hence, even forgotten devices will be displayed.

Other junk meta-classes could be unused functions in programs, transitions that are not consumed in statecharts, employees without appointment in organizational diagrams, ...

The method engineer can also define paths by composition of existing meta-relations. For instance, this mechanism will permit the method engineer to retrieve many-to-many meta-relations from two one-to-many meta-relations. This case is quite frequent. A meta-relation which occurs in a path cannot be itself a path, i.e., all the components must belong to MetaRelation. Moreover, the method engineer must indicate the direction, i.e., the role that the path must follow: from the owner to the members (`*:`) or from a member to the owner (`1:`).

$\diamondsuit$ ***directive-section*** $\leftarrow$ $\langle$***root-dir***$\rangle$ $\langle$***junk-dir***$\rangle$ $\langle$***composition-dir***$\rangle_{0,\infty}$

$\diamondsuit$ ***root-dir*** $\leftarrow$ `root` ":" ***meta-classes*** ";"

$\diamondsuit$ ***junk-dir*** $\leftarrow$ `junk` ":" ***meta-classes*** ";"

$\diamondsuit$ ***composition-dir*** $\leftarrow$ *ident* "=" $\langle$***comp-rel***$\rangle_{1,\infty}^{\text{"}\bullet\text{"}}$ ";"

$\diamondsuit$ ***comp-rel*** $\leftarrow$ "1:" *ident* | "*:" *ident*

$\diamondsuit$ ***meta-classes*** $\leftarrow$ $\langle$*ident*$\rangle_{1,\infty}^{\text{","}}$

Root and junk meta-classes must belong to the definition of the meta-model that is displayed, i.e.: $\mathsf{Root} \subseteq \mathrm{Mod}(M)$ and $\mathsf{Junk} \subseteq \mathrm{Mod}(M)$.

We note $\mathsf{Comp} : \mathsf{String} \to \big(\text{rel:}\mathsf{MetaRelation} \times \text{role:}\{\texttt{one, many}\}\big)^{[+]}$ the function which defines the paths. This function must denote valid paths, i.e., meta-relations that can be composed together. Let $p \in \mathrm{dom}(\mathsf{Comp})$ be a path, we note $\sigma = \mathsf{Comp}(p)$ its composition. Then $\forall i \in \{1, \ldots, \sharp\sigma\}$, we note $(r_i, m_i) = \sigma[i]$. The following equalities must hold depending on the values of m_i and m_{i+1}, $\forall i :\ 1 \leq i \leq \sharp\sigma - 1$:

m_i/m_{i+1}	one	many
one	r_i.owner = r_{i+1}.member	r_i.owner = r_{i+1}.owner
many	r_i.member = r_{i+1}.member	r_i.member = r_{i+1}.owner

Example

```
root:  <State>, <Transition> ;
junk:  <Event> ;
<send> = *:<incoming> @ 1:<outcoming> ;
```

The `root` line expresses that the DP must display all the states and all the transitions. The `junk` line mentions that some events could be present in a specification but would not be used by neither a state nor a transition. This directive will display all the events that are not "consumed". The method engineer could consider this directive as a way to produce a diagnostic about misused events. The last statement defines a new path that is the composition of the "incoming" and "outcoming" meta-relations. "send" is thus a reflexive path from "state" to "state".

6.4.4 The Main Section

The *main section* will give for each meta-class that needs to be displayed a GraSyLa expression describing its form on the basis of its semantics, i.e., its properties, its roles, and its supertypes. The relationship between a concept and the expression that describes it is noted \$ and is defined as a function

$$\$: \big(\mathsf{MetaClass} \cup \mathsf{MetaProperty}\big) \times \{\texttt{single}, \texttt{list}\} \times \mathrm{ID} \to \mathbb{E}$$

where ID ($\subset$ String) denotes a set of identifiers and $\mathbb{E}$ is the set of well-formed GraSyLa expressions. The domain of the \$ function is extended with the meta-properties in order to give them particular representations. Each sentence of the main section will describe one entry of the \$ function. The syntax of those entries is described below:

◇ ***main-section*** ← $\langle$***metaclass*** | ***metaproperty***$\rangle_{0,\infty}$
◇ ***metaclass*** ←
    `meta-class $` *ident* "=" ***G-expr***
  | `meta-class list $` *ident* "=" ***G-expr***
  | `meta-class` *ident* "(" `$` *ident* ")" "=" ***G-expr***
  | `meta-class` *ident* "(" `list $` *ident* ")" "=" ***G-expr***
◇ ***metaproperty*** ←
    `meta-property $` *ident* "." *ident* "=" ***G-expr***
  | `meta-property list $` *ident* "." *ident* "=" ***G-expr***
  | `meta-property` *ident* "(" `$` *ident* "." *ident* ")" "=" ***G-expr***
  | `meta-property` *ident* "(" `list $` *ident* "." *ident* ")" "=" ***G-expr***

The left-hand side part of these entries reflects the various kinds of "things" a display processor will have to care of. The rules describe in the order: *1)* a class, *2)* a sequence of classes, *3)* a class when its display depends on a functor, *4)* a sequence of classes when their display depends on a functor, *5)* a property (i.e., the value of a meta-property), *6)* a sequence of properties, *7)* a property when its display depends on a functor, and *8)* a sequence of properties when their display depends on a functor.

We can right here and now give the general principle of the display processor. Let us suppose that the DP has to display a class c in Def(s). The DP will start from the leaf of c in s to look for a GraSyLa expression that describes best c. Let us call this leaf l. If $\$(\Gamma(l), \text{single}, \epsilon^1)$ exists, then this expression can be used to display c, otherwise, the DP will visit recursively each supertype of c to look for this expression. If it does not find any expression, then the class is just omitted. The idea is similar for the other kinds of arguments. They will be explained below.

GraSyLa expressions are quite simple. They are composed of variables, elementary graphical objects and constructors. Variables denote a characteristic of the "thing" the expression endeavours to describe. The elementary graphical objects are the bricks of the language. Such objects are string constants, horizontal and vertical distances, pictures, ... The constructors permit the method engineer to build complex forms from simpler ones. For instance, a very common constructor is `boxH{ ... }` that gathers its arguments together in the horizontal direction. The expression `boxH{"my tailor " "is rich"}` will concatenate both strings.

The next enumeration describes the components that can occur in a GraSyLa expression (***G-expr***). For everyone, the description begins with the list of the preconditions the method engineer will have to respect to build well-formed expressions.

$ident

This variable denotes the graphical representation of a property.

- The LHS part must denote a single meta-class C.
- *ident* is the name of a meta-property P.
- $P \in \text{Prop}(C)$ or $\exists X \in \mathsf{MetaClass} : C\ \textbf{isa}^\star\ X \wedge P \in \text{Prop}(X)$.

This denotes the graphical representation of the property. If

$$\$(P, \texttt{if}\ \ P.\text{multi}\ \texttt{then}\ \text{list}\ \texttt{else}\ \text{single}, \epsilon)\exists$$

[1]We use the ϵ functor, when there is no explicit functor. ϵ is equivalent to the empty string "".

then we use it to display the value of the property. Otherwise, the value is represented in its literal form. The ϵ symbol denotes the empty functor (i.e., default functor). If the type of the meta-property is `picture`, `sound`, `document` or `video`, then the Display Processor represents them with ad-hoc icons. A double-click on these icons launches a multimedia player. See the `bitmap` constructor for more details.

Example

```
meta-class $State =  $name
```

States will be represented by their name.

`$1:`**ident** ⟨`restricted`⟩

This variable denote the graphical representation of the owner of a relation whose the LHS class is a member.

- The LHS part must denote a single meta-class (C).
- *ident* is the name of a meta-relation R.
- $C = R.\text{member}$ or $\exists D \in \mathsf{MetaClass} : \ C\ \mathbf{isa}^\star\ D \wedge D = R.\text{member}$.

Let us suppose that $c \in \Gamma^{-1}(C)$ is the class denoted by the LHS part. If ${\Re^\star_R}^{-1}(c)$ exists, then the DP will have to display this class. Otherwise, the DP does nothing. If the `restricted` keyword is present, then the class is displayed only if it belongs to the current specification's definition (s).

Example

```
meta-class $Actions = $1:<for-event>
```

`Actions` classes will be represented by the name of their event.

`$*:`**ident** ⟨`restricted`⟩

This variable denotes the graphical representation of the members of a relation whose the owner is the LHS class.

- The LHS part must denote a single meta-class (C).
- *ident* is the name of a meta-relation.
- $C = R.\text{owner}$ or $\exists D \in \mathsf{MetaClass} : \ C\ \mathbf{isa}^\star\ D \wedge D = R.\text{owner}$.

Let us suppose that $c \in \Gamma^{-1}(C)$ is the class denoted by the LHS part. Then the DP will have to display the sequence $\Re^\star_R(c)$. If the `restricted` keyword is present, then the DP will omit all the member classes that does not belong to the current specification (s).

Example

```
meta-class $State = boxH{ $name ":" $*:play }
```

States will be represented by their name and the representation of their actions.

Remark The use of GraSyLa variables (i.e., `$1:` ... and `$*:` ...) allows the Display Processor to "pump" information outside the boundary of the current specification (i.e., s) to complete the representation of a meta-class. The scope of this "pump" can be limited to the elements of the specification in using the `restricted` keyword. The example depicted in Fig. 6.3 shows such a case. Let us imagine that the s specification is displayed with this GraSyLa script:

`root:` C;
`meta-class` $\$C = \ldots$ `$*:`$R \ldots$ `$*:`E `restricted` $\ldots$
`meta-class` $\$D = \ldots$
`meta-class` $\$E = \ldots$

Then, classes c_1 and c_2 will be displayed because they are the roots. Classes d_1 and d_2 will also be displayed because they are required by the `$*:`R variable in the expression of $\$C$. But, e_1 will be skipped altough e_2 will be displayed because e_1 does not belong to the definition of s. Hence, we observe that the "scope" of the DP is much larger than the definition of the specification it has to display (*cfr.* Fig. 6.4). The reason is quite simple: a specification is not an isolated island in a semantics graph. The closure of a class (its surroundigs) contributes to its semantics whatever its belonging to a specification. A specification is just a point of view or a window on this graph (the repository).

The following cases demonstrate the pertinence of our approach:

The "pumping" : This principle makes it possible to complete the information displayed about an entity-type with elements borrowed from dataflow diagrams, statecharts, ...

Views : In ERA diagrams, people are used to define views on static diagrams. A view is an ERA diagram defined from another ERA diagram whose one retains only the pertinent concepts (entity-types, attributes, and relationships). In our approach, a "view" would be a meta-model derived from the "ERA" meta-model and its graphical representation would use the `restricted` feature to limit the display of an entity-type to the pertinent attributes, i.e., the attributes that belong to the view.

```
meta-class $<entity type> = ... $*:<has-properties> restricted ...
```

Nevertheless, the graphical objects presented by the Display Processor in a display are not on an equal footing. Indeed, the DP prevents the software engineer to edit/modify the graphical objects which denote "pumped classes", i.e., classes that do not belong to the displayed specification. For instance, software engineers who edit a "class diagram" are not permitted to change the name or the components of a "pumped" statechart. This prevents people to edit specifications that are outside their competences.

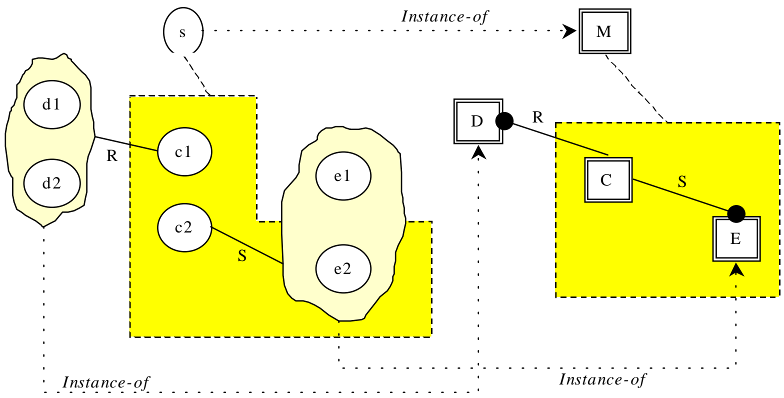


Figure 6.3: [**Restricted Scope**]

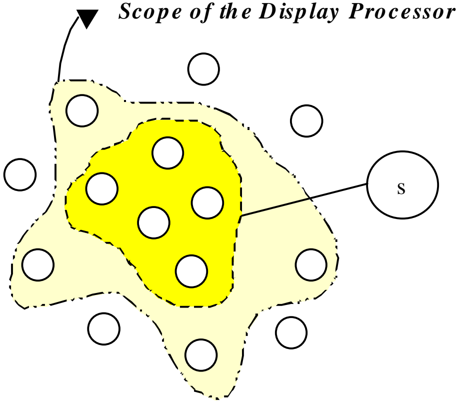


Figure 6.4: [**Scope of a Display Processor**]

`head`

This variable denotes the graphical representation of the first element of a sequence. The sequence can be composed of values or of classes. See `tail`.

- The LHS part must denote a list.

`tail`

This variable denotes the graphical representation of the queue of a sequence, i.e., the sequence minus its first element. The sequence can be composed of values or of classes.

- The LHS part must denote a list.

Example `meta-class list $Actions = boxV{ head tail }`
A sequence of actions will be represented as a vertical "stack" of actions.

Example `meta-class list $Actions = boxV{ tail head }`
A sequence of actions will be represented as a vertical "stack" of actions but in the inverse order.

> **Remark** There exists several strategies to decompose a sequence. The ambiguity arises when the sequence denotes a single element. Let us examine the $[e]$ sequence. It can be decomposed either as $e; [\,]$ or just e. If a GraSyLa expression is defined as `head "," tail`, then the comma character will act as a terminator in the first strategy and a separator in the last one. Nevertheless, when a sequence denotes a unique element, that last strategy would force the Display Processor to use directly a `$( ... ,single, ... )` instead of `$( ... ,list, ... )`. At this time, the Display Processor has been defined with the first strategy (terminator). The second strategy should also be proposed.

ident(G-expr)

The *ident* functor will be used to look for the rules describing the argument.

- The ***G-expr*** expression must denote a GraSyLa variable.

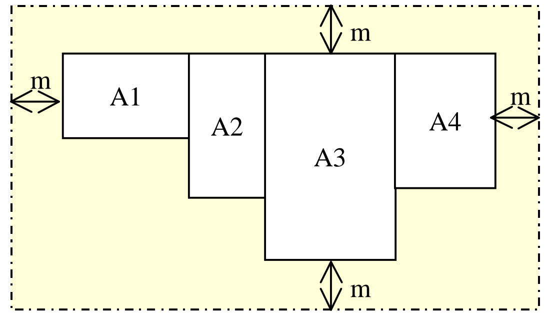
Example

```
meta-class $State = instate( $*:play)
meta-class $Transition = intransition( $*:play )
```

The use of distinct functors will guide the DP to specific GraSyLa expression to display sequences of `Actions`. The DP will use the "instate" (resp. "intransition") functor in place of ϵ to look for the entry of the $ function that describes best a sequence of actions.

> **Remark** In statements `meta-class list F( X )=...head..tail..`, the functor is systematically applied on the `head` and `tail` variables. This principle is the same when the sequence denotes meta-properties. The method engineer is not allowed to specify a functor for the `head` and `tail` variables[1].

[1] This semantics reflects the choices that have been made in the implemented prototype. There is no intrinsic reasons to keep this semantics, and therefore, it would be possible to generalize this principle in a further version.



boxH{ A_1 A_2 A_3 A_4 }{ stroke_color=black }
m is the margin between the components and the enclosing box.

Figure 6.5: [**The boxH Constructor**]

---

boxH"{" ⟨ **G-expr** $\rangle_{1,\infty}$ "}"⟨"{"**box-args**"}"⟩

This constructor will arrange n graphical objects (described by the ***G-expr***$_i$) horizontally in a box. The ***box-args*** is an optional component of this constructor that will control the aspect of the built box (its colour, its stroke, ...). The possible parameters are

stroke-color **:** defines the colour of the stroke. Its default value is "invisible".

stroke-width **:** the width of the stroke. Its default value is 1.

stroke-style **:** The style of the stroke. Its default value is "solid".

fill-color **:** the colour used to fill the box. Its default value is "white".

margin **:** the margin between the largest component and the inner side of the stroke. Its default value is 1.

handles **:** the number of handles to place on each side. Its default value is 0.

> **Remark** This number counts the handles from one coin included to another coin excluded. Hence, let h be this number, there will be exactly $4n$ distinct handles on the perimeter.

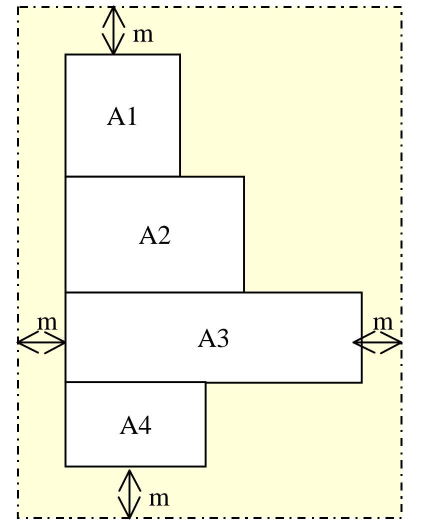
Example Figure 6.5 shows how the DP will gather the components together to form the new box (dashed frame).

boxV"{" ⟨ **G-expr** $\rangle_{1,\infty}$ "}"⟨"{" **box-args**"}"⟩

This constructor is similar to boxH but places its components in the vertical direction. Figure 6.6 shows the layout of the components.

circleH"{" ⟨ **G-expr** $\rangle_{1,\infty}$ "}"⟨"{"**box-args**"}"⟩

This constructor will arrange n graphical objects (described by the ***G-expr***$_i$) horizontally in a circle. The ***box-args*** is an optional component of this constructor that will control the aspect of the built box (its colour, its stroke, ...). The possible parameters are



boxV{ A_1 A_2 A_3 A_4 }{ stroke_color=black }

Figure 6.6: [**The boxV Constructor**]

stroke-color **:** defines the colour of the stroke. Its default value is “invisible”.

stroke-width **:** the width of the stroke. Its default value is 1.

stroke-style **:** The style of the stroke. Its default value is “solid”.

fill-color **:** the colour used to fill the box. Its default value is “white”.

margin **:** the margin between the largest component and the inner side of the stroke. Its default value is 1.

handles **:** the number of handles to place on the circle. The starting handle is placed at the North position. Its default value is 0.

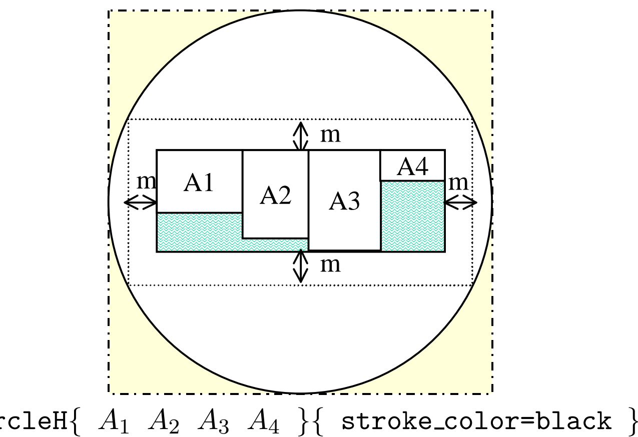
Example Figure 6.7 shows how the DP will gather the components together to form the circle. m is the margin between the components and the enclosing circle. Let us note that the DP only assembles rectangles. Hence, the DP considers that this constructor will build rectangles (the dashed frame).

circleV “{” $\langle$ **G-expr** $\rangle_{1,\infty}$ “}” $\langle$ “{” **box-args** “}” $\rangle$

This constructor draws a circle around components that are arranged in the vertical direction. See the circleH constructor for more details.

roundH “{” $\langle$ **G-expr** $\rangle_{1,\infty}$ “}” $\langle$ “{” **box-args** “}” $\rangle$

This constructor will arrange n graphical objects (described by the ***G-expr***$_i$) horizontally in a rectangle with rounded corners. The ***box-args*** is an optional component of this constructor that will control the aspect of the built box (its colour, its stroke, ...). The possible parameters are



circleH{ A_1 A_2 A_3 A_4 }{ stroke_color=black }

Figure 6.7: [**The** circleH **Constructor**]

stroke-color **:** defines the colour of the stroke. Its default value is "invisible".

stroke-width **:** the width of the stroke. Its default value is 1.

stroke-style **:** The style of the stroke. Its default value is "solid".

fill-color **:** the colour used to fill the box. Its default value is "white".

margin **:** the margin between the largest component and the inner side of the stroke. Its default value is 1.

handles **:** the number of handles to place on each side. Its default value is 0. **Remark:** This number counts the handles from one coin included to another coin excluded. Hence, let h be this number, there will be exactly $4n$ distinct handles on the perimeter.

round **:** defines the shape of the round. See Fig. 6.8.

Example Figure 6.8 illustrates the use of this constructor and explains the semantics of the `round` option.

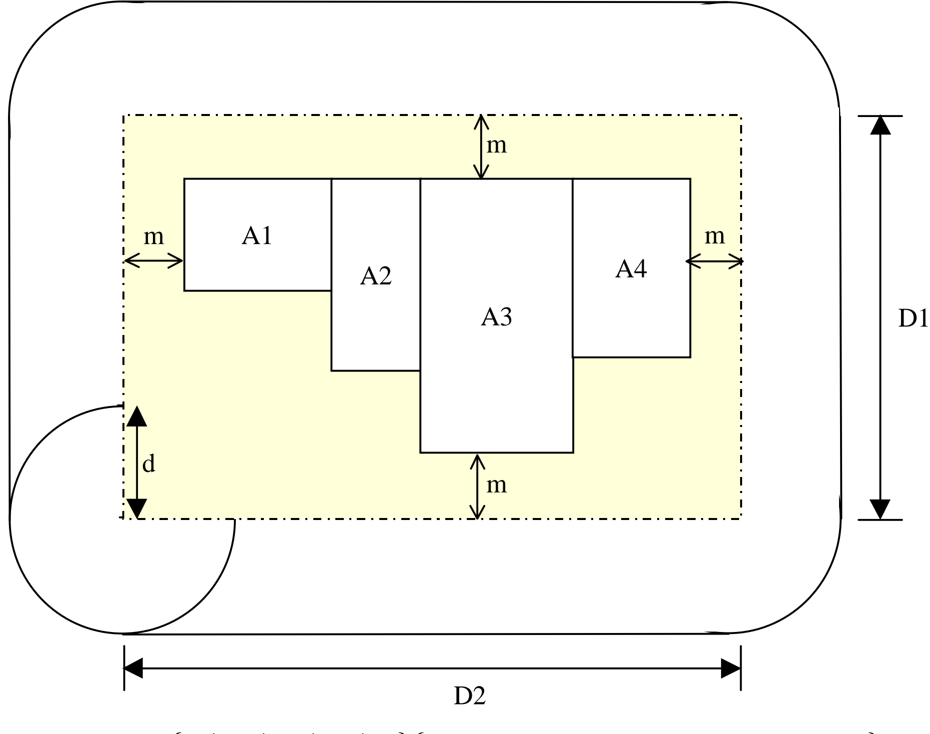
roundV "{" ⟨ **G-expr** $\rangle_{1,\infty}$ "}" ⟨ "{" **box-args** "}" ⟩

This constructor draws a circle around components that are arranged in the vertical direction. See the `roundH` constructor for more details.

if **cond** then **G-expr**$_1$ ⟨ else **G-expr**$_2$ ⟩

This constructor is a conditional statement that will direct the DP towards one GraSyLa expression among two possibilities.

- The LHS part must denote either a single meta-class or a single meta-property.
- Functors are allowed in the LHS part.
- The condition must respect this syntax:



`roundH`{ A_1 A_2 A_3 A_4 }{ `stroke_color=black, round=`σ }

Figure 6.8: [**The `roundH` Constructor**] The envelope is computed as follows: let D_1 and D_2 be the dimensions of the smallest rectangle that comprises the components. We compute the size of the rounded corner as $d = \frac{\min(D_1, D_2)}{2} \star \frac{\sigma}{100}$ where σ is the value of the `round` parameter (its default value is 50).

♢ ***cond*** ← (*ident* | `$$`) "=" ***constant***

♢ ***constant*** ← *integer* | *char* | *string* | *decimal* | *boolean*

The `$$` keyword is named *self* and the next table defines the lexemes used in these rules:

Lexeme	Regular expression	Examples
integer	`-?[0-9]+`	`123` — `-456`
char	`'`*x*`'`	`'i'` — `'''` — `'\'`
string	"..."	`"Hello"` — `"\""`
decimal	`-?[0-9]+"."[0-9]*`	`-0.13` — `12.32`
boolean	`true` or `false`	`true` — `false`

The double quote character can be escaped with the backslash character (`\"`). We will often replace the `"` character by its equivalent typographic characters: " and ".

- The self keyword is allowed only when the LHS part denotes a meta-property. Then this symbol denotes either the value of the property or its graphical representation if it appears in other expressions than a conditional statement.
- Both operands of the = operator must have identical types. Nevertheless, the following comparison are allowed: integer = real and document/sound/video/picture = string.

If the evaluation of the condition succeeds (resp. fails), then the DP will use the ***G-expr***$_1$ (***G-expr***$_2$) expression. If the `else` clause is lacking and the condition fails, then this statement just produces an empty box (null dimensions).

Example

```
meta-class $CompositeState =
           boxH{ $Name
                 if concurrent=true then " is concurrent" }
meta-property $CompositeState.Name =
              if $$=" " then "<default>"
              else $$
```

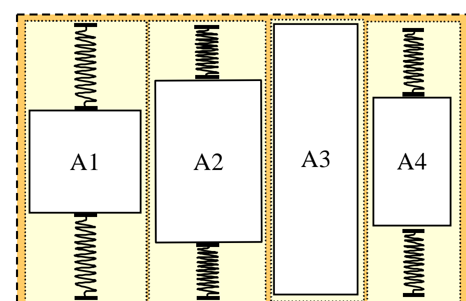
The two rules could obviously be merged but the second one illustrates the use of the self variable.

`spring`

This keyword introduces in the shape an invisible compressed spiral spring that will stretch out between the neighbouring faces. When several springs are placed side by side, their strength is added together. Hence, in `boxH{ spring spring` X `spring}`, the X symbol will be justified as $\frac{2}{3}$ / $\frac{1}{3}$. The orientation of the spring depends on the context, i.e., the box that contains it.

Example

```
boxH{
  boxV{ spring A1 spring }
  boxV{ spring A2 spring }
  boxV{ spring A3 spring }
  boxV{ spring A4 spring }
}
```



In the next script, the name of a state will centered although the description of its actions will left justified.

```
meta-class $State =
           boxV{ boxH{ spring $Name spring }
                 boxH{ $*:play spring }
               }
```

`H` **"{" dimension "}"**

This constructor produces a horizontal blank of the given dimension.

Example

```
boxH{ "|" H{1cm} "| is 1cm" }
```

will have as result this picture:

| | is 1cm .

`V` **"{" dimension "}"**

This constructor produces a horizontal blank of the given dimension.

`handle` ⟨**"{"handle-opt"}"**⟩

Handles permit the display processor to link together graphical objects. If the two objects to link together have handles, they will be used as ends. The default DP's behaviour is to use the handles which minimize the length of the drawn vertice. However, some methodologies[1] have distinct semantics based on the "bollard". For this reason, the method engineer can name his handles and reference them explicitely when defining the graphical representation of the vertices in the *connection section*.

$\diamondsuit$ ***handle-opt*** $\leftarrow$ $\langle$***opt-visible opt-index***$\rangle_{1,\infty}^{\text{“,”}}$

$\diamondsuit$ ***opt-visible*** $\leftarrow$ "visible"= *boolean*

[1]SADT for instance.

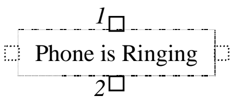
◇ ***opt-index*** ← "index"= *integer*

Options can be repeated and index values should be unique in a graphical object. The default value of the "visible" option is `false`. When this is `true`, then handles are displayed as small boxes. This is helpful for debugging unexpected behaviours.

Example

```
meta-class $State = boxV {
  boxH{ spring handle{visible=true, index=1} handle }
  boxH{ handle $Name handle }
  boxH{ spring handle{visible=true, index=2} handle }
}
```

This script places four handles around the name of a state. The top and bottom handles are named in order to attach them specific vertices (meta-relations/paths). Handles are by default invisible and take no place (dimensions are null).



`explode`

The `explode` keyword denotes the explosion of a specification.

1. The LHS part of the rule must denote a meta-model M'.
2. M' is either the current meta-model ($M' = M$) or a subtype (M' **isa*** M).

Remark Every script defined for a meta-model still is valid for its subtypes.

Once, the DP will encounter this keyword, the "thing" it is displaying will necessarily be a specification (s'). The DP will use the current GraSyLa script to present this specification on the display. Once this specification has been displayed, its smallest rectangular envelop is computed and used as the resulting box of the graphical representation of the `explode` keyword.

Example

```
meta-class $CompositeState=
  boxV{ boxH{  spring $Name spring }
        ruleH{ stroke_width=2pt }
        boxH{  spring explode spring }
      }{ stroke_width=2pt }
```

`bitmap` **string** ⟨ **"{" bitmap-opt "}"** ⟩

- The string must be a valid file name pointing to a bitmap picture (extension `.BMP`).

This constructor will display the image contained in the specified file. The following options are valid:

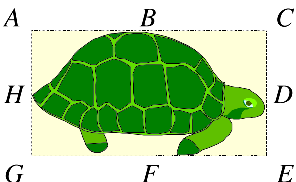


Figure 6.9: [**The** `bitmap` **Constructor and its handles**]

"factor" "=" *integer* : The integer value must be greater than 0 (≥ 1). This denotes the value of the magnifying glass. The 100 value denotes a 1:1 ratio.

"handle" "=" *identifier* : The name of this identifier must be formed of the letters 'A' ... 'H'. Each letter corresponds to a handle that will be managed by the DP. The position of these handles is shown in Fig. 6.9.

`bitmap` **$ident** ⟨ **"{" bitmap-opt "}"** ⟩

- The LHS part of the statement must denote a single meta-class C.
- *ident* must be a direct or inherited meta-property of C: P.
- This meta-property must denote a picture: P.type = *picture*.

`ruleH` ⟨ **"{" rule-opt "}"** ⟩

The `ruleH` acts as a horizontal spring that stretches over the free space. However, this spring does not have any effect on the position of its neighbour boxes as did the `spring` command. For instance, this GraSyLa expression

```
boxV{"XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
     ruleH{ stroke_color=dark_red,
            stroke_width=20pt,
            fill_style=diagcross }
     boxH{ ruleV{ stroke_color=dark_red, stroke_width=50pt}
           "YYYYYYYYYY" } }
```

will produce this shape:

Replacing the `ruleV` command with `ruleH` would erase the large black rule. Indeed, the space left to this spring would be null.

The possible options are:

stroke-color **:** defines the colour of the stroke. Its default value is "invisible".

stroke-width **:** the width of the stroke. Its default value is 1.

stroke-style **:** The style of the stroke. Its default value is “solid”.

fill-style **:** The style of the stroke. Its default value is “solid”. This option hides the ***stroke-style*** option and replaces it. The stroke is then considered as a filled rectangle.

Example

```
boxV{ H{2cm}
      boxH{ ruleH{ stroke_width=2pt }
            bitmap "bullet.bmp"
          }
    }
```

ruleV ⟨**“{”rule-opt“}”**⟩

The `ruleV` acts as a vertical spring that stretches over the free space.

Example

```
boxH{ ruleH{ stroke_width=2pt }
      bitmap "bullet.bmp"
    }
```

font **“{” G-expr “}” “{” ⟨font-opt⟩$_{1,\infty}^{\text{“,”}}$ “}”**

The `font` constructor defines the font that its components will use to display their texts. The parameters of the font are defined in the ***font-opt*** part. Let us notice that this command just modifies the current font and does not create a new one. The last font options are thus preserved unless they are redefined. The options are:

name **:** the name of the font. If this name does not correspond to any font on the used system, then the command has no effect. The default font is “arial” (or another one if it is not installed).

color-font **:** the colour of the font. The default value is “black”.

background **:** the background of the text, the default colour is “white”.

bold **:** (dis)active the bold font. The default value is `false`.

italic **:** (dis)active the italic font. The default value is `false`.

underline **:** (dis)active the underline font. The default value is `false`.

size **:** the size of the font. The default value is 10.

The syntax of the possible options is defined as:

♢ ***name*** ← “name” “=” *string*

♢ ***color-font*** ← “color” “=” (***color_cst*** | ***rgb_cst***)

◇ ***background*** ← "background" "=" (***color_cst*** | ***rgb_cst***)

◇ ***size*** ← "size" "=" *integer*

◇ ***bold*** ← "size" "=" *boolean*

◇ ***italic*** ← "italic" "=" *boolean*

◇ ***underline*** ← "underline" "=" *boolean*

Example The next script:

```
font{ boxH{ "a bold" font{ "string" }
                         { name="courier", underline=true }
          }
    { bold=true }
```

will display "**a bold** ***string***"

6.4.5 The Parameters of the Main Section

Some parameters in GraSyLa expressions are shared by several GraSyLa expressions and have both the same semantics and the same syntax. This section lists these parameters and defines their syntax. Their respective semantics has been explained above.

◇ ***stroke-color*** ← "stroke_color" "=" (***color_cst*** | ***rgb_cst***)

◇ ***color-cst*** ← "black" | "light_gray" | "gray" | "light_red" | "light_green" | "light_blue" | "light_magenta" | "light_cyan" | "white" | "dark_blue" | "dark_green" | "dark_cyan" | "dark_red" | "dark_magenta" | "dark_brown" | "yellow"

◇ ***rgb-cst*** ← `rgb` "(" *integer* "," *integer* "," *integer* ")"
The three integer values denote respectively the green/red/blue components of the colour and must be comprised between 0 and 255.

◇ ***stroke-width*** ← "stroke_width" "=" ***dimension***

◇ ***margin*** ← "margin" "=" *integer*

◇ ***round*** ← "round" "=" *integer*

◇ ***fill-color*** ← "fill_color" "=" (***color_cst*** | ***rgb_cst***)

◇ ***dimension*** ← *integer* ("pt" | "cm" | "mm")

◇ ***fill-style*** ← "margin" "=" ("horizontal" | "vertical" | "cross" | "diagcross" | "solid" | "bdiagonal" | "fdiagonal")

◇ ***stroke-style*** ← "stroke_style" "=" ("solid" | "dash" | "dot" | "invisible")

◇ ***handles*** ← "handles" "=" *integer*

6.4.6 The Connection Section

The *connection section* lists which meta-relations will be displayed as vertices linking together the root/junk graphical objects. Every statement in this section will correspond either to a meta-relation or to a path and will describe its appearance in the display. More particularly, they will define the shape of the extremities and the look of the vertices (i.e., the lines).

The general syntax of this section is defined as:

◇ ***connection-section*** ← ⟨connection-rule⟩$_{0,\infty}$

◇ ***connection-rule*** ← `meta-relation` "$" *identifier* "=" `connect`
"{" ***arrow-shape handle-connect*** "}"
"{" ***line-shape*** "}"
"{" ***arrow-shape handle-connect*** "}"

The identifier in the LHS part of the rules must denote a meta-relation or a valid path. The three arguments in the RHS part describe respectively the shape of the "owner" side, the body of the line, and the shape of the "member" side of the vertice. For a path (say $\rho_1 \bullet \cdots \bullet \rho_n$), the "owner" will denote the ρ_1's extremity and the "member" will denote the ρ_n's extremity. The ***handle-connect*** parameter restricts the handles to which the DP can attach the vertice. Each index mentioned in this parameter value should correspond to a handle defined in the connected class. Nevertheless, undefined handle indices will just be skipped by the DP.

The ***arrow-shape*** argument can be empty or a list of figures that will be assembled together like perls on a rope. Its syntax is defined as:

◇ ***arrow-shape*** ← ⟨***arrow-atomic***⟩$_{0,\infty}$

◇ ***arrow-atomic*** ← ***inner*** | ***outside*** | ***cross*** | ***bar*** | ***bullet***

◇ ***inner*** ← `inner`"{" ***arrow-options*** "}"

◇ ***outside*** ← `outside`"{" ***arrow-options*** "}"

◇ ***cross*** ← `cross`"{" ***arrow-options*** "}"

◇ ***bar*** ← `bar`"{" ***arrow-options*** "}"

◇ ***bullet*** ← `bullet`"{" ***arrow-options*** "}"

Each type of extremity will have its own parameters (***arrow-options***) as follows:

	inner	*outside*	*cross*	*bar*	*bullet*
stroke-color	×	×	×	×	×
stroke-width	×	×	×	×	×
fill-color	×	×			×
size	×	×	×	×	

The options have the same syntax and the same semantics as in the main section. The ***line-shape*** can be composed of the ***stroke-color***, ***stroke-width***, and ***stroke-style*** options.

The ***handle-connect*** part is defined as:

◇ ***handle-connect*** ← `handles` "{" ⟨*integer*⟩$_{1,\infty}$ "}"

Example The next script describes the display of a meta-relation. Extremities are both composed of three shapes[1].

[1]The method engineer can leave them empty.

```
meta-relation $incoming = connection
     { outside{ size=5, fill_color=black }
       inside{ size=5, fill_color=gray }
       bar{ size=10 } }
     { stroke_color=black, stroke_style=dash }
     { outside{ size=5, fill_color=black }
       bullet{ size=5, fill_color=white }
       inside{ size=5, fill_color=gray }
       cross{ size=5 } }
```

Each relation will appear like this ◀▶|———▶○◁× on the display.

6.4.7 The Display Processor

The GraSyLa architecture is depicted in figures 6.10 and 6.11. Each display device[1] is controlled by a display processor. This machine has to display a specification with respect to a GraSyLa script and must manage all the possible interactions with the software engineer. Every DP has its own memory, it can access the repository and can communicate via a common black-board[2] with other DPs. They are smart enough to detect any modification of their input script and they will refresh the display each time the engineer wishes to change a script definition. The software engineer can ask for several displays of a specification whatever the script. He can thus visualize several parts of a huge specification in distinct windows simultaneously.

The repository stores the (x, y) positions of the main objects only. Their shape description is automatically computed by the GraSyLa machine. This independence makes it possible to keep the layout of a diagram even if its definition changes. And last but not least, the repository schema definition[3] is the same whatever the visualizations (i.e., the GraSyLa scripts) defined by the method engineer!

6.5 GraSyLa and Inherited Definitions

The repository allows us to derive meta-models from other ones by inheritance. Such meta-models encompass the definitions of the inherited meta-models and axiom 9 (*cfr.* page 45) ensures us that all the views (i.e., the GraSyLa scripts) defined on the supertypes are still valid for the subtypes. Indeed, the root and junk meta-classes are necessarily meta-classes of the subtype meta-model.

The GraSyLa language allows us to derive new scripts from previous ones. This makes it possible to reuse GraSyLa scripts that were developped for other meta-models. Thus, when a meta-model M_1 inherits from another meta-model M_2, any GraSyLa script defined to display specifications of M_1 can reuse any script that has been defined for M_2. Moreover, it is possible to override statements that would have been defined for M_2 but which would not match the graphical requirements of M_1. But the reuse of scripts is not limited to the inheritance

[1] i.e., a window, a printer, a clipboard, . . .
[2] For instance, selected items are exchanged between displays via this black-board during Cut&Paste operations.
[3] i.e., the meta-meta-model

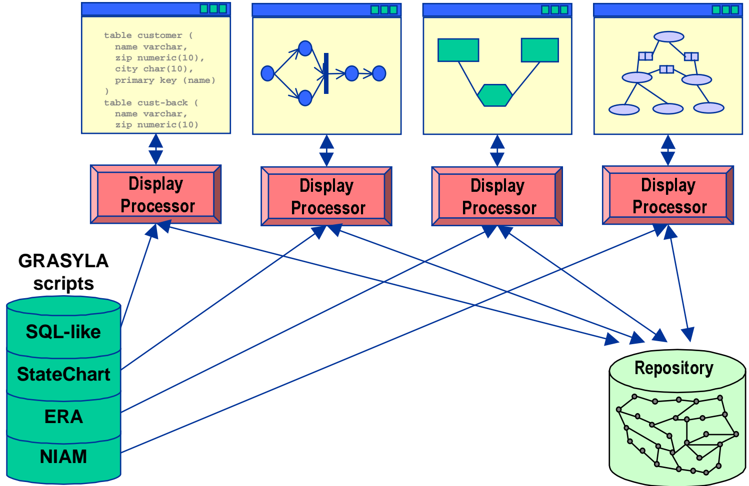


Figure 6.10: [**The** GraSyLa **Architecture**] They are as many Display Processors as displays. The events relative to a display are trapped and managed by its DP. DPs access the repository only to read the classes description and to update their positions once the display is closed. Several DPs can process a same specification whatever their scripts. This allows several displays with distinct views of a same specification.

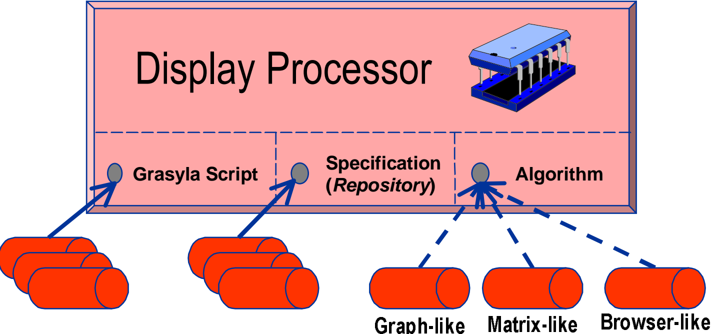


Figure 6.11: [**The Display Processor**] The explosion of the Display Processor shows three components: *1)* the GraSyLa script, *2)* the specification it will display, and *3)* the algorithm in charge of placing the graphical objects on the "display". The last component is hand-coded by the meta-CASE architect and can be selected with a "switch" in the GraSyLa script.

between meta-models. A script can also reuse scripts that were developped for the same meta-model. This makes it possible to customize very easily the graphical representation and to avoid redundancy between scripts (i.e., the same thing explained/defined several times in distinct scripts).

Scripts can thus inherit the entries of previous $-functions and, moreover, override partially (or completely) their definition. The ***use-clauses*** part[1] allows the method engineer to specify the inherited scripts. Each clause will force the display processor to load the respective scripts into its memory. When conflicts occur (the $-function is defined several times for the same arguments), the last occurrence overrides the previous one. The root and junk directives are skipped and are thus not taken into consideration. Only the root and junk directives of the main script are pertinent. The syntax of this part is defined as:

$$\diamondsuit\ \textit{\textbf{use-clauses}} \leftarrow \left\langle \texttt{use}\ \textit{string}\ \text{“; ”} \right\rangle_{0,\infty}$$

Every string must denote a file corresponding to a valid GraSyLa script.

Example The schema depicted in Fig. 6.12 shows how it is possible to derive and to extend a GraSyLa script from other scripts.

6.6 GraSyLa and Dedicated Display Processors

The hand-coded algorithm to place the classes (i.e., the roots) on a display is convenient for graph-like specifications very well. However, some views require special algorithms that cannot be modelled directly with GraSyLa : Matrices, Tables, Browsers, Sequence Diagrams, Screen Layouts, ...

For this reason, such algorithms have been encapsulated into a component with a well-delimited interface. The intelligence denoted in the GraSyLa scripts is itself stored in an autonomous component that serves the display processors which play then the *client* role. This architecture allows the meta-CASE architect to replace the DP component by another one. This architecture is illustrated in Fig. 6.13.

The meta-CASE architect can implement hard-wired dedicated graphical processors that will use the user-defined statements GraSyLa scripts to display the specifications. Hence, in a hypothetical security meta-model (which models the grants of a database system), we could think of writing a DP to display graphical matrices. For instance, a matrix with the pictures of users as x-axis and operations as y-axis. Cells of such a matrix could be a textual form of the respective rights granted (read/write/ ...).

When such specific algorithms are implemented, the method engineer can change the default algorithm with the ***engine-clause*** statement at the beginning of scripts. Its syntax is defined as follows:

$$\diamondsuit\ \textit{\textbf{engine-clause}} \leftarrow \texttt{engine}\ \textit{ident}\ \left\langle\ \text{“\{”}\ \left\langle \textit{ident}\ \text{“=”}\ \textit{string}\ \right\rangle_{1,\infty}^{\text{“,”}}\ \text{“\}”} \right\rangle\ \text{“;”}$$

The first identifier denotes some builtin algorithm name listed in the documentation. This one can accept some parameters between accolades. In pairs *ident* “=” *ident*, the first identifier denotes the name of the parameter and the next one is its value.

[1]See section 6.4.2 page 115.

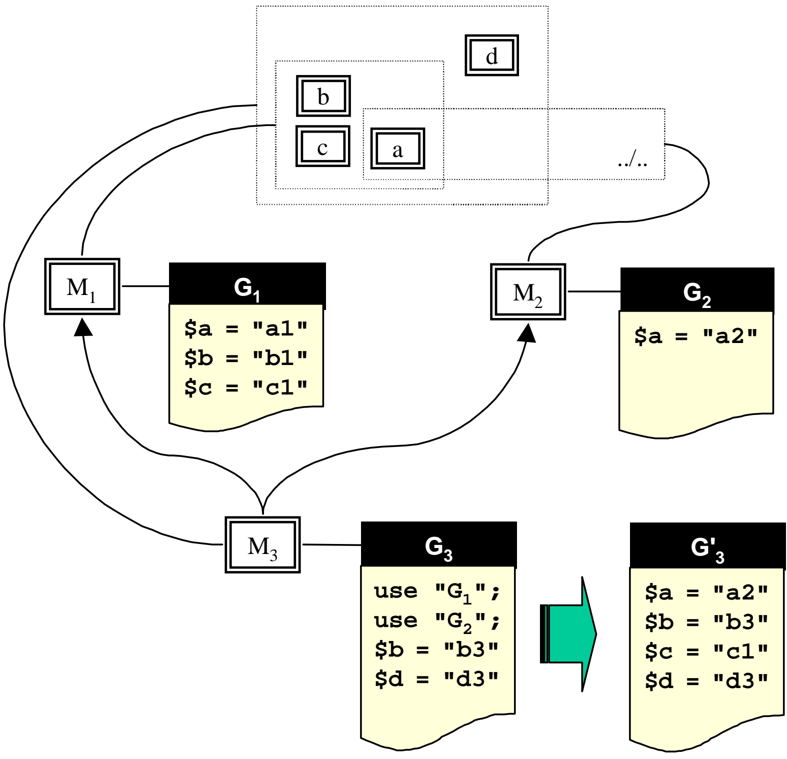


Figure 6.12: [GraSyLa **and Inheritance**] M_1, M_2 and M_3 denote three meta-models. G_1, G_2 and G_3 are three GraSyLa scripts that were defined in association with these respective meta-models. Script G_3 has been defined on top of the G_1 and G_2 scripts. G'_3 is a *virtual*[2] script that results from the three previous scripts. Each entry of the G_1 (resp. G_2) script will appear in G'_3 unless it has been overridden in $G_{2,3}$ (resp. G_3).

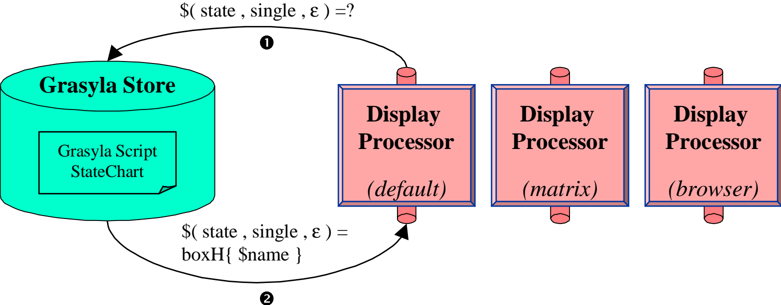


Figure 6.13: [**The Component Architecture of** GraSyLa] Each GraSyLa script is stored in a distinct component that acts as a server for the display processor. There is one store component per script, but the component does not change whatever the DP.

When the engine clause is specified, the root and junk clauses are not pertinent and will be skipped. When a script has ***use-clause***s, only the main engine (the last in the list) is taken into consideration.

Example This example specifies a matrix-like view as described above.

```
engine matrix { col="user", line="operation",
                cell="privilege",
                cocell="granted", licell="for" };
```

Example The next clause specifies a browser-like view that will display the meta-classes of the meta-model. The expansion of a meta-class will show its instances and the expansion of one instance, will show its properties, its supertypes, and its roles. Roles can themselves be expanded in instances and so-on. The classes will be displayed with the look defined in the GraSyLa script.

```
engine browser;
```

6.7 Examples

6.7.1 A Statechart Meta-Model

This section confronts the expressivity of GraSyLa with the graphical requirements of the Statechart diagrams defined in the UML reference guide [Rat97a]. Nevertheless, we will use the "Normal Statechart" as meta-model (page 92). It has about the same expressivity and is simpler than the original [Rat97b]. We use the examples depicted on pages 104 and 107 of [Rat97a] as benchmarks (with some minor extensions).

The GraSyLa script is depicted in section D.3.2 page 263. It is composed of only 10 GraSyLa equations and 2 connection statements. Moreover it defines a behaviour that is much more general than the one illustrated in the benchmark (for instance: there are no "fork" or "synchro" states). The script has only 72 lines with a normal indentation and the expressions were composed with less than 6 constructors (`boxH`, `font`, `H`, ...). Statements were thus written with less than 6 lines on average. This conciseness is quite oustanding when we compare it to other approaches such as that of MetaView [Fin94a, GFS+94, GLM94].

The original state diagram has been shown in Fig. 6.14 and the GraSyLa output in figures 6.15 and 6.16. Unfortunately, the meta-CASE prototype does not yet allow to explode specifications inside other specifications. Hence, the exploded view such as proposed in the UML reference guide cannot be showed yet. This implementation is still in progress.

6.7.2 Synchronized Textual and Graphical Views

Many situations require to have both graphical and textual representations of an information system. These views correspond to complementary requirements. One one side, graphical views will help the software engineer to grasp the semantics of a system in a quite natural and intuitive way. On the other side, textual views will be preferred for both their completeness and their conciseness[1].

[1] The information is presented as a sequence of lines, this takes generally less space than the layout of graphical diagram.

This section demonstrates the capability of GraSyLa to propose both graphical representations (i.e., graph-like diagrams) and textual views. Our exercice consists in proposing both synchronized views of a relational database modelled with high-level concepts that have a straightforward translation to the relational model. For instance, the software engineer can define mandatory one-to-many relationships between its entity-types (tables). The graphical representation shows the database schema with ERA conventions altough the textual view represents it with SQL statements. All the views are obvsiously synchronized and active. The software engineer can "double-click" on a foreign key and edit/delete the underlying relationship. This modification is automatically propagated to the other views that share this information. The screen dump of this exercice is shown in Fig. 6.17. The scripts and meta-model definition are listed in section D.3.3 page 264. This meta-model still contains marks of one-to-many relationships that are outside the relational ontology. Nevertheless, their straightforward translation to the relational model when the preconditions are fulfilled prompted us to call it "pseudo relational meta-model". This name is obviously an abuse of notation.

The GraSyLa script gives the impression to transform every relationships into a foreign key. Those transformations are of course very naive. The script just "pumps" the content of the primary key to add its contents in the table on the member side of the relationship. If the table/entity-type has no primary key, then the foreign key will be empty.

6.7.3 An Editor for GraSyLa Scripts

We will use GraSyLa to define a graphical editor for the GraSyLa scripts. GraSyLa scripts have a quite strict syntax and the statements must follow precise semantic rules. Although the meta-CASE allows the method engineer to test its scripts "on-line"[1], this task can be cumbersome. A natural idea to put this problem right is to propose a validating editor for GraSyLa itself. Thanks to the mirroring service, it is possible to define a meta-model which uses the elements of the meta-meta-model. Hence, it is possible to model the GraSyLa scripts with a meta-model that will associate meta-classes, meta-properties, and so forth with the GraSyLa expressions.

We will thus define a new meta-model that will model GraSyLa scripts. Unfortunately, this meta-model requires too many concepts and would be very difficult to visualize if it was defined in one go. For this reason, we define three meta-models which correspond to the projection of the resulting meta-model on *a)* the GraSyLa expressions, *b)* the connections, and *c)* the script's skeletons. Their integration is achieved by inheritance: the final meta-model is defined as a subtype of these three meta-models.

The GraSyLa's meta-model (`Script-Grasyla`) is illustrated in Fig. 6.18. It inherits from respectively three meta-models `G-expression` (Fig. 6.19), `G-script-rel` (Fig. 6.20), and `G-script-class` (Fig. 6.21.). These meta-models are virtual and can therefore not be instantiated without having a subtype. Indeed, they are here just intermediate meta-models that make the understanding of the global meta-model easier. We give hereafter an overview of the ontology they represent.

`G-expression` **:** It models the composition of the GraSyLa expressions. `G-expr` denotes an expression and it has as many subtypes as they are kinds of expressions in GraSyLa .

[1] It is possible to repeat the *load a script/change the script* operations several times in order to visualize the impact of the modifications on a sample specification.

G-arg-owner denotes constructors that are made of other expressions.

G-script-rel : It models the *connection section* of scripts. The key concept of this meta-model is G-connection. This meta-class will have as many instances as they are "connection" statements in the script. A connection is made of either a meta-relation or a path that is itself composed of other meta-relations. Finally, the connection is defined with a sequence of extremities (G-extremity) that can be crosses, bars, bullets, or inner/outside arrows.

G-script-class : It links together the GraSyLa expressions (*cfr.* G-expr), the functors and the concepts to display (meta-classes or meta-properties) into an equation (G-equation). The meta-model uses the particular semantics of the inheritance to mean that a meta-class can either be *root* or *junk* in one script but not both.

Figure 6.22 shows a snapshot of the meta-CASE. The front specification is a GraSyLa editor with some equations of the Statechart diagram. The window comprises the equations for the Initial and Final states. Each sentence can be edited independently. Although the view is "text-oriented", it remains a graphical display and the final text must be generated with a Voyager 2^+ script. The script should also perform some validation to check the conformance of the specification with constraints that were not defined in the meta-model. Indeed, a meta-model that would automatically check all the implicit constraints would make the edition too difficult. Such constraints would make "forward referencing" impossible although this is often required, since people often specify their requirements by need and not in a bottom-up way.

This example illustrates the use of the language to define textual views of specifications. GraSyLa can be used in a similar way to propose textual representation of programs, formal specifications, graphical ressources[1], but also hypertexts by using the explosion facility as an hyper-link mechanism.

Listing

The complete listing of the GraSyLa script that defines the GraSyLa editor is given in section D.3.1 page 260. We show hereafter the excerpts relative to the script displayed in Fig. 6.22.

```
view "Grasyla Editor" ;

begin

root: <G-root>, <G-junk>, <G-equation>, <G-connection>, <G-path> ;
junk: <G-functor>, <meta_class>, <G-expr> ;


/*******************************/
/* Display of Grasyla statement */
/*******************************/

meta-class $<G-equation> = boxH{ $1:<G-arg>
                                 used($1:<G-has-functor>)
                                 if <?list>=true then  "list" else ""
```

[1]The .RC files of a Windows program.

```
                                 " = "
                                 $1:<G-rel-expr>
                               }
meta-class $<G-object> = boxH{ arg($1:<G-rel-prop>)  arg($1:<G-rel-class>) }
meta-class arg($<meta_class>) = boxH{ "meta-class " $name }
meta-class list $<G-expr> = boxV{ head tail }

/***********************************/
/* display of a Grasyla expression */
/***********************************/


meta-class $<G-circle> =
    boxV{ boxH{ if <?horizontal>=true then "circleH {" else "circleV {"
                $*:<G-members> "}"
                spring
              }
          boxH{  "{" boxV{ if <stroke-color>="" then ""
                           else boxH{ "stroke-color= "  $<stroke-color> spring }
                           if <stroke-width>=0 then ""
                           else boxH{ "stroke-width= "  $<stroke-width> spring }
                           if <stroke-style>="" then ""
                           else boxH{ "stroke-style= "  $<stroke-style> spring }
                           if <fill-color>="" then ""
                           else boxH{ "fill-color= "  $<fill-color> spring }
                           if <margin>=0 then ""
                           else boxH{ "margin= "  $<margin> spring }
                           if <handles>=0 then ""
                           else boxH{ "handles= "  $<handles> spring }
                         }
                  "}" spring
              }
        }{ stroke_color=black, stroke_width=1, margin=2pt }
meta-class $<G-ident>= $name
meta-class $<G-rule>= boxH{ if <?horizontal>=true then "ruleH " else "ruleV " "{"
                            boxV{ boxH{ "stroke-color= "  $<stroke-color> spring }
                                  boxH{ "stroke-width= "  $<stroke-width> spring }
                                  boxH{ "stroke-style= "  $<stroke-style> spring }
                                  boxH{ "fill-style= "  $<fill-style> spring }
                                }
                            "}"
                          }
end
```

6.7.4 The "Firework" Sample

It is somewhat frustrating to limit the examples to didactic case studies. Indeed, their size and their complexity do not allow us to illustrate real GraSyLa 's capacity to face involved situations. The snapshot depicted in Fig. 6.23 shows how the meta-CASE (and GraSyLa) is able to manage and to display strongly inter-connected specifications with respect to distinct graphical requirements.

6.8 Summary

We have presented a declarative language (GraSyLa) to define the graphical representation of specifications with symbolic statements. This component is crucial since it will be one of the most visible aspects of our meta-CASE.

GraSyLa has been defined in such a way that the requirements about the meta-model definition and its graphical representations can be completely dissociated — they have no reciprocal influence. Besides this quality, GraSyLa takes into consideration important requirements that are, so far, still not supported by concurrent works (*cfr.* section 6.2): multiple textual and graphical active synchronized views, contextual and polymorphic representation, evolving requirements, and multimedia data. The language also matches very well the peculiar semantics of the repository (multivalued meta-properties, meta-relations, inheritance, . . .).

Despite all those requirements, the language remains simple, expressive, and very easy to use. Scripts have rarely more than some tens statements.

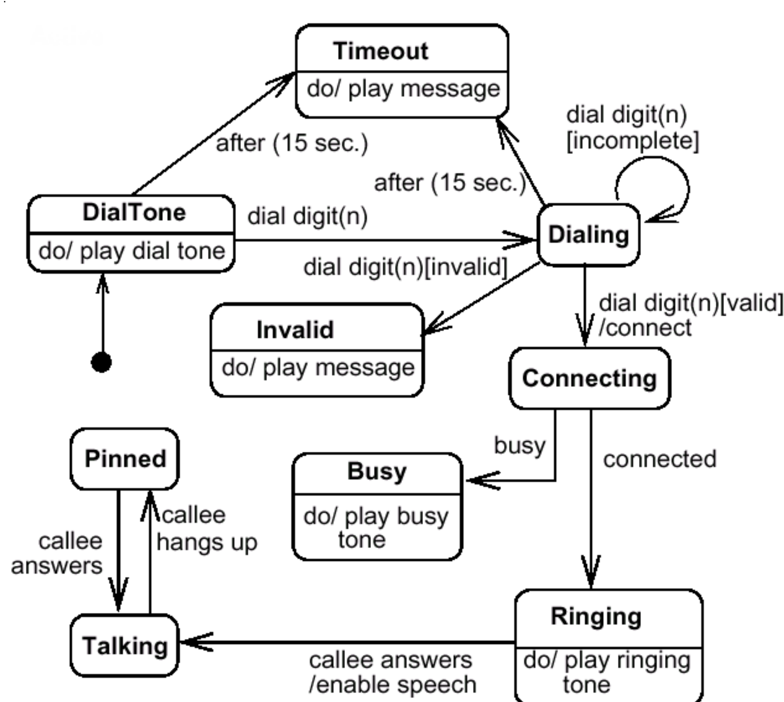


Figure 6.14: [**The Original "Phone" State Diagram**] The original state diagram of the "phone" example. This schema has been reproduced from page 104 of [Rat97a].

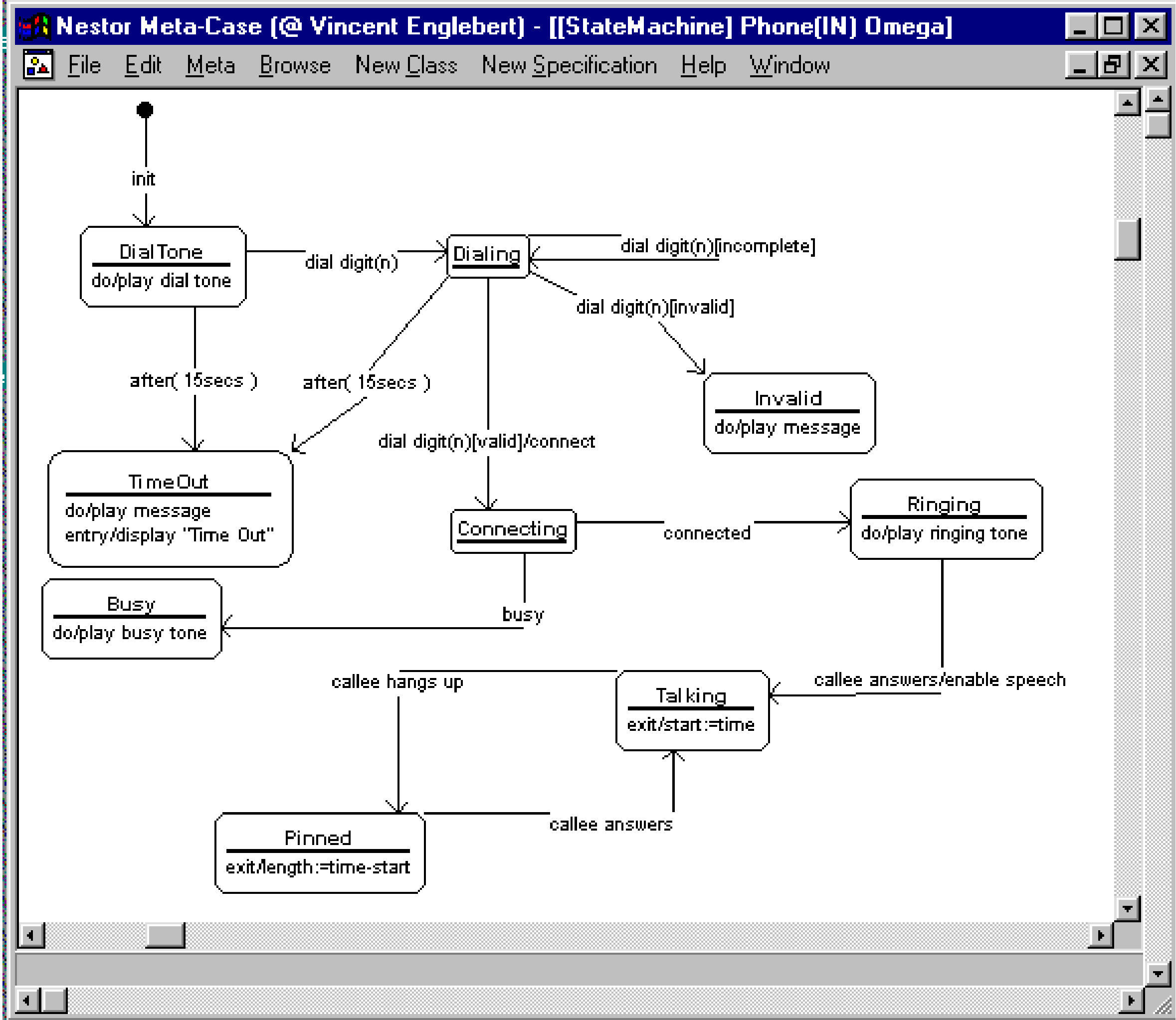


Figure 6.15: [**The** GraSyLa **Version of the "Phone" State Diagram**] This snapshot shows the "Phone" state diagram displayed with the GraSyLa "Statechart Editor" (*cfr.* section D.3.2).

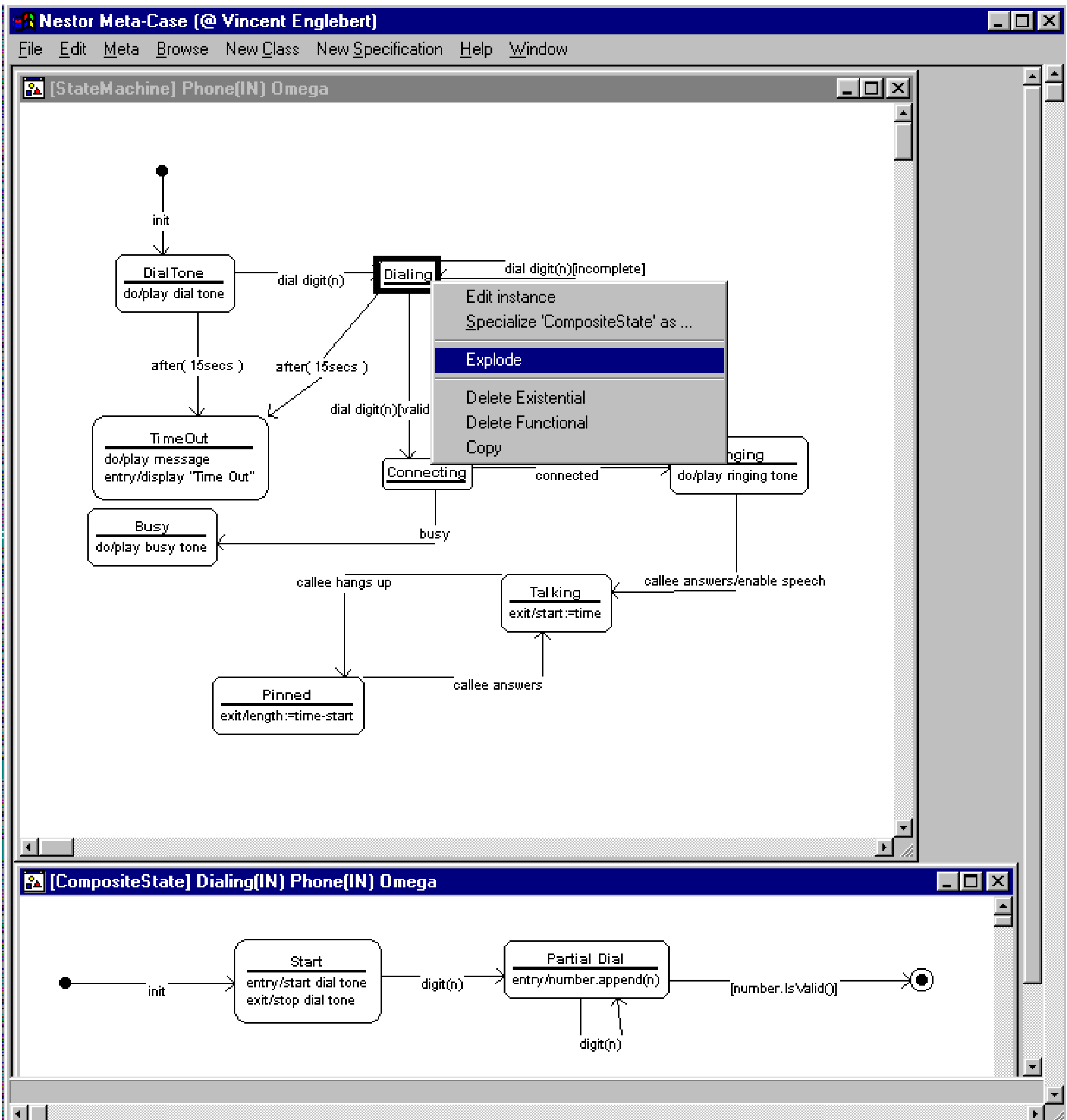


Figure 6.16: [**The** GraSyLa **Version of the "Phone" State Diagram (with explosion)**] This snapshot illustrates the use of the explosion mechanism on the `Dialing` composite state.

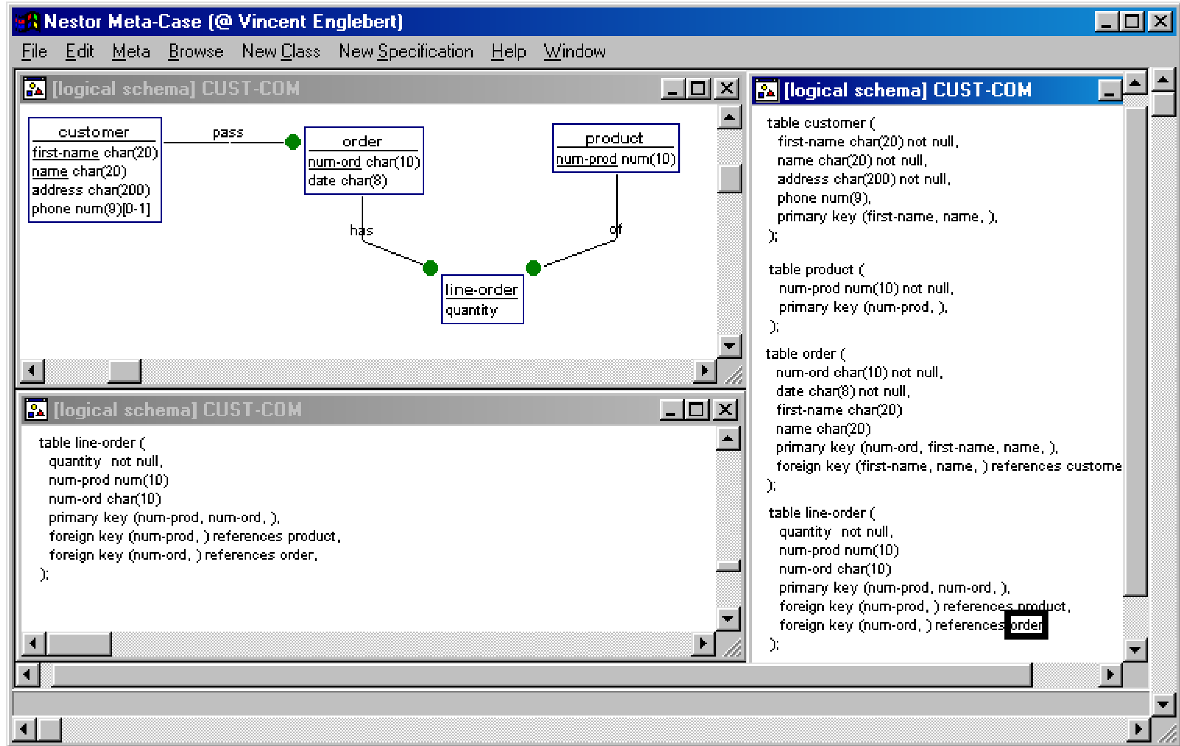


Figure 6.17: [**Textual and Graphical Synchronized Views**] The three panes represent the same information but displayed according to distinct requirements. The top-left window represents a relational schema with ERA-like graphical conventions, the other windows represent the SQL statements that implement this schema. Due to some weaknesses of the GraSyLa semantics, the comma character has been used as a "terminator" rather than a "separator". This explains the small divergences with the formal syntax of SQL.

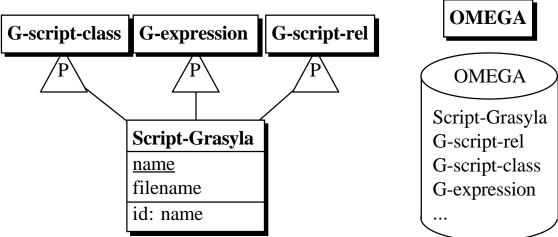
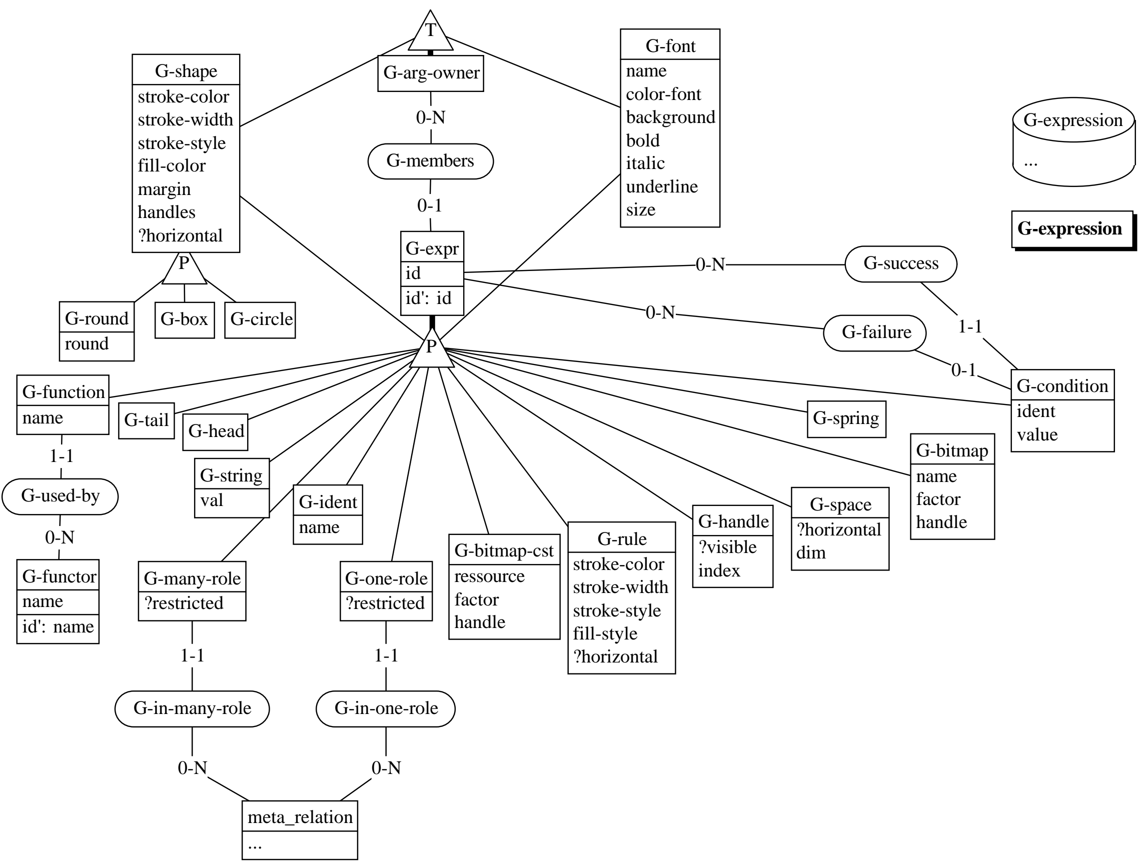


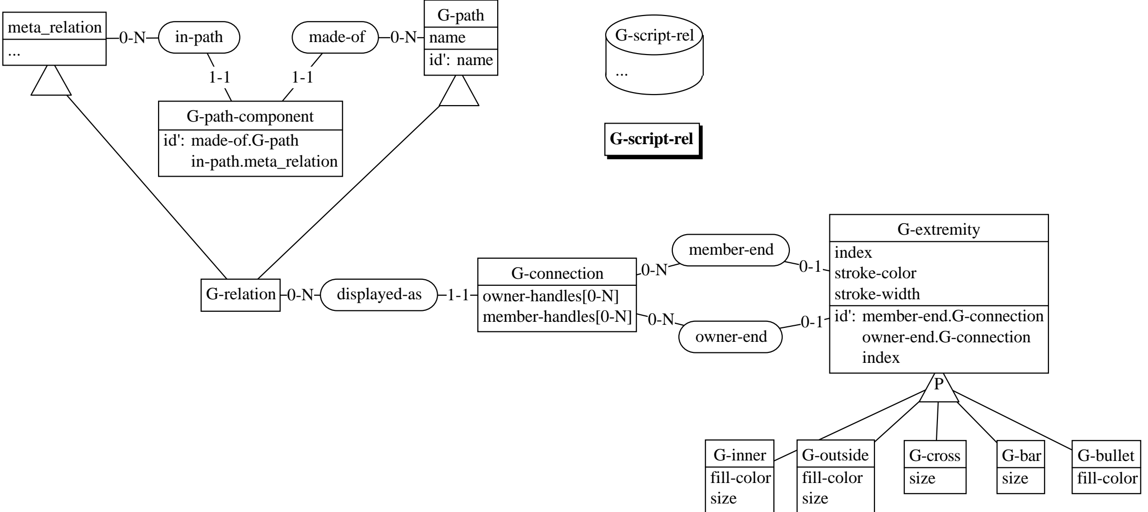
Figure 6.18: [**The `Script-Grasyla` Meta-Model**]



Additional constraints:

- Roles `G-success`—`G-expr` and `G-failure`—`G-expr` have 0-1 cardinalities.
- Roles `G-success`—`G-expr`, `G-failure`—`G-expr`, `G-members`—`G-expr` and `G-rel-expr`—`G-expr`[1] are exclusive.

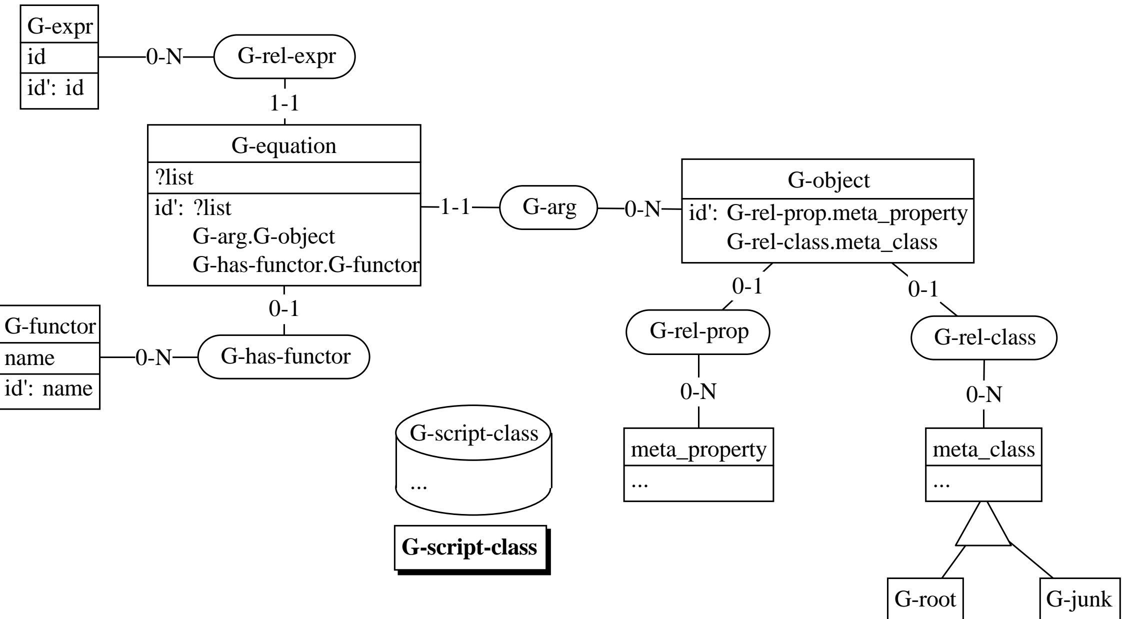
Figure 6.19: [**The `G-expression` Meta-Model**]



Additional constraint:

- Roles `member-end`—`G-extremity` and `owner-end`—`G-extremity` are exclusive and `G-extremity` must play one of these ones.

Figure 6.20: [**The** `G-script-rel` **Meta-Model**]



Additional constraints:

- The `G-rel-expr`—`G-expr` role have 0-1 cardinalities.
- The `G-object` meta-class must play exactly one role of either `G-rel-prop` or `G-rel-class`.

Figure 6.21: [**The** `G-script-class` **Meta-Model**]

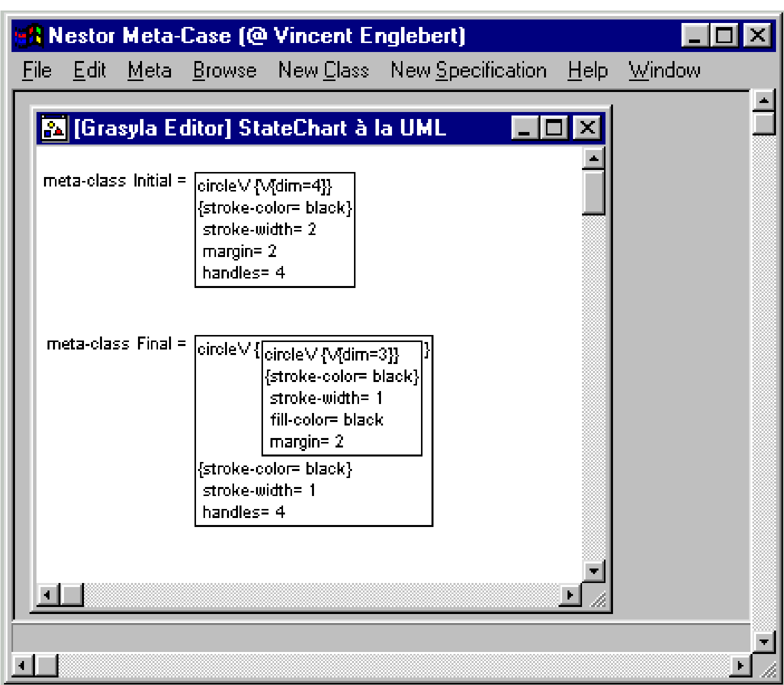


Figure 6.22: [**The** GraSyLa **Script of the** GraSyLa **Editor**]

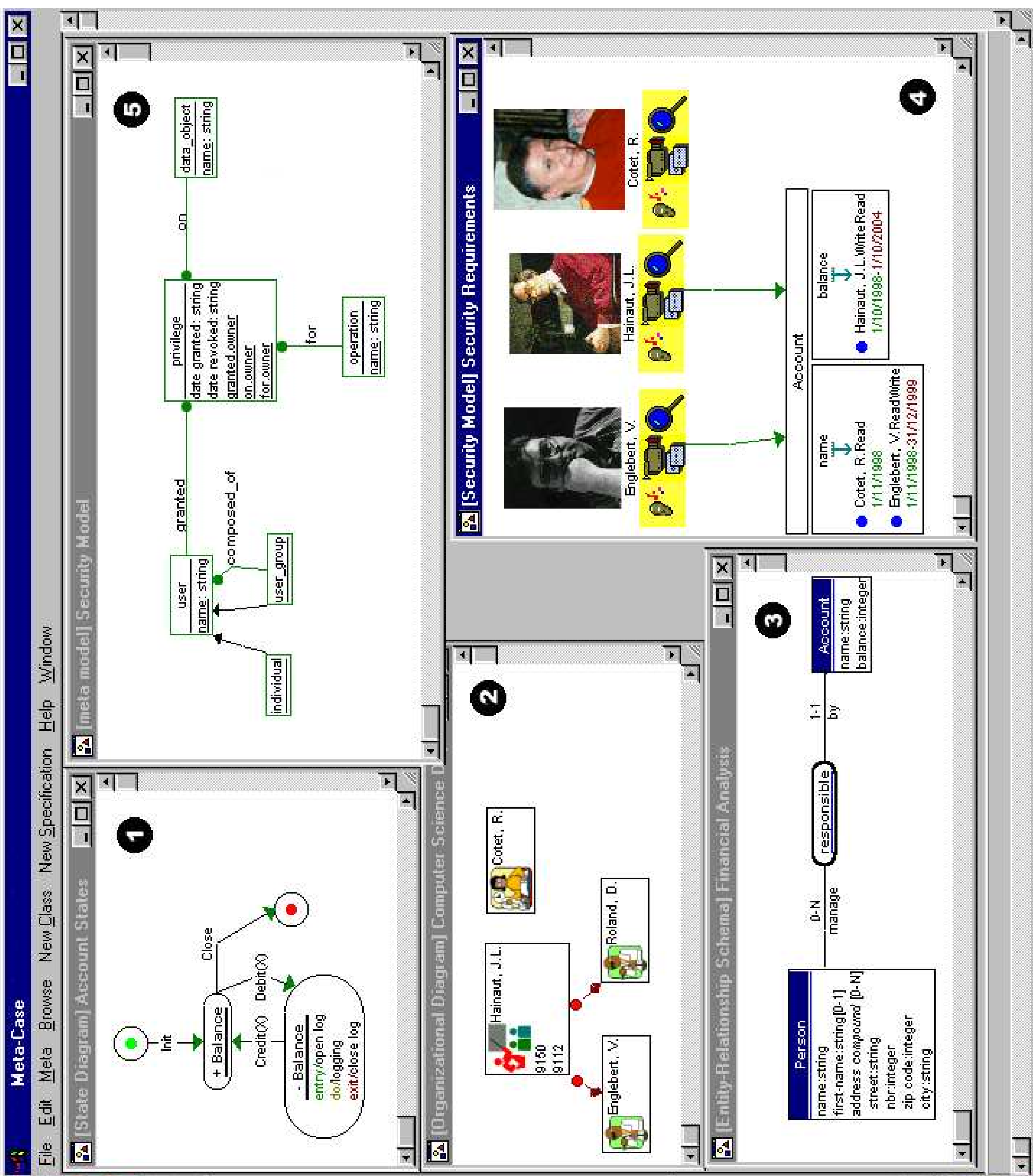


Figure 6.23: [**The Firework Sample**] Four specifications and one meta-model definition are visualized in this screen shot. Windows denote respectively *1)* a state diagram, *2)* an organization diagram, *3)* an entity-relationship schema, *4)* a specification about database's security, and *5)* the visualization of the database security's meta-model definition. In window *4*, the `account` entity-type is displayed as a table, and each column contains the grants (*who*, *what* and *when*). The arrows denote privileges on the whole table/entity type. Let us remark that the `account` concept is shared by specifications *3* and *4*, and that the `hainaut` person is itself shared by specifications *2* and *4*. Icons (sound/camera/magnifying glass) are active and a double click on them shows a multimedia document

> *"[. . .] I had not only to make language design decisions but also to implement them. Under these circumstances it is a question of whether to include more features, or to get ahead first with those needed to implement the compiler."*
>
> N. Wirth
> 1993 in Cambridge, Massachusetts.

Chapter 7

Voyager 2^+

7.1 Preliminaries

We believe that a meta-CASE without an integrated programming language has little chance to compete with hand-written CASE tools. Indeed, whatever the level of abstraction/expressiveness of a meta-meta-model, such an approach will always be limited to the representation of data.

Hence, a meta-CASE needs a programming language to define the behaviour of the objects and processing functions. But a wide variety of languages and paradigms exists. It is not so easy to choose the best language that will go with the meta-CASE. Of course, some languages like Prolog, Lisp, or Smalltalk are well-suited to support all the aspects of a meta-environment. They easily support meta-programming techniques; they can interpret both code and data structures that are built dynamically. Nevertheless, we have preferred another approach that consists in using both the requirements of the meta-CASE and the profile of the method engineer in order to define a new language that fits well its users and the way they will use it. We have thus to answer two critical questions:

1. What is the profile of a method engineer?
2. How will this programming language be used?

The next sections will try to answer these questions.

What is the Profile of a Method Engineer?

This question is multiple. Indeed, a meta-CASE has several faces depending on the enterprise that will use it. Enterprises can either be CASE tools manufacturers or CASE tools users. We make the hypothesis that CASE tools manufacturers have either engineers with a high degree in computer science or that they can invest in training to obtain qualified programmers. In this context, the method engineer profile is less important.

The situation is different in companies that just use the meta-CASE for their own needs. Indeed, Lending and Chevarny show in a recent study [LC98] that the major part of pro-

grammers who are using CASE tools in these companies have only a bachelor degree (65%) and that only a minor part has a master degree (25%).

In this last situation, it is desirable to limit the number of new concepts in order to make the training stage easier. Hence, the companion language of a meta-CASE must be close to the usual languages in this context, that is, the imperative language family.

How will this Programming Language be Used?

Several uses can be expected:

1. The programming language will have to face all the challenges that were left opened by the limits of expressiveness of the meta-meta-model. Some constraints cannot be represented with the inner mechanism of the repository — complex identifiers, one-to-one meta-relations, complex dependencies are good examples.

2. The language has to support an evolving repository and will have to deal with dynamic types.

3. CASE tools need to export and import their knowledge in various formats (exchange format such as CDIF; report format like: RTF, HTML, LaTeX; or code generation such as: SQL, C++, Java, XML, ...).

4. The meta-CASE must be able to execute tasks such as metrics, transformation, validation, verification, and so on.

5. Communication requirements are too often omitted in CASE tools. Nevertheless, with Internet, the information is now very often distributed on distinct places with heterogeneous technologies. Protocols like CORBA or ODBC should be supported by a meta-CASE in order to communicate with other applications. A programming language is a good candidate to control these protocols.

The Answer

To satisfy the first requirement, we have chosen the imperative paradigm with a Pascal-like language. This paradigm is taught in nearly all the degrees of computer science and should not disturb engineers with an average knowledge level.

The language is complete. In other words, it has the same expressiveness than Pascal or C. Moreover, it has been endowed with new concepts that will allow it to support all the expected challenges. For instance, it has the following features: lists, a garbage collector, a lexical analyser, predicative queries, functions and procedures, libraries, GUI functions, ... It is quite easy to teach, and the new concepts are very well integrated. The language has been designed in order to distinguish the *meta sphere* from the *common sphere*. The common sphere looks like any application dedicated language (for instance, Rose Script in Rational Rose [Rat] and Voyager 2[1] in DB-MAIN [Eng99]). The meta sphere effects the junction between the common sphere and the repository. Hence, only the meta sphere requires a good knowledge of the meta-level. That is, words like meta-class or meta-relation can be ignored in the common sphere. The engineer has only to deal with its domain specific types.

[1] Voyager 2 is the parent of Voyager 2^+ and could be described as an instance of Voyager 2^+.

Token class	Tokens
expression operators	+ - × / `mod` ++ ××
	`or and xor not`
	< > <= >= <> =
instruction operators	:= <− << >> +> <+
separators	. , ; ( ) [ ] { }

Table 7.1: Operators and Separators

Hence, Voyager 2^+ can act as a parametric language. Advanced method engineers can edit the parameters of the language to restrict the programming environment of other method engineers to their own ontologies.

7.2 Lexical Elements

7.2.1 Comments

A comment in a Voyager 2^+ program begins with an occurrence of the two characters `/*` not within a character or string constant and ends with the first occurrence of the two characters `*/`. Comments may contain any characters and may spread over several lines of a program. Comments do not have any effect on the meaning of the program.

A comment may also be any characters found after the two characters `//` in one line.

Example

```
/* Add comments to
** your programs, please ! */
...
x:=x+1; // and comments must be pertinent !
...
```

7.2.2 Operators

The operator tokens are divided in several groups as shown in the table 7.1.

Expression operators are used to build new expressions from other ones, instruction operators are a convenient way to replace classical functions by infix operators.

7.2.3 Identifiers

An *identifier* is a sequence of letters, digits and underscores. An identifier must begin with a letter; identifiers beginning with an underscore are reserved for keywords having a special meaning for the language. There is no restriction on the length of an identifier. Finally an identifier must be distinct from any reserved keyword (section 7.2.4) and any predefined constant name (section 7.2.5).

Example

`factorial`	`PI_31415`	`A_B__C_`	are all valid identifiers.
`_PI`	`314_PI`	`for`	are all incorrect identifiers.
`A_Einstein`	`A_EINSTEIN`	`a_einstein`	are three distinct identifiers

AddFirst	AddLast	AscToChar	BrowsePrint
BrowseRead	CallSystem	CharIsAlphaNum	CharIsAlpha
CharIsDigit	CharToAsc	CharToLower	CharToStr
CharToUpper	Choice	ClearScreen	CloseFile
DeleteFile	DialogBox	Environment	ExistFile
GetAllProperties	GetChar	GetDay	GetError
GetFirst	GetFlag	GetHour	GetID
GetLambda	GetLast	GetLast	GetMin
GetMonth	GetObject	GetOxoPath	GetProperty
GetSec	GetSpecification	GetTokenUntil	GetTokenWhile
GetType	GetType	GetWeekDay	GetYearDay
GetYear	IsActive	IsNoVoid	IsVoid
Length	MakeChoiceLU	MakeChoice	MessageBox
MetaClass	OpenFile	RenameFile	RetrieveID
SetFlag	SetParser	SetPrintList	SetProperty
SkipUntil	SkipWhile	StrBuild	StrCmpLU
StrCmp	StrConcat	StrFindChar	StrFindSubStr
StrGetChar	StrGetSubStr	StrIsInteger	StrItos
StrLength	StrSetChar	StrStoi	StrToLower
StrToUpper	UngetToken	UpdateProperty	Void
and	as	attach	begin
break	call	case	continue
create	delete	do	else
end	eof	export	for
function	get	goto	halt
if	in	kill	label
let	member	method	mod
neof	not	or	otherwise
package	printf	print	procedure
read	readf	relax	rename
repeat	return	self	switch
then	this	to	until
use	void	while	with
xor			

Table 7.2: Reserved Keywords

7.2.4 Reserved Words

Some words (table 7.2) are reserved for the language and cannot be redefined by the user.

7.2.5 Constants

In Voyager 2^+, *constants* are predefined variables with constant expressions. The constants names are listed in table 7.3 and in section D.1.2[1].

[1]See page 243.

FALSE	INT_MAX	INT_MIN	MAX_STRING
OMEGA_META_MODEL	OMEGA_SPECIFICATION	PROP_CORRUPTED	PROP_NOT_FOUND
TRUE	_A	_R	_W
_all	_single	_subtypes	_supertypes

Table 7.3: Miscellaneous Constants

7.3 The Program's Skeleton

A Voyager 2^+ program is composed of distinct sections given below:

◇ ***program-skeleton*** ← ***package-declaration***
  $\langle$***use-directive***$\rangle_{0,\infty}$
  global-variables-definitions
  functions-definitions
  `begin`
    main-body
  `end`

◇ ***functions-definitions*** ← $\langle$***def-method***$\rangle_{0,\infty}$

Section ***package-declaration*** is an optional header. The ***global-variables-definitions*** section contains the definition of all the global variables of the program. Their scope will be the whole program as well as the functions and the procedures. Constants can also be defined in this section. The ***use-directives*** are explained in section 7.10.3 and will import functions from other programs. The ***functions-definitions*** section will contain the definition of all the functions and all the procedures needed by the program. Functions will not be distinguished from procedures in this document. The scope of a function is the whole program. The ***main-body*** section is the main program, i.e., a list of instructions enclosed between the two keywords `begin` and `end`. Only the last section is mandatory in a Voyager 2^+ program. The main body is the place where the execution of the program will start.

The Package Declaration

The ***package-declaration*** section is optional and denotes the meta-class that will own the methods declared in the file. All the methods of a meta-class must be grouped together in a same file. This section is only required when the file is expected to contain the methods of a meta-class. Nevertheless, this still is an executable program as any other Voyager 2^+ program.

◇ ***package-declaration*** ← $\langle$`package` *cst-string* ";"$\rangle$

cst-string denotes the name of the meta-class the program will be attached to.

global-variables-definitions

The ***global-variables-definitions*** section contains the definition of all the global variables and all the constants of the program. This section is composed of ***definition-lines***, each one complies with the following syntax:

$\diamondsuit$ ***definition-line*** $\leftarrow$ ***type*** ":" $\langle$ *identifier* $\langle$"=" ***expr***$\rangle\rangle^{\text{","}}_{1,\infty}$ ";"

$\diamondsuit$ ***type*** $\leftarrow$ ***expr***

Types will be explained in section 7.4. In a line definition, when an expression is associated with an identifier, the identifier denotes a constant whose value is the evaluation of the expression. This evaluation is computed at the beginning of the program. The expression can be made up of identifiers (constants and functions available in its context).

Example

```
...
real: pi=3.1415;
meta_class: type_state=MetaClass("State");
type_state: rnd_state=GetFirst(type_state{self."name"="init"});
```

Functions and Procedures

The function definitions section will contain all the function/procedure definitions. The syntax of a function/procedure definition is fully explained in section 7.8. Each function/procedure can be called from anywhere in the program: from a function, from a procedure, or from the main body even if the call to the function/procedure appears before its definition. One will see later that functions can be called from other programs.

7.4 Types

Like many other imperative languages, Voyager 2^+ is based on typed variables. However, unlike its competitors, Voyager 2^+ needs very few elementary types. Indeed, all the types the language can use are modelled in the underlying meta-meta-model (see chap. 5). Hence, the language offers just one basic type: `meta_class`. All the other types one will present in the next sections are just predefined meta-class constants. In other words, the programmer* could imagine Voyager 2^+ with only one type (`meta_class`) and this line in front of each program:

```
meta_class: integer=MetaClass("integer"),
            real=MetaClass("real"),
            char=MetaClass("char"),
            string=MetaClass("string"),
            file=MetaClass("file"),
            list=MetaClass("list"),
            cursor=MetaClass("cursor") ;
```

Nevertheless, it would be illusory to implement all the integer operations in Voyager 2^+. For this reason, the language has its built-in types that are common to the other programming languages. Therefore, one will encounter integers, characters, strings, and some other well-known types. Nevertheless, it is important to notice that these types[1] are not keywords, but predefined variables. Program 7.1 illustrates the use of types as first-class objects (i.e., expressions, variables, ...). This section will present all the Voyager 2^+'s built-in types.

[1] i.e., the word that represents them in the language.

Program 7.1 [Types and First-Class Objects]
This program illustrates the use of types as values. Types are passed as parameters to a procedure.

```
procedure display_type(meta_class: mc){ ... }
...
begin
  display_type(integer);
  display_type(CompositeState);
end
```

7.4.1 Integers

Integer type covers all the integer values from `INT_MIN` to `INT_MAX`. Integers are signed and the integer constant `INT_MIN` (resp. `INT_MAX`) is the smallest (resp. greatest) value of this type. Integer constants are signed literals composed of digits 0,1, ... ,8,9 but the unary operator + is not allowed. The integer type is named `integer`.

Examples

1, 123, -458, -1021 are valid integer constants
+458, 3.1415, 3E+6 are not valid integer constants

7.4.2 Characters

The character type covers the whole ASCII character set from code 0 to 255. All the characters having a graphical representation have a corresponding constant in this type: the graphical representation itself enclosed between simple quotes. Otherwise characters can be represented by their ASCII value like `'^`*val*`^'`.

Example

```
char:  a='a', Z='Z', plus='+';
char:  bell='^7^', strange='^236^';
```

Some non-graphic characters have a special representation illustrated in table 7.4.

7.4.3 Strings

Strings are sequences of characters. Although the programmer must take care of details like the size of the memory block where the string is stored in Pascal and C, these mechanisms are completely transparent in Voyager 2^+. Hence, the sentence "the size of the string *s*" means the number of characters stored in the string *s*. String constants are sequences of characters between double quotes. The length of a string must be less then `MAX_STRING`.

Example

Character	Representation
backspace	’\b’
form feed	’\f’
newline	’\n’
carriage return	’\r’
tab	’\t’
’	’’’

Table 7.4: Conventions about Special Characters

The statement `print("Albert Einstein")`
will produce `Albert Einstein`
and `print("1\tone\n2\ttwo\n^51^\tthree\n")`
will produce

```
1    one
2    two
3    three
```

Let us note that in string constants some characters have a special representation depicted in table 7.5. The second part of the table uses the same conventions as for the characters. With these conventions, the way the compiler interprets a string is not completely sound without some other rules:

1. Characters are examined from left to right
2. If the `^` character is followed by a sequence of digits denoting a number between 0 and 255 followed by `^`, then the whole sequence is replaced by exactly one character whose ASCII code is the mentioned number found in the string. Otherwise, the first `^` character is interpreted literally, and the interpreter scans the right part of the string *w.r.t.* these rules.
3. If the \ character is followed by a letter (λ) and if $\backslash\lambda$ does not denote a meta-character as depicted in table 7.5, then the sequence is replaced by λ. The \ character is thus removed from the string.

Example

The instruction `print("^1234^88^\^\\\h");`
will print `^1234X^\h`

7.4.4 Lists

Lists are ordered collections of values. These values can be of any type (`list` included). Because lists belong to a basic type (`list`), operations on lists are often easier than in other languages like Pascal and C. Another type (`cursor`) is strongly associated with lists and will be discussed in the next section. All the operations and operators available with this type are explained here below.

A programmer can directly enter a constant list in a program simply by specifying the components of the list between brackets like that:

Character	Representation
backslash \	`\\`
double quote "	`\"`
hat ^	`\^`
backspace	`\b`
form feed	`\f`
newline	`\n`
carriage return	`\r`
tab	`\t`

Table 7.5: Meta-Characters Used in String Constants

```
list: A,B;
begin
  A:=[1..20];
  B:=[1,2,3,5,8,13,21];
  print(A**B); // Operator ** means the intersection of lists
end
```

This program will print all the common values of the two lists: "1 2 3 5 8 13". List A was defined *in expansion* although the second one (B) was defined *in extension*. The syntax of list constants is:

$$\diamondsuit\ \textbf{\textit{list constant}} \leftarrow \text{“[”}\ \langle \textbf{\textit{expr}} \rangle_{0,\infty}^{\text{“,”}}\ \text{“]”}\ \mid \text{“[”}\ \textbf{\textit{expr}}\ \text{“..”}\ \textbf{\textit{expr}}\ \text{“]”}$$

More complicated list constant expressions follow:

Examples

```
[1,[1..fact(1)],2,[1..fact(2)],3,[1..fact(3)],4,[1..fact(4)]]
[[],[1,[]],[2,[1,[]]],[3,[2,[1,[]]]]]
[1,2,3..10,11]
```
→ **Error: .. is not valid here!**

7.4.5 Cursors

Cursors are references to elements of lists. They can have several states described below. Let us suppose that c is a variable which denotes a cursor.

a) The cursor is not attached to any list (c is `void`).

b) The cursor is positioned on a value in a list and this value can be consulted, removed, etc. The cursor is distinct from `void` and the value can be obtained by `get(`c`)`.

c) Let us suppose that c is positioned on value 2 of the list $l \equiv$`[1,2,3]`. If the value 2 is removed from the list l, then c is `void`.

Example

```
cursor: cur;
```

```
begin
  attach cur to [1..10];
  cur>>3;          // to move the cursor 3 times
  print(get(cur)); // print 4
end
```

The first instruction attaches the cursor to the list [1,2,3, ... ,8,9,10]. The cursor is positioned on the first element. The next statement pushes the cursor to the right three times. The last statement retrieves the value denoted by the cursor and prints it.

7.4.6 Files

Objects of type `file` are references to operating system files stored on disks. This object becomes a real reference after the call to the function `OpenFile` whose first argument is the name of the file and second argument is an integer constant. This constant indicates the mode: `_W` if the file is created for writing, `_R` if the file is opened for reading or `_A` is the file is opened for appending. Depending on the mode, the program may read or write information. Writing always occurs at the end of the file and characters are read from the *current position*. Programs must close all the opened files before leaving. More details are found in section D.2.4.

7.4.7 Meta-Classes & Meta-Models

A meta-class (resp. meta-model) variable is a reference to a meta-class (resp. meta-model). This reference can be `void`. In this case, the variable is useless. Otherwise, this variable can be used anywhere a type is expected. Voyager 2^+ recognises the `meta_class` identifier as a type.

Although meta-models and meta-classes are parent concepts, there are no reserved words to define meta-model variables, unlike meta-classes! The reason is quite simple, `meta_class` is the only type that Voyager 2^+ will recognise as built-in type. All the other types are user-defined types that were declared in separated files (see section D.1.1 for more details). The type is declared in the file `TYPES-CST.I`.

Example

```
meta_class: State, Transition;
meta_model: Statechart;
```

7.4.8 Meta-Relation

A meta-relation variable denotes a meta-relation between two meta-classes. If such a variable is not `void`, then it can be used in queries to navigate throughout the repository. The `meta_relation` variable (or type) is declared in the file `TYPES-CST.I`.

Example `meta_relation: incoming, outcoming;`

7.4.9 Meta-Property

A meta-property variable denotes a meta-property. If this variable is not `void`, then it can be used to consult or modify the meta-property value of a class. The `meta_property` variable (or type) is declared in the file `TYPES-CST.I`.

Example `meta_property: StateName, TransitionName;`

7.5 Block Clauses

In conventional programming languages, a strict dichotomy is respected between the values and the type of these values, that is, a type is not a value and a value is not a type. In meta-programming languages the frontier[1] between types and values often vanishes. However one strives to keep Voyager 2^+ as simple as possible and easy to teach and learn. Block declarations seemed to be the easiest way to reach this goal. Blocks allow the definition of new variables anywhere in a program with types that can be left unknown at compilation time.

The syntax of a block clause is

◇ ***block-clause*** ← `let` $\langle$***definition-line***$\rangle_{1,\infty}$ `in` "{" ***body*** "}"

◇ ***body*** ← ***list-instruction***

Each expression denotes a type and each identifier is a new variable declared with the corresponding type and has the body as scope. The body of the clause is a list of statements. Block clauses can be imbricated. Inner variables overrides outer variables when conflicts occur and variables must have distinct names inside the same block.

Example

This function uses a block clause to create a variable with the ad-hoc type to manipulate its instances. The type (i.e., meta-class) is passed as argument.

This program use a block-clause to define a new variable whose type is defined by the parameter[2].

```
meta_class: State=MetaClass("State"),
            Transition=MetaClass("Transition");
...
function integer ProceedStateOrTransition(meta_class: mc)
{ let mc: instance;
  in { for instance in mc{TRUE} do {
         switch (mc){
         case State:
          /* to process a state */
         case Transition:
          /* to process a transition */
         }
```

[1]Other meta-programming languages consider also the programs as data.

[2]`mc` could also have been declared as a local variable.

```
      }
    }
}
```

This program will print the class-id value of all the classes in the repository.

```
meta_class: mc;
begin
  for mc in meta_class{TRUE} do {
    // for each meta-class 'mc' do
    let mc: aclass;
    in { for aclass in mc{TRUE} do {
           // for each class 'aclass' of 'mc' do
           print(GetID(mc));
           print("\n");
         }
       }
  }
end
```

This example illustrates how the types and their values can be used jointly. The `let` clause structures and isolates distinct level of abstractions but the language remains identical whatever the abstraction level.

Let us note that although the function is generic, there is no need to type cast the `instance` variable onto the right type, the variable has automatically the right type!

7.6 Expressions

Expressions are classified into several categories depending on the type returned by the evaluation process. Some expressions are not typed mainly due to access to the repository and to lists, for these particular cases, the type verification is delayed until the execution time. The first subsection discusses the operators used in expressions. Next subsections describe operators and functions provided by the language for each type.

7.6.1 Precedence and associativity of operators

Each expression operator in Voyager 2^+ has a precedence level and a rule of associativity. Where parentheses do not explicitly indicate the grouping of operands with operators, the operands are grouped with the operators having higher precedence. If two operators have the same precedence, there are grouped with respect to their associativity rules (left/right associativity). The table 7.6 defines the precedence and associativity rules of each operator.

Example

Following complex expressions may be reduced as follows with the precedence/associativity rules:

Token	Operator	Class	Associates	Operands[a]
`not`	logical not	prefix	no	i,r
`-`	unary minus	unary	no	i,r
`*`	multiplicative	binary	left	i,r
`/`	division	binary	left	i,r
`mod`	modulo	binary	left	i,r
`**`	list intersection	binary	left	l
`+`	addition	binary	left	i,r,s
`-`	difference	binary	left	i,r
`++`	list concatenation	binary	left	l
`<`	less than	binary	left	i,r,c,s
`>`	greater than	binary	left	i,r,c,s
`<=`	less than or equal	binary	left	i,r,c,s
`>=`	greater than or equal	binary	left	i,r,c,s
`<>`	different	binary	left	any
`=`	equal	binary	left	any
`and`	logic and	binary	left	i,r
`or`	logic or	binary	left	i,r
`xor`	logic xor	binary	left	i,r
`,`	separator	binary	left	any
`.`	dot notation	binary	left	class`.`{s,p}
`->`	method call	binary	left	class`->`{s,method}
`::`	method call	binary	left	class`::`{s,method}
`%%`	typecasting	binary	right	meta-class`%%`class

Lines separate operators depending on their precedence. One line separates two groups of operators and each operator inside one group have the same precedence. If a group is above another one, then the precedence of its operator is higher than the other group. For instance: prec(`*`)>prec(`+`).

[a]Operands must always be of the same type. Nevertheless, if an integer is used with a real number in an arithmetic operation, the integer value is converted to a real number. Following letters denote expected types by previous operators: `l`: list, `i`: integer, `r`: real, `c`: char, `s`: string, `p`: meta-property, `any`: any type.

Table 7.6: Operators: Precedence and Associativity rules

Original expression	Equivalent expression
a+b×c	a+(b×c)
a=not b and c or d	a=(((not b) and c) or d)
a.length > 10 = 1	((a.length)>10)=1

7.6.2 Arithmetic Expressions

Operators +[1], -, ×, /, `mod`, `and`, `or`, <, >, <=, >=, <>, = denote arithmetic operators. Their operands are fully evaluated before their own evaluation but the order is left unspecified. The definition of +, -, ×, /, and `mod` is respectively addition, subtraction, multiplication, division and remainder. Their arguments can be either integers or real numbers (except for `mod` that requires two integers). When the divisor is zero ($x/0$), then the result of the division is 0 and the error register is set to `DIV_BY_ZERO`. The following table gives a formal definition of the other operators.

$$
\begin{array}{rl}
a > b & \text{returns : if } a > b \text{ then 1 else 0} \\
a < b & \text{returns : if } a < b \text{ then 1 else 0} \\
a <= b & \text{returns : if } a \leq b \text{ then 1 else 0} \\
a >= b & \text{returns : if } a \geq b \text{ then 1 else 0} \\
a = b & \text{returns : if } a = b \text{ then 1 else 0} \\
a <> b & \text{returns : if } a \neq b \text{ then 1 else 0} \\
\texttt{not}(a) & \text{returns if } a = 0 \text{ then 1 else 0}
\end{array}
$$

The program is aborted with an ad-hoc message if the types of the operands are not correct.

> **Remark** The `=` and `<>` operators can be used with strings or lists values with the usual semantics. Nevertheless, if the list denotes an infinite structure (a reference to itself for instance), then the evaluation will not terminate.

7.6.3 The "dot" Notation

The dot notation references the property of a class. Its syntax is

◇ ***dot-notation*** ← $\textit{\textbf{expr}}_c$ "." $\textit{\textbf{expr}}_p$

The $\textit{\textbf{expr}}_c$ and $\textit{\textbf{expr}}_p$ expressions denote respectively a class and a meta-property. If $\textit{\textbf{expr}}_c$= `void` then the evaluation of the dot expression causes an error and the program is aborted. The $\textit{\textbf{expr}}_p$ expression can denote either a meta-property or a string expression.

If we note $c.P$ the $\textit{\textbf{expr}}_c$ "." $\textit{\textbf{expr}}_p$ expression, where c denotes the class and P stands for the meta-property, then P must belong to $\text{Prop}(\Gamma(c))$ or there must exist a meta-class D in $\text{super}^\star(\Gamma(c))$ such that $P \in \text{Prop}(D)$. Another case occurs when this field is a string expression (s). Then the field denotes the name of a meta-property. In such a case, there must exists a meta-property P and a meta-class D such that $D \in \text{super}^\star(\Gamma(c)) \cup \Gamma(c)$ and $P \in \text{Prop}(D)$ and $P.\text{name} = s$. If these conditions are satisfied, the dot expression denotes the value $\xi^\star_P(c)$. This notation can be used either as an expression (and its value is $\xi^\star_P(c)$) or as a reference to the property if it is used in the left-hand side part of an assignment.

[1]The + operator is overloaded in order to behave like the `StrConcat` function with strings.

7.6.4 MetaClass

The `MetaClass` keyword denotes a function that permits us to "fuse" a Voyager 2^+ program and to make it aware of new types.

function meta_class : r MetaClass (String : n)

Precondition. String n should denote the name of a meta-class in the repository.

Postcondition. $r \in \mathsf{MetaClass} : \; r.\text{name} = n \;\; \otimes \;\; r = \texttt{void}$.

7.6.5 List Expressions

Overview

Lists in Voyager 2^+ have no similar counterparts in Pascal and C. As explained in the subsection 7.4.4, lists are ordered collections of values. A list has an existence which is not directly linked to the scope of variables representing it. Moreover, values in lists can be of any type, even `list`, `cursor`, ... A list exists in memory until the program can no longer use the values of this list. A garbage collector retrieves the lost lists and the programmer does not need to care for this mechanism.

Concatenation of Lists The infixed operator `++` takes two distinct lists and returns the concatenation of both. Let us note that the arguments are detached of their body after execution. For instance, let us suppose that the cursor C is attached to the list L_1 and that the instruction $R := L_1$++L_2 is performed right now! Then the cursor C is now attached to the list R and no more to L_1 whose the value is the empty list [ ].

function list : r ++ (list : l_1, list : l_2)

Precondition. lists l_1 and l_2 are two list expressions **referencing distinct lists**.

Postcondition. $l_{1\mathsf{new}} = [\,]$ and $l_{2\mathsf{new}} = [\,]$. Let us suppose that list $l_{1\mathsf{old}} = [v_1, \ldots, v_n]$ and $l_{2\mathsf{old}} = [w_1, \ldots, w_m]$, then the result will be a new list: $[v_1, \ldots, v_n, w_1, \ldots, w_m]$.

Example

```
[1,2,3]++[4,5,6] → [1,2,3,4,5,6]
L1:=[['a',1]]; L1:=L1++[['b',2]] → [['a',1],[['b',2]]]
L2:=L1++L1; → error !
L2:=L1; L3:=L1++L2; → error !
```

Remark This operator is defined in such a way that the concatenation does not need to copy the content of the arguments to get the new lists. The time ellapsed by this operator does not depend on the size of the arguments, and moreover, lists with circular references can be used without causing an infinite loop. Besides the obvious advantages of this operator, we have also observed that deep copies of lists were rarely needed.

Intersection of Lists The infixed operator $**$ is used between two lists to compute all the common elements. There is no restrictions on the arguments of this operator.

`function list:`r `**` (`list:`l_1, `list:`l_2)

Precondition. l_1 and l_2 are two lists. The type of the items stored in both lists is not necessarily identical.

Postcondition. r is the list of all the values common to lists l_1 and l_2. If one object is present in both lists but with different types (one super-type and one sub-type for instance), they are considered as distinct. The order of the returned list is left unspecified.

Examples

```
[1,1,2,4] ** [1,5,2]
  →[1,2]
[[1,2],[3,4],'b',7] ** ['a',7,[3,4],["ab",factorial(12)]]
  →[[3,4],7]
l1:=[1,2,1,3,'a','a','b'];
l1**l1
  →[1,2,3,'a','b']
```

7.7 Statements

Each statement must be terminated by one semi-colon except for compound instructions where this character is optional. The empty statement is not allowed in Voyager 2^+, however a compound statement may be empty.

7.7.1 Assignment

Assignment statements must respect the following syntax:

◇ ***assignment-inst*** ← ***lhs*** ":=" ***rhs***

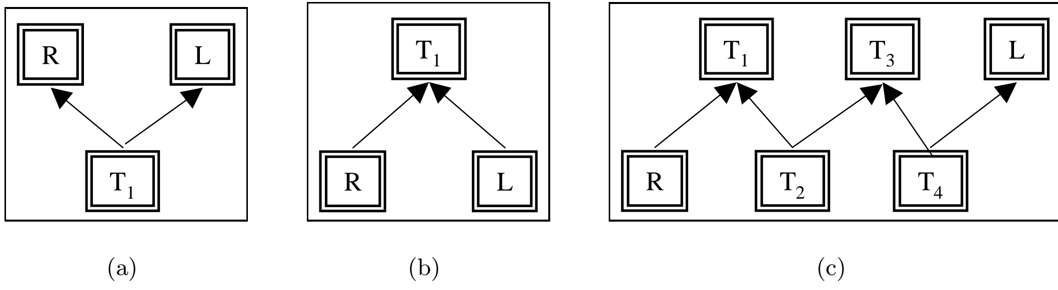
◇ ***rhs*** ← ***expr***

◇ ***lhs*** ← *identifier* | *identifier* "." ***expr***

The types of the *rhs* expression and *lhs* parts must be compatible, i.e., their types must belong to the same inheritance graph. When the types of the *rhs* (R) and *lhs* (L) parts differ, two cases must be examined.

1. R and L do not belong to the same inheritance graph, then there is no way to assign the variable with the *rhs* expression. The execution is stopped and an error message is displayed.

2. R and L belong to the same inheritance graph. This means that there exists a sequence $[T_1, \ldots, T_n] \subseteq \mathsf{MetaClass}$ such that $\rho(R, T_1) \wedge \rho(T_1, T_2) \wedge \cdots \wedge \rho(T_{n-1}, T_n) \wedge \rho(T_n, L)$. The complexity of this case can easily be explained in showing some patterns of inheritance graph:



(a) (b) (c)

Figure 7.1: [**Some Patterns of Inheritance Graph**]

As illustrated in these patterns, the path that link together two meta-classes can be complex and can pass via common supertypes, subtypes or even both of them.

Let us note r the value of the *rhs*-expression. If there exists $n+1$ classes $[t_1, t_2, \ldots, t_{n-1}, t_n, l]$ such that $\Gamma(t_i) = T_i$ and $\Gamma(l) = L$ and $\varrho(r, t_1)$ and $\varrho(t_i, t_{i+1})$ and $\varrho(t_n, l)$ then l is the value that will be stored into the variable denoted by the *lhs*-expression. Otherwise, this variable is `void`. The assignment statement is thus capable of type casting the right expression to the type of the left variable.

If the *lhs*-part denotes a "dot" expression[1], then the `CanUpdate()` method is called. Let us remember that if the dot expression is noted "*c.p*", then c must be $\neq$ `void` and hence, the type of c is known. Then, c`->CanUpdate( ... )` is implicitly called with the *rhs*-value as argument. If this call returns **false**, then the property is let unchanged and the error register is set to `ERR_NO_UPDATE`.

Example

```
State: my_state;
CompositeState: my_composite;
begin
  my_state:= ...
  my_composite:=my_state;
  print(my_state."Name");
  if IsNoVoid(my_composite)
  then { print(" is a composite state"); }
  else { print(" is an atomic state"); }
end
```

7.7.2 Selection Statement

Selection statements direct the flow of control depending on the value of an expression.

The `if-then` Statement

The `if-then` statement executes a list of instructions if the evaluation of the condition is different from 0. The syntax is:

[1]See section 7.6.3.

◇ ***if-then-statement*** ← `if` ***condition*** `then` "{" ***success*** "}"

◇ ***condition*** ← ***expr***

◇ ***success*** ← ***list-instruction***

The evaluation of the expression *condition* must return an integer value d. If the value d is not null then the list of instructions *list-instruction* is executed.

Example

```
if n=0 then { n:=1; }
```

The `if-then-else` Statement

The `if-then-else` statement executes a list of instructions among the candidates ***success*** and ***failure*** depending on the evaluation of the expression ***condition***. The evaluation of this expression must return an integer value.

◇ ***if-then-else-statement*** ← `if` ***condition*** `then` "{" ***success*** "}" `else` "{" ***failure*** "}"

◇ ***condition*** ← ***expr***

◇ ***success*** ← ***list-instruction***

◇ ***failure*** ← ***list-instruction***

The evaluation of the expression ***condition*** must return an integer value (d). The flow of control is directed to the list of instructions ***success*** (resp. ***failure***) if the evaluation of the expression ***condition*** is not null (resp. *zero*).

Example

```
if m<n then {
    v:=m;
} else {
    v:=n;
}
```

The `switch` Statement

The `switch` statement chooses one of several flows of control depending upon a criterion. The criterion must be either a variable or a variable with a field whose the type must be compatible with respect to the "="" operator with the values found in the `case` statements. Its syntax is:

◇ ***switch-statement*** ← `switch` "(" ***variable*** ")" "{" $\langle$***case-sttmt***$\rangle_{0,\infty}$ $\langle$***default***$\rangle$ "}"

◇ ***case-sttmt*** ← `case` ***expr*** ":" ***list-instruction***

◇ ***default*** ← `otherwise` ":" ***list-instruction***

The meaning of such a statement can be described by another equivalent `if-then-else` statement as it is showed below:

```
switch (v) {
case e1:  s1
case e2:  s2
     ⋮
case en:  sn
otherwise:  s0
}
```

$\Longleftrightarrow$

```
if v = e1 then { s1 }
else {
     if v = e2 then { s2 }
     else {
          ⋮
             if v = en then { sn }
             else { s0 }
          ⋮
     }
}
```

If the ***default*** clause is not present, then just consider that the s_0 list is empty. This translation of the `switch` statement is in no way an explanation of the compilation process, so the order of evaluation of the e_i expressions is not guaranteed by the language Voyager 2^+. Therefore, expressions for which the evaluation has a side effect are discouraged since the semantics is unspecified.

When the `switch` statement is executed the value of the variable is compared to each expression e_i until values are equal. Once this condition is satisfied, the respective list of instructions is executed. If all tests fail, then no instruction is executed unless the ***default*** case is present and then its list of instructions is executed.

Example

```
switch (letter) {
case 'B':
    print("Belgium");
    print(" (Belgique)");
case 'F':
    print("France");
case 'S':
    print("Spain");
    print(" (Espagne)");
otherwise:
    print("I don't know!");
}
```

7.7.3 Iteration Statement

Iteration statements are the `while`, `repeat` and `for` instructions.

The `while` Statement

The `while` statement has the following syntax:

♢ ***while-statement*** ← `while` ***condition*** `do` “{” ***body*** “}”

♢ ***condition*** ← ***expr***

♢ ***body*** ← ***list-instruction***

The evaluation of the ***condition*** must be an integer value. While the evaluation of this expression will be not null, the ***body*** will be executed. The iteration stops when the evaluation of the ***condition*** returns the value 0.

Example

```
f:=1;
while n>0 do {
    f:=f*n;
    n:=n-1;
}
```

The `repeat` Statement

The `repeat` statement has the following syntax:

♢ ***repeat-statement*** ← `repeat` “{”***body*** “}” `until` ***condition***

♢ ***body*** ← ***list-instruction***

♢ ***condition*** ← ***expr***

The evaluation of the expression ***condition*** must return an integer value. The ***body*** is executed until the evaluation of the ***condition*** returns a not null value.

Example

```
n:=read(integer);
repeat {
    n:=n-1; } until n=0;
```

The `for` Statement

The `for` statement is probably the most usual instruction for doing iterations among a set of values. The original feature of this statement is certainly the iteration through elements of lists. This characteristic allows it to be used to visit references coming from the evaluation of a request. Its syntax is:

♢ ***for-statement*** ← `for` ***iterator*** `in` ***list*** `do` “{” ***body*** “}”

♢ ***iterator*** ← ***variable***

♢ ***list*** ← ***expr***

♢ ***body*** ← ***list-instruction***

The evaluation of the expression *list* must return a value of type `list`. Moreover, each element of this list must be of the same type as the variable *iterator*. If the list is empty, this statement has no effect except the evaluation of the *list*. Otherwise, the variable *iterator* is instantiated with the first value found in the list, and the *body* is executed. Afterwards, we try to iterate through the suffix of this list until the prefix becomes empty. When the `for` statement is completed, the variable *iterator* is instantiated with the last value found in the list (the value is unspecified if the list is empty).

Let us note that one does not tell neither how nor when the list is evaluated! So we recommend to avoid any instruction that could have a side-effect on the evaluation of the list. The following examples show very dangerous programming styles.

Examples

```
for i in [1..n] do {
  print(i);
  n:=n+1;
}
```

```
b:=1;
for i in [a,b] do {
  b:=2;
}
```

```
l:=[1..10];
for i in l do{
  l:=l++[11];
}
```

Here follow some correct uses of the `for` instruction.

Example

```
for i in [1..5+5] do {
    print(i);
}
for tr in transition{@tigger:[event]
                     with self.name<>""} do {
    print(tr.name);
    for st in state{incoming:[tr]}++state{outcoming:[tr]} do {
        print(st.name);
    }
}
for c in ['a','b']++['e','f'] do {
    print(CharToUpper(c));
}
for my_list in [[1,2],[3,4],[5]] do {
    print(my_list);
}
```

The `goto` Statement

The `goto` statement directs the flow of control to a statement labelled by an identifier. If this instruction is used inside a function, the flow of control cannot go out the body of the function. In the same way, the control flow cannot be directed from the main body to the inside of a function. See section 7.7.3 for a detailed example.

◇ ***goto-statement*** ← `goto` *identifier*

The `label` Statement

The **`label`** statement is used to put a label in front of a statement. The syntax is:

◊ ***label-statement*** ← `label` "identifier"

Example

```
i:=0;
label loop;
if i<10 then {
   i:=i+1;
   goto loop;
}
```

The `continue` Statement

The **`continue`** instruction can only be called inside **`for-do-in`**, **`while`** and **`repeat`** instructions. In the **`while`**/**`repeat`** instructions, **`continue`** will skip the rest of the instruction's body and causes the re-evaluation of the condition. In a **`for-in-do`** statement, the **`continue`** instruction is allowed only when the list expression is not denoting a repository request. If the list expression is not a request, then the **`continue`** statement will just skip the rest of the body and will process the next value in the list. (See also sections 7.7.3 and 7.9.1)

The `break` Statement

The **`break`** instruction can only be used in **`for-in-do`**, **`while`** and **`repeat`** instructions. For the **`while`** and **`repeat`** instructions, the effect of this instruction is equivalent to a **`goto`** instruction to a label put just after the **`while`**/**`repeat`** instruction. This "breaks" the loop. In **`for-in-do`** statements, the instruction can only occur when the list expression of the loop is not denoting a repository request and it breaks the iteration as for the other statements. . (See also section 7.9.1)

Example

The aim of the following program is to check the existence of transitions which come into states **`s1`** or **`s2`** and come from other states (i.e., distinct from **`s1`** and **`s2`**). The loop will iterate through all the transitions that come into one of these states. The body of the loop checks if the transition comes from a state that is distinct from **`s1`** and **`s2`**. If this test succeeds, then the **`break`** instruction is executed.

```
State: s1,s2;
Transition: t;
 ...
for t in Transition{@incoming:[s1,s2]} do {
   if State{outcoming:[t]}**[s1,s2] <> [] then {
     print("Error: foreign transition");
     break;
   }
}
```

Unfortunately, the use of the `break` statement is illegal in this program since the list expression of the loop is a query. Fortunately, it is possible to rewrite/transform the program in order to make possible the use of the `break` statement. The first transformation consists in adding an empty list to the result of the query. The body is no more query but just a list expression (a concatenation).

```
for t in Transition{@incoming:[s1,s2]} ++ [] do {
   if State{outcoming:[t]}**[s1,s2] <> [] then {
     print("Error: foreign transition");
     break;
  }
}
```

Another transformation consists in moving the restriction directly in the request in order to avoid the `continue` statement.

```
if Transition{@incoming:[s1,s2]
          with State{outcoming:[self]}**[s1,s2] <> []} <>[]
then { print("Error: foreign transition"); }
```

The last transformation is probably nicer, but all the transitions will be visited.

The `halt` Statement

The `halt` instruction can be called anywhere in the program where an instruction is expected. This instruction will stop the program. As when your program terminates, this instruction will not close your opened files!

7.8 Functions, Procedures and Methods

Overview

Functions and procedures are abstractions of program slices. We make no distinction between both terms except if it is explicitly mentioned (in this case, the term is underlined like that: function). The scope of a function is the whole program. Each function is identified by its name in the program where it is defined. The definitions can occur anywhere between the last global variable definition and the main program. Functions may have local variables (their scope is restricted to the body of the function) and return a value (the result) of any type (but not the procedures). A local variable overrides a global variable when they have the same name.

◊ ***def-method*** ← ***def-function*** | ***def-procedure***

◊ ***def-function*** ← ⟨`export`⟩ ⟨`method`⟩ `function` ***expr*** *identifier*
"(" $\langle$***param***$\rangle_{0,\infty}^{\text{“,”}}$ ")"
explain-clause $\langle$***definition-line***$\rangle_{0,\infty}$ "{" $\langle$***instr***$\rangle_{0,\infty}$ "}"

◇ ***def-procedure*** ← ⟨`export`⟩ ⟨`method`⟩ `procedure` *identifier*
"(" ⟨***param***⟩$_{0,\infty}^{\text{“,”}}$ ")"
explain-clause ⟨***definition-line***⟩$_{0,\infty}$ "{" ⟨***instr***⟩$_{0,\infty}$ "}"

◇ ***param*** ← {{`relax` ⟨***expr***⟩ } | ***expr*** } ":" *identifier*

Functions

For functions, the flow of the executed instructions must pass through a `return` instruction before reaching the end of the function. If this condition is not respected, then, an error will occur at run time and it will be trapped by the abstract machine that will display an ad-hoc error message. Functions returning no values have no sense in Voyager 2^+.

In the body of a function, `return` instructions can occur anywhere in the body and must be followed by an expression whose the type is compatible with the one specified in the definition. When this instruction is reached, all the local variables disappear from the environment, the memory is cleaned and the expression is returned as the result of the function.

In the body of a procedure, the `return` instruction has the same meaning as before except that no expression can follow it[1]. If the flow of the executed instructions reaches the end of the body or a `return` instruction, then the execution of the procedure is completed and all the local variables are removed from the environment.

When a function is completed, the flow is directed to the instruction which follows the function call.

Parameters

A function can accept parameters. Each one is identified with a name (that must be distinct from the local variables) and must be typed. The type is an expression which must denote a meta-class $\neq$ `void`. There is no special restriction on the components of this expression. It can thus be a global variable, a function or any other valid expression. The context of this expression consists of global variables, the functions[2], as well as the parameters on the left of the "type". This characteristics is quite unusual, but since types are first-class objects in our approach, there is no reason to limit us to elementary types (that are, anyway, nothing but predefined meta-class constants). An example of this characteristics has already been illustrated on page 61. The `relax` keyword will be explained later in this section.

All the arguments are passed by value except for lists[3]. This means, that for each call, the parameter is in fact a copy of the argument. Nevertheless, there is one exception: lists are passed by *reference*. This means that all the operations performed on a list parameter are also performed on the argument.

The arguments of a call are evaluated from left to right. This property is important since,

[1] If an expression follows the `return` statement, then this expression is not evaluated. Its presence is not a syntax error but has no influence on the program.

[2] They can be either built-in or user-defined.

[3] This is not a real exception. Nevertheless, people often believe that calls like that `foo([1,2,3])` will pass a list to the function. The semantics of lists in Voyager 2^+ is defined in such way that lists are autonomous objects and that a program manipulates them only via references! Hence, passing a list always means passing a reference to a list.

Program 7.2 [**The** `relax` **Directive**]

```
procedure foo(relax integer: n){
  if n="botch" then {  // (*)
     print("botched code");
  }
}
begin
  foo("botch");
  foo(1); // The program will stop in (*)
end
```

This program illustrates the use of the `relax` directive. Although the `foo` function expects an integer, the `relax` directive will prevent the compiler to generate the code to check the type of the argument. The program is not "not typed" for all that! The checking will be delayed until the use of the parameter in the body. The type is after the directive is just an indication that can be considered here as a comment.

besides the well-known side effects[1], the order of this evaluation is exploited in order to pass types as first-class citizens that can be used in the right parameters.

The remark about `list` arguments also holds for lists returned by functions. Let us suppose that l is a local `list` variable, and its value before the call to the `return` instruction is `[1,2,3]`. Then this list is still valid after the call although we said that all the local variables were destroyed when a function exits. The reason is very simple: lists are managed by a *garbage collector* and this one sees that the list is used both by a local variable and by the program calling the function. Hence, the garbage collector does not destroy the list.

Type Casting and the `relax` Directive

An implicit type casting is carried out on all the arguments to match the type of the function's parameters. Nevertheless, it is possible to prevent this operation on some arguments in placing the `relax` keyword in front of them. Indeed, the type casting operation is quite complex, and can thus slow down some functions. Such arguments are not checked at all. Their verification is made by needs, i.e., they are passed as they are. The verification will be delayed until they will be used. So it is possible to write "botched" code like in program 7.2 (and this will work!). Such programs must be avoided as much as possible. Nevertheless, this technique will beyond doubt render service to programmers when they will write parsers, pieces of mathematical code and so on. When a type follows the `relax` keyword, it plays the same role as a comment, with this difference that it must respect the syntax.

Methods

When the ***package-declaration*** is present at the beginning of the program, then the programmer can use the `method` prefix in front of the functions. This keyword will add a new implicit parameter named `this` whose the type is the meta-class declared in the ***package-declaration*** part. It denotes the current class on which the method is invoked.

[1] The semantics of propositions like `p(f(),g())` can be non-deterministic if the order of the evaluation is not specified.

Program 7.3 [[**The Factorial Program**]

```
function integer fact(relax integer: n)
{ if n<=1 then { return 1; }
          else { return fact(n-1)*n; }
}

procedure PrintFact(integer: i, integer: j)
  integer: z;
  list: l;
{ for z in [i..j]
  do { l:=l++[fact(z)]; }
  print(l);
}

begin
  PrintFact(2,5);
end
```

This program illustrates the use of functions and procedures to print lists of factorials.

Voyager 2^+ programs, like any PASCAL or C program, have a body, some auxiliary functions, global variables and can be executed alone. Nevertheless, Voyager 2^+ programs can also be considered as libraries or collectors of methods. That is, a text that groups together some functions for a later use.

We have explained in section 4.3.6[1] that methods could be attached to meta-classes. A program will naturally host these method definitions and it will be considered no more as an executable program but rather like a store of methods. Every meta-class will be associated with a Voyager 2^+ program that will contain its method definitions.

To invoke a method attached to some meta-class C, one will use this notation:

$$\boldsymbol{expr}_{mc} :: \boldsymbol{expr\text{-}method} \; (\; \boldsymbol{expr}_1 \; , \ldots , \; \boldsymbol{expr}_n) \qquad (n \geq 0)$$

$\boldsymbol{expr}_{mc}$ is an expression whose the evaluation denotes a meta-class and ***expr-method*** is an expression that will denote either a string or a method[2]. When this expression is a string, the call will look like this $C::s(\ldots)$, where s is a string[3], then there must exist a method $M \in \mathrm{Meth}(C)$ such that $M.\mathrm{name} = s$, and we can replace the string with M. Thus, in both cases, this statement denotes the call $C::M(\ldots)$ that has been explained in section 4.3.6.

Nevertheless, another notation is accepted to call a method. Its syntax is very close to other languages like C++ or Java:

$$\boldsymbol{expr}_c \texttt{ ->} \boldsymbol{expr\text{-}method} \; (\; \boldsymbol{expr}_1 \; , \ldots , \; \boldsymbol{expr}_n \;)$$

This notation is allowed only when the `method` keyword is mentioned in the definition of the function. Expression $\boldsymbol{expr}_c$ denotes a class and, as above, ***expr-method*** must denote either

[1]See page 47.

[2]i.e., an expression whose the type is `meta_method`.

[3]i.e., an expression whose the type is `string`.

a string or a method. If that is a string, then the above-mentioned rule can be applied and this call will always be equivalent to this construct:

$$\Gamma(\boldsymbol{expr}_c)::\texttt{expr-method} \ (\ \boldsymbol{expr}_c \ , \ \boldsymbol{expr}_1 \ , \ \ldots \ , \ \boldsymbol{expr}_n \ )$$

If ***expr-method*** denotes a method, $\boldsymbol{expr}_c$ will be type casted on the meta-class that owns the method. Let us suppose that the expression is something like this:

$$x\texttt{->}M(a_1, \ldots, a_n) \tag{7.1}$$

Then it can be rewritten as

$$\mathrm{Meth}^{-1}(M)::M(y, a_1, \ldots, a_n) \tag{7.2}$$

where either $\Gamma(y) = \mathrm{Meth}^{-1}(M)$ and $\exists y_1, \ldots, y_m \in \mathsf{Class}\,(m \geq 0) \ : \ \varrho(y, z_1) \wedge \varrho(z_m, x) \wedge \forall i,\ 1 \leq i < m : \ \varrho(z_i, z_i + 1)$ or $y = \texttt{void}$. Relation ϱ has been defined in section 4 on page 54. Thus propositions 7.1 and 7.2 are equivalent. The main operand can be referenced in the body of the method with the `this` keyword.

> **Remark** The type casting of the main operand respects the rule that has been defined for the implicit type casting of the arguments in functions calls.

This semantics is quite unusual in OO programming language. Generally, methods are either inherited or specialized with respect to the rules of the language that control the polymorphism. In section 4.3.13[1], we explained that the language made no hypothesis on the semantics of the polymorphism. And although it is not defined, the method engineer can define its own rules. This allows him, for instance, to implement a limited[2] form when methods are called from an meta-model and an extended form in other meta-models. Anyway, the programmer can select "manually" both the method and the type of the main operand (the expression on the left of the `->`symbol).

Let us note that the main operand can be `void`. Although this has no real advantages, this case can be treated by the called method and can thus be easily fixed. Let us remark that since the main operand can become `void` after a type casting, this characteristic allows the programmer to trap this case and to manage it.

Programs 7.4 and 7.5 illustrate the concepts presented in this section.

Other Directives

The `export` and `explain` clauses that appear in the syntax are outside the scope of this section and will respectively be described in sections 7.10.2 and 7.10.5.

Résumé

When the ***package-declaration*** is omitted, a Voyager 2^+ program is an usual program with variables, functions and a body. The `method` keyword is not allowed in such programs.

[1]See page 60.

[2]We could imagine that some calls will search the whole inheritance graph for the best method although other strategies will limit themselves to the meta-classes which belong to the meta-model that the software engineer is editing at the moment of the call.

Program 7.4 [The "Count" Program]

```
let meta_class: State=MetaClass("State");
    meta_method: Count;
    integer: n;
in { Count:=GetFirst(meta_method{@rel_methods:[State]
                                    with self.name="count"});
     n:=State::Count();
     print(["There are ",n," states!\n"]);
   }
```

This chunk of program will count the number of instances in the extent of the `State` meta-class. The first statement initializes the `Count` variable with the reference to the ad-hoc method named "`count`". The next statement denotes a call to the "`count`" method in using the `Count` variable.

Otherwise, the file is intended to be the store of all the methods relative to the meta-class mentioned in the package directive. In such a case, all the functions (and the procedures) are called methods, and when the `method` keyword is explicitely mentioned in the definition, the functions will accept a main operand, named `this`. The role of the global variables in packages will be explained in section 7.10.2[1].

7.9 Repository Access Methods

7.9.1 Predicative Queries

CASE programs need to navigate through the repository. Flows between the repository and the program are so important that a repository access technology with a high impedance mismatch would make the programming task too uphill. The interface between the repository and the program should thus be without any transition. Moreover, meta-models definitions are often quite complicated:

1. High-level concepts like inheritance, meta-relations and decomposition/explosion require a high level of abstraction from the programmer's point of view.

2. Reasonning about types often confuses the programmer's mind.

3. Meta-models are generally complex and they often encompass more semantics than the programmer is aware of.

4. The "navigation" throughout the meta-model's definition can use 5 or more distinct meta-relations and move the programmer from his concerns[2].

5. Unlike user-centered applications, CASE programs have to deal with meta-models which are often abstractions of ideas that do not have material counterparts. This lack of concrete reference makes the programming task tricky.

[1] See page 194.

[2] A validation process could take the programmer out for a visite of quite independent ontologies like database schema, statecharts and class diagrams.

Program 7.5 [The "Simulate" Program]

```
package "State";

meta_class: State=MetaClass("State"),
            Transition=MetaClass("Transition"),
            Event=MetaClass("Event");

meta_relation:
  incoming=GetFirst(meta_relation{self."name"="incoming"}),
  outcoming=GetFirst(meta_relation{self."name"="outcoming"}),
  trigger=GetFirst(meta_relation{self."name"="trigger"});
```

This part (above) defines both the meta-classes and the meta-relations that will be needed in the further part. This will make easier the understanding of the code.

```
procedure simulate_once(Event: e)
  State: i_state;
{ for i_state in State{self."current"=TRUE}++[] do {
     i_state->"throw_transition"(e);
  }
}
```

The `simulate_once` method receives an event `e` and tries to simulate the *life* of the Statecharts. The `State` meta-class has been endowed with a new meta-property named "*current*" for this purpose. The method will visit all the states in the repository that have the property `current` set to **true** and it will activate them with this event. Because `throw_transition` can modify the "current" property and thus alter the query of the loop, its evaluation has been forced with the `++` operator (*cfr.* section 7.9.1 page 185).

```
method procedure throw_transition(Event: e)
  State: i_state;
{  /* Precondition: this."current"=TRUE */
   for i_state
   in State{incoming:   Transition{@outgoing:[this]}
                     ** Transition{@trigger:[e]}
          } do {
     this."current":=FALSE;
     i_state."current":=TRUE;
   }
}
```

The `throw_transition` will move the "*current*" status from the current state to the target state of an hypothetical transition that would leave the `this` state and that would be triggered by event `e`.

Remark The simulation should be limited to one Statechart to be more realistic. Nevertheless, we have prefered to keep the code as simple as possible. For the same reason, we omitted the "fork" and "join" pseudo states in the simulator.

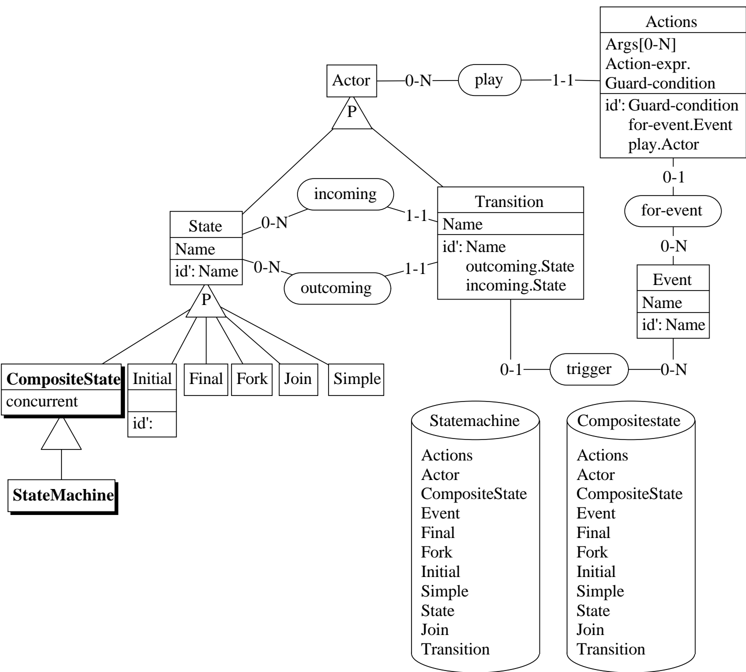


Figure 7.2: The Normal Statechart Meta-Model

These requirements lead to a query language that must be efficient, expressive and well-integrated. For these reasons, we propose a declarative construct that will relieve the programmer of a lot of technical details and that will have the same efficiency than an procedure form[1].

> **Remark** This section will present several examples to illustrate queries. We will use the "normal statechart meta-model" to express them. The latter has been defined and commented in section 4.7.1 and is depicted in Fig. 7.2.

Definition

A *declarative query* is an expression which returns a list of classes that satisfy a constraint. The general idea behind a query is to retrieve all the instances of the C meta-class that satisfy a constraint Π. Hence, we could express this idea with this mathematical expression:

$$\{x \in \Gamma^{-1}(C) \dagger \Pi(x)\} \tag{7.3}$$

The syntax of a query is akin to this notation:

$\diamondsuit$ ***query*** $\leftarrow$ ***mc-expr*** "{" Π-***expr*** "}"

mc-expr is any expression that denotes a meta-class and Π-***expr*** is the constraint. Depending on the nature of the constraint, one will distinguish two kinds of queries: the *exhaustive*

[1] DB-MAIN's Voyager 2 proposes builtin functions to access the repository. Nevertheless, few programmers use them and they can always be replaced with predicative queries.

Program 7.6 Operational Semantics of Exhaustive Queries

```
begin
  let self be a variable of type C
  in { S := []
       for self in Γ^{-1}(C) do
           if Π = true then
              S := S ++[self]
     }
end
```

and the *path oriented* queries. Let us note that to every query corresponds one special variable named `self`. This variable will play an analogous role to the x of equation 7.3. It will design the "object" of the question. Its semantics and scope will be explained in the forthcoming sections.

Exhaustive Queries

Exhaustive queries look in the whole repository for classes satisfying a constraint. The constraint is any integer expression built with the elements of its context and with the `self` variable that is relative to the corresponding query. When the boolean component is absent, it is supposed to be **true**. The syntax of the constraint that figures in an exhaustive query will be:

◇ Π-***expr*** ← ***integer-expr***

The operational semantics of this kind of query is defined in program 7.6. S denotes the result of the query. The semantics matches exactly the idea depicted in equation 7.3: the result is the list of all the classes in the extent of the meta-class that will satisfy the constraint.

Example

```
State{ TRUE }
State{ self."Name"="Init" or  self."Name"="Final"}
State{ StrToUpper(self."Name")="INIT" }
```

The first query returns the list of all the states (whatever their specification/Statechart). The second query will be the list of all the states named "Init" or "Final". Finaly, the last one will be the list of all the states named "init" (case insensitive).

Path Oriented Queries

Exhaustive queries do not provide any help to navigate through meta-relations. *Path oriented* queries comprise a component in their constraint which makes it possible to restrict the search to the elements that participate in some meta-relation. Their constraints have two components: the ***path-constraint*** and the ***boolean-constraint***. Although the first part is mandatory, the second one is not. The syntax of this constraint is:

◇ Π-***expr*** ← ***path-constraint*** ⟨`with` ***boolean-expr*** ⟩

◇ ***path-constraint*** ← ⟨"@"⟩ ***path-expr*** ":" ***list-expr***

The ***path-expr*** component is any expression denoting a valid meta-relation. The ***boolean-expr*** component is any boolean expression built from the elements of its context and the `self` variable of the corresponding query. The scope of this special variable is limited to the ***boolean-expr*** and, hence, the ***path-constraint*** component is outside its scope. The "@" character will indicate the direction of the meta-relation: "*do we use the one-to-many (owner-to-members) direction or the many-to-one (member-to-owner) direction?*".

In order to explain the semantics of this query, we will start from one pattern that will illustrate the case with the "@" character.

$$A\{\texttt{@}R{:}L\ \texttt{with}\ \Pi\} \tag{7.4}$$

This query expresses the question: "*which are the instances of meta-class A which verify the constraint* Π *and which play the member role in the R meta-relation with at least one instance of the list L?*". Nevertheless, a small difference exists between the intuitive semantics and the operational semantics. Indeed, that semantics does not take into account the order between the members of a relation. Although the intuitive semantics is easier to explain, the operational semantics will much more often match the programmer's needs. This semantics is defined by the translation of the query into the pseudo-code depicted in program 7.7. The list S denotes the result of the query. As the reader can observe it, the reading direction of the query is exactly the inverse as the direction suggested in the intuitive semantics. The teaching of the DB-MAIN's Voyager 2 language revealed that most programmers read and validate the query with respect to the intuitive semantics but expect a result as defined by the operational semantics. This experience convinced us to deal with this paradox. This kind of query must fulfill some preconditions to avoid execution errors[1].

1. $A \neq$ `void` and $R \neq$ `void`.

2. The meta-class of the query must be the member of the meta-relation (i.e., R.member $=$ A).

   Remark This information is thus redundant with the meta-relation expression. Nevertheless, this allows the abstract machine to control and to detect programming errors.

3. Every element of the list L can be assigned to a variable of type R.owner with respect to the rules of the assignment statement (*cfr.* page 168).

The last restriction indicates that the query will accept very rich forms. For instance, `void` values can figure in the list and will be omitted during the computation[2].

The dual case occurs when the "@" character is omitted in the ***path-constraint*** component. The pattern looks like this:

$$A\{R{:}L\ \texttt{with}\ \Pi\} \tag{7.5}$$

and the intuitive semantics is "*which are the classes in the extent of the meta-class A that own some member of the list L and fulfill the* Π *condition*". The programmer expects to

[1] These errors will be trapped by the abstract machine that will stop the execution and display an ad-hoc message.

[2] The elements of the list are typecasted on the role of the meta-relation. This typecasting can sometimes return `void`values when it fails (e.g., in a statechart, all the states do not denote initial states.

Program 7.7 Operational Semantics of Path-Oriented Queries. Part I

```
begin
  let owner be the meta-relation "owner" and
      x be a brand-new variable of type R.owner and
      self be a brand-new variable of type R.member and
      S be brand-new variable of type list
  in { S := []
       for x in L do      ♣
           if x ≠ void then
              for self in ℜ_R(x) do
                  if Π = true then
                     S := S + [self]
     }
end
```

In statement ♣, if L denotes a query, then the statement is still expanded with an ad-hoc renaming of the intermediate variables.

Program 7.8 Operational Semantics of Path-Oriented Queries. Part II

```
begin
  S = []
  for x in L do      ♣
      if x ≠ void then
         if ∃self ∈ Γ^{-1}(R.owner) :  x ∈ ℜ_R(self) then
            if Π = true then
               S := S + [self]
end
```

In statement ♣, if L denotes a query, then the statement is still expanded with an ad-hoc renaming of the intermediate variables.

retrieve the classes that play the owner role in relations that would have one element of L as member. Moreover, this owner should satisfy the Π constraint. The operational semantics is defined in program 7.8. Variable S denotes the result of the query.

1. $A \neq$ `void` and $R \neq$ `void`.
2. The meta-class of the query must match the member of the meta-relation (R.member = A).
3. Every element of the list L can be assigned to a variable of type R.member with respect to the rules of the assignment statement (*cfr.* page 168).

Query Expansion

Queries are like any list expression and for this reason, they can figure in any place where a list expression is expected. Queries are then evaluated and their computation returns a list. Nevertheless the loops (`for-in-do` statement) are an exception to this rule. When the list of a loop (the `for-in-do` statement) denotes a query, this one is always expanded such that

the list is not explicitely built. The compiler proceeds to expand the query as its operational semantics suggests it.

Examples

This example shows the expansion strategy that the compiler will apply on several cases.

```
L:=State{TRUE}                                          ⟶ evaluation
for x in State{TRUE} do ...                             ⟶ expansion
for x in Transition{@trigger:Event{TRUE}} do ...        ⟶ expansion
for x in Event{trigger:Transition{TRUE}} do ...         ⟶ expansion
for x in Event{trigger:Transition{TRUE}}++[e] do ...    ⟶ evaluation
for x in Event{trigger:Transition{TRUE}++L} do ...      ⟶ partial
```

In the last statement, the "`Transition{TRUE}++L`" expression is evaluated although the "`Event{trigger: ... }`" query is expanded.

The expansion can sometimes have an unexpected behaviour with respect to the intuitive semantics — such a case has already been encountered in program 7.5. Indeed, if the body of the loop alters the classes the query will traverse, then, the query will be directly influenced. Nevertheless, it suffices to place the query in a *neutral* expression to force the evaluation and, hence, to solve the problem. Simple neutral expressions are `*Query*++[]` or `Eval(*Query*)` where the `Eval` function is defined as

```
function list Eval(list: X){ return X; }
```

This expansion policy had some consequences on the semantics of the `break` and `continue` statements. They cannot be used in loop with query as list expressions. (*cfr.* section 7.7.3).

Queries and List Operators

As we mentioned it, queries can be used anywhere where a list expression is expected and therefore in list operators (see section 7.6.5 page 167). The latter can be used to extend the expressiveness and to overcome some weakness of queries. The `++` operator can be used to concatenate queries and will be used as an union operator. Moreover, this operator will preserve the order. The `**` operator will be used to compute the intersection of two queries. Nevertheless, it does not maintain the order, and the result will be a list of classes without redundancy. This "side effect" is sometimes interesting to easily transform a list into a unique list.

Examples

Let `s` be some instance of the `State` meta-class.

```
L:=Transition{@incoming:[s]}++Transition{@outcoming:[s]};
L:=L**L;
```

The first statement retrieves the transitions that either "outcome" or "income" state `s`. In order to remove the multiple occurences of reflexive transitions (a transition from state `s` to `s`), the second statement computes the intersection of the query with itself. Finally, `L` denotes the expected list: which are the transitions attached to state `s`.

7.9.2 Class Creation

The `create` function will be invoked to create new instances of meta-classes. Due to the genericity of our inheritance semantics, the call of this function will reflect all the richness of the mechanism and will look a bit complex. `create` takes two arguments: *1)* the type of the new instance (a meta-class) and *2)* the "context" of this new instance, that is, its subtypes or supertypes, its properties, its relations and so on.

`function` $T : r$ `create` (`meta_class`: T, `list`:L)

Precondition. The list L denotes a list of classes. Those classes either already exist or need to be created, and they will be all connected together in the inheritance graph unless the precondition fails. The components of this list can be described with the following rules which denote the content of such a list once it has been expanded. Of course, the list can be any expression of type `list` that respects these conventions[1]:

♢ ***arg-create*** ← [$\langle$ ***new-class*** | ***old-class*** $\rangle^{,}_{1,\infty}$]
♢ ***old-class*** ← [`"ancient"` , *class-expr*]
♢ ***new-class*** ← [***meta-class-expr*** , [$\langle$***state-elem***$\rangle^{,}_{0,\infty}$]]
♢ ***state-elem*** ← ***state-prop*** | ***state-owner*** | ***state-member*** | ***state-spec***
♢ ***state-prop*** ← [***meta-property-expr*** , ***expr***]
♢ ***state-owner*** ← [***meta-relation-expr*** , ***class-expr***]
♢ ***state-owner*** ← [***meta-relation-expr*** , [$\langle$***class-expr***$\rangle^{,}_{0,\infty}$]]
♢ ***state-spec*** ← ***expr-specification***

In order to describe precisely the semantics of this operation, we summarize L with these mathematical concepts: OLD is the set of all the "***old-classes***", NEW is the set of all the "***new-classes***". OLD denote a set of classes that already exist and that will be connected with the classes that will be created during the creation process. NEW comprises descriptions of classes that the user wishes to create. Each one describes the future state of the class, i.e., its properties, its roles, and the specifications it will belong to. In order to make the use of these elements easier, one will define new functions that will give us access to their components:

The type of the class to create

$$\text{type} : \text{NEW} \rightarrow \texttt{eMetaClass}$$

The value of a property

$$\text{prop} : \text{NEW} \rightarrow (\texttt{eMetaProperty} \rightarrow \mathsf{ValUniv})$$

[1] Some literal expressions (like `[ ]` or `,`) have not been enclosed between double quotes (" ... "). The BNF rules of a list are well-known and these ones are just intuitive indications of the form of the list argument in the `create` instruction.

The owner of the newly created class w.r.t. a meta-relation

$$\text{rel_owner} : \text{NEW} \rightarrow (\texttt{eMetaRelation} \rightarrow \texttt{eClass})$$

The members of the newly created class w.r.t. a meta-relation

$$\text{rel_member} : \text{NEW} \rightarrow (\texttt{eMetaRelation} \rightarrow \texttt{eClass}^{[\star]})$$

The specifications the newly created class will belong to

$$\text{spec} : \text{NEW} \rightarrow \texttt{eSpecification}^{\star}$$

Moreover, these conditions must hold:

1. All the entries of the list must denote distinct meta-classes and must be no `void`.

$$\begin{array}{l}\forall x \in \text{OLD} :\ x \neq \texttt{void} \\ \forall x \in \text{NEW} :\ y.\text{type} \neq \texttt{void} \\ \forall x \in \text{OLD},\ \forall y \in \text{NEW} :\ \Gamma(x) \neq y.\text{type} \\ \forall x, y \in \text{OLD} :\ x \neq y \Rightarrow \Gamma(x) \neq \Gamma(y) \\ \forall x, y \in \text{NEW} :\ x \neq y \Rightarrow x.\text{type} \neq y.\text{type}\end{array}$$

2. Classes must be described with ad-hoc meta-properties and meta-relations.
   $\forall x \in \text{NEW} :$
   (a) The meta-properties must describe the corresponding meta-class and the associated value must have the right type.
   $\forall p \in \text{dom}(x.\text{prop}) :$
      i. $p \neq \texttt{void} \ \wedge\ p \in \text{Prop}(x.\text{type})$.
      ii. $x.\text{prop}(p)$ is a value compatible with the type of p.
   (b) Every meta-property must be described.

$$\text{Prop}(x.\text{type}) = \text{dom}(x.\text{prop})$$

   (c) When a meta-relation component denotes an owner, the meta-relation must be attached to the corresponding meta-class with the right role. Moreover, the class passed as the candidate owner for this relation must be well-typed.
   $\forall r \in \text{dom}(x.\text{rel_owner}) :$
      i. $r \neq \texttt{void}$.
      ii. $x.\text{rel_owner}(r) \neq \texttt{void}$.
      iii. $x.\text{rel_owner}(r) \in \Gamma^{-1}(r.\text{owner})$.
   (d) All the meta-relations that have a mandatory role and whose the member role is played by $\Gamma^{-1}(x.\text{type})$ must be present in the creator.

$$\exists r \in \mathsf{MetaRelation} :\ r.\text{member} = \Gamma^{-1}(x.\text{type})\ \wedge\ r.\text{mandatory}\ \Rightarrow x.\text{rel_owner}(r)\exists$$

(e) When a meta-relation denotes members, the meta-relation must be attached to the corresponding meta-class with the right role. And all the classes which belong to the list of members must be well-typed.
$\forall r \in \text{dom}(x.\text{rel_member}),\ \forall y \in x.\text{rel_member}(r) :\ (y \downarrow r.\text{member}) \neq \texttt{void}$.

(f) The specifications must be potential holders of the new class.

$$\forall s \in \text{spec}(x) :\ s \neq \texttt{void}\ \wedge\ x.\text{type} \in \text{Mod}(\Gamma(s))$$

(g) There must be at least one specification.

$$\sharp\text{spec}(x) \geq 1$$

(h) If the meta-class is virtual, there must exist at least one subtype in the new inheritance graph.

$$x.\text{type.virtual}\ \Rightarrow\ (\exists y \in \text{OLD} :\ \Gamma(y)\ \textbf{isa}\ x.\text{type}) \vee (\exists y \in \text{NEW} :\ y.\text{type}\ \textbf{isa}\ x.\text{type})$$

3. The classes which are present in the list will be connected together in an inheritance graph. For this reason, their meta-class must all be interconnected with the ISA relation. Let $S_O = \{x \dagger \exists y \in \text{OLD} : x = \Gamma(y)\}$ and $S_N = \{x \dagger \exists y \in \text{NEW} : x = y.\text{type}\}$. Then $\forall x, y \in S_N,\ \exists z_1, \ldots, z_n \in S_O \cup S_N :\ \rho(x, z_1) \wedge \forall i : 1 \ldots n-1 : \rho(z_i, z_{i+1}) \wedge \rho(z_n, y)$ (relation ρ has been defined in section 1 on page 44).

These clauses will ensure us that the list denote a "well-formed" information. Nevertheless, several factors could enfringe the axioms of the repository. All these axioms will be verified by the function.

Postcondition. `create` can be considered as atomic, and if one clause fails, then the whole operation will fail. Moreover, the meta_class::`CanCreate` method will be called on each element of NEW to ensure the programmer that its precondition is fulfilled (see program 7.9).

Example

In the context of a statechart specification (denoted by the `CoffeeMachine` variable), the folowing statements create respectively a simple state, an initial state and a transition that will link together the two previous states.

```
s:=create(Simple, [ [Simple, [ CoffeeMachine ]]
                    [State, [ [ StateName, "CupEmpty" ],
                              CoffeeMachine
                            ]],
                    [Actor, [ CoffeeMachine ] ]
                  ]);
i:=create(Initial, [ [Initial, [ CoffeeMachine ]],
                     [State, [ [ StateName, "Initial" ],
                               CoffeeMachine
                             ]],
                     [Actor, [ CoffeeMachine ]]
                   ]);
t:=create(Transition, [ [Transition, [ [ TransitionName, "0" ],
                                       [ incoming, s],
```

Program 7.9

```
begin
  let list: A;
      boolean: ok;
  in { ok := true;
      for A in L do
          if GetType(GetFirst(A))=meta_class then
             % if this item belongs to NEW, then check the predicate
             ok := GetFirst(A)::CanCreate(GetLast(A)) ∧ ok
             if ¬ ok then break
      % ok=true means: all the preconditions are fulfilled
      if ok=true then
         for A in L do
             if GetType(GetFirst(A))=meta_class then
                x := GetFirst(A)::Constructor(GetLast(A))
    }

end
```

```
                                  [ outcoming, i],
                                  CoffeeMachine
                                ]],
                  [Actor, [ CoffeeMachine ]]
                ]);
```

7.9.3 Class Deletion

The `delete` function deletes a class (and its dependent classes). This function is an interface above the `DeleteClass` function defined on page 67.

The removal of one class from the repository is done by a call to the `delete` Function. This function accepts one argument: the class to remove and return either **true** if the operation succeeds and **false** otherwise. The operation can fail for several reasons: the removal will corrupt some axioms or is forbidden by a CanDelete method.

`function integer:` r `delete` ($T : c$, `integer:` m)

Precondition. Meta-class T must not denote an elementary type and c may be any class. m must be one of those constants:

`_single` : the function will delete only c.

`_all` : the function will delete the whole inheritance graph.

`_supertypes` : the function will delete c and its supertypes (direct and indirect).

`_subtypes` : the function will delete c and its subtypes (direct or indirect).

Postcondition. The class and its dependent classes are removed from the repository (r = **true**) or the repository is left intact (r = **false**). If c is `void`, nothing happens.

Example

The `RemoveAll` deletes all the classes of the list it receives. In order to avoid "reference dangling", we transform the list of classes into a list of "class id". The second loop will use those id's to retrieve the classes (unless they have already been deleted).

```
procedure RemoveAll(list: L,integer: mode)
  cursor: c;
  list: Li;
  integer: id;
{ // we use a cursor instead of a 'for' statement because we do not
  // know the type of the elements of 'L'
  attach c to L;
  while IsNoVoid(c) do {
    AddLast(Li,GetID(get(c));
    c>>;
  }
  for id in Li do {
    delete(RetrieveID(id),mode);
  }
}
begin
  // delete all the transitions
  RemoveAll(Transition{TRUE},_single);
  // delete all the ``init'' events (whatever the statechart)
  RemoveAll(Event{self.name="init"},_single);
end
```

7.9.4 Predefined Methods

_MetaRelation

`function` $\mathbb{B}:r$ _MetaRelation::`CanInsert` (_MetaRelation$:R$, Class $:o$, Class $:m$, Class $:p$)

Precondition.

- $o \in \Gamma^{-1}(R.\text{owner})$.
- $m \in \Gamma^{-1}(R.\text{member})$.
- $p = \texttt{void}; \vee\, p \in \Gamma^{-1}(R.\text{member})$.

> **Remark** The `void` value will be presented in chapter 7. All the elements of the Class set are of course pertinent. Nevertheless, Voyager 2^+ endows the set with a special value named `void` which is distinct from any other value. It denotes the absence of value.

This method controls the deliberate deletion of a class from a relation. Deliberate means here that the method is not invoked when a member class is withdrawn due to

the mandatory characteristics of the meta-relation. In other words, if the deletion is caused by the deletion of the owner class, the method is not called.

Postcondition. This method controls the insertion of a class in a relation. $r =$ **true** iff class m can be inserted after p in the sequence $\Re_R(o)$. This method is called everytime a relation is modified, even while the creation of a new class.

`function` $\mathbb{B}:r$ _MetaRelation::`CanRemove` (_MetaRelation : R, Class : m)

Precondition.

- $\Gamma(m) = R$.member.
- $\exists o \in \Gamma^{-1}(R.\text{owner}) : \ m \in \Re_R(o)$.

Postcondition. This method controls the deliberate deletion of a class from a relation. Deliberate means here that the method is not invoked when a member class is withdrawn due to the mandatory characteristics of the meta-relation. In other words, if the deletion is caused by the deletion of the owner class, the method is not called. r is **true** iff m can be withdrawn from the relation.

_MetaProperty

`function` $\mathbb{B}:r$ _MetaProperty::`CanUpdate` (_MetaProperty : P, Class : c, ValUniv : v)

Precondition.

- $c \neq$ `void`.
- $P \in \text{Prop}(\Gamma^{-1}(c))$.
- v matches the type of P.

Postcondition. This method is called each time the value of a property changes. v denotes the value $\xi_{\mathsf{old}\,P}(c)$. The method returns **false** when the update operation is not allowed. This predicate is not called on the creation of a class.

7.10 Modular Programming

Modular programming allows the programmer to explode a program into smaller components or to define libraries which share some common semantics. Some aspects of this mechanism have already been presented in section 7.8 where packages were associated with meta-classes. Meta-classes played then the role of "warehouseman". Nevertheless, this section will take up this problem from a procedural point of view and give some hints on the implementation of the packages (and their methods) in the meta-CASE. The libraries are intimately related to the processes implemented by the abstract machine (see chapter 9).

The "Presentation" section will present the underlying architecture that supports the processes. The "Voyager 2^+ Process" section will introduce the main concepts in a step by step way. The reader will learn how to use the Voyager 2^+ processes and will thus discover the hidden machine room of the Modular Programming. Once the processes have been described, the libraries will be presented in section 7.10.3. Section "Literate Programming" explains

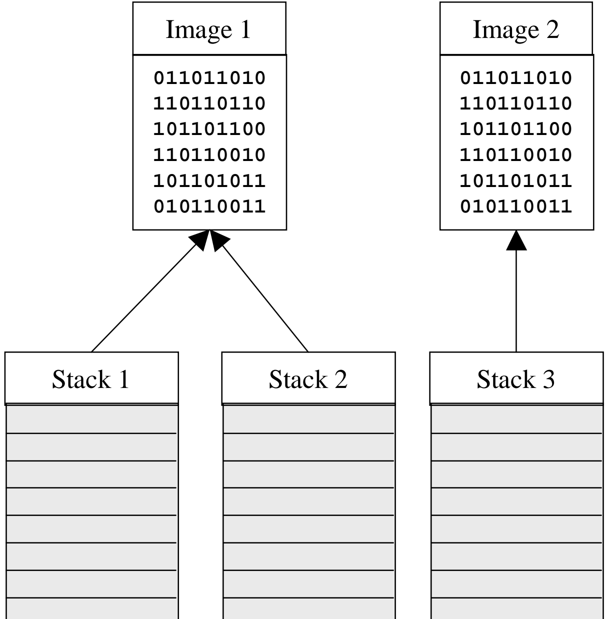


Figure 7.3: [**Image and Stack in Voyager 2$^+$**] This schema depicts the memory state with two programs and three running processes.

how the language allows the programmer to recover the documentation of a library (or a program) from the code. The "Include Directive" section presents the `include` directive to insert files into a program. Finally, section 7.10.7 will give some hints on the implementation of the meta-classes packages.

7.10.1 Presentation

The abstract machine is mainly composed of two memory blocks. The first one contains the program code[1], the second one is the memory[2] used during the execution of the program. They are respectively called the "*image*" and the "*stack*". The architecture supports several images and stacks at once. Let us use the program "`foo.oxo`" as example. Once it is loaded, this program can be stored in one or more images (as many times it has been loaded). Once a program is loaded, it program can be executed. This entails the creation of one stack to give a working space to the new process. But from the same image, another process can be run that involves the creation of another stack, that is distinct from the first one. Thus, one have one image and two stacks. But the architecture makes it possible to load another program, and hence, a new process, ie. another stack. The situation is depicted in figure 7.3.

The image stores all the "instructions" to be executed to run a program, and therefore each function/procedure has its representation in the image. One could execute either the whole program (ie. the body) or simply one function/procedure. Let us remember that as long as a stack is preserved, the "memory" of the process is preserved. So, if one executes one process, this one can leave the global variables in some state (S) at the end of its execution, and retrieve the same state at the next execution.

[1]The program code is a quite straighforward representation of the content of the compiled file (`.oxo`).
[2]This memory is used to store the data.

We have now sufficient information to describe more precisely this mechanism. The language is endowed with two dedicated types: `program` and `lambda`. A value of type `program`[1] denotes either a process or an image, and one will see that once a value of this type is correctly initialized, one can start an execution of the associated program. The second type denotes an entry in the image of a program corresponding to a procedure or a function. Such a value, once initialized, can be used to start the execution of the associated function/procedure in using the stack of a process.

7.10.2 Voyager 2^+ Process

This section will present intuitively the main concepts and operators to manage processes and foreign functions in Voyager 2^+. Their formal definitions will be presented in section 7.10.4. The first step in the use of a process is the declaration of one variable of type `program`.

```
program: p;
```

Once this variable is declared, we can initialize it with the `use` instruction.

```
p := use("c:\\foo.oxo");
```

The `use` instruction has only one argument (a string) that denotes the program to be loaded into a newly created image. Once the `p` variable is initialized, you can call this program as follows:

```
call(p,a1,...,an);
```

`call` is a builtin function that can be invoked either as a function or as a procedure. The $a_1, \ldots, a_n$ values are the arguments passed to the process. Programs can return a value, and its type must not be defined. Hence, a program can be executed either as a procedure (an instruction) or as a function (an expression) depending on the importance of the returned value in the calling program[2]. So, another call to the program `foo` could have been:

```
v := call(p,a1,...,an);
```

We mentioned in the beginning of this section that functions/procedures could be called separately. The first step to call such a function is to get the "*handle*" of the function in the program/process. This is achieved in calling the `GetLambda` function which accepts two arguments. The first argument denotes the process, and the second one is a string that denotes the name of the function/procedure. `GetLambda` returns a value of type `lambda`. This result should obviously be stored in a variable to be used later as it is illustrated in this example:

```
lambda: fct;
...
fct:=GetLambda(p,"my_function");
```

[1] The confusion between both terms is due to automatic management of the abstract machine of both processes and images.

[2] This approach is close to the `C` philosophy, where functions can be used as procedures.

Once this variable has been correctly initialized, it can be used to call the function like this:

```
x := fct(y1,...,ym);
```

This statement calls and executes the `my_function` function inside the "*sleeping*" process[1] `p`. The values $y_1, \ldots, y_m$ denote the arguments of the function. The situation for procedures is similar. So the complete program could be:

```
program: p;
lambda: fct;
string: result;

begin
  p:=use("c:\\foo.oxo");
  fct:=GetLambda(p,"FormatDate");
  result:=fct(13, "Feb", 1968);
  print(result);
end
```

Of course, those constructs can be used in an orthogonal way:

```
begin
  print(GetLambda(use("c:\\foo.oxo"),"FormatDate")(13, "Feb", 1968));
end
```

In the called program, functions/procedures that can be called from outside must be prefixed with the `export` keyword. So the `FormatDate` function would have the following syntax:

```
export function string FormatDate( integer:d, string:m, integer:y )
```

The syntax of a procedure is similar. The `export` keyword allows the programmer to define functions whose use is either private or public.

Definition 33 (Foreign Function — Foreign Procedure)
A foreign function/procedure *is a function or a procedure that has been defined with the* `export` *keyword and that is used outside its program.*

The stack is preserved as long as it could be needed to execute the process or a function/procedure attached to this process. Let us take a look at these two programs

[1] This state is described as `alive` or `virgin` in chapter 9.

Program 7.10 [**Use of Several Processes**]

```
program: p1,p2;
lambda: fct1,fct2;

begin
  p1:=use("c:\\foo.oxo");
  p2:=use("c:\\foo.oxo");
  fct1:=GetLambda(p1,"nestor");
  fct2:=GetLambda(p2,"nestor");
  call(p1);
  call(p2);
  print(fct1(1));
  print(fct2(3));
end
```

The execution of this program will print the values 2 and 4. The execution of the first function will not influence the stack of the second function!

<table>
<tr><th>Calling program</th><th>Called program (foo.v2)</th></tr>
<tr><td><pre>program: p;
lambda: fct;

begin
  p:=use("c:\\foo.oxo");
  fct:=GetLambda(p,"nestor");
  call(p);
  print(fct(1));
  print(fct(3));
end</pre></td><td><pre>integer: n;

export function integer nestor(integer: a){
  n:=n+a;
  return n;
}

begin
  n:=1;
end</pre></td></tr>
</table>

The execution of the first program will call the body of the foo program and put the 1 value into the global variable n. Next, the function nestor is called. This function will use the variable n as the previous execution let it in the stack. In our example, n will be 1 and the function will return 2 ($= 1 + 1$). The second call will return 5 ($= 2 + 3$). Nevertheless, functions do no need to share the same stack. It suffices to initialize two (or more) processes (see program 7.10) in order to have distinct stacks during the execution of foreign functions.

When a program is called from another process, it should be able to access the arguments of the call and to return a result. Each program is endowed with a predefined variable named Environment of type list. It contains all the arguments specified in the call. For instance, this call: call(p,$a_1, \ldots, a_n$) will initialize the list with this value: $[a_1, \ldots, a_n]$. Program sum uses this feature (see program 7.11). The body of a program can use the return statement (as in a function) to return a value to the calling process. There is no restriction on the type of this value.

The programmer has not to care about the process unloading. As for strings and lists, the Voyager 2^+ language ensures that processes are automatically unloaded from the memory (image and stack) once a process is no more needed.

Program 7.11 [The Sum Program]

In the following extract, the `sum` program is called two times. The first call tests the result of the program (which denotes an error code) to display a message. The second call is more confident and does not test the error code. Let us remark that the number of arguments can change from one call to another.

```
program: Sum;
begin
  Sum:=use("c:\\sum.oxo");
  if not call(Sum,1,2,3) then {
    print("error message");
  }
  call(Sum,4,5);
end
```

The `sum` program consults the `Environment` list to access its parameters. If the list is empty, then it detects a possible error and returns 0.

```
integer: sum,i;
begin
  if Length(Environment)>0 then {
    sum:=0;
    for i in Environment do {
      sum:=sum+i;
    }
    print(sum);
    return TRUE;
  } else {
    return FALSE;
  }
end
```

`process` and `lambda` values are first order classes, and can be manipulated as any other value. They can be passed as arguments to functions (even foreign functions), stored in lists, etc. Only IO output are not allowed: there is indeed no reason to print or read a `process`/`lambda` value.

Sometimes, the process loading can fail (not enough memory, `.oxo` file corrupted, security failure, mistyped filename, ...). In such cases, the loading mechanism returns a `program` or a `lambda void` value that can be tested with the usual functions[1]. The programmer must explicitely test these values to insure the program correctness.

7.10.3 Libraries

Libraries are a convenient cosmetic ointment over the `program` and `lambda` concepts. The language offers powerful syntactic shortcuts to define and use user-defined libraries.

The `use` directive allows the programmer to define and use a program and its functions without having to execute specific instructions for this purpose. Indeed, the foreign functions can be declared in the front of the program. Moreover, the mechanism makes possible to rename functions. The syntax of this directive is:

◊ ***use-directive*** ← ***use-library*** | ***use-function***

◊ ***use-library*** ← `use` *cst-string* `as` $identifier_{lib}$ ";"

◊ ***use-function*** ← `use` $identifier_{lib}$ "." $identifier_{fct}$ `as` $identifier_{alias}$

The `use` keyword denotes here two possible uses. The first one (***use-library***) gives a logical name to a process/library. The second one (***use-function***) gives a logical name to a function/procedure inside a library. $identifier_{lib}$ will reference the program/process and $identifier_{alias}$ will be a `lambda` value to the corresponding function in the library. The ***use-directive*** is thus a shortcut to define `program` and `lambda` variables with their initialization in the same time. ***use-library*** directives must always precede the use of its $identifier_{lib}$ name in other directives or expressions.

Example

```
/*************************/
/* libraries declaration */
/*************************/

use "c:\\lib\\tree.oxo" as treelib;
use treelib.DisplayTree as DisplayTree;
use treelib.ErrorProcess as TreeError;
use treelib.ComputeDepth as Depth;

use "c:\\lib\\assoc.oxo" as associative_list;
use associative_list.SetAssoc as SetAssoc;

integer: a,b,c;        /* Global Variables */
...
begin
```

[1] i.e., `IsVoid` or `IsNoVoid`.

```
    ...
    DisplayTree(MyTree,File);
    print(Depth(MyTree));
    ...
  end
```

As we already explained, this syntax is just a cosmetic layer on known concepts. Nevertheless, this allows us to initialize processes without having to explicitely execute their body. This characteristic is important when the programmer whishes to use a library which needs another library. If the program uses the `tree.oxo` library and if it needs the `record.oxo` library, then `record.oxo` must be loaded before any function/procedure of `tree.oxo` is called. This is now ensured by the `use` directive at the beginning of the program. Without this capability, the program should execute the process first before executing any function from this process.

To make easier the programming job, the compiler produces a file with the `.ixi` extension that contains all the needed `use` directives for each function/procedure[1] defined with the `export` keyword. This file can be included with the `include` directive explained in chapter 7.10.6.

Hence, the previous program can be rewritten like this:

```
#include "tree.ixi"
integer: a,b,c;
...
begin
  ...
  DisplayTree(MyTree,File);
  print(ComputeDepth(MyTree));
  ...
end
```

7.10.4 Formal Definitions

The `use` Function

`function program:`p `use ( string:`s `)`

Precondition. s is a string that denotes the name of a Voyager 2^+ program.

Postcondition. p denotes a new process that has been loaded in memory with a newly created stack. The process is just loaded and no execution has been launched!
on error: The void value is returned.

The `call` function

The `call` statement can be used either as a function or as a procedure depending on the context and the way the programmer has defined the body of the called program.

[1]The methods are will also be declared in this file. Nevertheless, the programmer can ignore them, since they are easily accessible from the meta-class they depend on.

`function any call ( program:`P, $a_1, \ldots, a_n$ $(n \geq 0)$ `)`

Precondition. P denotes a `program` expression.

Postcondition. If $P \neq$ `void`, then program P is called in using the $a_1, \ldots, a_n$ values as arguments. Otherwise, nothing happens. If this statement is used inside an expression, then the called program must necessarily return a well-typed value as expected depending on the context. If it is used as a procedure, the returned valued is ignored.

Calls of Foreign Functions

We observed in the presentation (section 7.10.1) that contrary to other languages, the functors[1] in the calls are not necessarily identifiers denoting functions or procedures, but could also be any `lambda` expression. When the functor is the identifier of a function (resp. a `lambda` expression), the call is *static* (resp. *dynamic*). In this last case, the semantics of the call is a bit more complex.

Let `F(`$a_1, \ldots, a_n$`)` be a call (to a function or a procedure, it does not matter). F can be any expression that will be evaluated as a `lambda` value F'. Depending on the nature of F', the function/procedure denoted by F' is called in using the values $a_1, \ldots, a_n$ as arguments. The number of arguments must be strictly the same as the number specified in the definition of the foreign function/procedure. If F' is `void`, then nothing happens and the execution is aborted with an error message. The arguments are type casted with respect to the usual rules.

The compiler shows here some weakness:

1. The compiler cannot check if the number of arguments is correct. If such a case occurs, the stack will be corrupted and the program will stop.

2. The compiler does not have enough information to check that functions are not used as procedures or reciprocally.

Remark

- This thesis describes the Voyager 2^+ language that is under development. A further version will fix these problems.
- The abstract machine can have several images with inconsistent functions: the signature of the call does not match the one of the function. For instance, a library can be loaded, edited, recompiled and reloaded while the main program is still running. This capability allows the method engineer to edit and adapt the methods of a meta-class while the meta-CASE is running.

The `GetLambda` Function

`function program:`L `GetLambda ( program:`P, `string:`N `)`

Precondition. $\emptyset$

Postcondition. This function returns a `lambda` expression that corresponds to a function/procedure named N and defined in P. The stack of P will be used during the

[1] If `F(`$a_1, \ldots, a_n$`)` is a call, then `F` is the functor.

execution of the function/procedure. R is `void` if $P =$ `void` or if the N string does not match a function/procedure P. Otherwise, $R \neq$ `void` and is a valid handle to a foreign function that is ready to use.

7.10.5 Literate Programming

Programmers often document their programs with comments but the retro-documentation of such programs from these comments is very rarely supported by the programming environments[1]. Voyager 2^+ uses comments present in the programs to document the `.ixi` file produced by the compiler. In the Voyager 2^+ syntax, ***explain-clauses*** can occur in the head of the program and inside each function/procedure. They are used by the compiler to produce documented `.ixi` files. Figure 7.4 shows how the compiler works on a sample.

♢ ***explain-clause*** ← `explain` "(*" *anything but *)* "*)"

7.10.6 The Include Directive

The `include` directive makes it possible to include text files in a Voyager 2^+ program. Although this technique is quite raw, it is easy to implement, to teach, and to use. It has been used for a long time in the the C language. The syntax of this directive is

♢ ***include-directive*** ← `include` *cst-string*

This directive may appear anywhere in the program: in the library section, in the global variables declaration, between statements, or even inside an expression. The semantics of this directive is quite simple: the compiler replaces the directive with the content of the file specified in argument. The backslash characters must not be escaped.

Example

```
#include "c:\lib\tree.ixi"
integer: n;
#include "c:\lib\rtf_cst.h2"
function char foo(){ ... }

begin
  #include "c:\misc\copyright.h2"
  ...
end
```

The language does not force the extension names. However, we stringly recommend the use of the `.ixi` extension for libraries[2] and `.h2` for the other files.

If an error occurs in included files, then the compilation fails. However, if the compiler fails in opening an included file, this one is simply skipped and the compiler produces a warning[3].

[1] "*Literate Programming*" first appeared in the `WEB` programming language that is a mix of TEX sentences and `Pascal` statements. `WEB` was defined by D. Knuth.

[2] The compiler already produces files with the `.ixi` extension.

[3] This sometimes avoids attack of nerves.

```
explain (* Factorial Library. Author: Nestor Burma *)

export function integer fact1(integer: a)
explain (* fact1 computes the factorial of its argument
           in using a recursive algorithm
        *)
{ if a=0 then { return 1; }
  else { return a* fact1(a-1); }
}

export function integer fact2(integer: a)
explain (* fact2 computes the factorial of its argument
           in using an iterative algorithm
        *)
  integer: i, result;
{ result:=1;
  for i in [1..a]
  do { result:=result*i; }
  return result;
}

begin
end
```

$\Downarrow$

```
/* FILE GENERATED ON: 23/IX/1999 at 10:44,24 secs
** Compiled with Version 1 Release 0 */

use "facto.OXO" as facto;

/******************
 * Documentation: *
 ******************

Factorial Library. Author: Nestor Burma */

use facto.fact1 as fact1;

/* FUNCTION returns integer
   Arguments:
    1) integer: a
   EXPLAIN:
fact1 computes the factorial of its argument in using
a recursive algorithm */

use facto.fact2 as fact2;

/* FUNCTION returns integer
   Arguments:
    1) integer: a
   EXPLAIN:
fact1 computes the factorial of its argument in using
an iterative algorithm */

/* IXI file completed */
```

Figure 7.4: [**Literate Programming**] The `.ixi` file (bottom) is produced from the Voyager 2^+ program (top). The *explain clauses* document exported functions as well as the library itself.

Remark We also recommend to avoid as much as possible the use of this directive to include functions and procedures definitions[1]. The use of libraries should be preferred in place of such practices.

7.10.7 Processes and Packages

Packages and methods are implemented with respectively `program` and `lambda` attributes: in the logical schema[2], entity-type `eMethod` (resp. `eMetaClass`) is endowed with an attribute of type `program` (resp. `lambda`). Every time a package is loaded, these attributes are updated in order to denote the corresponding process that will host it (and its methods).

7.11 Examples

7.11.1 The Statechart Meta-Model

To illustrate the use of Voyager 2^+ on the Statechart meta-model, we will translate the integrity constraints specified in the UML reference manual (*cfr.* [Rat97b] pages 104–107). We will review every concepts that has been expressed in the "Normal Statechart" meta-model (*cfr.* Fig. 4.28 page 92). Each concept must respect well-formedness rules that will be translated in Voyager 2^+.

A constraint is "delayed" if for some reasons we prefer to postpone its validation until a further analysis step in the building process. Delayed constraints would render the editing stage too constraining if they were systematically checked.

CompositeState

1. "*A composite state can have at most one initial vertex.*" This constraint is implicitely verified by the empty local identifier that will allow only one initial state in every composite state.

2. "*There have to be at least two composite substates in a concurrent composite state*" and "*A concurrent state can only have composite states as substates*"[3] These constraints are delayed.

```
package "CompositeState";
...
method function string Constraint()
{  if Length(CompositeState{indef(self)})<2
   or Length(State{indef(self,this) and IsVoid(CompositeState %% self)})>0
   then { return "CompositeState: Constraint not respected\n"; }
   else { return ""; }
}


export method procedure Analysis()
explain (* Proceed to the analysis of the Statechart
```

[1] The `include` directive enlarges the `.oxo` file size.

[2] See Fig. 4.25 page 83.

[3] From [Gro99].

```
                to check the integrity constraints
             *)
       Fork: a_fork;
       Join: a_join;
     { print(this->Constraint());
       for a_fork in Fork{indef(self,this)} do {
         print(a_fork->Constraint(this));
       }
       for a_join in Join{indef(self,this)} do {
         print(a_join->Constraint(this));
       }
     }
```

Transition

1. "*An initial state can have at most one outgoing transition and no incoming transitions.*"
2. "*A final state cannot have outgoing transitions.*"
3. "*A fork segment should always target a simple state.*"
4. "*A join segment should always originate from a state.*"

These constraints are grouped together in the `CanCreate` predicate of the `Transition` meta-class.

```
package "Transition";
...
export function integer CanCreate(list: args)
  State: origin, target;
  list: aclass;
  meta_relation: a_relation;
{
  for aclass in args do { // for each class
    if GetFirst(aclass)<>Transition then { continue }
    // this entry describes a transition
    for cur in GetLast(aclass) do {
      if IsNoVoid(meta_relation %% GetFirst(cur)) then {
        a_relation:=GetFirst(cur);
        switch (a_relation){
          case incoming: target:=GetLast(cur);
          case outcoming: origin:=GetLast(cur);
        }
      }
    if IsVoid(incoming) or IsVoid(outcoming)
    then { return FALSE; }
    if IsNoVoid(Initial %% target) then {
      print("A transition can not target the initial state\n");
      return FALSE;
    }
```

```
    if IsNoVoid(Final %% origin) then {
      print("The final state can not be origin of a transition\n");
      return FALSE;
    }
    if IsNoVoid(Initial %% origin) then {
      if Length(Transition{@outcoming:[origin]})>0 then {
        print("The initial state can be origin of only one transition\n");
        return FALSE;
      }
    }
    if IsNoVoid(Fork %% origin) then {
      if IsNoVoid(Fork %% target) or IsNoVoid(Synchro %% target) then {
        print("A fork segment should always target a simple state.\n");
        return FALSE;
      }
    }
    if IsNoVoid(Synchro %% target) then {
      if IsNoVoid(Fork %% origin) or IsNoVoid(Synchro %% origin) then {
        print("A synchro segment should always originate from a state.\n");
        return FALSE;
      }
    }
  }
  return TRUE;
}
```

Join

1. "*A join segment must have at least two incoming transitions and exactly one outgoing transition.*"

```
package "Join";
...
method function string Constraint(CompositeState: spec)
{  // pre: indef(this,spec)
   if    Length(Transition{@incoming:[self] with indef(self,spec)})<2
      or Length(Transition{@outcoming:[self] with indef(self,spec)})<>1
   then { return "Fork: bad transitions\n"; }
   else { return ""; }
}
```

2. "*A join segment should not have guards or triggers.*" This constraint is not consistent[1] in UML, since the meta-model as depicted on page 99 of [Rat97b] prevents us to associate triggers with a pseudo-state (i.e, state that denote complex transitions such as fork and join). Hence, this constraint will not be translated in our meta-model.

[1]Several errors were observed in the UML reference guide [Rat97b], mainly in the constraint part written in OCL. These errors have not been fixed in [Gro99].

Fork

1. "*A fork segment must have at least two outgoing transitions and exactly one incoming transition.*" This constraint is delayed.

```
package "Fork";
...
method function string Constraint(CompositeState: spec)
{  // pre: indef(this,spec)
   if    Length(Transition{@incoming:[self] with indef(self,spec)})<>1
      or Length(Transition{@outcoming:[self] with indef(self,spec)})<2
   then { return "Fork: bad transitions\n"; }
   else { return ""; }
}
```

2. "*A fork segment should not have guards or triggers.*"

```
package "Fork";
...
export function integer CanCreate(list: args)
  list: aclass, entry;

{ for aclass in args do {
    if GetFirst(aclass)=Fork then {
      for entry in GetLast(aclass) do {
        if IsNoVoid(meta_property ~ GetFirst(entry)) then {
          if GetFirst(entry)=prop_guards then {
            if GetLast(entry)<>"" then { return FALSE; }
        } }
        if IsNoVoid(meta_relation ~ GetFirst(entry)) then {
          if GetFirst(entry)=trigger then {
            if IsNoVoid(GetLast(entry)) then {
              return FALSE;
  } } } } } }
  return TRUE;
}
```

3. "*Join segments should originate from orthogonal states.*"

4. "*Fork segments should target orthogonal states.*"

7.11.2 An XML Generator

XML* is a subset of SGML* that has been defined by the W3C consortium to enable generic SGML to be served, received, and processed on the Web. More particularly, the OMG* has defined in 1998 a XML Metadata Interchange (XMI*) format to exchange meta-models and specifications between modelling tools (based on the OMG UML) [Béz99]. We can roughly describe XML documents as files made of two components: the DTD* and the entities. The DTD component is optional and defines the syntax of the entities. Entities denote the information the document will represent.

This language is thus well-suited for exporting the information of the meta-CASE to another framework. The program we will present in this section will generate XML files from the repository. This program is interesting for several aspects:

1. The generation of both the DTD and the entities sections requires to visit both the meta-model and the specification levels, and

2. the browsing through the specifications will be made dynamically and it will depend on the current meta-models definitions.

The prefix of "meta-language" begins to make sense in this example.

We reproduce here two excerpts of a report generated from the specification encountered in Fig. 6.17 on page 146. The first extract is the DTD definition that corresponds to the `entity-type` meta-class. The second one is an entity that matches the previous definition. The complete listing is reproduced in section D.4.3[1] The code of the program is split up into two programs: file `xml.v2` (section D.4.1 page 266)denotes the application and uses methods of the `meta-class` meta-class (file `meta-class.v2` section D.4.2 page 269).

```
<!-- meta-class entity-type -->

<!ELEMENT entity-type (
supertypes?,
entity-type.name,
entity-type.owner*,
entity-type.member*,
entity-type.has*
)>
  <!ATTLIST entity-type cid ID #REQUIRED>
<!ELEMENT entity-type.name EMPTY>
  <!ATTLIST entity-type.name val CDATA #REQUIRED>
<!ELEMENT entity-type.owner EMPTY>
  <!ATTLIST entity-type.owner ref IDREF #REQUIRED>
<!ELEMENT entity-type.member EMPTY>
  <!ATTLIST entity-type.member ref IDREF #REQUIRED>
<!ELEMENT entity-type.has EMPTY>
  <!ATTLIST entity-type.has ref IDREF #REQUIRED>


<entity-type cid='x1'>
<supertypes>
</supertypes>
<entity-type.name val="customer"/>
<entity-type.owner ref='x13'/>
<entity-type.has ref='x5' />
<entity-type.has ref='x6' />
<entity-type.has ref='x7' />
<entity-type.has ref='x8' />
</entity-type>
```

Remark The XML generator does not meet completely the XML's requirements. Indeed, the program make the hypothesis that names are XML compliant. Although checking this property is quite easy, this would make the program more difficult to understand.

[1]See page 270.

7.12 Summary

In this chapter we have described several scenarios (i.e., uses of the meta-CASE) which needed a programming language. Every one of them had its own programmer profile and served specific goals. The necessity to endow the meta-CASE with a complete programming language that meets several requirements (sometimes contradictory[1]) has leaded us to propose a brand-new programming language: Voyager 2^+.

The latter takes advantage of the reflective architecture (*cfr.* the mirroring service described in chapter 5) to fulfill the requirements of both the *meta-sphere* and the *common-sphere* with a conventional approach. In other words, types and their values are both first-class citizens (i.e., values).

Its main strong points are: *1)* a semantics that is compliant with the repository's features, *2)* the low impedance of the query expressions, *3)* the composition of the queries with the list operators to obtain a powerful query language, *4)* a garbage collector, *5)* the possibility to query the meta-models and their specifications, *6)* the dynamic loading of packages, and *7)* the capability to manage the meta-models and theirs specifications in the same way.

The marriage of a conventional language with those remarkable features made Voyager 2^+a very expressive language that is easy to learn.

[1]Simplicity and expressivity are often contradictory requirements.

– Create a new statechart diagram!
– *Why do you say create a new statechart diagram?*
– I must describe the behaviour of a class.
– *Is it because of your plans that you say you must describe the behaviour of a class?*
– I am implementing an object-oriented program.
– *Maybe your life have something to do with this.*
– no.
– *Why not?*
Excerpt of a dialogue with a meta-Doctor (Emacs)

Chapter 8

The Dialogues

We have deliberately chosen to reduce the dialogues in the meta-CASE to an over-simplified form. We could caricature the modelling of the interface with two scenarios:

1. the CASE tool must show information to the software engineer. This is mainly modelled with the GraSyLa language.

2. the software engineer must add and edit information.

The last point is the subject of this chapter.

We adopt here a similar approach to the other meta-CASEs, i.e., dialogues are intimately bound with the meta-model definition. As far as we know[1], dialogues have never been the object of in-depth research in the meta-CASE realm. Little documentation exists and the state of the art is probably represented by MetaEdit+ [Con99, Met99]. The method engineer can associate a dialogue box (dialbox) with every meta-class and define it with the help of an authoring tool. Each dialogue box has its own behaviour that is deduced from the meta-model definition.

Our approach is similar but we replaced the authoring tool with a deterministic process which can build dialogue boxes for any meta-class without any human intervention. This approach is sufficient as long as the interpretation of the meta-model definition is intuitive and can easily be understood. Nevertheless, this limitation also holds for the concurrent products.

The main reason why meta-CASE architects restricted the requirements about the modelling of the dialogues in meta-CASEs can easily be explained. So far, meta-CASEs have focused their interest on the representation of the information that underlies the main software engineering tasks. Hence, the main expectations concerned only the graphical representation of the information and its translation into other formats, e.g., code and reports. This observation is still true if we analyze recent CASE tools which support modern method (like

[1]Few companies agree to distribute their documentation.

UML). Nevertheless, some advanced tools which cope with complex problems like reverse engineering, reengineering, optimization, transformations, GUI definition, ... need more complex dialogues based on a semantics whose the goal is not just drawing.

Their definition and their requirements are then deduced from the task the software engineer has to complete (see [BS99, Sze96] for more information on the modelling of task-based user interface). They can no longer be deduced from the meta-model definition. But such tasks are very versatile and complex. It is then very difficult to model them in a simple way. Such dialogues must be defined and managed with the same tools than a hand-coded CASE tools. Such experiences have already been successfully accomplished by using a delegation principle: while the behaviour of the task is managed by Voyager 2^+, the program delegates the advanced dialogues to sub-programs written in JAVA, C++ or Delphi.

8.1 Representative Dialogues

This section uses simple scenarios that entail the use of representative dialogues. The first scenario consists in creating a transition between two states of a statechart. The second one will add a new action to a simple state.

8.1.1 To Create a new Transition

The software engineer chooses the meta-class he will create from a menu (Fig. 8.1). The menu contains all the meta-classes pertinent for the specification that is being edited — virtual meta-classes are not represented in this menu. The meta-CASE consults the meta-model definition to build the dialogue box that corresponds to it. We get the interface shown in Fig. 8.2. Each meta-property produces an entry in the dialogue and buttons will be added for all the roles the meta-class can play. The names are automatically deduced from the aliases[1]. The activation of these buttons will launch new dialogues that will allow the software engineer to edit the "extremity" of each role (i.e., $\xi_R(x)$ or $\xi_R^{-1}(x)$ — x denotes here the current transition). For instance the `Origin State` and `Target State` buttons will launch respectively the dialogues depicted in figures 8.3 and 8.4. The software engineer can either choose the class that will play the "distant" role or even create a new class "from scratch" for this purpose. The dialogue allows him to choose a class from another specification, whatever its meta-model. This feature allows us to link classes which belong to distinct specifications. Once the origin and target states of the transition have been chosen, the software engineer can close (or cancel) the dialbox, and we get the expected result, illustrated in Fig. 8.5.

8.1.2 To Add a new Action

The next scenario will add a new action to the `Filled-Cup` state. A double-click on this state launches the dialbox of the `Simple State`. The `Actions` button allows us to see and edit each action of this state. A simple click on it shows the dialbox (*cfr.* Fig. 8.6). A click on the `Add New First` button will add a new action to the list and produces the ad-hoc dialbox (*cfr.* Fig. 8.7). The `On event...` button allows us to choose the event associated with this action (*cfr.* Fig. 8.8). Closing those successive dialboxes will come us back to the previous steps (*cfr.* figures 8.9 and 8.10). Finally the display is updated and the new action is visible.

[1]Aliases are user-friendly names that the method engineer can give to roles. This feature has not been documented so far, since it does not influence on the repository's semantics.

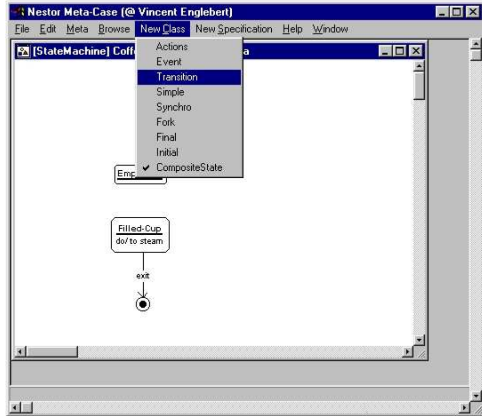


Figure 8.1: [**Create a new Transition**]

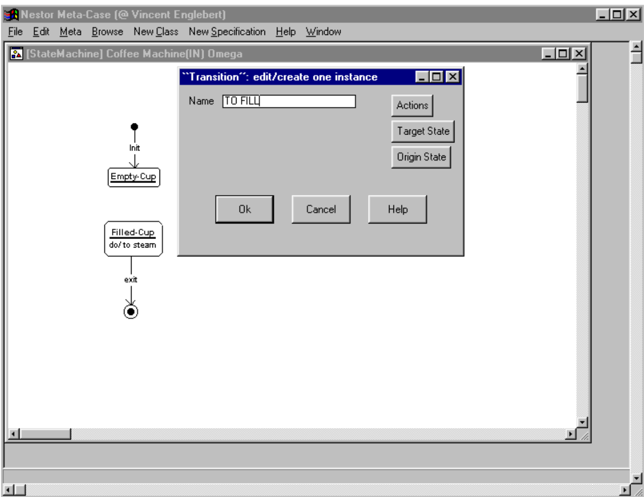


Figure 8.2: [**The Transition's Dialogue Box**]

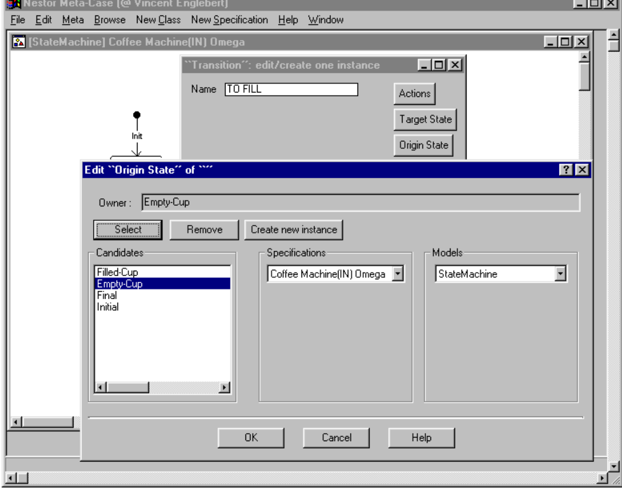


Figure 8.3: [**Edit the Origin of a Transition**]

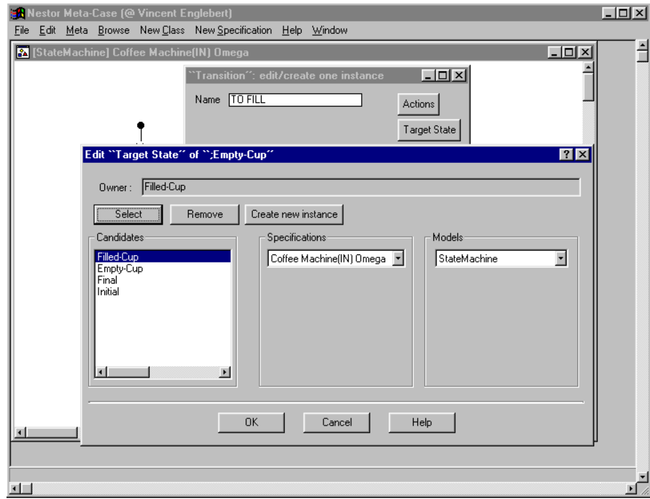


Figure 8.4: [Edit the Target of a Transition]

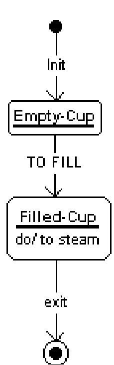


Figure 8.5: [Result]

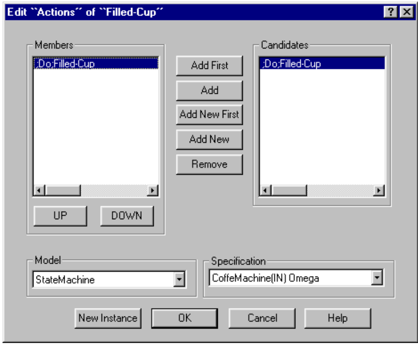


Figure 8.6: [Add a new Action]

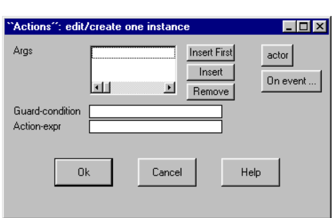


Figure 8.7: [Edit a new Action]

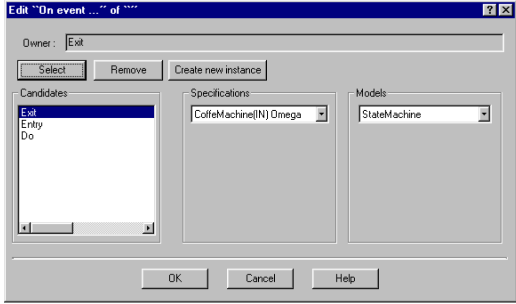


Figure 8.8: [Choose an Event]

`Do`, `entry`, and `Exit` actions are modelled with pseudo-event (like in UML).

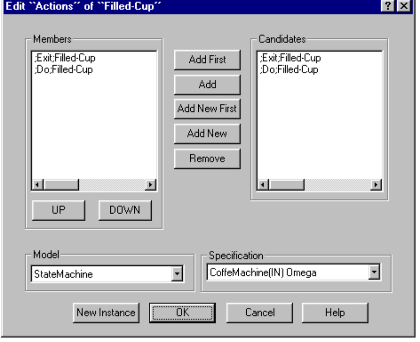


Figure 8.9: [The "Edit Actions" Dialogue Box is Completed]

8.1.3 The Meta-Properties in Dialogues

When the meta-CASE produces the dialogue box of a meta-class, it automatically adds as many fields as they are meta-properties. Their nature depends on the type and the multiplicity of the corresponding meta-property. These forms are illustrated in figures 8.12 and 8.13.

8.2 Some Built-In Dialogue Boxes

Although the reflective architecture could be satisfying (this provides us dialogue boxes to edit the mirrored meta-meta-model), the meta-CASE has been given a "library" of built-in dialogue boxes to edit and create the meta-models. *Cfr.* figures 8.14, 8.15, 8.16, 8.17, 8.18, 8.19.

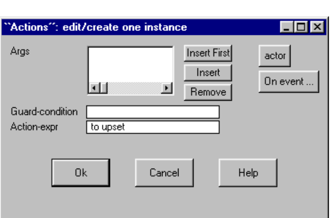


Figure 8.10: [**The "Actions" Dialogue Box is Completed**]

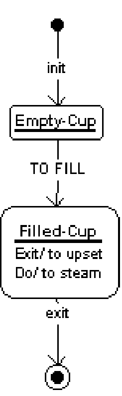


Figure 8.11: [**The Result**]

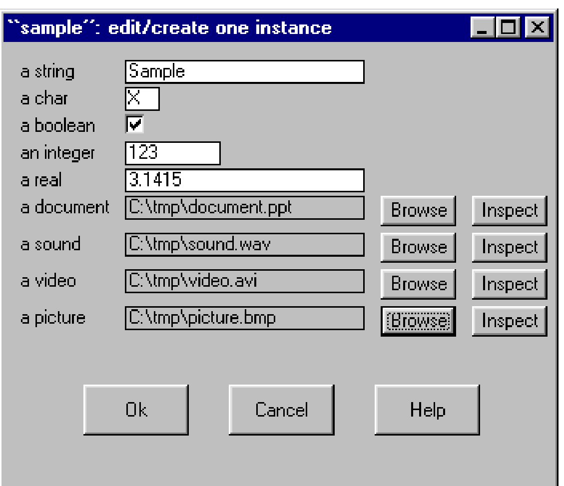


Figure 8.12: A Dialogue Box with all the Kinds of Meta-Properties (Mono-Valued)

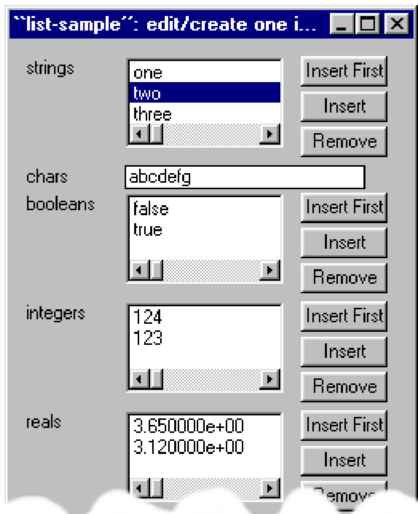


Figure 8.13: A Dialogue Box with all the Kinds of Meta-Properties (Multi-Valued)

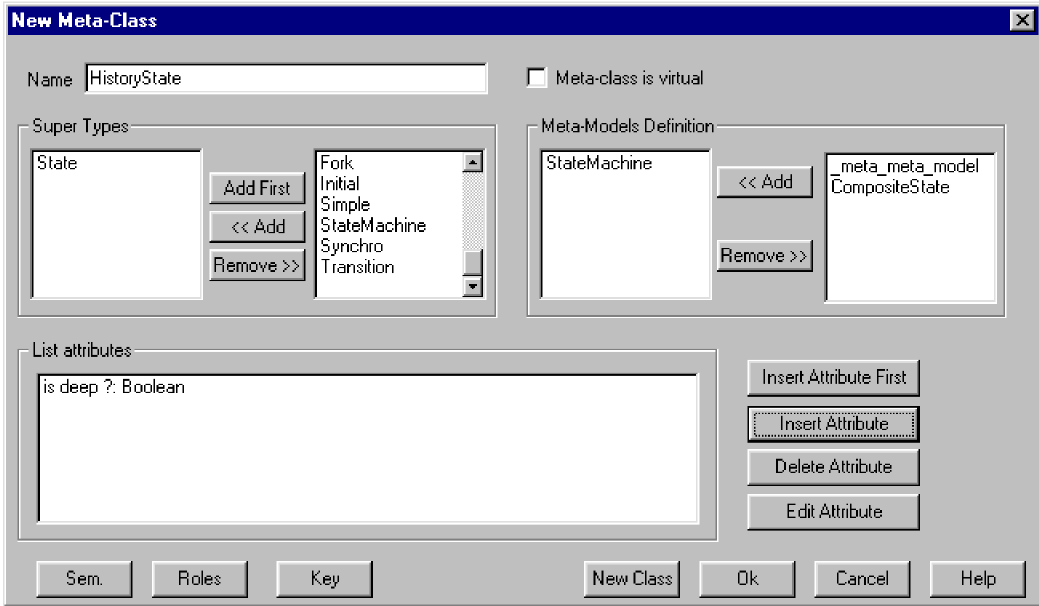


Figure 8.14: [**Create a new Meta-Class**] This dialbox allows the method engineer to define a new meta-class. He can define its meta-properties (Fig. 8.15), new relationships between this newly create meta-class and older ones (Fig. 8.16 and 8.17), define the supertypes, the meta-models it will belong to, and its identifier.

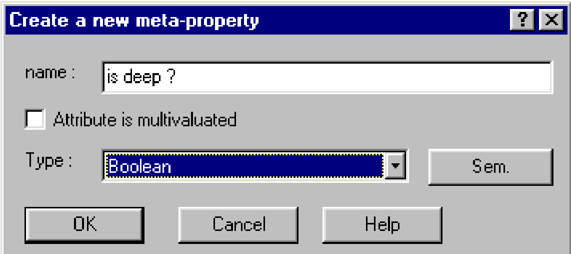


Figure 8.15: [**Create a new Meta-Property**] The method engineer can define the features of a meta-property: its name, its type and its multiplicity

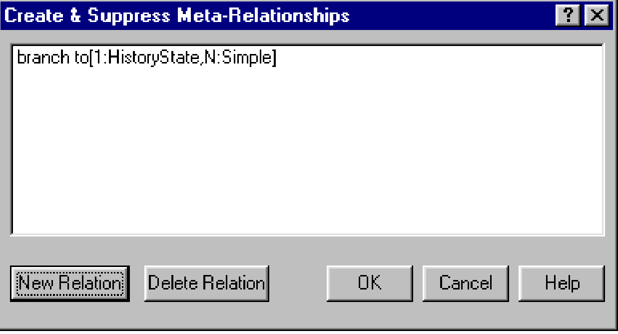


Figure 8.16: [**Edit the Meta-Relations**] The method engineer can visualize the meta-relations that the new meta-class owns. He can either suppress a meta-relation or create a new one.

Figure 8.17: [**Create a new Meta-Relation**] This dialbox defines a new meta-relation. This panel represents all its features: its name, its roles (member/owner), and the aliases.

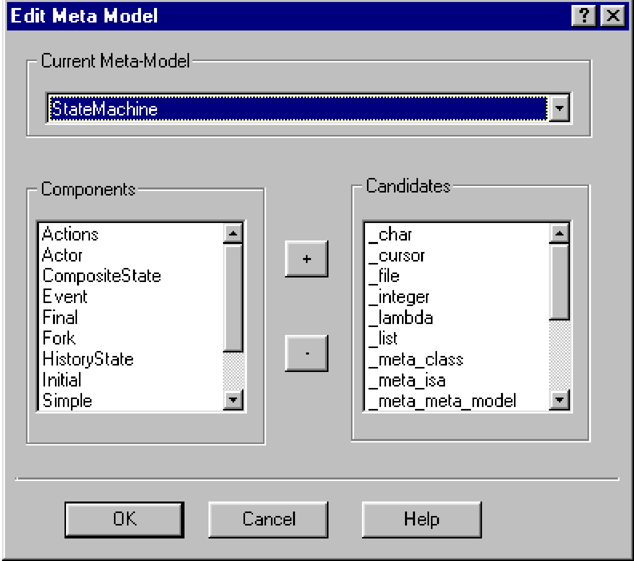


Figure 8.18: [**Edit the Definition of a Meta-Model**] This dialbox allows the method engineer to add new components (i.e., meta-classes) to the definition of a meta-model. Nevertheless, he is not allowed to remove meta-class that belonged to it before.

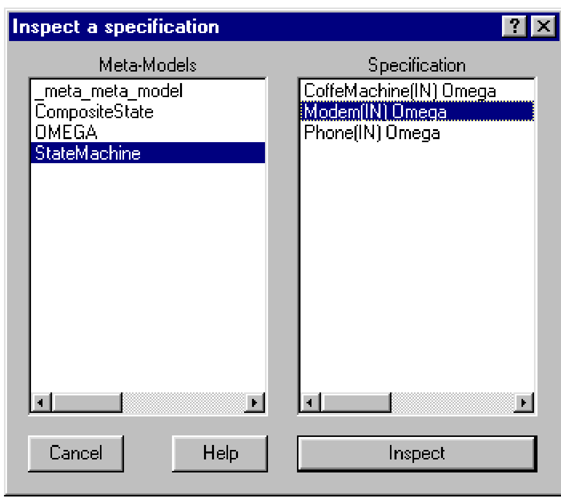


Figure 8.19: [**Inspect a Specification**] The software engineer can choose a specification in the extension of meta-model to visualize/edit it. If several GraSyLa scripts are attached to this meta-model, then a list box is displayed in order to choose the adequate graphical representation.

"It's like a light bulb. When it's broken, unplug it, throw it away and plug in another."
TED HOFF (1937–)

Chapter 9

The Voyager Abstract Machine

Voyager 2^+ programs have been presented in chapter 7. In order to execute these programs, the Meta-CASE hosts an abstract machine* specially defined to execute Voyager 2^+ programs. The idea of virtual machines* were introduced by IBM in 1959 in his VM operating system and since this date, several other machines were defined and used for distinct architectures or languages. In the 1970s, an abstract machine was defined in order to execute Smalltalk programs independently of the platform. In 1983, David Warren defined his Warren's Abstract Machine (WAM) for Prolog. The WAM is still used by academic and commercial Prolog compilers. More recently, the Java Virtual[1] Machine was defined by SUN to support the JAVA language [AK91, MD97]. Of course, these examples are not exhaustive and many other abstract machines exist.

Abstract machines offer several advantages that neither compilation to native CPU codes nor textual interpretation[2] can offer. One will cite here only the few elements that have motivated our choice:

1. Abstract machines ensure that compiled programs can be ported from one system to another one.

2. Abstract machines are generally faster than the textual interpretation mechanisms. Text is parsed only once by the compiler and is no longer used afterwards. Moreover, the cross-references (addresses resolution) are resolved during the compilation.

3. An abstract machine is an additional layer that allows a better control on both the execution of programs and the correctness of the architecture (i.e., the interpreter and the compiler).

4. Pseudo-code can ensure that compiled programs are distributed unmodified. Hence, method engineers can keep their source code confidential and preserve their code from undesirable[3] modifications.

[1]Although the word "abstract" would be more appropriate, SUN probably preferred "virtual" to avoid puns.

[2]We call "textual interpretation" the technology which uses either directly the original file or an intermediate syntactic tree as support.

[3]This requirement has been observed in the DB-MAIN research group. Indeed, once CASE tools are sold, developers have the right to protect their investments (i.e., the programs).

Unfortunately, abstract machines slow programs down. This drawback is not too severe in the environment we are developing since the programs we consider should not be too CPU greedy. In the same way that repositories address complex data although DBMSs address huge data sets, meta-programs will have to deal with very complex data sets but not necessarily large. Hence, an abstract machine is a good trade-off between textual interpretation and native compilation.

The abstract machine emulates a virtual computer whose instruction set is close to an assembler language (called Voyager 1^+). This language has both a concrete syntax and a binary representation[1]. The first representation is a user-friendly text. The second form is a binary representation of the textual representation and is defined in order to be loaded quickly (cross-references are already resolved in this file). The language has 169 statements.

As any computer, the Voyager 1^+ abstract machine has a memory to store data. One will define this memory as a sequence of tuples (type/value). This sequence will be managed like a stack with direct access to its content[2]. Voyager 1^+ will obsiously support any kind of information supported by the repository, but more types are needed in order to respond to the common programming practices like files, lists, etc.

Definition 34 (Abstract Machine's Memory)
The memory of an abstract machine is a sequence of tuples (type,value) and is noted `DATA` .

$$\texttt{DATA} \in \left((type : \texttt{eMetaClass}) \times (value : V) \right)^{[\star]}$$

where the set V is the universe of constants. This set will be defined more precisely with the restriction of the following axiom.

Axiom 47 (Well-Typed Memory)

$$\forall x \in \texttt{DATA} : \; x.\text{value} \neq \texttt{void} \Rightarrow \Gamma(x.\text{value}) = x.\text{type}$$

$$\forall x \in \texttt{DATA} : \; x.type \in \{\mathsf{String}, \mathsf{Char}, \mathsf{Document}, \mathsf{Sound}, \mathsf{Video}, \mathsf{List}, \mathbb{Z}, \mathbb{R}\} \Rightarrow (x.value \neq \texttt{void})$$

Another "memory" is also defined and will be used for relative addressing in the `DATA` stack. This memory will be a sequence of integer values. Each integer will denote an address that will be the basis of an environment. Such environments will be used to address local variables, block variables, intermediate variables and so on. Such environments are often implemented in abstract machines as linked list inside the memory stack itself. Unfortunately this forces the instructions to loop through lists and this slows referencing down. An explicit stack allows environment referencing in a constant time.

Definition 35 (Environment Stack)
One defines the `ENV` *stack as:* $\texttt{ENV} \in \mathbb{N}^{[\star]}$.

---

[1] The files are endowed with a CRC code to ensure that they have not been neither modified nor edited.

[2] The P-machine that has been defined in Zürich introduced this abuse of notation. Its store is called a stack although it is possible to reference its elements whatever their place in the "stack". [WM94]

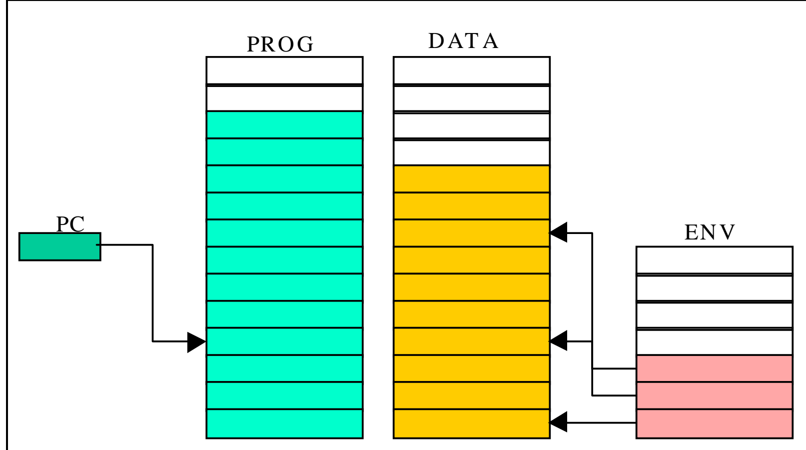


Figure 9.1: The Distinct Memory Areas

Axiom 48 (Well-Formed ENV)
All the indices of the ENV *sequence are valid* DATA *addresses.*

$$\forall x \in \texttt{ENV} : 0 \leq x \leq \sharp\texttt{DATA}$$

The code is stored in a separate memory called PROG that is defined as a sequence of tuples (pcode,x) where pcode is an integer denoting the code of a primitive instruction of the abstract machine. The second part of the tuple (x) can be any kind of value; its type and its interpretation will depend on the pcode value. For instance the PUSH-INT instruction will require an integer and the PUSH instruction will require one integer and one address.

The last component of the Voyager 1^+ abstract machine will be its P-Counter, that denotes the current instruction to execute.

Definition 36 (PROG Memory / P-Counter)
The program to execute is stored as a sequence of tuples (*pcode*, *args*) *and is formally defined as*

$$\texttt{PROG} \in (\mathbb{N}\times ?)^{[\star]}$$

The ? indicates that the second component depends on the pcode value. The current instruction to execute in this sequence is denoted by the P-Counter noted PC*. Its formal definition is* $\texttt{PC} \in \mathbb{N}$*. When* PC=0*, then the abstract machine is stopped.*

The distinct memory areas are depicted in the figure 9.1. One will call "*program*" a sequence of pcodes and "*process*" the execution of a program. Hence, a program will be characterized by its PROG memory and a process by a tuple (DATA, ENV, PC). A process can have four distinct states:

***virgin*:** The process is not running. The P-Counter is null. Its memory is virgin (DATA = [ ] and ENV = [0]).

***alive*:** The process is not running. The P-Counter is null. The memory (DATA) has been initialized: the global variables have been reserved.

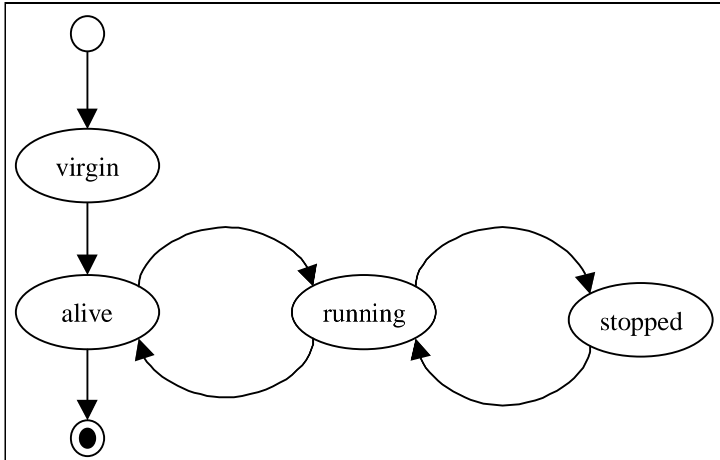


Figure 9.2: [**The possible transitions between the process states**]

***running*:** The process is being executed. The P-Counter is not null.

***stopped*:** The process was stopped during its execution and is waiting for a signal to wake up. The P-Counter is not null.

The figure 9.2 shows the possible transitions between these states. When a process moves from the *running* state to the *alive* state, it can left some information in the `DATA` memory for a further use i.e., the global variables of this process can keep their original values.

The architecture of the Meta-CASE is defined in such a way that several programs can be executed in the same time. Several programs can coexist together in the Meta-CASE with distinct `PROG` chunks and each one can have distinct processes. These processes will have their own P-Counter and memory (`DATA` and `ENV`). Moreover, a process can call another process (in *virgin* or *alive* state). This has alrady been commented in section 7.10 and Fig. 9.3 describes the machines in greater detail.

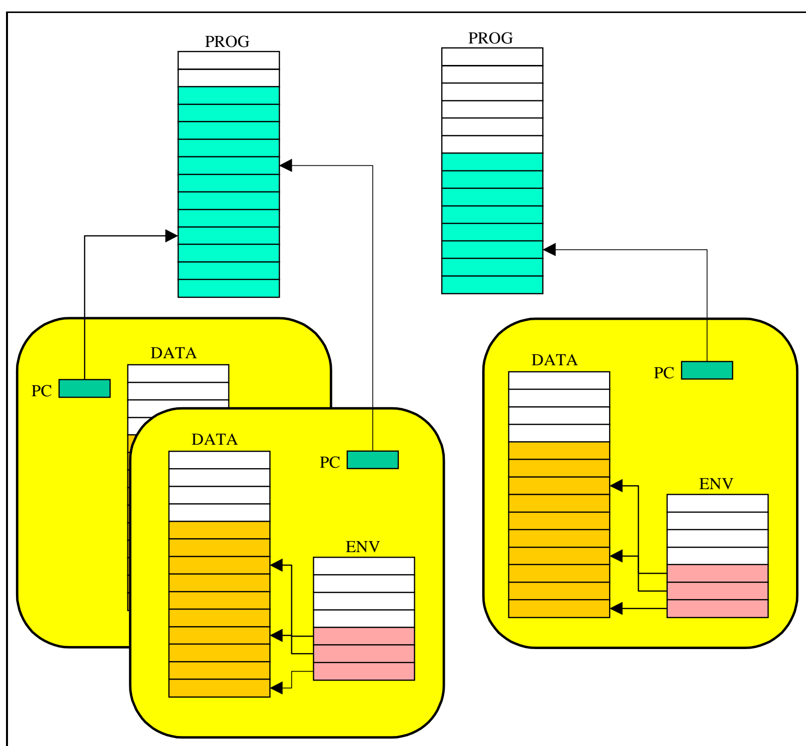


Figure 9.3: Programs & Processes

Chapter 10

Conclusions

An analysis of the state of the art in the meta-CASE realm guided us in the definition of new precepts which should improve the impact of those tools on the software engineering practices. From an analogy with a hypothetical "Turing test" between a CASE tool and a meta-CASE, we have proposed five aspects on which they will have to compete: information modelling, representation of this information, intelligence, user interface, and communication channels. We have proposed four brand-new models for the first dimensions that will be summarized below. So far, the last dimension (i.e., the communication channel) has been shrugged off in most of the CASE tools, and hence, in most meta-CASEs.

The Results

We summarize in the following sections the main results of this thesis as they were implemented in our prototype.

A) The Information Modelling

Our repository uses a minimal set of concepts (*meta-class*, *meta-model*, *meta-property*, *meta-relation*, *method*, and *inheritance graph*) to reach a great expressivity that can be compared with other current works. Its main strengths can be summarized as follows:

1. Its conciseness and its similarity with other common OO models make it easy to teach and to learn.

2. The simplicity of the meta-meta-repository enables us to make it evolve very easily. Meta-models can be incrementally modified (even if their extension is not empty).

3. The generalization of meta-models as meta-classes turns them into first-class citizens. This facilitates the modelling at the various "floors" of the "software engineering house". In other words, meta-classes can be grouped together to form meta-models, and meta-models can themselves be grouped together (even with meta-classes) to form new meta-models and so on. For instance, this architecture would make the definition of process models very simple.

4. The dynamic inheritance makes possible a posteriori specializations (along with the sharing of classes between specifications with the appropriate semantics).

5. The repository helps engineers integrate meta-models. The sharing of both meta-classes and classes between respectively meta-models and specifications makes it possible to define not only bridges (meta-relations) between meta-models, but also to reuse specialized meta-classes (and classes).

6. The mirroring service endows the repository with a reflective architecture that benefits the other components (GraSyLa , Voyager 2^+, ...).

7. The formal definition of the repository with axioms defines precisely the semantics of both the meta-meta-model and its meta-models. This is an indisputable basis for further improvements, implementations or teaching lessons.

B) Representation of the Information (GraSyLa)

The modelling of this component is maybe the most original contribution of our work. GraSyLa is a new language that models the requirements about the representation of the specifications whatever their nature. Meta-models can be displayed according to distinct requirements in the same time and all their representations are synchronized. The representation can be either graphical or textual. Besides this, the architecture is generic enough to make possible its extension to other representations like browsers, matrices, tables, ... We can summarize its features as follows:

Symbolic: Method engineers can concentrate on the requirements and the result. The computations are postponed until the display of the specifications.

Polymorph: The display processor chooses the best representation depending on the nature of the displayed specification and the type of the graphical object.

Aggregate: The display processor can access to the surroundings of a class to make up its representation.

Independent: The requirements of the representation of the specification do not influence the way the method engineer will define the meta-model.

Conditional: The representation is no more directed by the structural definition of the meta-model but can also change depending on the nature of the specifications themselves.

Multiple: Meta-models can now have several synchronized representations (e.g., textual and graphical).

Evolving: GraSyLa scripts can be edited on the fly without disturbing the meta-CASE.

Specialization: GraSyLa is defined in compliance with the semantics of the inheritance between meta-models.

Multimedia: GraSyLa supports multimedia data like sounds, videos, bitmaps, URLs, ...

Besides its expressivity, GraSyLa is very concise and simple. Moreover, it could be easily adapted in other frameworks (CASE tools, repositories, applets, ...).

C) The Intelligence (Voyager 2^+)

This component is crucial in the definition of a meta-CASE. Our experience with DB-MAIN tends to prove that the adoption of the generated CASE tool in the long term will certainly depend on this. Nevertheless, we had to answer two contradictory requirements: *a.* the language must be simple enough to be used by engineers who have a low degree and *b.* it must have a richer semantics than conventional languages to take into account the complexity of its environment (i.e., the meta-CASE). Moreover, it has often to fill the gap left by the other components.

Voyager 2^+ has fulfilled those requirements in proposing both a Pascal-like language with well-known paradigms and extensions that endow it with meta-facilities. Its similarity with Pascal or C makes methods engineers comfortable and this allows teachers to speed up the learning curve of this time-consuming learning unit. Nevertheless, the language supports extensions (listed below) which make it well suited for meta-developments.

Garbage collector: Voyager 2^+considers lists as first-class values and relieves the method engineer of managing the memory (de)allocation.

Predicative queries : They make the access methods very concise (this often cuts dozens of statements down to one alone expression) and the utilization of lists makes the queries very well integrated in the language's semantics.

Meta-queries : Queries are built on the basis of expressions rather than literals. This makes it possible to define meta-queries, i.e., under specified queries with possible a posteriori specializations.

Meta-homogeneity: The language confuses types and values, meta-classes and classes. Hence, types and values are without discrimination first-class objects. This makes it possible to express queries both on the meta-model and the specification levels.

Dynamic packages: The abstract machine can load several packages (or programs) in the same time, and furthermore packages can be unloaded and loaded safely while the meta-CASE is running. This makes the debugging stage significantly more comfortable.

Libraries: The language proposes numerous libraries such as: lexical analyzers, GUI dialogues, textual properties, system calls, ...

D) The User Interface

The user interface is certainly a crucial aspect since, along with the graphical representation, this is the more visible dimension of the "CASE-tool" object. However, UI designs is a creative activity and cannot be fully automated. Moreover, the recent UI models do not enable a developer to work from an abstract specification of the system to produce a prototype design [VP99, SR98, SLN93]. Those limitations combined with the CASE tool's needs of complex UIs would lead any attempt of modelling it to either a failure or an oversimplified solution.

Hopefully, some requirements defined with quite precise "boundaries" can be more or less modelled. This situation is clearly inadequate but provides the meta-CASE with a necessary comfort. Those requirements are the same as for all meta-CASEs and we do not escape them. They concern mainly the UI to create, add and edit information. We have succeeded

in generating automatically a user-friendly UI. The enriched semantics of our repository was a great help to achieve this goal.

E) The Communication

This dimension has been deliberately neglected for this very simple reason: we cannot model things that do not exist. This extreme stance must obviously be moderated: some CASEs have a distributed repository, can exchange data, can consult database's repository, ... But those features are already in the scope of the previous dimensions. We think that meta-CASEs should go one step further and propose a generic framework to help teamwork, cooperative work, concurrent engineering tasks, distributed process control, ... Although these examples were not coped with this work, we are convinced that they can be modelled with the existing components and hence, be bootstrapped. Moreover, the integration of our framework in a distributed architecture (as described in the chapter 5) would widen the meta-CASE's horizon.

Retrospective Criticism

Although we have gotten interesting results (they have just been commented), we necessarily have unsatisfied ambitions. Those ambitions either existed before the project and could not be fulfilled or they appeared as the fruit of my experience.

The initial mission my advisor entrusted to me was to extend the DB-MAIN CASE tool with still more advanced meta-facilities (the tool already had meta-properties and the Voyager 2 programming language). Unfortunately, this tool was already operational while we were analysing the requirements of a hypothetical meta-extension, and it continued to evolve with "*unfair means*" (i.e., the efforts of our three talented colleagues: Didier, Jean, and Jean-Marc). This forces us to develop the prototype as a separate framework. Nevertheless, those tools (DB-MAIN and the meta-CASE prototype) have now reached such a complexity (either hidden[1] or visible[2]) that we are now convinced that our initial decision to split the development into two distinct pieces was a wishful thinking. The integration of those both tools in using the mirroring service above a middleware component such as CORBA would certainly be a very exciting experiment.

Bootstrapping is certainly a major ingredient in the implementation of a meta-CASE to take advantage of noteworthy qualities such as: correctness, reflectivity, reification, and self-configuration. Although we have followed this way, we think that we could go one step further. For instance, GraSyLa and the structures of its associated interpreter could be bootstrapped in using the repository. The advantages could be significant: *a.* Voyager 2^+ could access the GraSyLa scripts and the graphical positions of the objects, and *b.* the visualization of the GraSyLa scripts (*cfr.* section 6.7.3) could now been synchronized. Although Voyager 2^+ could be another candidate, we are less confident in the opportunity to follow that way.

Our prototype has been developed from scratch and does not really support persistent data. Like DB-MAIN, it loads its information from a file and unloads them once the job is finished. We would like to add a persistent layer to our prototype with a fine-grained object-oriented database management system. This would allow us to generalize the tool to several

[1] The repository for instance.

[2] The user interface, the ergonomics, the methodology, the user guide, ...

users at the same time. We believe that the security management could be bootstrapped as an extension of the meta-meta-model with the help of the intrinsic mechanism of the OODBMS.

The User Interface component should of course be improved, and the use of a library like Tcl/Tk along with Voyager 2^+ should allow the method engineer to define advanced dialogue boxes. So far, the interaction with the repository is only possible through dialogue boxes although other possibilities can be envisaged: active drag&drop[1] and contextual edges[2].

Concerning the repository, we would like to add one-to-one meta-relations. This extension is not complicated and is just a specific case of the one-to-many relation. Moreover, this could improve the quality of the generated user interface and suppress some cumbersome constraints. We would also like to propose a multiple explosion mechanism and make the name of the concepts depend on the meta-model that uses them.

Description of the Prototype

This work has been implemented with Borland C++ 5.0 for Windows 98. So far, this project contains more than 300 classes, 100 templates, and it represents about 62.000 lines of code. All the bitmap screen dumps used in this thesis have been produced by this tool. The repository, the GraSyLa display processors, and the generators of user interfaces are operational. The Voyager 2^+ abstract machine is ready and waits for an adaptation of the compilers that have been developed for the original Voyager 2 programming language. The mirroring service is still under development[3].

This tool is still under development and important improvements are planned in the medium term:

1. So far, the repository is not persistent. The tool loads data from a file, and unload them afterwards. We are investigating the use of an OODBMS.

2. The current graphical library is specific to Windows and Borland. Other portable libraries are under examination.

[1] The displacement of one class to another class could trigger either a predefined action or a Voyager 2^+ method. For instance, the displacement of one “UML class C” to a “UML package P” could add X to P. Moreover, this mechanism could be used to control the execution of Petri-nets, statecharts, or process models.

[2] The software engineer links together two classes and the meta-CASE proposes the possible meta-relations depending on the type of the two selected objects.

[3] We are still looking for a cheap object oriented DBMS. `:-(`

Part III

Appendices

Appendix A

The BNF Conventions

A BNF syntax is composed of rules. Each one is expressed like this:

$\diamondsuit$ ***head*** $\leftarrow$ body

Where the head denotes a non terminal term, i.e., a construct that is defined in another rule. Non-terminal terms are noted like this ***foo***. When several rules have the same head, we suppose that their bodies are grouped together with a disjunctive operator. The body can be composed of:

literals : A literal denotes simply itself: "foo".

terminal : A terminal is an atomic concept which hides information: (*integer*, *identifier*, ...)

keyword : Keywords are literals that play a crucial role in the syntax. For this reason, the user prefers to note them in a distinct way and will often require that identifiers be distinct from keywords for instance: `begin`, `while`, ... The distinction between literals and keywords is subjective.

lists : Lists are composed of 0 or more elements: $\big\langle \text{foo} \big\rangle_{0,\infty}$.

sequence : Sequences are composed of 1 or more elements: $\big\langle \text{foo} \big\rangle_{1,\infty}$.

separated lists : Lists composed of 0 or more elements separated by an expression (sep): $\big\langle \text{foo} \big\rangle_{0,\infty}^{\text{sep}}$.

separated sequences : Sequences composed of 1 or more elements separated by an expression (sep): $\big\langle \text{foo} \big\rangle_{1,\infty}^{\text{sep}}$.

alternatives : A choice, for instance: a | b.

nothing : The empty element, noted $\emptyset$. For instance, $\big\langle \text{foo} \big\rangle_{0,\infty} \equiv \big\langle \text{foo} \big\rangle_{0,\infty}^{\emptyset}$.

optional element : The element can occur 0 or 1 times: $\big\langle \text{foo} \big\rangle$.

Elements can be grouped together with the $\big\{$and $\big\}$symbols.

Example

$\diamondsuit$ ***expression*** $\leftarrow$ $\big\langle$***expression***$\big\rangle_{1,\infty}^{\text{“+”}}$

$\diamondsuit$ ***expression*** $\leftarrow$ “(” ***expression*** `mod` ***expression*** “)”

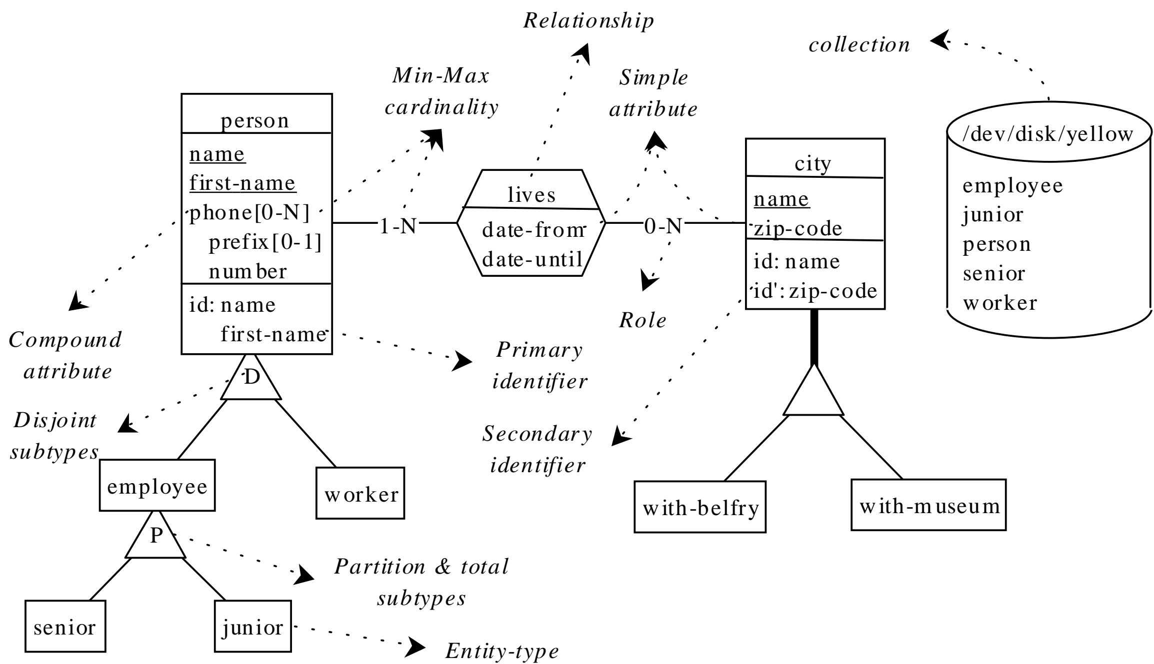
$\diamondsuit$ ***expression*** $\leftarrow$ ***integer***

$\diamondsuit$ ***integer*** $\leftarrow$ $\big\langle$“-”$\big\rangle\big\langle digit\big\rangle_{1,\infty}$

Appendix B

The DB-MAIN Conventions

This sections introduces the graphical conventions of the DB-MAIN CASE tools for ERA schemas. The graphical notations have been used in this work but their semantics has been twisted according to the rules explained in chapter 4.



Appendix C

The Voyager 2^+ Syntax

The syntax of the Voyager 2^+ programming language is given below.

$\diamondsuit$ ***program*** $\leftarrow$ $\langle$***explain-clause***$\rangle$
    package-declaration
    $\langle$***use-directive***$\rangle_{0,\infty}$
    global-variables-definitions
    functions-definitions
    `begin`
        list-instruction
    `end`

$\diamondsuit$ ***explain-clause*** $\leftarrow$ `explain` "`(*`" *anything but* `*)` "`*)`"

$\diamondsuit$ ***package-declaration*** $\leftarrow$ $\langle$`package` *cst-string* ";"$\rangle$

$\diamondsuit$ ***use-directive*** $\leftarrow$ ***use-library*** | ***use-function***

$\diamondsuit$ ***use-library*** $\leftarrow$ `use` *cst-string* `as` $identifier_{lib}$ ";"

$\diamondsuit$ ***use-function*** $\leftarrow$ `use` $identifier_{lib}$ "." $identifier_{fct}$ `as` $identifier_{alias}$

$\diamondsuit$ ***global-variables-definitions*** $\leftarrow$ $\langle$***definition-line***$\rangle_{0,\infty}$

$\diamondsuit$ ***definition-line*** $\leftarrow$ ***type*** ":" $\langle$ *identifier* $\langle$"=" ***expr***$\rangle\rangle_{1,\infty}^{\text{“,”}}$ ";"

$\diamondsuit$ ***type*** $\leftarrow$ ***expr***

$\diamondsuit$ ***functions-definitions*** $\leftarrow$ $\langle$***def-method***$\rangle_{0,\infty}$

$\diamondsuit$ ***def-method*** $\leftarrow$ ***def-function*** | ***def-procedure***

$\diamondsuit$ ***def-function*** $\leftarrow$ $\langle$`export`$\rangle$ $\langle$`method`$\rangle$ `function` ***expr*** *identifier*
    "(" $\langle$***param***$\rangle_{0,\infty}^{\text{“,”}}$ ")"
    explain-clause $\langle$***definition-line***$\rangle_{0,\infty}$ "{" $\langle$***instr***$\rangle_{0,\infty}$ "}"

$\diamondsuit$ ***def-procedure*** $\leftarrow$ $\langle$`export`$\rangle$ $\langle$`method`$\rangle$ `procedure` *identifier*
    "(" $\langle$***param***$\rangle_{0,\infty}^{\text{“,”}}$ ")"
    explain-clause $\langle$***definition-line***$\rangle_{0,\infty}$ "{" $\langle$***instr***$\rangle_{0,\infty}$ "}"

$\diamondsuit$ ***param*** $\leftarrow$ {{`relax` $\langle$***expr***$\rangle$ } | ***expr*** }":" *identifier*

♢ ***instr*** ← ∅
| ***assignment-statement***
| ***block-clause*** ⟨“;”⟩
| ***if-then-statement*** ⟨“;”⟩
| ***if-then-else-statement*** ⟨“;”⟩
| ***switch-statement*** ⟨“;”⟩
| ***while-statement*** ⟨“;”⟩
| ***repeat-statement*** ⟨“;”⟩
| ***for-statement*** ⟨“;”⟩
| ***goto-statement***
| ***label-statement***
| ***continue-statement***
| ***break-statement***
| ***halt-statement***
| ***call-fp*** “;”
| ***return-statement***
| ***attach-statement***
| ***move-statement***
| ***addcursor-statement***
| ***setval-statement***

♢ ***assignment-statement*** ← ***lhs*** “:=” ***expr***

♢ ***lhs*** ← *identifier* | *identifier* “.” ***expr***

♢ ***if-then-statement*** ← `if` ***expr*** `then` “{” ***list-instruction*** “}”

♢ ***if-then-else-statement*** ← `if` ***expr*** `then` “{” ***list-instruction*** “}” `else` “{” ***list-instruction*** “}”

♢ ***list-instruction*** ← $\langle$***instr***$\rangle_{0,\infty}$

♢ ***while-statement*** ← `while` ***expr*** `do` “{” ***list-instruction*** “}”

♢ ***repeat-statement*** ← `repeat` “{”***list-instruction*** “}” `until` ***expr***

♢ ***block-clause*** ← `let` $\langle$***definition-line***$\rangle_{1,\infty}$ `in` “{” ***list-instruction*** “}”

♢ ***goto-statement*** ← `goto` *identifier*

♢ ***for-statement*** ← `for` *identifier* `in` ***expr*** `do` “{” ***list-instruction*** “}”

♢ ***switch-statement*** ← `switch` “(” *identifier* “)” “{” $\langle$***case-sttmt***$\rangle_{0,\infty}$ ⟨***default***⟩ “}”

♢ ***case-sttmt*** ← `case` ***expr*** “:” ***list-instruction***

♢ ***default*** ← `otherwise` “:” ***list-instruction***

♢ ***label-statement*** ← `label` “identifier”

♢ ***continue-statement*** ← `continue` “;”

♢ ***break-statement*** ← `break` “;”

♢ ***halt-statement*** ← `halt` “;”

♢ ***call-fp*** ← ***call-functor*** “(” $\langle$***expr***$\rangle^{“,”}_{0,\infty}$ “)”

◇ ***call-functor*** ← *identifier*

| ***expr***[1]

| ***expr*** "`::`" ***expr***

| ***expr*** "`->`" ***expr***

◇ ***return-statement*** ← `return` ⟨***expr***⟩";"

◇ ***attach-statement*** ← `attach` *identifier* `to` ***expr*** ";"

◇ ***move-statement*** ← *identifier* {"`<<`" | "`>>`" }⟨***expr***⟩ ";"

◇ ***addcursor-statement*** ← ***expr*** {"`<+`" | "`+>`" }***expr*** ";"

◇ ***setval-statement*** ← ***expr*** "`<--`" ***expr*** ";"

◇ ***expr*** ← `self`
| `this`
| ***expr omega2 expr***
| ***lhs*** "`:==`" ***expr***
| "-" ***expr***
| `not` "(" ***expr*** ")"
| "(" ***expr*** ")"
| "[" $\langle \textbf{\textit{expr}} \rangle_{0,\infty}^{\text{“,”}}$ "]"
| "[" ***expr*** "`..`" ***expr*** "]"
| ***query-expr***
| ***call-fp***
| ***lhs***
| *cst-integer*
| *cst-char*
| *cst-string*
| *cst-real*

◇ ***query-expr*** ← ***expr*** "{" Π-***expr*** "}"

◇ Π-***expr*** ← ***expr*** | {***path-constraint*** ⟨`with` ***expr*** ⟩ }

◇ ***path-constraint*** ← ⟨"@"⟩ ***expr*** ":" ***expr***

◇ ***omega2*** ← "`-`" | "`+`" | "`*`" | "`<`" | "`>`" | "`<=`" | "`>=`" | "`=`" | "`<>`" | "`++`" | "`**`" | `and`
| `or` | `xor` | `mod` | `%%`

Remark The `#include` directive does not appear in the Voyager 2^+ syntax since it is systematically replaced with the content of the specified file.

[1]The parser has enough information to distinguish identifiers which denote functors of builtin/user-defined functions from identifiers which denote `lambda` expressions.

Appendix D

Listings

D.1 The Voyager 2^+'s Constants

D.1.1 The Builtin Types

```
/**********************************************************/
/*  This file should be included in any voyager 2+ program. */
/*  It contains all the built-in types of the language.     */
/**********************************************************/
/*  file: TYPES-CST.I */
/*********************/

// meta_class=MetaClass("meta_class");
// this constant is hard-coded in the system.

meta_class:
  integer=MetaClass("integer"),
  real=MetaClass("real"),
  char=MetaClass("char"),
  string=MetaClass("string"),
  file=MetaClass("file"),
  list=MetaClass("list"),
  cursor=MetaClass("cursor"),
  lambda=MetaClass("lambda"),
  program=MetaClass("program");

meta_class:
  meta_relation=MetaClass("meta_relation"),
  meta_isa=MetaClass("meta_isa"),
  meta_model=MetaClass("meta_model"),
  meta_property=MetaClass("meta_property"),
  meta_method=MetaClass("method");

meta_relation:
  rel_prop=GetFirst(meta_relation{self."name"="prop"}),
```

```
  rel_methods=GetFirst(meta_relation{self."name"="methods"}),
  rel_contains=GetFirst(meta_relation(self."name"="contains"),
  rel_belongs_to=GetFirst(meta_relation(self."name"="belongs_to")),
  rel_subtype_of=GetFirst(meta_relation(self."name"="subtype_of")),
  rel_supertype_of=GetFirst(meta_relation(self."name"="supertype_of")),
  rel_owner=GetFirst(meta_relation(self."name"="owner")),
  rel_member=GetFirst(meta_relation(self."name"="member")) ;

/* eof */
```

D.1.2 The Integer Constants

```
/* Code errors
*/

/* -------------- */
/* file: INT-CST.I */
/* -------------- */

integer:
  ERR_CALL=1,
  ERR_CANCEL=2,
  ERR_DIV_BY_ZERO=3,
  ERR_ERROR=4,
  ERR_FILE_CLOSE=5,
  ERR_FILE_OPEN=6,
  ERR_NO_UPDATE=7,
  ERR_PATH_NOT_FOUND=8,
  ERR_PERMISSION_DENIED=9;

/* eof */
```

D.2 The Voyager 2^+ Predefined Operations

D.2.1 Operations on Characters

`function char:` c `AscToChar` (`integer:` d)

Precondition. $0 \leq d \leq 255$

Postcondition. Character c has the ASCII code: d.
on error: $c =$`^0^`.

`function integer:` d `CharIsAlpha` (`char:` c)

Precondition. **true**

Postcondition. $d = 1$ is $c \in \{a, \ldots, z, A, \ldots, Z\}$ and 0 otherwise.

`function integer:` d `CharIsAlphaNum` (`char:` c)

Precondition. **true**

Postcondition. $d = 1$ if $c \in \{0, \ldots, 9, a, \ldots, z, A, \ldots, Z\}$ and 0 otherwise.

`function integer:` d `CharIsDigit` (`char:` c)

Precondition. **true**

Postcondition. $d = 1$ if $c \in \{0, \ldots, 9\}$ and 0 otherwise.

`function integer:` d `CharToAsc` (`char:` c)

Precondition. **true**

Postcondition. d is the ASCII code of the character c.

`function char` c' `CharToLower` (`char` c)

Precondition. **true**

Postcondition. if $c \in \{A, \ldots, Z\}$ then c' is the respective lower case letter. All other characters are left unchanged.

`function string:` s `CharToStr` (`char:` c)

Precondition. **true**

Postcondition. s is a string composed of the c character.

`function char:` c' `CharToUpper` (`char:` c)

Precondition. **true**

Postcondition. if $c \in \{a, \ldots, z\}$ then c' is the respective upper case letter. All other characters are left unchanged.

D.2.2 Operations on Strings

We will define here some definitions to make easier the explanations that will follow. First of all, let s denote a string and d be a positive number. Then we note s_d the dth character of the string s, and $s_{d\rightarrow}$ the suffix of the string s starting at the position d (included) in the string and $s_{d\rightarrow d+l}$ the substring comprised between positions d and $d+l$ where l is a positive number such that $d+l$ does not exceed the length of the string. Let us remember a last detail: the first character of a string is placed at the position 0, and thus if l is the length of s, the last character is placed at the position $l-1$. The following operations are safe with respect to two criteria:

- The program can never write a character outside strings.
- The program can never place the null character inside a string[1].

This is a valuable guaranty against frequent bugs that C and Pascal programmers certainly know.

`function string:` s `StrBuild` (`integer:` d)

Precondition. $d \geq 0$ and $d \leq$`MAX_STRING`

Postcondition. s is a string composed of d space characters (`' '`).
on error: s is the empty string.

[1] By convention, the null character ends strings. Therefore such a possibility is troubling the memory manager.

`function integer StrCmp ( string:` s_1`, string:` s_2 `)`

Precondition. **true**

Postcondition. Returns 0 if $s_1 = s_2$, 1 if $s_1 > s_2$ and -1 otherwise.

`function integer StrCmpLU ( string:` s_1`, string:` s_2 `)`

Precondition. **true**

Postcondition. Returns 0 if $s'_1 = s'_2$, 1 if $s'_1 > s'_2$ and -1 otherwise, where $s'_i =$ `StrToUpper`(s_i) and $i \in \{1, 2\}$.

`function string:` s `StrConcat ( string:` s_1`, string:` s_2 `)`

Precondition. **true**

Postcondition. This function appends the string s_2 at the end of s_1 and the result is stored in s. The length of the resulting string is `StrLength(`s_1`)+StrLength(`s_2`)`. The infix operator "+" can also be used in place of the `StrConcat` function.

`function integer:` r `StrFindChar ( string:` s`, integer:` d`, char:` c `)`

Precondition. $0 \leq d <$ `StrLength`(s).

Postcondition. If $r \geq 0$ then $s_r = c$ and $\forall i,\ d \leq i < r :\ s_i \neq c$. Otherwise if $r = -1$ then $\forall i,\ d \leq i <$ `StrLength`$(s) :\ s_i \neq c$.
on error: $r = -1$.

`function integer:` r `StrFindSubStr ( string:` s`, integer:` d`, string:` t `)`

Precondition. $0 \leq d <$ `StrLength`(s).

Postcondition. If $r \geq 0$ then t is a prefix of $s_{d+r\rightarrow}$. Otherwise if $r = -1$ then $\forall i \geq 0$, t never is a prefix of $s_{d+i\rightarrow}$.
on error: $r = -1$

`function char:` c `StrGetChar ( string:` s`, integer:` d `)`

Precondition. $0 \leq d <$ `StrLength`(s).

Postcondition. $c = s_d$.
on error: $c =$`'^0^'`

`function string:` r `StrGetSubStr ( string:` s`, integer:` d`, integer:` l `)`

Precondition. $0 \leq d <$`StrLength`$(s)\ \wedge\ 0 \leq l \leq$`StrLength`$(s) - d$

Postcondition. $r = s_{d\rightarrow d+l}$
on error: if $d < 0$ then $d \leftarrow 0$; if $d \geq$`StrLength`(s) then $d \leftarrow$`StrLength`(s); if $l < 0$ then $l \leftarrow 0$; if $l >$`StrLength`$(s) - d$ then the function will consider that $l-$`StrLength`$(s) + d$ space characters are added at the end of the string s.

`function integer StrIsInteger ( string:` s `)`

Precondition. **true**

Postcondition. Returns 1 if `CharIsDigit`(s_i)=1 $\forall 0 \leq i \leq$ `StrLength`$s - 1$ and 0 otherwise.

`function string:` s `StrItos ( integer:` d `)`

Precondition. **true**

Postcondition. Converts the integer d into the string s.

`function integer:` d `StrLength ( string:` s `)`

Precondition. **true**

Postcondition. d is the length of the string s.

`function string:` s' `StrSetChar ( string:` s, `integer:` d, `char:` c `)`

Precondition. $0 \leq d <$`StrLength`(s) and $c \neq$`'^0^'`.

Postcondition. $\forall i \in \{0 \ldots$ `StrLength`$(s) - 1\} \setminus \{d\} : s'_i = s_i$ and $s'_d = c$.
on error: $s' = s$.

`function integer:` d `StrStoi ( string:` s `)`

Precondition. 1) The number represented by s is a number between `INT_MIN` and `INT_MAX`. 2) The string must match this regular expression: `[\t ]*[+-]?[0..9]+`. The string may start with spaces or tabular characters but must end with a number. A number may have a sign (+ or -) and must have at least one digit.

Postcondition. Converts the longest prefix of s satisfying the above regular expression to an integer d. Space and tabular characters at the beginning of s are omitted.
on error: If the value d is outside the integer range, then the result d is undefined. If the string s does not match the regular expression, then $d = 0$.

`function string:` s' `StrToLower ( string:` s `)`

Precondition. **true**

Postcondition. All the characters $c \in \{A, \ldots, Z\}$ in the string s are replaced by their corresponding lowercase letters, the result is stored in s'. No other characters are changed.

`function string:` s' `StrToUpper ( string:` s `)`

Precondition. **true**

Postcondition. All the characters $c \in \{a, \ldots, z\}$ in the string s are replaced by their corresponding upper case letters, the result is stored in s'. No other characters are changed.

See also `MakeChoice` and `MakeChoiceLU` in chapter D.2.4 (pages 253).

D.2.3 Operations on Lists and Cursors

`procedure AddFirst ( list:` $l1$, `relax` e `)`

Precondition. **true**

Postcondition. After evaluation of the expression e, the result is added to the list $l1$ at the first position. If the expression is a list, this list is shared by $l1$.

`procedure AddLast ( list:` $l1$, `relax:` e `)`

Precondition. **true**

Postcondition. After evaluation of the expression e, the result is added to the list $l1$ at the last position. If the expression is a list, this list is shared by $l1$.

`function relax:` r `GetFirst ( list:` l `)`

Precondition. l is a non-empty list

Postcondition. r is the first element of the list l. Of course, if the first element of a the list is a list, then the result is not a copy of it but shares it.
on error: the program is halted.

`function relax:` r `GetLast ( list:` l `)`

Precondition. l is a non-empty list

Postcondition. r is the last element of the list l. Of course, if the last element of a the list is a list, then the result is not a copy of it but shares it.
on error: the program is halted.

`function integer :` n `Length ( list:` l `)`

Precondition. **true**

Postcondition. n is the number of elements found in the list l.

`function cursor:` c `member ( list:` l, `relax :` m `)`

Precondition. **true**

Postcondition. If the element m occurs in the list l, then the cursor c points to this element. If the elements occurs more than once, then c points to the first occurrence. Elements that have a type different of the element m are omitted. If m does not belong to the list then the cursor c is void.

D.2.4 Operations on Files

`procedure CloseFile ( file: f )`

Precondition. f denotes an handle to a file opened with the instruction `OpenFile`.

Postcondition. The file is closed, and the value of f is undefined.
on error: The error register is set to `ERR_FILE_CLOSE`.

`function file OpenFile ( string: s, integer: m )`

Precondition. s is the name of a file. m is an integer constant among: `_W` for the write mode and `_R` for the read mode and `_A` for the append mode.

Postcondition. Depending on the value of m:

`_W` **:** If the file s exists then it is destroyed and the result is a handle to a new file opened for writing only. If s is not a valid name, the result is `void` and the error register is set to `ERR_FILE_OPEN`.

`_R` **:** If the file s does not exist, the result is the value `void` and the error register is set to `ERR_FILE_OPEN`. Otherwise the result is a handle to the file opened for reading. The *current position* is either the first character of the file or the end of file if the file is empty.

`_A` **:** If the file s does exist then the function returns an handle to this file opened for writing at the end-of-file. Otherwise, the file is created and the function behaves like the mode was `_W`.

`procedure printf ( file: f, relax: v )`

Precondition. f denotes a file opened for writing and v is any expression whose the type is `string` or `char` or `integer` or `real` or `list`.

Postcondition. v is written on the v file. If the type of v is `list` then all the values found in this list are written on the file surrounded by the string constants `LEFT` and `RIGHT`. They are separated by the `COMMA`[1] string constant (*cfr.* `SetPrintList` for more details about these constants). Values that do not belong to types `string`, `char`, `integer`, `list` are skipped.
on error: The behavior is undefined.

Let us remark that very depth or recursive lists perturb this procedure. The procedure `print` only accepts the second argument of `printf` and write the value on the console.

`function any readf ( file: f, meta_class: t )`

Precondition. f denotes a file opened for reading and t denotes the type of the information the instruction has to read in the file f. Following constants can be used: `_integer`, `_char`, `_string`.

Postcondition. Upon the value of t, the instruction will behave like this:

[1] These constants are internal and are not visible.

`integer` **:** The longest sequence of decimal digits optionally preceded by - or + is read from the *current position*. At the end, the *current position* is either the first character after the sequence or the end of file. If *current position* is either the end of file or is not indicating a number, then 0 is returned. If the sequence denotes a number outside the range `[INT_MIN ... INT_MAX]` then the instruction returns a random integer.

`real` **:** The longest sequence of characters that matches this regular expression `[0-9]+(\.[0-9]+)?` is read from the *current position*. At the end, the *current position* is either the first character after the sequence or the end of file. If *current position* is either the end of file or is not indicating a number, then 0 is returned. The range of the number is not checked and thus cause overflow errors.

`string` **:** The longest sequence of characters before either the end of file or the first character '\n' or the `MAX_BUFFER`*nth* character after the *current position*. If the *current position* is the end of file or is indicating the end of line character, then the empty string is returned. The *current position* becomes either the end of file or the first character after the sequence.

`char` **:** The character under the *current position* is returned. If the end of file is reached, the ASCII code 0 is returned.

The function `read` behaves like `readf` except that characters are read from the console.

`function integer eof ( file:` f `)`

Precondition. The file f is opened.

Postcondition. `eof` returns 1 if the end of file is reached and 0 otherwise. For files opened for writing, the function always returns 1. (see also `neof`)

`function integer neof ( file:` f `)`

Precondition. The file f is opened.

Postcondition. `neof` returns 0 if the end of file is reached and 1 otherwise. For files opened for writing, the function always returns 0. (see also `eof`)

`procedure SetPrintList ( string:` l`, string:` r`, string:` c `)`

Precondition. l, r and c are strings with no more than `MAX_DELIM` characters. Strings can be empty.

Postcondition. Strings l, r, c are respectively assigned to the "constants" `LEFT`, `RIGHT` and `COMMA`.

Miscellaneous Operations on Files

Some other functions are discussed here although they have no concern with the `file` type.

`procedure DeleteFile ( string: f )`

Precondition. f is the name of an existing file.

Postcondition. File f is deleted. On errors, the error register is set to one of the following values: `ERR_PERMISSION_DENIED`, `ERR_PATH_NOT_FOUND`.

`function integer ExistFile ( string: f )`

Precondition. f is a valid file name[1]. The file may not exist.

Postcondition. The function returns 1 if the file exists. Otherwise, error codes `ERR_PERMISSION_DENIED` and `ERR_PATH_NOT_FOUND` can be returned. The error register is not modified.

`procedure RenameFile ( string: o, string: n )`

Precondition. o is the name of an existing file. n is a file name that does not yet exist. Both expressions must denote files on a same physical device.

Postcondition. File o is renamed n. If paths are different, then this instruction will move the file. On errors, the error register is set to `ERR_ERROR`.

Example

```
file: f;
begin
  f:=OpenFile("c:\\tmp\\foo.txt",_W);
  SetPrintList("(",")",",");
  printf(f,[1,[1,2,3],4,'\n']);
  SetPrintList("\/*","*/\n","\n");
  printf(f,["line comment 1","line comment 2","line comment 3"]);
  CloseFile(f);
end
```

The program will print the next characters in the file named `foo.txt`:

```
(1,(1,2,3),4)
/*line comment 1
line comment 2
line comment 3*/
```

Lexical Analyzer

All these functions use the same *input stream* which is initialized by the function `SetParser`. The input stream may be either a file or a string. Because a stream is a little bit more sophisticated than a normal file (*cfr.* `OpenFile`), usual functions associated with the `file` type cannot be used with this *input stream*.

[1] See the operating system's reference manual for more details.

Example

The general skeleton of a program that will use these operations will look like this:

```
file: f;
...
begin
  f:=OpenFile("input.txt",_R);
  SetParser(f);  // or SetParser("123 654 789  ");
  while nseof() do {
    SkipUntil("0-9");
    print(GetTokenWhile("0-9"));
  }
  CloseFile(f);
end
```

`procedure SetParser` (τ: sf)

Precondition. τ is either the `string` type or the `file` type. This instruction specifies from which stream the functions from the lexical library will read the input. If the argument is a string, then all the lexical functions will read characters from a "virtual" file initialized with the argument. Otherwise, if it is a file, characters are read from the file itself.

Postcondition. The input stream is initialized.

`function string` : r `GetTokenWhile` (`string` : s)

Precondition. The input stream is initialized.The argument must be a literal string – the value of s must be known at the compilation time. This string denotes a pattern that defines the behaviour of the lexical analyzer. The pattern specifies a range of characters. This range may be defined either in extension or in expansion. The first character of the pattern is always interpreted literaly. For the other ones, the pattern is expanded like that: for each occurence of "α-β" where α, β denote any character, this substring is replaced in the pattern with the set of characters $\gamma : \alpha \leq \gamma \leq \beta$. Thus the following pattern : "`-a-d0-4`" is equivalent to the string: "`-abcd01234`".

Postcondition. Let $(\alpha_i)_1^n$ be the characters present in the input stream of the lexical library. The result of this function is the string $(\alpha_i)_1^m$ where $0 \leq m \leq n$ and $\forall i \in 1..m : \alpha_i \in \mathcal{P}$ and if $m < n$ then $\alpha_{m+1} \notin \mathcal{P}$ where $\mathcal{P}$ is the pattern. After the call, the input stream is replaced with $(\alpha_i)_{m+1}^n$.

`function string` : r `GetTokenUntil` (`string` : s)

Precondition. The input stream is initialized. The argument must be a literal string – the value of s must be known at the compilation time. This string denotes a pattern that defines the behaviour of the lexical analyzer. The pattern specifies a range

of characters. This range may be defined either in extension or in expansion. The first character of the pattern is always interpreted literaly. For the other ones, the pattern is expanded like that: for each occurence of "α-β" where α, β denote any character, this substring is replaced in the pattern with the set of characters $\gamma : \alpha \leq \gamma \leq \beta$. Thus the following pattern : "`-a-d0-4`" is equivalent to the string: "`-abcd01234`".

Postcondition. Let $(\alpha_i)_1^n$ be the characters present in the input stream of the lexical library. The result of this function is the string $(\alpha_i)_1^m$ where $0 \leq m \leq n$ and $\forall i \in 1..m : \alpha_i \notin \mathcal{P}$ and if $m < n$ then $\alpha_{m+1} \in \mathcal{P}$ where $\mathcal{P}$ is the pattern. After the call, the input stream is replaced with $(\alpha_i)_{m+1}^n$.

`procedure SkipWhile ( string : s )`

Precondition. The input stream is initialized. The argument must be a literal string – the value of s must be known at the compilation time. This string denotes a pattern that defines the behaviour of the lexical analyzer. The pattern specifies a range of characters. This range may be defined either in extension or in expansion. The first character of the pattern is always interpreted literaly. For the other ones, the pattern is expanded like that: for each occurence of "α-β" where α, β denote any character, this substring is replaced in the pattern with the set of characters $\gamma : \alpha \leq \gamma \leq \beta$. Thus the following pattern : "`-a-d0-4`" is equivalent to the string: "`-abcd01234`".

Postcondition. Let $(\alpha_i)_1^n$ be the characters present in the input stream of the lexical library. After the call, the input stream is replaced with $(\alpha_i)_{m+1}^n$ where m is defined as $0 \leq m \leq n$ and $\forall i \in 1..m : \alpha_i \in \mathcal{P}$ and if $m < n$ then $\alpha_{m+1} \notin \mathcal{P}$ where $\mathcal{P}$ is the pattern.

`procedure SkipUntil ( string : s )`

Precondition. The argument must be a literal string – the value of s must be known at the compilation time. This string denotes a pattern that defines the behaviour of the lexical analyzer. The pattern specifies a range of characters. This range may be defined either in extension or in expansion. The first character of the pattern is always interpreted literaly. For the other ones, the pattern is expanded like that: for each occurence of "α-β" where α, β denote any character, this substring is replaced in the pattern with the set of characters $\gamma : \alpha \leq \gamma\beta$. Thus the following pattern : "`-a-d0-4`" is equivalent to the string: "`-abcd01234`".

Postcondition. Let $(\alpha_i)_1^n$ be the characters present in the input stream of the lexical library. After the call, the input stream is replaced with $(\alpha_i)_{m+1}^n$ where m is defined as $0 \leq m \leq n$ and $\forall i \in 1..m : \alpha_i \notin \mathcal{P}$ and if $m < n$ then $\alpha_{m+1} \in \mathcal{P}$ where $\mathcal{P}$ is the pattern.

`procedure UngetToken ( string : s )`

Precondition. The input stream is initialized. Let $(\alpha_i)_1^n$ be the input stream. α_1 is the first character. Let $(\sigma_j)_1^m$ be the sequence of letters in s.

Postcondition. The input stream is replaced with $(\sigma_j)_1^m \circ (\alpha_i)_1^n$ where $\circ$ is the "append" operator for lists.

`function char : c GetChar ( )`

Precondition. The input stream is initialized. There is at least one character in the input stream.

Postcondition. The first character of the input stream is removed and returned.

`function integer seof ( )`

Precondition. The input stream is initialized.

Postcondition. Let n be the value returned by this function. Then $n = 1$ if there is one character in the input stream and 0 otherwise.

`function integer nseof ( )`

Precondition. The input stream is initialized.

Postcondition. Let n be the value returned by this function. Then $n = 0$ if there is one character in the input stream and 1 otherwise.

Remark Althouth the place of the `MakeChoice` and `MakeChoiceLU` functions should be in section D.2.2, their definition was motivated by the use of the lexical analyzer. Therefore, we decided to let them in this section.

`function integer: d MakeChoice ( string: s, list: l )`

Precondition. l is a literal list of strings —This value must be known at the compilation time. We note $(\sigma_i)_1^n$ the list l where σ_1 is the first element. All the values should be distinct.

Postcondition. if $\exists i \in 1..n : \ \sigma_i = s$ then $d = i$ otherwise $d = 0$. The complexity of the this function is $\Theta(\log_2 n)$. **on error:** If one value occurs several times in the list l, then the result is random. The compiler prints a warning message.

`function integer: d MakeChoiceLU ( string: s, list: l )`

Precondition. l is a literal list of strings —This value must be known at the compilation time. We note $(\sigma_i)_1^n$ the list l where σ_1 is the first element. All the values should be distinct. labeli-makechoicelu

Postcondition. if $\exists i \in 1..n : \texttt{StrCmpLU}(\sigma_i, s) = 0$ then $d = i$ otherwise $d = 0$. This comparaison is case **in**sensitive. The complexity of the this function is $\Theta(\log_2 n)$. **on error:** If one value occurs several times (*case insensitive*) in the list l, then the result is random. The compiler prints a warning message.

D.2.5 Interface Operations

Remark The French sentences in the snapshots are due to the default language of the operating system.

`function string BrowsePrint` (`string` : t, `string` :m, `string`: e)

Precondition. **true**

Postcondition. Create a dialog box shown in figure D.1 (page 256) from the argument interpreted as:

t**:** the title (`<TITLE>` in the figure)

m**:** the message (`<MESSAGE>` in the figure)

e**:** suggested extensions. This string is formatted as a list of pairs like this:
`"name|ext|name|ext|name|ext"`
where `name` is the associated name of one extension ("`text file`" for instance) and `ext` is its extension ("`*.v2`" for instance).

If the user chooses the `CANCEL` button, the result is the empty string and the error register is set to `ERR_CANCEL`. Otherwise, the result is name of the selected file (with its path). The user may either choose an existing file or type a new name.

`function string BrowseRead` (`string` : t, `string` :m, `string`: e)

Precondition. **true**

Postcondition. Create a dialog box shown in figure D.1 (page 256) from the argument interpreted as:

t**:** the title (`<TITLE>` in the figure)

m**:** the message (`<MESSAGE>` in the figure)

e**:** suggested extensions. This string is formatted as a list of pairs like this:
`"name|ext|name|ext|name|ext"`
where `name` is the associated name of one extension ("`text file`" for instance) and `ext` is its extension ("`*.v2`" for instance).

If the user chooses the `CANCEL` button, the result is the empty string and the error register is set to `ERR_CANCEL`. Otherwise, the result is name of the selected file (with its path). Although file names are greyed, the user can type a new file name.

`function integer`: r `Choice` (`string` : t, `list` :L, `integer`: s, `integer`: m)

Precondition. **true**

Postcondition. Create a dialog box that let the user to choose an item among several items.

t**:** the title (`<TITLE>` in the figure)

L: the list that will denote the possible items. The elements of *L* which do not match the `string` type will be omitted.

s: the listbox will be sorted if $s \neq 0$.

m: whenever the user will click on the OK button, an item must be selected

If the user chooses the `CANCEL` button, the result is -2. If the choice is not mandatory ($m =$`FALSE`) and if no item is selected, then the result is -1. Otherwise, the result is the index of the string in the list (the first index is 0). (See Fig. D.2).

`function string DialogBox ( string : t, string :m, integer: s, string: d )`

Precondition. **true**

Postcondition. Create a dialog box shown in figure D.3 (page 256) from the argument interpreted as:

t: the title (`<TITLE>` in the figure)

m: the message (`<MESSAGE>` in the figure)

s: the maximum length of the input area in characters

d: the default string displayed in the input area (`<DEFAULT VALUE>` in the figure)

If the user chooses the `CANCEL` button, the result is the empty string and the error register is set to `ERR_CANCEL`.

`procedure MessageBox ( string : t, string :m )`

Precondition. **true**

Postcondition. Create a messge box displayed as in the figure D.4 (page 256). Arguments are interpreted as:

t: the title (`<TITLE>` in the figure)

m: the message (`<MESSAGE>` in the figure)

D.2.6 Time Operations

`function integer GetDay ( )`

Precondition. **true**

Postcondition. returns the current day (1-31).

`function integer GetHour ( )`

Precondition. **true**

Postcondition. returns the hour (0-23).

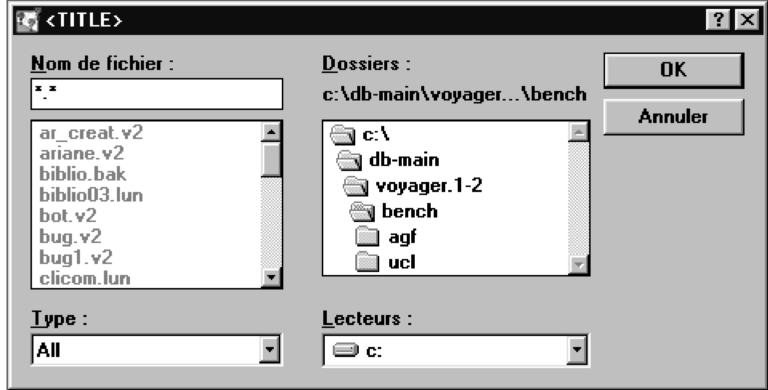
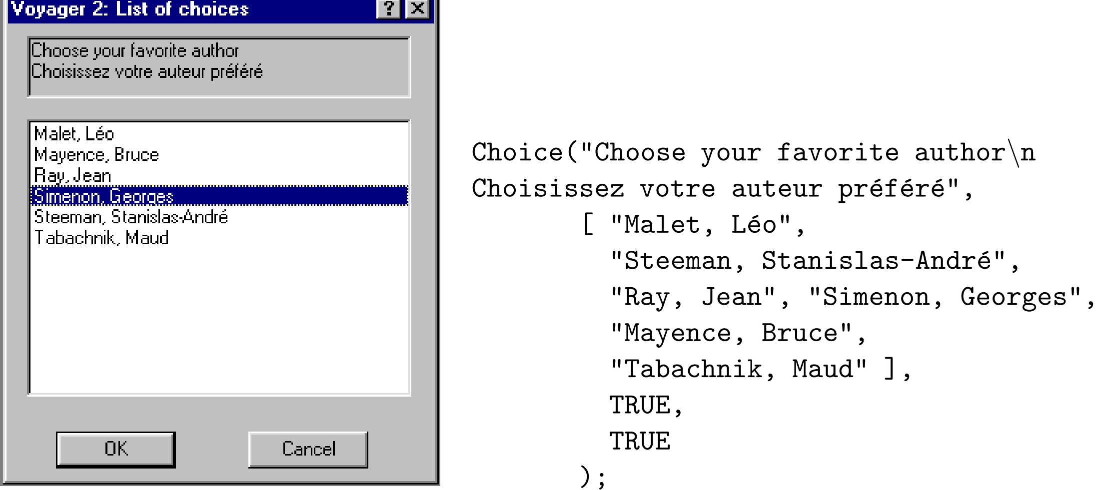


Figure D.1: A File Browsing Window




```
Choice("Choose your favorite author\n
Choisissez votre auteur préféré",
       [ "Malet, Léo",
         "Steeman, Stanislas-André",
         "Ray, Jean", "Simenon, Georges",
         "Mayence, Bruce",
         "Tabachnik, Maud" ],
         TRUE,
         TRUE
       );
```

Figure D.2: A Choice Dialog

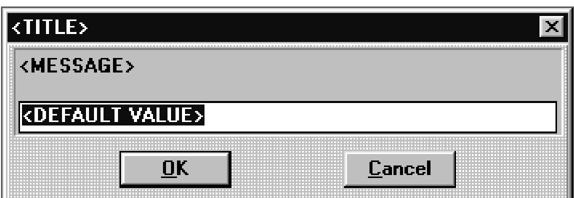


Figure D.3: A DialogBox Window

Figure D.4: A Message Box

`function integer GetMin ( )`

Precondition. **true**

Postcondition. returns the minute (0-59).

`function integer GetMonth ( )`

Precondition. **true**

Postcondition. returns the month (1-12).

`function integer GetSec ( )`

Precondition. **true**

Postcondition. returns the second (0-59).

`function integer GetWeekDay ( )`

Precondition. **true**

Postcondition. returns the day in the week (1-7).

`function integer GetYear ( )`

Precondition. **true**

Postcondition. returns the year.

`function integer GetYearDay ( )`

Precondition. **true**

Postcondition. returns the day in the year (1-366).

D.2.7 Flag Operations

Integers are sometimes used as arrays of bits. These functions allows the programmer to access and modify each bit of a 32bit integer.

`function integer:` r `GetFlag ( integer:` d, `integer:` p `)`

Precondition. **true**

Postcondition. Returns the bit stored at position p in the integer d. The value p is a binary mask to access the bit.

`function integer:` r `SetFlag ( integer:` d, `integer:` p, `integer:` v `)`

Precondition. **true**

Postcondition. Builds a new flag from d where the bit at position p has been set to v. All the other bits are let unchanged.

D.2.8 General Operations

`procedure CallSystem ( string: s )`

Precondition. s denotes a windows application with optional arguments.

Postcondition. The program s is executed and the Voyager 2^+ program continues its execution.
on error: The error register is set to the constant `ERR_CALL`.

`procedure ClearScreen ( )`

Precondition. **true**

Postcondition. The console window is cleared.

`function a class GetObject ( )`

Precondition. **true**

Postcondition. Returns the class that has been selected in the active window by the software engineer. The class is `void` if no class is selected. By default, the class is the leaf of the selected item in the current specification.

`function a specification GetSpecification ( )`

Precondition. **true**

Postcondition. Returns the specification that is displayed in the active window. If no window is active, the result is `void`.

`function integer: e GetError ( )`

Precondition. **true**

Postcondition. e is the value found in the error register. The call puts the value 0 in the error register.

`function integer: r GetID ( relax: x )`

Precondition. The type of x must not be an elementary type.

Postcondition. $x = \texttt{void} \wedge r = 0 \otimes r = \texttt{cid}(x)$

`function string: s GetOxoPath ( )`

Precondition. **true**

Postcondition. Returns a string with the path of the `.oxo` file that contains the current program.

`function` meta_class: r `GetType` (a class: c, specification: s)

Precondition. **true**

Postcondition. If c = `void` or s is `void`, then r denotes the type of c (Let us remember that all the values are typed in Voyager 2^+, even the `void` values). Otherwise, r denotes the leaf of c in s. If $c \notin \mathrm{Def}(s)$ then r = `void`.

`function integer`: r `IsNoVoid` (`relax`: v)

Precondition. **true**

Postcondition. returns `not IsVoid(v)`. See `IsVoid` for more details.

`function integer`: r `IsVoid` (`relax`: v)

Precondition. **true**

Postcondition. Returns `TRUE` if the argument denotes the `void` value. If the type of the argument is `list` or `string` the result is always `FALSE`. For integers and characters, this predicate is true if the value is the integer 0 or the character `'^0^'`.

`function` c `RetrieveID` (`integer`:i)

Precondition. **true**

Postcondition. If there is a class c such that GetID(c)=i, then the function returns c. Otherwise, the result is `void`.

`function` t: r `Void` (`meta_class`: t)

Precondition. $t \notin$ {`void`, `list`, `string`}

Postcondition. r is a value of type t whose the value is `void`. **on error:** The program halts.

D.3 GraSyLa Scripts

D.3.1 The GraSyLa Editor's Listing

```
view "Grasyla Editor" ;

begin

root: <G-root>, <G-junk>, <G-equation>, <G-connection>, <G-path> ;
junk: <G-functor>, <meta_class> ;

/****************************************************/
/* Display the root and junk meta-classes of the script */
/****************************************************/

meta-class list $<root> =
     boxV{ boxH { spring bold{"Root Meta-Classes"} spring }
           ruleH
           head
           tail
         }
meta-class list $<junk> =
     boxV{ boxH { spring bold{"Junk Meta-Classes"} spring }
           ruleH
           head
           tail
         }
meta-class $<meta_class> = boxH{ $<name> spring }
meta-class list $<meta_class> = boxV{ head tail }

/*********************************/
/* Display of Grasyla a statement */
/*********************************/

meta-class $<G-equation> = boxH{ $1:<G-arg>
                                 used($1:<G-has-functor>)
                                 if <?list>=true then  "list"
                                 " = "
                                 $1:<G-rel-expr>
                               }
meta-class $<G-object> = boxH{ arg($1:<G-rel-prop>)  arg($1:<G-rel-class>) }
meta-class arg($<meta_class>) = boxH{ "meta-class " $name }
meta-class arg($<meta_property>) = boxH{ "meta-property " $name }
meta-class list $<G-expr> = boxV{ head tail }

/**********************************/
/* display of a Grasyla expression */
/**********************************/


meta-class $<G-circle> =
    boxV{ boxH{ if <?horizontal>=true then "circleH {" else "circleV {"
                $*:<G-members> "}"
                spring
              }
          boxH{  "{" boxV{ if <stroke-color>="" then ""
                           else boxH{ "stroke-color= "  $<stroke-color> spring }
```

```
                          if <stroke-width>=0 then ""
                          else boxH{ "stroke-width= "  $<stroke-width> spring }
                          if <stroke-style>="" then ""
                          else boxH{ "stroke-style= "  $<stroke-style> spring }
                          if <fill-color>="" then ""
                          else boxH{ "fill-color= "  $<fill-color> spring }
                          if <margin>=0 then ""
                          else boxH{ "margin= "  $<margin> spring }
                          if <handles>=0 then ""
                          else boxH{ "handles= "  $<handles> spring }
                        }
                 "}" spring
             }
       }{ stroke_color=black, stroke_width=1, margin=2pt }
meta-class $<G-box> =
     boxV{ boxH{ if <?horizontal>=true then "boxH {" else "boxV {"
                 $*:<G-members> "}"
                 spring
               }
           boxH{ "{" boxV{ if <stroke-color>="" then ""
                          else boxH{ "stroke-color= "  $<stroke-color> spring }
                          if <stroke-width>=0 then ""
                          else boxH{ "stroke-width= "  $<stroke-width> spring }
                          if <stroke-style>="" then ""
                          else boxH{ "stroke-style= "  $<stroke-style> spring }
                          if <fill-color>="" then ""
                          else boxH{ "fill-color= "  $<fill-color> spring }
                          if <margin>=0 then ""
                          else boxH{ "margin= "  $<margin> spring }
                          if <handles>=0 then ""
                          else boxH{ "handles= "  $<handles> spring }
                        }
                 "}" spring
               }
         } { stroke_color=black, stroke_width=1, margin=2pt }
meta-class $<G-round> =
     boxV{ boxH{ if <?horizontal>=true then "circleH {" else "circleV {"
                 $*:<G-members> "}"
                 spring
               }
           boxH{  "{" boxV{ if <stroke-color>="" then ""
                           else boxH{ "stroke-color= "  $<stroke-color> spring }
                           if <stroke-width>=0 then ""
                           else boxH{ "stroke-width= "  $<stroke-width> spring }
                           if <stroke-style>="" then ""
                           else boxH{ "stroke-style= "  $<stroke-style> spring }
                           if <fill-color>="" then ""
                           else boxH{ "fill-color= "  $<fill-color> spring }
                           if <margin>=0 then ""
                           else boxH{ "margin= "  $<margin> spring }
                           if <handles>=0 then ""
                           else boxH{ "handles= "  $<handles> spring }
                           if <round>=0 then ""
                           else boxH{ "round= "  $<round> spring }
                         }
                  "}" spring
               }
```

```
         } { stroke_color=black, stroke_width=1, margin=2pt }
meta-class $<G-function> = boxH{ used( $1:<G-used-by> ) "(" $name ")" }
meta-class $<G-tail>= "tail"
meta-class $<G-head>= "head"
meta-class $<G-ident>= $name
meta-class $<G-rule>= boxH{ if <?horizontal>=true then "ruleH " else "ruleV " "{"
                            boxV{ boxH{ "stroke-color= "  $<stroke-color> spring }
                                  boxH{ "stroke-width= "  $<stroke-width> spring }
                                  boxH{ "stroke-style= "  $<stroke-style> spring }
                                  boxH{ "fill-style= "  $<fill-style> spring }
                                }
                            "}"
                          }
meta-class $<G-handle> = boxH{ "handle { visible=" $<?visible> ", index=" $<index> "}" }
meta-class $<G-space> = boxH{ if <?horizontal>=true then "H{" else "V{"
                              "dim=" $dim "}"
                            }
meta-class $<G-condition> = boxH{ "if " $<ident> "=" $<value>
                                  " then "  $1:<G-success>
                                  " else " $1:<G-failure>
                                }
meta-class $<G-spring> = bitmap "images/spring.bmp"


meta-class $<G-font> =
     boxV{ boxH{ "font {" $*:<G-members> "}" spring }
           boxH{ spring "{" boxV{ boxH{ "name =" $<name> }
                                  boxH{ "color-font= " $<color-font> }
                                  boxH{ "background= " $<background> }
                                  boxH{ "bold= " $<bold> }
                                  boxH{ "italic= " $<italic> }
                                  boxH{ "underline= " $<underline> }
                                  boxH{ "size= " $<size> }
                                }
                         "}"
               }
          }
meta-class $<G-many-role> = boxH{ "$*:" $1:<G-in-many-role>
                                  if <?restricted> = true
                                  then " restricted"  }
meta-class $<G-one-role> = boxH{ "$1:" $1:<G-in-one-role>
                                 if <?restricted> = true
                                 then " restricted"  }
meta-class $<meta_relation>  = $name
meta-class $<G-bitmap-cst> =
     boxV{ boxH{ spring "factor=" $factor ", handle=" $handle spring }
           boxH{ spring bitmap $ressource spring }
         }
meta-class $<G-bitmap> = boxH{ "bitmap name=" $name " factor=" $factor " handle=" $handle }

/*******************************************************/
/* Display of the functors (that are not used, cfr junk) */
/*******************************************************/

meta-class $<G-functor> = boxH{ "functor <" $name "> is not used!" }
meta-class used($<G-functor>) = $name
```

```
/******************************/
/* Display of the connections */
/******************************/

meta-class $<G-connection> =
     boxH{ "connection"
           $1:<displayed-as>
           $*:<owner-end> boxV{ spring
                                boxH{ spring "----------" spring }
                                boxH{ spring "stroke-color=" $<stroke-color> spring }
                                boxH{ spring "stroke-width=" $<stroke-width> spring }
                              }
           $*:<member-end>
         }
meta-class list $<G-extremity> = boxV{ head tail }
meta-class $<G-inner> = boxH{ bitmap "images/inner.bmp"
                              $<stroke-color> $<stroke-width> $<fill-color> $<size> }
meta-class $<G-outside> = boxH{ bitmap "images/outside.bmp"
                                $<stroke-color> $<stroke-width> $<fill-color> $<size> }
meta-class $<G-cross> = boxH{ bitmap "images/cross.bmp"
                              $<stroke-color> $<stroke-width> $<size> }
meta-class $<G-bar> = boxH{ bitmap "images/cross.bmp"
                            $<stroke-color> $<stroke-width> $<size> }
meta-class $<G-bullet> = boxH{ bitmap "images/cross.bmp"
                               $<stroke-color> $<stroke-width> $<fill-color> }

/*********************/
/* Display of a path */
/*********************/

meta-class $<G-path> = boxH{ "path " $name " = " $*:<made-of> }
meta-class list $<G-path-component> = boxH{ head "->" tail }
meta-class $<G-path-component> = $1:<in-path>
meta-class $<meta_class> = boxH{ "meta-class: " $name }
end
```

D.3.2 The Statechart Editor

```
view <StateChart Editor> ;
begin
root: <State>, <Transition> ;

meta-class $<Simple> =
    roundV{ boxH{ spring font{ $Name }{ bold=true, size=12 } spring }
            ruleH{ stroke_width=3 }
            $*:<play>
          }{ stroke_width=1pt, stroke_color=black, round=10, handles=10 }

meta-class $<Initial> =
    circleH{  H{6pt} V{6pt}
           }{ stroke_color=black, fill_color=black, handles=4 }

meta-class $<Final> =
    circleH{ circleH{ H{6pt} V{6pt}
                    }{ fill_color=black, stroke_width=1pt }
           }{ margin=2pt, stroke_width=1pt, fill_color=white, handles=4  }
```

```
meta-class $<Transition> =
    boxV{ boxH{ spring handle{index=1} spring }
          boxH{ handle{index=1} spring handle{index=2}}
          boxH{ spring $Name $*:<play> spring }
          boxH{ spring handle{index=2} spring }
        }

meta-class list $<Actions> = boxV{ head tail }

meta-class $<Actions> = boxH{ $1:<for-event>
                              $<Args>
                             if <Guard-condition>=""
                               then ""
                               else boxH{ "[" $<Guard-condition> "]" }
                             if <Action-expr>=""
                               then ""
                               else "/"
                             $<Action-expr>
                           }

meta-class $<Event> = $Name

meta-class $<Fork> = boxH{ V{20pt}
                          boxV{ spring handle spring }
                          ruleV{ stroke_width=5pt, stroke_color=black }
                          boxV{ spring handle spring }
                        }

meta-class $<Synchro> = boxH{ V{1cm}
                             boxV{ spring handle spring }
                             ruleH{ stroke_width=3pt, stroke_color=black }
                             boxV{ spring handle spring }
                           }

meta-class $<CompositeState> =
    roundV{ boxH{ spring font{ $Name }{ bold=true, size=12 } spring }
            ruleH{ stroke_width=3 }
            $*:<play>
            // explode (not yet implemented)
          }{ stroke_width=2pt, stroke_color=black, handles=10, round=10 }


meta-relation $<incoming> = connection
      { outside{ size=8, stroke_color=black} }
      { stroke_width=1pt, stroke_color=black }
      { handle{2} }

meta-relation $<outcoming> = connection
      { }
      { stroke_width=1pt, stroke_color=black }
      { handle{1} }
end
```

D.3.3 The Relational Editor

The relational meta-model is presented in Fig. D.5.

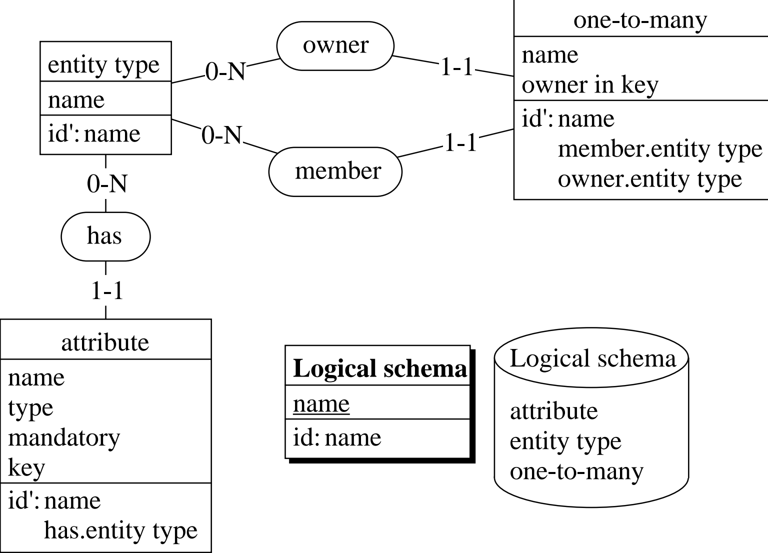


Figure D.5: [**A Pseudo Relational Meta-Model**] Page 139 explains why we consider this as a "pseudo relational meta-model".

The Graphical GraSyLa Script

```
view "Logical" ;
begin

root: $<entity type>, $<one-to-many> ;

meta-class $<entity type> = boxV{ boxH{ spring bold { $<name> }  spring }
                                  ruleH{ stroke_width=2pt }
                                  $*:<has>
                                }{ margin=3pt, stroke_width=1pt,
                                   stroke_color=dark_blue, handles=4 }

meta-class list $<attribute> = boxV{ head tail }
meta-class $<attribute> = boxH{ if key=true then underline{ $<name> }
                                            else $<name>
                                " " $<type>
                                if mandatory=true then "" else "[0-1]"
                                spring
                              }

meta-class $<one-to-many> = boxV{ $<name>
                                  boxH{ spring handle spring }
                                }

meta-relation $<owner> = connect{}{ stroke_width=1pt }{};
meta-relation $<member> = connect{ bullet { size=5, fill_color=dark_green,
                                            stroke_color=dark_green } }
                                 { stroke_width=1pt }
                                 { }
end
```

The Textual (SQL-like) GraSyLa Script

```
view "StateChart Editor" ;
begin
root: $<entity type> ;

meta-class $<entity type> = boxV{ boxH{ "table " $<name> " (" spring }
                                  boxH{ H{6pt} $*:<has> spring }
                                  boxH{ H{6pt} FKdef( $*:<member> )  }
                                  boxH{ H{6pt} "primary key (" PKdef( $*:<has> )
                                               PKdef( $*:<member> ) ")," }
                                  boxH{ H{6pt} FKref( $*:<member> )  }
                                  boxH{ ");" spring }                             }

meta-class list $<attribute> = boxV{ head tail }
meta-class $<attribute> = boxH{ $<name> " " $<type>
                                if mandatory=true then " not null"
                                ","                                  }

/***** PKdef *****/

meta-class list PKdef( $<attribute> ) = boxH{ head  tail }
meta-class PKdef( $<attribute> ) = if key=true then boxH{ $<name> ", " }

meta-class list PKdef( $<one-to-many> ) = boxH{ head tail };
meta-class PKdef( $<one-to-many> ) = if <owner in key>=true then FKref( $1:<owner> )

/***** FKdef *****/

meta-class list FKdef( $<one-to-many> ) = boxV{  head  tail }
meta-class FKdef( $<one-to-many> ) = FKdef( $1:<owner> )

meta-class FKdef( $<entity type> )= FKdef( $*:<has> )

meta-class list FKdef( $<attribute> ) = boxV{ head tail }
meta-class FKdef( $<attribute> ) = if key=true then boxH{ $<name> " " $<type> }

/***** FKref *****/

meta-class list FKref( $<one-to-many> ) = boxV{ head tail };
meta-class FKref( $<one-to-many> ) = boxH{ "foreign key (" FKref($1:<owner>)
                                           ") references " FKrefent($1:<owner>) "," };

meta-class FKref( $<entity type> ) = PKdef( $*:<has> );
meta-class FKrefent( $<entity type> ) = $<name>;

end
```

D.4 Voyager 2^+ Programs

D.4.1 The XML-Generator Program

This file is the main part of the XML generator presented in section 7.11.2 on page 206. The program calls methods of the "meta-class" package that is presented in the next section.

```
explain (*
```

```
This program generates an XML report with
the information stored in the repository.
Author: Vincent Englebert
Date: nov 1999
*)

#include "types-cst.ixi"
#include "io-cst.ixi"
#include "meta_class.ixi"
#include "meta_relation.ixi"

procedure XML_generate_DTD(file: f,meta_class: mc)
  meta_model: mm;
  meta_property: mp;
  meta_relation: mr;
  string: prefix;
  string: s;
{ printf(f,"\n\n<!-- meta-class "+mc."name"+" -->\n\n");
  // print the meta-class entity.
  printf(f, [ "<!ELEMENT ",mc."name"," (\n"]);
  printf(f,"supertypes?");
  prefix:=",\n"+(mc."name")+".";
  for mp in meta_property{@rel_prop:[mc]} do {
    printf(f,prefix+mp."name");
    if mp."multi" then {
      printf(f,"*");
    }
    printf(f,"\n");
  }
  for mr in meta_relation{@rel_owner:[mc]} do {
    printf(f,prefix+(mr."name")+"*");
  }
  mm=mc;
  if IsNoVoid(mm) then {
    printf(f,",\ncomponents");
  }
  printf(f,"\n)>\n ");
  printf(f,["\t<!ATTLIST ", mc."name", " cid ID #REQUIRED>\n"]);
  // print the meta-property entities
  for mp in meta_property{@rel_prop:[mc]} do {
    s:=(mc."name")+"."+(mp."name");
    printf(f,["<!ELEMENT ", s, " EMPTY>\n" ]);
    printf(f,["\t<!ATTLIST ", s, " val CDATA #REQUIRED>\n"]);
  }
  // print the "owner" meta-relation entities
  for mr in meta_relation{@rel_owner:[mc]} do {
    s:=(mc."name")+"."+(mr."name");
    printf(f,["<!ELEMENT ",s," EMPTY>\n"]);
    printf(f,["\t<!ATTLIST ",s," ref IDREF #REQUIRED>\n"]);
  }
  printf(f,"\n");
}

function file ChooseAFileAndOpenIt()
explain (* Display a dialog box and open the file *)
  string: name_file;
{ name_file:=BrowsePrint("XML Saver",
```

```
                         "Select a file or type a new name:",
                         "Text file|*.txt|XML file|*.xml");
  if GetError()=ERR_CANCEL then {
    return Void(file);
  } else {
    return OpenFile(name_file,_W);
  }
}

procedure PrintHeader(file: f)
{ printf(f,"<?xml version='1.0' ?>\n\n");
  printf(f,"<!DOCTYPE repository [\n\n");
  printf(f,"<!-- DTD BEGINS HERE -->\n\n");
  printf(f,"<!ELEMENT xref EMPTY>\n");
  printf(f,"\t<!ATTLIST xref ref IDREF #REQUIRED >\n\n");
  printf(f,"<!ELEMENT components (xref)*>\n\n");
  printf(f,"<!ELEMENT supertypes (xref)*>\n\n");
}

procedure PintSkeleton(file: f)
  meta_class: mc;
  integer: print_comma;
{
  print_comma:=FALSE;
  printf(f,"<!ELEMENT repository ( ");
  for mc in meta_class{TRUE} do {
    if print_comma then {
      printf(f,", ");
    } else {
      print_comma:=TRUE;
    }
    printf(f,(mc."name")+"*");
  }
  printf(f,")>\n\n");
}

procedure PrintMetaClassSkeleton(file: f)
  meta_class: mc;
{ for mc in meta_class{TRUE} do {
    XML_generate_DTD(f,mc);
  }
}


export procedure generate_XML()
explain (*
#menu=&Save project (XML)
#help-line=Save the whole project in a XML file
#help-text=This program download both the DTD and the .. sections
whatever the meta-models in the repository.
#end
*)
  meta_class: mc;
  file: f;
{ f:=ChooseAFileAndOpenIt();
  if IsVoid(f) then {
    MessageBox("Error","Operation Halted!");
```

```
    halt;
  }
  PrintHeader(f);
  PrintSkeleton(f);
  PrintMetaClassSkeleton(f);
  printf(f,"<!-- DTD ENDS HERE -->\n");
  printf(f,"]>\n\n");
  printf(f,"<!-- CLASS DESCRIPTION BEGINS HERE -->\n\n");
  printf(f,"<repository>\n\n");
  for mc in meta_class{TRUE} do {
    mc->XML_generate_instances(f);
  }
  printf(f,"</repository>\n\n");
  printf(f,"<!-- CLASS DESCRIPTION ENDS HERE -->\n\n");
  CloseFile(f);
}

begin
  generate_XML();
end
```

D.4.2 The "meta-class" Package

This file is the "meta-class" package which contains the methods used by the XML-generator. This file is not exhaustive and can of course contain methods that would be outside the scope of the XML generator.

```
package "meta_class";

#include "types-cst.ixi"
#include "io-cst.ixi"

procedure XML_generate(file: f,
                       meta_class: type_instance,
                       type_instance: instance)
  meta_property: mp;
  meta_relation: mr;
  meta_class: super,member_type;
  string: name, prefix;
  meta_model: mm;
{
  name:=type_instance."name";
  // print the cid information
  printf(f,["<",name," cid='x",instance->GetID(),"'>\n"]);
  // print the supertypes information
  printf(f,"<supertypes>\n");
  for super in meta_isa{rel_supertype_of:meta_isa{@rel_subtype_of:[type_instance]}} do {
    let super: superclass
    in {  superclass:=instance;
          if IfNoVoid(superclass) then {
            printf(f,["<xref ref='x",superclass->GetID(),"'>\n"]);
          }
       }
    }
  }
  printf(f,"</supertypes>\n");
  // print the properties information
```

```
  for mp in meta_property{@rel_prop:[type_instance]} do {
    printf(f,["<",name,".",mp."name"," val=\"",instance.mp,"\"/>\n"]);
  }
  // print the relations information
  for mr in meta_relation{@rel_owner:[type_instance]} do {
    prefix:=(type_instance."name")+"."+(mr."name");
    member_type:=GetFirst(meta_class{rel_member:[mr]});
    let member_type: member
    in {  for member in member_type{@mr:[instance]} do {
            printf(f,["<",prefix," ref='x",member->GetID(),"'/>\n"]);
          }
       }
  }
  // print the definition of the specification
  mm:=type_instance;
  if IsNoVoid(mm) then {
    // the instance denotes a specification.
    // and we must generate its "definition"
    attach c to DefinitionOf(mm %% instance);
    printf(f,"<components>\n");
    while IsNoVoid(c) do {
      printf(f,["<xref ref ='x",get(c)->GetID(),"'/>\n"]);
      c>>;
    }
    printf(f,"</components>\n");
  }
  printf(f,["</",name,">\n\n"]);
}

export method procedure XML_generate_instances(file: f)
explain (*
Flush the instances of the 'this' meta-class on
file denoted by 'f'
*)
  this: instance;
{ for instance in this{TRUE} do {
    XML_generate(f,this,instance);
  }
}

/*************************************************/
/* ....... the other methods of this package .....*/
/*************************************************/

begin
end
```

D.4.3 Example

This file represents an output of the XML generator. The specification represented here corresponds to the specification depicted in Fig. 6.17 on page 146. The file has been purged of all the details related to the meta-model level.

```
<?xml version='1.0' ?>

<!DOCTYPE repository [
```

```
<!-- DTD BEGINS HERE -->


<!ELEMENT xref EMPTY>
  <!ATTLIST xref ref IDREF #REQUIRED >

<!ELEMENT components (xref)*>

<!ELEMENT supertypes (xref)*>

<!ELEMENT repository ( entity-type*, attribute*,
                       one-to-many*, Logical-schema*)>



<!-- meta-class entity-type -->

<!ELEMENT entity-type (
supertypes?,
entity-type.name,
entity-type.owner*,
entity-type.member*,
entity-type.has*
)>
  <!ATTLIST entity-type cid ID #REQUIRED>
<!ELEMENT entity-type.name EMPTY>
  <!ATTLIST entity-type.name val CDATA #REQUIRED>
<!ELEMENT entity-type.owner EMPTY>
  <!ATTLIST entity-type.owner ref IDREF #REQUIRED>
<!ELEMENT entity-type.member EMPTY>
  <!ATTLIST entity-type.member ref IDREF #REQUIRED>
<!ELEMENT entity-type.has EMPTY>
  <!ATTLIST entity-type.has ref IDREF #REQUIRED>

<!-- meta-class attribute -->

<!ELEMENT attribute (
supertypes?,
attribute.name,
attribute.type,
attribute.mandatory,
attribute.key
)>
  <!ATTLIST attribute cid ID #REQUIRED>
<!ELEMENT attribute.name EMPTY>
  <!ATTLIST attribute.name val CDATA #REQUIRED>
<!ELEMENT attribute.type EMPTY>
  <!ATTLIST attribute.type val CDATA #REQUIRED>
<!ELEMENT attribute.mandatory EMPTY>
  <!ATTLIST attribute.mandatory val CDATA #REQUIRED>
<!ELEMENT attribute.key EMPTY>
  <!ATTLIST attribute.key val CDATA #REQUIRED>

<!-- meta-class one-to-many -->

<!ELEMENT one-to-many (
supertypes?,
```

```
one-to-many.name,
one-to-many.owner-in-key
)>
  <!ATTLIST one-to-many cid ID #REQUIRED>
<!ELEMENT one-to-many.name EMPTY>
  <!ATTLIST one-to-many.name val CDATA #REQUIRED>
<!ELEMENT one-to-many.owner-in-key EMPTY>
  <!ATTLIST one-to-many.owner-in-key val CDATA #REQUIRED>

<!-- meta-class Logical-schema -->

<!ELEMENT Logical-schema (
supertypes?,
Logical-schema.name,
components
)>
  <!ATTLIST Logical-schema cid ID #REQUIRED>
<!ELEMENT Logical-schema.name EMPTY>
  <!ATTLIST Logical-schema.name val CDATA #REQUIRED>

<!-- DTD ENDS HERE -->
]>

<!-- CLASS DESCRIPTION BEGINS HERE -->

<repository>

<entity-type cid='x1'>
<supertypes>
</supertypes>
<entity-type.name val="customer"/>
<entity-type.owner ref='x13'/>
<entity-type.has ref='x5' />
<entity-type.has ref='x6' />
<entity-type.has ref='x7' />
<entity-type.has ref='x8' />
</entity-type>

<entity-type cid='x2'>
<supertypes>
</supertypes>
<entity-type.name val="order"/>
<entity-type.owner ref='x14'/>
<entity-type.member ref='x13'/>
<entity-type.has ref='x9' />
<entity-type.has ref='x10' />
<entity-type.has ref='x7' />
<entity-type.has ref='x8' />
</entity-type>

<entity-type cid='x3'>
<supertypes>
</supertypes>
<entity-type.name val="product"/>
<entity-type.owner ref='x15'/>
<entity-type.has ref='x11' />
</entity-type>
```

```
<entity-type cid='x4'>
<supertypes>
</supertypes>
<entity-type.name val="line-order"/>
<entity-type.member ref='x14'/>
<entity-type.member ref='x15'/>
<entity-type.has ref='x12' />
</entity-type>

<attribute cid='x5'>
<supertypes>
</supertypes>
<attribute.name val="first-name"/>
<attribute.type val="char(20)"/>
<attribute.mandatory val="true"/>
<attribute.key val="true"/>
</attribute>

<attribute cid='x6'>
<supertypes>
</supertypes>
<attribute.name val="name"/>
<attribute.type val="char(20)"/>
<attribute.mandatory val="true"/>
<attribute.key val="true"/>
</attribute>

<attribute cid='x7'>
<supertypes>
</supertypes>
<attribute.name val="address"/>
<attribute.type val="char(200)"/>
<attribute.mandatory val="true"/>
<attribute.key val="false"/>
</attribute>

<attribute cid='x8'>
<supertypes>
</supertypes>
<attribute.name val="phone"/>
<attribute.type val="num(9)"/>
<attribute.mandatory val="false"/>
<attribute.key val="false"/>
</attribute>

<attribute cid='x9'>
<supertypes>
</supertypes>
<attribute.name val="num-ord"/>
<attribute.type val="char(10)"/>
<attribute.mandatory val="true"/>
<attribute.key val="true"/>
</attribute>

<attribute cid='x10'>
<supertypes>
```

```
</supertypes>
<attribute.name val="date"/>
<attribute.type val="char(8)"/>
<attribute.mandatory val="true"/>
<attribute.key val="false"/>
</attribute>


<attribute cid='x11'>
<supertypes>
</supertypes>
<attribute.name val="num-prod"/>
<attribute.type val="num(10)"/>
<attribute.mandatory val="true"/>
<attribute.key val="true"/>
</attribute>

<attribute cid='x12'>
<supertypes>
</supertypes>
<attribute.name val="quantity"/>
<attribute.type val=""/>
<attribute.mandatory val="true"/>
<attribute.key val="false"/>
</attribute>

<one-to-many cid='x13'>
<supertypes>
</supertypes>
<one-to-many.name val="pass"/>
<one-to-many.owner-in-key val="true"/>
</one-to-many>

<one-to-many cid='x14'>
<supertypes>
</supertypes>
<one-to-many.name val="has"/>
<one-to-many.owner-in-key val="true"/>
</one-to-many>

<one-to-many cid='x15'>
<supertypes>
</supertypes>
<one-to-many.name val="of"/>
<one-to-many.owner-in-key val="true"/>
</one-to-many>

<Logical-schema cid='x16'>
<supertypes>
</supertypes>
<Logical-schema.name val="CUST-COM"/>
<components>
<xref ref='x1'/>
<xref ref='x2'/>
<xref ref='x3'/>
<xref ref='x4'/>
<xref ref='x5'/>
```

```
<xref ref='x6'/>
<xref ref='x7'/>
<xref ref='x8'/>
<xref ref='x9'/>
<xref ref='x10'/>
<xref ref='x11'/>
<xref ref='x12'/>
<xref ref='x13'/>
<xref ref='x14'/>
<xref ref='x15'/>
</components>
</Logical-schema>

</repository>

<!-- CLASS DESCRIPTION ENDS HERE -->
```

Glossary

Abstract Machine A design for a processor that is not meant for implementation but that represents a model for processing an intermediate language, called abstract machine language, used by an interpreter or compiler. Its instruction set can use instructions that more closely resemble the compiled language than the instructions used by an actual computer. It can also be used to make the implementation of the language more portable to other platforms.

Argument An *argument* is a value passed to either a function or a procedure. Operands of operators are sometimes also called *arguments*.

Bootstrap A process is *bootstrapped* when it uses its own components to complete its construction (for instance: a jip crane). Bootstrapping is used by compilers to port them easily on other architectures or to test their correctness.

Class A *class* is a set of objects that share a common structure and a common behaviour [Boo91]. In this document, a class is just an instance of a meta-class. One observed that this concept corresponds to the first definition in most cases in the software engineering realm. This level of abstraction is of course relative and a class could be a synonym of object in some particular cases (i.e., meta-models).

CORBA Common Object Request Broker Architecture. CORBA defines a framework for transparent communication between application objects [Obj97a, Obj99, Obj98, YD, OHE96].

DCOM Distributed Component Object Model. Microsoft's extension of their Component Object Model (COM) to support objects distributed across a network.

Display A *display* is a device to show a view of a specification — for instance, screens (windows), printers and clipboards.

Display Processor The *display processor* is a control process that manages the views of a specification and all the possible interactions with the software engineer.

DTD Document Type Definition (*cfr.* XML).

ERA Entity-Relationship model with Attributes.

Extension The *extension* denotes the instances of an abstract concept. For instance, the extension of a meta-class is the set of its classes, the extension of a relational database is the set of its tables.

Identifier *Identifiers* can have two meanings depending on their context. An identifier can either denote a token in a grammar (that is, a lexical value) or a set of characteristics that identify a concept amongst its family (cfr. page 63).

Meta-CASE Architect The *meta-CASE architect* is the engineer who defined and implemented the meta-CASE tool.

Meta-Class The class of a class; a class whose instances are themselves classes [Boo91].

Meta-Model A *meta-model* is a model of a modelling technique [LL94].

Meta-Property *Meta-properties* are identifiers that attach information to meta-classes. This information can be of several kinds (integer, string, ...), simple or multivalued (sequences). See section 4.3.5 page 45.

Meta-Relation A *meta-relation* denotes a possible link/relation between instances of meta-classes. In this document, meta-relations denote functional binary relations.

Method A *method* is a procedure or a function that is defined in the context of a meta-class. The execution of method is always associated with a class in the extension of the corresponding meta-class.

Method Engineer *Method engineers* define methods (i.e., their meta-models, their semantics and their graphical representation). See also *Software Engineer*.

Methodology An organised, documented set of procedures and guidelines for one or more phases of the software life cycle, such as analysis or design. Many methodologies include a diagramming notation for documenting the results of the procedure; a step-by-step "cookbook" approach for carrying out the procedure; and an objective (ideally quantified) set of criteria for determining whether the results of the procedure are of acceptable quality.

MOF Meta Object Facilities. The main purpose of the OMG MOF is to provide a set of CORBA interfaces that can be used to define and manipulate a set of interoperable meta-models. The MOF is a key building block in the construction of CORBA based distributed development environments. [Obj97b]

OMG Object Management Group (*cfr.* `http://www.omg.org`).

Ontology An *ontology* is an explicit and precise description of concepts and relations that exist in a particual domain such as a given organization, a study field, an application area, etc. There are several levels and kinds of ontologies. The advantage of an ontotology is that we are dealing with concepts and getting rid off several nasty problems usually linked to natural language vocabularies like homonymy, metonymy, polysemy, etc. [Béz98, Kay98]

Parameter A *parameter* denotes a value passed as argument to a function (or a procedure) in its definition.

PCTE Portable Common Tool Environment. [LM93]

Programmer In chapter 7, *programmer* is used as a synonym of method engineer. A programmer is a method engineer who programs.

Property A *property* is the value a meta-property assigns to a class (see section 4.3.12, page 58).

Rational Tree A *rational tree* is a tree where infinite branches can be replaced by a reference to an upper node.

Reflection Generally, reflection is an operation that a program P_0 executes and that turns the program and its current state into a data structure and turns execution over another program P_1. The program P_1 can examine and change P_0, or it can *reflect* and turn execution over to P_2, or it can turn control back to P_0, which is called *reification* [FD98, WF86].

Reification See *reflection*.

Relation **1.** A *relation* is a subset of a Cartesian product ($R \subseteq S_1 \times \cdots \times S_n$).
2. A *relation* can denote a link between several classes. Albeit such relations still verify the mathematical definition (previous item), this definition stresses the compliance with the repository's axioms.

Repository A *repository* is where information (such as objects, table, matrix, relationship definitions, constraint rules, menus, dialogs, graphic representation information, and so on) about a software development method or a specific CASE project developed with specific methods is held. (G. Maokai & L. Scott)

Sequence A *sequence* is an ordered collection of values. For instance $[1, 2, 3]$ and $[3, 1, 2]$ are two distinct sequences. We sometimes abuse of the notation and confuse sequences with sets. The sequence is then used as a set where the order is not taken into account. For instance $s = [3, 1, 2, 1]$ could interpreted as a set when we note $2 \in s$.

SGML Standard Generalized Markup Language.

Software Engineer *Software Engineers* define specifications with respect to some meta-models. See also *Method Engineer*.

Specification A *specification* is a piece of information that describes a point of view on the definition of a program (software). For instance, the ERA schema "Bank More&Money" is a specification.

Subtype A *subtype* denotes a specialization of either a meta-class or a class. Subtypes inherit the meta-properties (resp. the properties) of their supertypes.

Supertype A *supertype* denotes a father of either a meta-class or a class in an inheritance graph. A meta-class (resp. a class) may have 0, 1 or more supertypes.

Virtual Machine A *virtual machine* is an illusion of a real machine. [Dei90]

XMI XML Metadata Interchange (*cfr.* [OMG98]).

XML Extensible Markup Language (*cfr.* [XML98]).

Bibliography

[ACE99] Albert Alderson, J. W. Cartnell, and A. Elloit. Toolbuilder: From CASE tool components to method engineering. In CoSET'99 [CoS99].

[AK91] Hassan Aït-Kaci. *Warren's Abstract Machine: A tutorial reconstruction.* MIT Press, Cambridge, Massachussetts, 1991.

[BBD+89] P. Bergsten, J. Bubenko, R. Dahl, M.R. Gustafsson, and L-A. Johansson. RAMATIC - a CASE shell for implementation of specific CASE tools. Technical report, SISU, Stockholm, Sweden, 1989.

[BCPC+97] Fethi Bounaas, Monique Chabre-Peccoud, Pierre-Yves Cunin, Jean-Pierre Giraudin, Philippe Morat, and Dominique Rieu. Objets et méta-modélisation. In Oussalah and *alii* [O*alii*97], chapter 5, pages 157–204.

[Béz98] Jean Bézivin. Who is afraid of ontologies? In Jean Bézivin, Johannes Ernst, and Woody Pidcock, editors, *OOPSALA'98 Workshop ♯25*, Vancouver, BC, USA, October 1998. Model Engineering, Methods and Tools Integration with CDIF.

[Béz99] Jean Bézivin. UML, le MOF et XMI: L'architecture réflexive de méta-modélisation de l'OMG. Slides of the presentation *Systèmes informatiques de confiance* organised on feb. 1999 by the group *Objectif Zéro Défaut* about *"UML et techniques formelles"*, February 1999.

[Boo91] G. Booch. *Object-Oriented Design With Applications.* Benjamin/Cummings, Redwood City, California, 1991.

[Bra83] Ronald J. Brachman. What IS-A is and isn't: an analysis of taxonomic links in semantic networks. *Computer*, pages 30–36, October 1983.

[BS99] Birgit Bomsdorf and Gerd Szwillus. CMF: A coherent modelling framework for task-based user interface design. In Vanderdonckt and Puerta [VP99], pages 293–304.

[CG99] Sophie Cluet and Sophie Gamerman. OQL: le solide challenger 100% objet. *Programmez*, 1(6):14–23, January 1999.

[Cod90] E.F. Codd. *The Relational Model for Database Management version 2.* Addison-Wesley, 1990.

[Con99] MetaCase Consulting. MetaEdit+ method workbench 3.0 simple user evaluation version, 1999.

[CoS99] *First International Symposium on Constructing Software Engineering Tools (CoSET'99)*, Los Angeles, USA, May 1999.

[Dah97] Ajantha Dahanayake. *An Environment to Support Flexible Information Modeling*. PhD thesis, Delft University of Technology (Netherlands), 1997.

[Dat95] C.J. Date. *An introduction to database systems*. Addison-Wesley, sixth edition, 1995.

[Dei90] Harvey M. Deitel. *Operating Systems*. Addison-Wesley, second edition edition, 1990.

[DK95] D. Däberitz and U Kelter. Rapid prototyping of graphical editors in an open SDE. In *Proc. Of the 7th Conference on Software Engineering Environments (SEE'95)*, pages 61–72. IEEE Computer Society Press, 1995.

[EC94] Jürgen Ebert and Martin Carstensen. *Ansatz und architektur von KOGGE*. Fachberichte informatik, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1, D-56075 Koblenz, version 1.0 (DRAFT) edition, June 1994.

[EG93] H. Elshazly and V. Gover. A study on the evaluation of CASE technology. *Journal of Information Technology Management*, 4(1), 1993.

[EH99a] Vincent Englebert and Jean-Luc Hainaut. DB-MAIN: A next generation meta-CASE. *Information Systems*, 24(2):99–112, June 1999. Special issue on meta-modelling and methodology engineering.

[EH99b] Vincent Englebert and Jean-Luc Hainaut. GRASYLA: Modelling CASE tool GUIs in Meta-CASEs. In Vanderdonckt and Puerta [VP99].

[EJ91] S. Eherer and M. Jarke. Knowledge-bases support for hypertext co-authoring. In *2nd International Conference on Database and Expert Systems Applications (DEXA'91)*, pages 465–470, Berlin, Germany, August 1991.

[Ele94a] Electronic Industries Association. CDIF Technical Committee. *CDIF CASE Interchange Format - Overview*, eia/is-106 edition, January 1994.

[Ele94b] Electronic Industries Association. CDIF Technical Committee. *CDIF Framework for Modeling and Extensibility*, eia/is-107 edition, January 1994.

[Ele94c] Electronic Industries Association. CDIF Technical Committee. *CDIF Transfer Format Syntax* `SYNTAX.1`, eia/is-109 edition, January 1994.

[Ele94d] Electronic Industries Association. CDIF Technical Committee. *CDIF Transfert Format General Rules for Syntaxes and Encodings*, eia/is-108 edition, January 1994.

[Ele95a] Electronic Industries Association. CDIF Technical Committee. *CDIF Integrated Meta-model Common Subject Area*, eia/is-112 edition, December 1995.

[Ele95b] Electronic Industries Association. CDIF Technical Committee. *CDIF Integrated Meta-model Data Flow Model Subject Area*, eia/is-115 edition, December 1995.

[Ele96a] Electronic Industries Association. CDIF Technical Committee. *CDIF Integrated Meta-model Data Modeling Subject Area*, eia/is-114 edition, December 1996.

[Ele96b] Electronic Industries Association. CDIF Technical Committee. *CDIF Integrated Meta-model Presentation Location and Connectivity Subject Area*, eia/is-118 edition, December 1996.

[Ele96c] Electronic Industries Association. Engineering Department. *CDIF – Integrated Meta-model – Object-oriented Analysis and Design Core Subject Area*, cdif-draft-ooad-v01 edition, July 1996.

[Ele97] Electronic Industries Association. CDIF Technical Committee. *CDIF – Transfer Format – OMG IDL Bindings*, eia/is-734 edition, August 1997.

[Eng99] Vincent Englebert. *Voyager 2 Version 5*. University of Namur - DB-MAIN, Rue grandgagnage 21. 5000 Namur. Belgium, dec 1999.

[Ern97] Johannes Ernst. Introduction to CDIF. `http://www.cdif.org`, May 1997.

[ES97] Jürgen Ebert and Roger Süttenbach. An OMT metamodel. Fachberichte Informatik 13/97, Universität Koblenz-Landau, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1, D-56075 Koblenz, 1997.

[ESU97] Jürgen Ebert, Roger Süttenbach, and Ingar Uhe. Meta-CASE in practice: a case for KOGGE. In Olivé and Pastor [OP97], pages 203–216.

[ESU98] Jürgen Ebert, Roger Süttenbach, and Ingar Uhe. Meta-CASE worldwide. Technical Report 24/98, Fachberichte informatik, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1, D-56075 Koblenz, 1998.

[EWD+96] J. Ebert, A. Winter, P. Dahm, A. Franzke, and R. Süttenbach. Graph based modeling and implementation with EER/GRAL. Fachberichte Informatik 11/96, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1, D-56075 Koblenz, 1996.

[FD98] Ira R. Forman and Scott H. Danforth. *Putting Metaclasses to Work: A New Dimension in Object-Oriented Programming*. Addison-Wesley, 1998.

[Fin92a] Piotr Findeisen. The EARA/GE model for Metaview. Technical report, University of Alberta, November 1992.

[Fin92b] Piotr Findeisen. The graphical extension of EARA model. Technical report, University of Alberta, September 1992.

[Fin93a] Piotr Findeisen. The Metaview software. Technical report, University of Alberta, March 1993.

[Fin93b] Piotr Findeisen. MGED reference manual – release 2.0. Technical report, University of Alberta, May 1993.

[Fin94a] Piotr Findeisen. A complete definition of Data Flow diagram environment for Metaview. Technical report, University of Alberta, July 1994.

[Fin94b] Piotr Findeisen. *The EARA model for Metaview. A reference.* University of Alberta, June 1994.

[Fin94c] Piotr Findeisen. Environment Constraint Language for MetaView. Technical report, University of Alberta, May 1994.

[Fin94d] Piotr Findeisen. *Project Daemon – reference manual.* Computing Science Department. University of Alberta, version 2.3 edition, August 1994.

[FPD+99] R. I. Ferguson, N. F. Parrington, P. Dunne, J. M. Archibald, and J. B. Thompson. MetaMOOSE - an object-oriented framework for the construction of CASE tool. In CoSET'99 [CoS99].

[Fra97] Angelika Franske. GRAL 2.0 a reference manual. Technical Report 3/97, Fachberichte informatik, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1, D-56075 Koblenz, 1997.

[FTS95] Garry Froehlich, Jean-Paul Tremblay, and Paul Sorenson. Providing support for process model enaction in the Metaview metasystem. In Müller and Norman [MN95].

[GFS+94] Dinesh Gadwal, Piotr S. Findeisen, Paul G. Sorenson, J. Paul Tremblay, and L. Beth Millar. *Generating customizable software specification environment using Metaview.* University of Saskatchewan and University of Alberta, March 1994.

[GH93] John C. Grundy and John G. Hosking. Constructing multi-view editing environments using MViews. In *IEEE Symposium on Visual Languages*, pages 220–224, Bergen, Norway, 1993. IEEE CS Press.

[GLM94] Dinesh Gadwal, Pius Lo, and Beth Millar. EDL/GE users's manual. Technical report, University of Alberta and University of Saskatchewan, 1994.

[Gro] DB-MAIN Research Group. *DB-MAIN: Documentation.*

[Gro99] Object Management Group. OMG unified modeling language specification (draft). draft Version 1.3 alpha R2, OMG, January 1999.

[Hai91] J.-L. Hainaut. Entity-generating schema transformations for entity-relationship models. In 10th *Entity-Relationship Conference*, San Mateo, 1991. North-Holland.

[HEH+96] Jean-Luc Hainaut, Vincent Englebert, Jean Henrard, Jean-Marc Hick, and Didier Roland. Database reverse engineering : from requirement to CARE tools. *Journal of Automated Software Engineering*, 3(2), 1996.

[Hon99a] Honeywell. *DOME. Alter Programmer's Reference Manual.* Honeywell, 1999. Version 5.2.1.

[Hon99b] Honeywell. *DOME Extensions Manual.* Honeywell, 1999. Version 5.2.1.

[Hon99c] Honeywell. *DOME Guide*, 1999. Version 5.2.1.

[HSB98] B. Henderson-Sellers and A. Bulthuis. *Object-Oriented Metamethods*. Springer-Verlag, 1998.

[HSTE95] Colin Hardy, Simon Stobart, Barrie Thompson, and Helen Edwards. A comparison of the results of two surveys on software development and the role of CASE in the UK. In Müller and Norman [MN95].

[IPS98] IPSYS. Toolbuilder, the MetaCASE tool. `http://www.ipsys.com`, 1998.

[JGJ+95] M. Jarke, R. Gallersdorfer, M.A. Jeusfeld, M. Staudt, and S. Eherer. ConceptBase – a deductive object base for meta data management. *Journal of Intelligent Information Systems, Special Issue on Deductive and Object-Oriented Databases*, 4(2):167–192, March 1995.

[JH98] Stan Jarzabek and Riri Huang. The case for user-centered CASE tools. *Communications of the ACM*, 41(8):93–99, August 1998.

[JJNS98] Manfred A. Jeusfeld, Matthias Jarke, Hans W. Nissen, and Martin Staudt. ConceptBase: Managing conceptual models about information systems. In P. Bernus, K. Mertins, and G. Schmidt, editors, *Handbook on Architectures of Information Systems*, Germany, 1998. Springer-Verlag.

[JJQ98] Matthias Jarke, A. Jeusfeld, Manfred, and Christoph Quix. *ConceptBase V5.0 User Manual*. RWTH Aachen, Germany, April 1998.

[Joh99] Peter Johnson. Theory based design: From individual users and tasks to collaborative systems. In Vanderdonckt and Puerta [VP99], pages 21–32.

[Kai97] Janne Kaipala. Augmenting CASE tools with hypertext: desired functionality and implementation issues. In Olivé and Pastor [OP97].

[Kay98] D. Kayser. Ontologically yours. In M.L. Mugnier and M. Chein, editors, *ICCS'98*, number 1453 in LNAI, Montpellier, France, 1998. Springer-Verlag. invited talk.

[KD98] Nabil N. Kamel and Robert M. Davison. Applying CSCW technology to overcome traditional barriers in group interactions. *Information & Management*, 34:209–219, 1998.

[Kel97] Steven Kelly. *Towards a Comprehensive MetaCASE and CAME Environment: Conceptual, Architectural, Functional and Usability Advances in MetaEdit+*. PhD thesis, University of Jyväskylä, 1997. ISBN 951-39-0118-1.

[KLR96] S. Kelly, K. Lyytinen, and M. Rossi. MetaEdit+: A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment. In P. Constantopoulos, J. Mylopoulos, and Y. Vassiliou, editors, *Proceedings of the* 8th *International Conference CAiSE'96 on Advanced Information Systems Engineering*, volume 1080 of *LNCS*, pages 1–21, Heraklion, Crete, Greece, May 1996. Springer-Verlag.

[LC98] Diane Lending and Norman L. Chevarny. The use of CASE tools. In *SIGCPR'98. Proceedings of the 1998 conference on Computer personnel research*, pages 49–58, 1998.

[Lin98] Lincoln Software — what is “MetaCASE”. `http://www.ipsys.com`, 1998.

[LKNF95] U. Leonhardt, J. Kramer, B. Nuseibeh, and A. Finkelstein. Decentralised Process Enactment in a Multi-Perspective Development Environment. In *Proceedings of the 17th International Conference on Software Engineering*, pages 255–264, April 1995.

[LL94] V. Lalioti and P. Loucopoulos. Visualisation of conceptual specifications. *Information Systems*, 19(3):291–309, April 1994.

[LM93] Fred Long and Ed Morris. An overview of PCTE: A basis for a Portable Common Tool Environment. Technical Report CMU/SEI-93-TR-1, Software Engineering Institute. Carnegie Mellon University, Pittsburgh, Pennsylvania 15213, March 1993.

[LMT+98] K. Lyytinen, P. Martiin, J.-P. Tolvanen, M. Jarke, K. Pohl, and K. Weidenhaupt. CASE environment adaptability: Bridging the islands of automation. In *Eight annual Workshop on Information Technologies and Systems (WITS'98)*, 1998.

[Mar95a] M.P. Martin. The case against CASE. *Journal of Systems management*, 46:54–57, Jan/Feb 1995.

[Mar95b] Pentti Marttiin. Towards flexible process support with a CASE shell. In Gerard Wijers, Sjaak Brinkkemper, and Anthony I. Wasserman, editors, *Advanced Information Systems Engineering*, number 811 in LNCS, Utrecht, The Netherlands, June 1995. CaiSE'94, Springer-Verlag.

[Mar98] P. Marttiin. Customizable process modeling support and tools for design environments. Technical report, Department of Computer Science and Information Systems. University of Jyväskylä, 1998.

[McI95] D.W. McIntyre. *Design and Implementation with Vampire*, chapter 7, pages 129–159. Prentice-Hall, 1995.

[McL89] Carma McLure. *CASE is software automation*. Prentice-Hall, 1989.

[MD97] Jon Meyer and Troy Downing. *JAVA Virtual Machine*. O'Reilly, 1997.

[Met96] MetaCase Consulting. Developping new methods with the MetaEdit personal environment, February 1996. white paper.

[Met99] MetaCase Consulting. *MetaEdit+ 3 Method Workbench Users Guide. Version 3.0*. Ylistönmäentie 31. FIN-40500 Jyväskylä. Finland, 1999.

[MHR96] P. Marttiin, F. Harmsen, and M. Rossi. A functional framework for evaluating method engineering environments: The case of Maestro II/Decamerone and MetaEdit+. In Sjaak Brinkkemper, Kalle Lyytinen, and Richard J. Welke, editors, *Proceedings of the IFIP TC8, WG8.1/8.2, Working Conference on Method Engineering: Principles of Method Construction and Tool Support*, Atlanta, August 26-28 1996. Georgia, Chapman & Hall.

[MI99] Jari Maansaari and Juhani Iivari. The evolution of CASE usage in Finland between 1993 and 1996. *Information & Management*, 36:37–53, 1999.

[MN95] Hausi A. Müller and Ronald J. Norman, editors. 7th *International Workshop Computer-Aided Software Engineering (CASE'95)*, Toronto, Ontario, Canada, July 1995. IEEE Computer Society Press.

[MO97] Martine Magnan and Chabane Oussalah. Objets et composition. In Oussalah and *alii* [O*alii*97], chapter 2, pages 61–92.

[MRTL93] P. Marttiin, M. Rossi, V.P. Tahvanainen, and K. Lyytinen. A comparative review of CASE-shells - a preliminary framework and research outcomes. *Information & Management*, 25:11–31, 1993.

[Myl92] J. Mylopoulos. Conceptual modeling and telos. In P. Loucopoulos and R. Zicari, editors, *Conceptual Modeling, Databases, and CASE. An Integrated View of Information Systems Development*, chapter 2, pages 49–68. Wiley Professional Computing, 1992.

[NSC+91] Ronald J. Norman, Wayne Stevens, Elliot J. Chikofsky, John Jenkins, Burt L. Rubenstein, and Gene Forte. CASE at the start of the 1990's. In *ICSE '91. Proceedings of the* 13th *international conference on Software engineering*, pages 128–139, Burlington, MA, USA, 1991. International Workshop on CASE.

[NTH89] C.R. Necco, N.W. Tsai, and K.W. Holgeson. Current usage of CASE software. *Journal of Systems Management*, pages 6–11, May 1989.

[O*alii*97] Chabane Oussalah and *alii*, editors. *Ingénierie objet: Concepts et techniques.* InterEditions, Masson, 1997.

[Obj97a] Object Management Group. *A Discussion of the Object Management Architecture*, jan 1997.

[Obj97b] Object Management Group Inc. *Meta Object Facility (MOF) Specification*, September 1997. OMG Document ad/97-08-14.

[Obj98] Object Management Group. *CORBAservices: Common Object Services Specification*, dec 1998.

[Obj99] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, oct 1999.

[OHE96] Robert Orfali, Dan Harkey, and Jeri Edwards. *The Essential Distributed Objects Survival Guide.* Wiley Professional Computing, 1996.

[OMG98] OMG. XML metadata interchange (XMI). Technical Report OMG Document ad/98-07-01, Object Management Group, July 1998. Proposal to the OMG OA&DTF RFP 3: Stream-based Model Interchange Format (SMIF).

[OP97] A. Olivé and J. A. Pastor, editors. *Advanced Information Systems Engineering.* 9th *International Conference CAiSE'97*, number 1250 in LNCS, Barcelona, Catalonia, Spain, jun 1997. Springer-Verlag.

[Par] Parallax Software Technologies. *GraphTalk & LEdit. Guide d'intégration.* Parallax – Software Technologies.

[Par94] Parallax Software Technologies. *GraphTalk 2.5 Refrence Manual*, 1994.

[Pid96] Woody Pidcock. *CDIF – Intergrated Meta-Model – Business Process Modeling Subject Area.* CASE Data Interchange Format Technical Committee, The BOEING Company, draft version edition, March 1996.

[Pro94] Protosoft. *Paradigm+/Cadre Edition Reference Manual.* Protosoft, Providence, 1994.

[Rana] Rank Xerox. *GraphTalk Métamodélisation. Interface de programmation. Version 2.4.* Rank Xerox France, Direction Informatique Avancée et Génie Logiciel. 7 rue Touzet Gaillard. 93586 Saint-Ouen CEDEX.

[Ranb] Rank Xerox. *GraphTalk Métamodélisation. Manuel de référence. Version 2.4.* Rank Xerox France, Direction Informatique Avancée et Génie Logiciel. 7 rue Touzet Gaillard. 93586 Saint-Ouen CEDEX.

[Ranc] Rank Xerox. *GraphTalk Modélisation. Guide utilisateur. Version 2.3.* Rank Xerox France, Direction Informatique Avancée et Génie Logiciel. 7 rue Touzet Gaillard. 93586 Saint-Ouen CEDEX.

[Rand] Rank Xerox. *GraphTalk Modélisation. Manuel d'installation et tutorial. Version 2.3.* Rank Xerox France, Direction Informatique Avancée et Génie Logiciel. 7 rue Touzet Gaillard. 93586 Saint-Ouen CEDEX.

[Ran91] Rank Xerox. *GraphTalk Environnement objets de développement et d'utilisation d'atelier de génie logiciel.* Rank Xerox France, Direction Informatique Avancée et Génie Logiciel. 7 rue Touzet Gaillard. 93586 Saint-Ouen CEDEX, 1991.

[Ran92] Rank Xerox. *GraphTalk Métamodélisation. Tutorial. Version 2.4.* Rank Xerox France, Direction Informatique Avancée et Génie Logiciel. 7 rue Touzet Gaillard. 93586 Saint-Ouen CEDEX, January 1992.

[Rat] Rational. Rational Rose 4.0.

[Rat97a] Rational Software Corporation. *UML Notation Guide. Version 1.1*, September 1997.

[Rat97b] Rational Software Corporation. *UML Semantics. Version 1.1*, September 1997.

[RH97] D. Roland and J-L. Hainaut. Database engineering process modeling. In *International Conference on The Many Facets of Process Engineering*, Gammarth, Tunis, 1997.

[RK99] Matti Rossi and Steven Kelly. Construction of a CASE tool: The case for MetaEdit+. In CoSET'99 [CoS99].

[SE97] Roger Süttenbach and Jürgen Ebert. A Booch Metamodel. Fachberichte Informatik 5/97, Universität Koblenz-Landau, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1, D-56075 Koblenz, 1997.

[SFT96] Paul G. Sorenson, Piotr S. Findeisen, and Jean Paul Tremblay. Supporting viewpoints in Metaview. In *Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints'96) on SIGSOFT'96*, pages 237–241, San Francisco, CA, USA, 1996.

[SLN93] P. Szekely, Ping Luo, and Robert Neches. Beyond interface builders: Model-based interface tools. In *Bridges Between Worlds: Human Factors in Computing Systems (INTERCHI'93)*, 1993.

[Spi92] J.M. Spivey. *The Z Notation: A Reference Manual*. Internation Series in Computer Sciences. Prentice-Hall, second edition, 1992.

[SR96] Julian Smart and Robert Rae. *Hardy. User Guide*. Artificial Intelligence Applications Institute, 80 South Bridge, Edinburgh EH1 1HN. UK, February 1996.

[SR98] R.E. Kurt Stirewalt and Spencer Rugaber. Automating UI generation by model composition. In *13th Conference on Automated Software Engineering (ASE'98)*, Honolulu, Hawaii, oct 1998. IEEE Computer Society Press.

[STM88] Paul G. Sorenson, Jean-Paul Tremblay, and A. J. McAllister. The Metaview system for many specification environments. *IEEE Software*, 5(2):30–38, March 1988.

[STM91] Paul G. Sorenson, Jean-Paul Tremblay, and A. J. McAllister. The EARA/GI model for software specification environments. Technical Report TR 91-14, University of Alberta, June 1991.

[Str97] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, third edition, 1997.

[Sum92] M. Sumner. Making the transition to computer-assisted software engineering. In A.L. Lederer, editor, *1992 ACM SIGCPR Conference*, pages 81–92, New York, 1992. ACM Press.

[SW95] J.A. Senn and J.L. Wynekoop. The other side of CASE implementation: Best practices for success. *Information Systems Management*, 12:7–14, 1995.

[Sze96] Pedro Szekely. Retrospective and challenges for model-based interface development. In F. Bodart and J. Vanderdonckt, editors, *Proc. Of the Eurographics Workshop (Design, Specification and Verification of Interactive Systems '96)*, Namur, Belgium, June 1996. Springer-Verlag.

[Tea98] ConceptBase Team. *ConceptBase Tutorial*, April 1998.

[Ten81] R.D. Tennent. *Principles of Programming Languages*. International Series in Computer Science. Prentice-Hall, 1981.

[tHV96] A.H.M. ter Hofstede and T.F. Verhoef. Meta-CASE: Is the game worth the candle? *Information Systems Journal*, 6(1):41–68, 1996.

[Tou86] David S. Touretzky. *The Mathematics of Inheritance Systems.* Morgan Kaufmann, Los Altos, California, 1986. PhD Thesis, Carnegie-Mellon University.

[UES98] Ingar Uhe, Jürgen Ebert, and Roger Süttenbach. Meta-CASE Worldwide. Technical Report 24/98, Universität Koblenz-Landau, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1, D-56075 Koblenz, 1998.

[Uni96a] Unisys. *Universal Repository. Capabilities Overview for ISVs. Embedding the Unisys Object Repository in Resale Commercial Solutions.* Unisys Corp., 1996.

[Uni96b] Unisys. *Universal Repository. Technical Overview. Release 1.2.* Unisys Corp., August 1996.

[VP99] J. Vanderdonckt and A. Puerta, editors. *Proceedings of the 3rd International Conference on Computer-Aided Design of User Interfaces (CADUI'99)*, Dordrecht, October 1999. Louvain-la-Neuve, Kluwer.

[WCH87] M.E. Winston, R. Chaffin, and D.J. Hermann. A taxonomy of part-whole relations. *Cognitive Science*, 11:417–444, 1987.

[WF86] M. Wand and D. Friedman. The mystery of the tower revealed. In *Proc. Of the 1986 LISP and Functional Programming Conference*, 1986.

[WL95] X. Wang and P. Loucopoulos. The development of Phedias: a CASE shell. In Müller and Norman [MN95].

[WM94] R. Wilhelm and D. Maurer. *Les Compilateurs. Théorie, Construction, Génération.* Masson, 1994.

[XML98] Extensible markup language (XML) 1.0, February 1998. http://www.w3.org/TR/1998/REC-xml-19980210.

[Yan99] Heng-Li Yang. Adoption and implementation of CASE tools in Taiwan. *Information & Management*, 35:89–112, 1999.

[YD] Zhonghua Yang and Keith Duddy. CORBA: A platform for distributed object computing (a state-of-the-art report on OMG/CORBA). Technical report, CRC for Distributed Systems Technology (DSTC).

[ZFS95] Yong Zhuang, Piotr Findeisen, and Paul G. Sorenson. Object-oriented modeling in Metaview. Technical report, University of Alberta, December 1995.

[ZK95] Esteban Zimányi and Manuel Kolp. Using Prolog to implement a CASE shell for object-oriented development. In 8th *Symposium and Exhibition on Industrial Application of Prolog, INAP'95*, pages 41–48, Tokyo, Japan, October 1995.

[ZK96] Esteban Zimányi and Manuel Kolp. A Prolog-based architecture for an object-oriented CASE shell. In 4th *International Conference on Practical Application of Prolog, PAP'96*, pages 497–520, London, UK, April 1996.

Index

Symbols

$\Re$, 61

$\Re^{\star}$, 62

$\Re^{-1}$, 61

Ω, *see* meta-model

$\downarrow$, *see* type casting

$\ll$, *see* relation

$\lll$, *see* relation

$\lll_S$, 67

ω, *see* specification

ρ, 44

$\Gamma(\ldots)$, 258

ϱ, 54

ξ-valuation, 58

$\xi^{\star}$, **60**

$\Re_R^{\star\,-1}$, 62

DV_T, 59

Θ , *see* elementary type

ValUniv, *see* domain

A

abstract machine, 277

argument, 277

B

boolean, *see* type

bootstrap, 277

bootstrapping, *see* reflection

C

CASE tool

  Horizontal, 4

  Lower-CASE, 4

  Upper-CASE, 4

  Vertical, 4

CDIF, 15

char, *see* type

character, *see* type

cid, 50, 258, 259

class, **50**, 277

  inheritance, **54**

Computer Sciences Corporation, 8

ConceptBase, 7, 111

cooperative work, 20

CORBA, 29, 108, 277

cut&paste, *see* operation

D

DB-MAIN, 4, 33, 38, 45, 111, 112, 154, 182, 184

DCOM, 108, 277

deductive database, 7

default value, 59

DeleteClass, *see* operation

deletion, *see* operation

display, 113, 277

display processor, 277

display-processor, 113

document, *see* type

DTD, 206, 277

E

EARA/GE, 13

EER/GRAL, 8

EIA, 15

elementary type, **45**

ERA, 278

extension, 278

G

global, *see* identifier

GQL, *see* GraphTalk

GRAL, 10

GraphTalk, 8

  GQL, 8

  Ledit, 8

grasyla, **111**

  constructor

    `bitmap`, 129

    `boxH`, 123

circleH, 123
boxV, 123
circleV, 124
explode, 129
font, 131
H, 128
handle, 128
if-then-else, 125
roundH, 124
roundV, 125
ruleH, 130
ruleV, 131
spring, 127
V, 128
directive section, 115
functor, 122
main section, 117
the GraSyLa language, **111**
variable
head, 122
tail, 122
meta-property, 118
meta-relation, 119

H
Horizontal CASE tool, *see* CASE tool
hypertext
GraSyLa , 140

I
identifier, 38, **63**, *see* cid, 278
idval, 64
global, 63
graphical representation, 66
id value, 64
local, 63
primary, 66
secondary, 66
well-formed, 64
inheritance
class, **54**
graph, 84
meta-class, **43**
integer, *see* type

K
KOGGE, 8
urKOGGE, 10
Kogge, 6

L
Ledit, *see* GraphTalk
literate programming, 201
local, *see* identifier
Lower-CASE tool, *see* CASE tool

M
Mod, 39
Meta Object Facilities, *see* MOF
meta-CASE architect, 85, 103, 136, 209, 278
meta-class, **37**, 278
identifier, 38
inheritance, **43**
meta-model, 39
meta-model, **38**, 278
Ω, **40**
meta-property, **45**, 278
meta-relation, **48**, 278
mandatory, 49
member, 49
name, 48
owner, 49
uniqueness, 49
MetaEdit+, 6
MetaView, 6, 13
method, **47**, 85, 278
CanCreate(), 47
CanDelete(), 48
Constructor(), 47
Delete(), 48
OnCreate(), 47
ToDelete(), 48
method engineer, 278
methodology, 278
methods
predefined, 47–48
MFC, 28
mirroring, **95**
MOF, 15, 108, 278
Motif, 28
mutation, 75

O
O-Telos, 7
ODBC, 29

OMG, 15, 206, 278
ontology, 279
operation
  cut&paste, 72
  `DeleteClass`, 67
  deletion, 67
  remember&paste, 73
  specialize, 75
OQL, 59

P
Paradigm+, 4
Parallax, 8
parameter, 279
PCTE, 111, 279
Phedias, 6
picture, *see* type, 129
process modelling, 20
programmer, 279
property, 58, 85, 279
pumping, 120

R
Rank Xerox, 8
rational tree, 279
real, *see* type
reflection, **104**, 279
  bootstrapping, 108
reification, 279
relation, 61, 84, 279
  $\ll$, 40
  $\lll$, 53
  ρ, 44
  ϱ, 54
remember&paste, *see* operation
repository, 33, 279

S
sequence, 279
SGML, 206, 279
software engineer, 280
sound, *see* type
specialization, 75
specialize, *see* operation
specification, **51**, 280
  ω, 52
SQL, 59
string, *see* type
subtype, 280
subtypes
  function, 67
subtypes*
  function, 67
supertype, 280

T
Tcl/Tk, 28
Telos, 7
TGraph, 10
ToolBuilder, 6
Toolframe, 4
transaction, 20
type, *see* Voyager 2
  boolean, 45
  char, 45
  character, 45
  document, 45, 119
  integer, 45
  picture, 45, 119
  real, 45
  sound, 45, 119
  string, 45
  video, 45, 119
type casting, 55

U
Universal Repository, 15, 111
Upper-CASE tool, *see* CASE tool
UREP, *see* Universal Repository, 108
urKOGGE, *see* KOGGE
user interface, 28

V
Value Universe, **45**
versioning, 20
Vertical CASE tool, *see* CASE tool
video, *see* type
view
  compact, 33
  extended, 33
virtual machine, 280
Voyager 1
  memory, 220
    data, **220**
    `DATA` , 220
    environnement, 220

PROG , 221
relative address, **220**
ENV , 220
P-Counter, **221**
Voyager
process, 221
program, 221
Voyager 2
+, 245
\, 160
Γ(. . .), 258
'\n', 160
'\t', 160
*, 166
+, 166
-, 166
/, 166
/*...*/, 155
//, 155
:=, 168
_A, 248
_R, 248
_W, 248
_MetaProperty::CanUpdate , **192**
_MetaRelation::CanInsert , **191**
_MetaRelation::CanRemove , **192**
eClass::AddFirstRelation , **85**
eClass::AddNextRelation , **85**
eClass::GetFirstDef , **87**
eClass::GetFirstRelation , **85**
eClass::GetNextDef , **87**
eClass::GetNextRelation , **85**
eClass::GetOwnerRelation , **85**
eClass::GetProperty , **87**
eClass::GetSubtype , **86**
eClass::GetSupertype , **86**
eClass::RemoveRelation , **86**
eClass::UpdateISA , **86**
eClass::UpdateProperty , **87**
eSpecification::AddComponent , **87**
eSpecification::GetFirstComponent , **87**
eSpecification::GetNextComponent , **87**
** , **168**
++ , **167**
AddFirst , **247**
AddLast , **247**
and, 166
AscToChar , **243**
assignment, **168**
associativity, **164**
backslash, 160
\, 160
block declaration, **163**
BrowsePrint , **254**
BrowseRead , **254**
call, 199
call , **199**
CallSystem , **258**
char, 159
character, **159**
CharIsAlpha , **243**
CharIsAlphaNum , **243**
CharIsDigit , **243**
CharToAsc , **244**
CharToLower , **244**
CharToStr , **244**
CharToUpper , **244**
Choice , **254**
cid, 258, 259
ClearScreen , **258**
CloseFile , **248**
COMMA, 248, 249
comment, **155**
constant, **156, 157**
character, 159
integer, **159**
list, **160**
string, **159**
constants
_A, 162
_R, 162
_W, 162
constraint, 182
create , **187**
cursor, **161**
Cut&Paste , **72**
delete , **190**
DeleteFile , **249**
DialogBox , **255**
directive
include, **201**
use, **198, 199**
dynamic call, 200
Environment, **196**

eof , **249**
ERR_CALL, 243
ERR_CALL, 258
ERR_CANCEL, 243
ERR_CANCEL, 254, 255
ERR_DIV_BY_ZERO, 243
ERR_DIV_BY_ZERO, 166
ERR_ERROR, 243
ERR_ERROR, 250
ERR_FILE_CLOSE, 243
ERR_FILE_CLOSE, 248
ERR_FILE_OPEN, 243
ERR_FILE_OPEN, 248
ERR_NO_UPDATE, 243
ERR_NO_UPDATE, 169
ERR_PATH_NOT_FOUND, 243
ERR_PATH_NOT_FOUND, 250
ERR_PERMISSION_DENIED, 243
ERR_PERMISSION_DENIED, 250
error
  register, 258
ExistFile , **250**
expression, **164**
expression operators, 155
fct-name , **xix**
file, **162**
foreign, **195**
function, **175**, 200
  foreign, *see* foreign
garbage collector, 177
GetChar , **252**
GetDay , **255**
GetError , **258**
GetFirst , **247**
GetFlag , **257**
GetHour , **255**
GetID , **258**
GetLambda , **200**
GetLast , **247**
GetMin , **255**
GetMonth , **257**
GetObject , **258**
GetOxoPath , **258**
GetSec , **257**
GetSpecification , **258**
GetTokenUntil , **251**
GetTokenWhile , **251**
GetType , **258**
GetWeekDay , **257**
GetYear , **257**
GetYearDay , **257**
identifier, **155**
image, **193**
include, *see* directive
instruction operators, 155
integer, **159**
IsNoVoid , **259**
IsVoid , **259**
keyword, **156**
  method, 178
  relax, 177
lambda, 200
LEFT, 248, 249
Length , **247**
library, **198, 199**
list, **160**
  ** operation, 168
  ++ operation, 167
  overview, 167
MakeChoice , **253**
MakeChoiceLU , **253**
MAX_STRING, 159
member , **247**
MessageBox , **255**
meta-class, **162**
meta-model, **162**
meta-property, **163**
meta-relation, **162**
MetaClass , **167**
method, 178
mod, 166
modular programming, **192**
neof , **249**
newline, 160
not, 166
nseof , **253**
OpenFile , **248**
operations
  character, 243–244
  cursor, 247
  file, 248–250
  list, 247
  misc., 258
  string, 244–246

operator
  **, 168
  ++, 167
operators, **155**, 164
`or`, 166
precedence, **164**
`print`, 248
`printf` , **248**
`proc-name` , **xx**
procedure, **175**, 200
  foreign, *see* foreign
programming, *see* modular programming
query, **180**
  constraint, 182
  exhaustive, 183
  expansion, 185
  path oriented, 183
quote, double, 160
`read`, **249**
`readf` , **248**
register
  error, *see* error, register
`relax`, 177
`Remember&Paste` , **73**
`RenameFile` , **250**
reserved, word, **156**
`RetrieveID` , **259**
`return`, **196**
`RIGHT`, 248, 249
`seof` , **253**
`SetFlag` , **257**
`SetParser` , **251**
`SetPrintList` , **249**
`SkipUntil` , **252**
`SkipWhile` , **252**
`Specialize` , **75**
stack, **193**
statement
  `break`, 174
  `continue`, 174
  `do`, 172
  `else`, 170
  `for`, 172
  `goto`, 173
  `halt`, 175
  `if`, 169, 170
  `if-then`, 169
  `if-then-else`, 170
  `in`, 172
  `label`, 174
  `repeat`, 172
  `switch`, 170
  `then`, 169, 170
  `while`, 171
static call, 200
`StrBuild` , **244**
`StrCmp` , **245**
`StrCmpLU` , **245**
`StrConcat` , **245**
`StrFindChar` , **245**
`StrFindSubStr` , **245**
`StrGetChar` , **245**
`StrGetSubStr` , **245**
string, **159**
`StrIsInteger` , **246**
`StrItos` , **246**
`StrLength` , **246**
`StrSetChar` , **246**
`StrStoi` , **246**
`StrToLower` , **246**
`StrToUpper` , **246**
tab, 160
`this`, 179
type casting, 169
  in function calls, 177
type, definition, **158**
`UngetToken` , **252**
`use`, *see* directive
`use` , **199**
variable, **157**
void
  cursor, 161
`Void` , **259**

X

XMI, 206, 280
XML, 206, 280