Institutional Repository - Research Portal
Dépôt Institutionnel - Portail de la Recherche

researchportal.unamur.be

University of Namur

UNIVERSITÉ DE NAMUR

THESIS / THÈSE

DOCTOR OF SCIENCES

Database engineering process modelling

Roland, Didier

Award date:
2003

Awarding institution:
University of Namur

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 23. Jun. 2020

FUNDP
Institut d'Informatique
rue Grandgagnage 21
B-5000 Namur
Belgium
Tél. +32 81 72 49 64
Fax. +32 81 72 49 67
http://www.fundp.ac.be

Database engineering process modelling

Didier Roland

Supervisor

Prof. Jean-Luc Hainaut

Jury

Jean Fichefet (president, FUNDP, Namur, Belgium)

Jean-Luc Hainaut (FUNDP, Namur, Belgium)

Colette Rolland (Sorbonne, Paris I, France)

Éric Dubois (CRP Henri Tudor, Luxembourg)

Naji Habra (FUNDP, Namur, Belgium)

Academic year 2002-2003

A thesis submitted for the degree of PhD in sciences, computer science option

Abstract

An important research subject in Software engineering is concerned with modelling the development process of huge software in order to bring some help to engineers when designing and maintaining an application. In general, every design process is seen a rational application of transformation operators to one or more products (mainly specifications) in order to produce new products that satisfy some given criteria: O=*f*(I). I and O being sets of products compliant with formalisable models, *f* is a transformation composition whose specifications are the properties of I and O. This modelling is a sound basis for a methodological guidance. Indeed, at each step of the process, the set of pertinent activities and types of products are proposed to the designer, without any other. This guidance can be reinforced with some help. Furthermore, this modelling allows to document the process with its history, i.e. with a representation of performed activities. This history is itself the basis of maintenance activities.

The thesis holds in four phases:

- elaboration of a general model of design processes, a method specification language (MDL), and a history representation
- basic methodological recommendation proposals for the elaboration of engineering methods according to the defined model
- development and integration of some methodological control functions in the DB-MAIN CASE tool, including an extension of the repository, the definition of the interface of the methodological functions, the development of the methodological engine and the development of history processors (recording, replay, analysis,...)
- evaluation of this model with case studies using classical methods.

Résumé

Un sujet de recherche important dans le monde de l'ingénierie logicielle concerne la modélisation des processus de développement de grosses applications afin d'apporter de l'aide aux ingénieurs pour concevoir et maintenir leurs applications. En général, chaque processus de conception est vu comme l'application rationnelle d'opérateurs de transformation à un ou plusieurs produits (généralement des spécifications) pour obtenir de nouveaux produits qui satisfont un ensemble défini de critères: S=*f*(E). E et S étant des ensembles de produits conformes à des modèles formalisables, *f* est une composition de transformations dont les spécifications sont les propriétés de E et S. Cette modélisation permet, principalement, un suivi méthodologique. En effet, à chaque étape du processus, seul l'ensemble des outils pertinents est mis à la disposition du concepteur. Ce guidage peut éventuellement être renforcé par des messages d'aide. De plus, cette modélisation permet de documenter le processus avec son historique, c'est-à-dire avec une représentation des actions entreprises. Cet historique peut lui-même être à la base d'activités de maintenance.

La thèse tient en quatre parties :

- élaboration d'un modèle général pour la définition de processus d'ingénierie, d'un langage de spécification de méthodes (MDL) et d'une représentation des historiques;
- propositions de recommandations méthodologiques pour l'élaboration de méthodes d'ingénierie selon le modèle défini;
- développement et intégration de fonctions de contrôle méthodologique dans l'atelier DB-MAIN; ceci inclut l'extension du référentiel, la définition de l'interface homme-machine pour les fonctions méthodologiques, le développement du moteur méthodologique et le développement de processeurs d'historiques (enregistrer, rejouer, analyser,...);
- évaluation de ce modèle avec des études de cas utilisant des méthodes classiques.

Acknowledgement

This thesis is the result of a long work. A long time during which I met many people who helped me and who showed some interest. All these people deserve to be thanked.

First of all I want to Thank Jean-Luc Hainaut, with who I did all the job. I want to thank him for the opportunity he gave to me, for his support and for his availability in the framework of this thesis, but also for his great job in founding and leading the LIBD research team. I want to thank all this team too, including its present and former members, for their support and collaboration. In particular, many thanks to Jean Henrard, Jean-Marc Hick and Vincent Englebert for the great job we did in the DB-MAIN project, and to Virginie Detienne with who I also worked on another interesting project.

Many thanks to all the other people from the Insistut d'Informatique and from many other research labs for all the interesting discussion I had with them all. I will not list them all because I should fill several pages with all their names, but they can really be sure I do not forget them.

Many thanks to the readers of this thesis and to the members of the jury for their interest in it.

I also want to thank all the friends and family members who supported me along the years. Special thanks to Raoudha who particularly supported me along the whole work while preparing her own PhD and to Xia who particularly supported me in the last months. Many thanks to Renaud, Lysia, Sivilay, Olfa and many other friends who also supported me a lot.

But this thesis could not have been written if I had not received a very good education before. For this education, and for supporting me all the time from my birth, many, many, many thanks to my parents. Many thanks too to all my other family members (in the broader sense) and to all the people who participated to this education, including all the teachers at kindergarten, primary school, secondary school, and all the years at university before PhD.

Contents

Glossary 14

Chapter 1
Introduction 1

1.1. Process modelling presentation 2

1.2. State of the art and related works 3
1.2.1. History of data and process engineering 3
1.2.2. Process modelling in the large 4
1.2.3. CASE tools and meta-CASE tools 7
1.2.4. History recording 7

1.3. Database specifics 8

1.4. Goals 9

1.5. Structure of the thesis 10

1.6. DB-MAIN 11

Part 1
Models and Methods 13

Chapter 2
Basics 15

2.1. Basic definitions 16

2.2. Architecture 19

Chapter 3
Product models 23

3.1. Basic considerations 24

3.2. The GER model 25
3.2.1. Schema 25
3.2.2. Entity types 26
3.2.3. Relationship types (rel-types) 26
3.2.4. Attributes 26
3.2.5. Roles 27
3.2.6. Constraints 28
3.2.7. Is-a relations 31
3.2.8. Processing units 32
3.2.9. Collections 32
3.2.10. Dynamic properties 32

3.3. Schema model 33

3.4. Text model 45

3.5. Product model hierarchies 47

Chapter 4
Product types and process types **49**

4.1. Defining product types 50

4.2. Modelling engineering process types 51
4.2.1. Engineering process type decomposition 51
4.2.2. Engineering process type interface 52
4.2.3. Engineering process type strategy 55

4.3. Comparison with other modelling techniques 70

Chapter 5
The MDL language **73**

5.1. Requirements 74

5.2. Language definition 74
5.2.1. Generalities 74
5.2.2. Method 75
5.2.3. Product Models 76
5.2.4. Global product types 79
5.2.5. Toolboxes 80
5.2.6. External function declarations 81
5.2.7. Process types 82

5.3. Language analysis 96
5.3.1. The syntax is unambiguous 97
5.3.2. Syntactical analysis 98
5.3.3. The semantics is unambiguous 99
5.3.4. Compliance with the requirements 99

Part 2
Histories 101

Chapter 6
Histories **103**

6.1. Usefulness of histories 104
6.1.1. Documentation 104
6.1.2. Undo 104
6.1.3. Database design recovery 104
6.1.4. Database evolution 105
6.1.5. History analysis 105
6.1.6. Method induction 105

6.2. Expectations for histories 106

6.3. Structure of histories 106
6.3.1. Products 106
6.3.2. Processes 107
6.3.3. Primitive processes 107
6.3.4. Engineering processes 112
6.3.5. Decisions 113
6.3.6. The history of a project 114

6.4. History representation 116
6.4.1. Representation of the tree structure 116
6.4.2. Representation of primitive process histories 116
6.4.3. Representation of engineering process graphs 117

6.5. History construction 120
6.5.1. Primitive processes 120
6.5.2. Engineering processes 120
6.5.3. Hypotheses, versions and decisions 121

Chapter 7
History processing 125

7.1. Basic hypotheses 126

7.2. History replay 126
7.2.1. Replaying primitive processes of automatic basic type 126
7.2.2. Replaying primitive processes of automatic configurable type 126
7.2.3. Replaying primitive processes of automatic user configurable type 126
7.2.4. Replaying primitive processes of manual type 126
7.2.5. Replaying every primitive processes 128
7.2.6. Replaying engineering processes 129

7.3. History evolution 129

7.4. History transformation 132
7.4.1. History characteristics 132
7.4.2. Excerpts 135
7.4.3. Independent history excerpts 136
7.4.4. Equivalent history excerpts 137
7.4.5. Minimal history excerpts 137
7.4.6. Operations on history excerpts 137
7.4.7. History transformation 139

7.5. History cleaning 139
7.5.1. History cleaning 140
7.5.2. Primitive process history cleaning 140
7.5.3. Engineering process history cleaning 142

7.6. History flattening 143

7.7. History inversion 145

Part 3
In practice 147

Chapter 8
Method design: basic elements 149

8.1. Product model declarations 150

8.2. Product type declarations 151

8.3. Process type declarations 152
8.3.1. Loops 152
8.3.2. Sequences and each structures 155
8.3.3. Sub-process use 155
8.3.4. Degrees of freedom 158

Chapter 9
CASE tool usage **161**

9.1. Requirements 162
9.1.1. Method development environment requirements 162
9.1.2. CASE environment requirements 163

9.2. HMI proposals 166
9.2.1. Method development environment 167
9.2.2. Method visualisation and browsing 168
9.2.3. Following a method 170
9.2.4. Recording a history 178
9.2.5. Complementary tools 181
9.2.6. Configuring the CASE environment 186
9.2.7. Browsing through a history 187
9.2.8. History replay and transformation 187

Chapter 10
Architectural issues **189**

10.1. General architecture 190

10.2. The repository 191
10.2.1. Notations 192
10.2.2. The original repository of the DB-MAIN CASE environment 192
10.2.3. The repository extension 193

10.3. Parsing an MDL source file 202

10.4. The GUI 203
10.4.1. Loading a method 203
10.4.2. History window extension 203
10.4.3. The methodological engine 204
10.4.4. The GUI look and feel 204

10.5. The methodological engine 207
10.5.1. Following a method 207
10.5.2. Product and expression evaluation 209

Chapter 11
Case studies **211**

11.1. First case study: a simple forward engineering project 212
11.1.1. Defining the method 212
11.1.2. Performing the project 216
11.1.3. The resulting history 222

11.2. Second case study: a complex reverse engineering project 223
11.2.1. Method description 223
11.2.2. Project performance 229
11.2.3. The resulting history 239
11.2.4. Design recovery 241

11.3. Conclusion 241

Chapter 12
Professional use **247**

12.1. List of questions 248

12.2. Relational database applications evolution 248

12.3. XML Engineering 248

12.4. Conclusion 250

Part 4
Future work 251

Chapter 13
Method evolution 253

13.1. Presentation 254

13.2. The problem 254
13.2.1. Product models and product types 254
13.2.2. Process types 255
13.2.3. The method evolution problem 255

13.3. Solution proposal 256
13.3.1. Temporal databases 256
13.3.2. A solution proposal for the method evolution problem 260

Chapter 14
Conclusion and future works 263

14.1. Conclusion 263

14.2. Future works 264
14.2.1. Method evolution implementation 264
14.2.2. Method engineering methodology 264
14.2.3. Method recovery 265
14.2.4. Graphical method development environment 265
14.2.5. Extending to software engineering in general 266
14.2.6. Supporting a Meta-CASE 266
14.2.7. Supporting co-operative design 266

Bibliography 267

Appendix A
Schema analysis predicates 277

A.1. Constraints on schema 277

A.2. Constraints on collections 277

A.3. Constraints on entity types 278

A.4. Constraints on is-a relations 281

A.5. Constraints on rel-types 282

A.6. Constraints on roles 284

A.7. Constraints on attributes 284

A.8. Constraints on groups 286

A.9. Constraints on entity type identifiers 287

A.10. Constraints on rel-type identifiers 290

A.11. Constraints on attribute identifiers 293

A.12. Constraints on access keys 295

A.13. Constraints on referential groups 296

A.14. Constraints on processing units 298

A.15. Constraints on names 298

A.16. Using DYN_PROP_OF_... constraints 300

A.17. Using Voyager 2 constraints 302

Appendix B
The PDL syntax 303

B.1. BNF notation 303

B.2. The PDL language 303

Appendix C
Global transformations 305

C.1. Transformations 305

C.2. Control structures 307

Appendix D
The MDL syntax 309

D.1. BNF notation 309

D.2. Miscellaneous rules 309
D.2.1. Spaces and comments 309
D.2.2. Forward references 310

D.3. Multi-purpose definitions 310

D.4. Expressions 310

D.5. Method description 311

D.6. External declaration 311

D.7. Schema model description 312

D.8. Text model description 312

D.9. Product type description 312

D.10. Toolbox description 313

D.11. Process type description 313

Appendix E
DB-MAIN functions 315

Appendix F
Case study listings 321

F.1. The first case study: a forward engineering method 321

F.2. The first case study: the interview report 327
F.3. The first case study: the script of actions performed by the engineer 327
F.4. The second case study: a reverse engineering method 330
F.5. The Order.cob program analysed in the second case study 339
F.6. A small C program to clean log files 343

Glossary

This glossary is a list of the main terms used in this thesis. They are fully defined in the thesis. These definitions are summarised here for reference, as a reminder for the reader.

actor: An actor is a person, or a machine, that can perform actions and conduct processes.

automatic primitive process type: A primitive process type that can be performed by the CASE environment without the intervention of an engineer.

decision: A decision is either a choice of one or several product versions to abandon among several ones, or a *yes* or *no* answer to a question imposed by the method.

engineering process: An engineering process is a goal-driven process, i.e. a process that tries to make its output products comply with specific design requirements.

GER: The Generic Entity-Relationship model used as the basis for defining database schema models within the DB-MAIN CASE environment and the MDL language.

history: A history is the recording of everything that happens during the life cycle of an engineering project. It also includes all the products that are used or produced during the project, as well as all the rationales according to which the processes are carried out.

hypothesis: An hypothesis is a statement that confines a problem to a particular context in order to solve it.

log file: A log file is an ASCII-based text file containing the trace of performed actions. In this thesis, log files are used to store primitive process histories.

manual primitive process type: A primitive process that must be performed by a human being, using the tools provided by the CASE environment.

MDL: The Method Definition Language is a non-deterministic procedural language aimed at defining database engineering method in order to configure a CASE environment.

method: A method is a way-of-working commonly agreed among engineers to perform a given work.

methodological engine: The methodological engine is a piece of program which is added to a CASE environment in order for it to be able to follow a defined method.

methodology: A methodology is a system of methods and principles for doing something[1], database engineering in this thesis.

method-free project: A project performed without the guidance of a declared method. The engineer may follow an implicit method anyway.

method-supported project: A project performed according to a method defined with the MDL language.

1 This definition is from [COLLINS,95].

primitive process: A primitive process is an atomic process, that is to say, a process that comprises a single operation. It is a single step on the path towards the goals of an engineering process.

process: A process is an activity that is carried out by an actor in order to transform products.

process type: A process type describes the general properties of a class of processes that have the same purpose, and that use, update or generate products of the same types.

product: A product is a document used, modified or produced during the design life cycle of an information system. They are database schemas and database-related texts.

product model: A model defines a general class of products by stating the basic components they are allowed to be included and the assembly constraints that must be satisfied.

product type: A product type describes a class of products that play a definite role in the system life cycle. A product type is expressed into a product model. A product is an instance of a product type.

product version: A version of a product is the result of solving a problem in a particular context after specifying an hypothesis. Several hypotheses may lead to several versions of a product.

schema: Database schemas can be any data structure description that can be of interest during the whole life cycle of the database engineering project, in any phase, at every abstraction level, ranging from conceptual entity-relationship, object oriented or UML schemas, to physical Oracle or COBOL schemas.

strategy: The strategy of an engineering process type specifies how any process of this type must be, or can be, carried out in order to solve the problems it is intended to, and to make it produce output products that meet its requirements. In particular, a strategy mentions what processes, in what order, are to be carried out, and following what reasoning.

text: A text is any relevant character-based document that is not a schema. This concept encompasses program source files, SQL-DDL scripts, help files, word processing files, forms, etc.

tool: A tool is a function of the CASE environment that can be used through the menus, the toolbars, keyboard shortcuts, or with the mouse when it points to a window.

toolbox: A toolbox is a collection of tools provided by the CASE environment. A toolbox can be put at the engineer's disposal by the CASE environment when required by the method.

Chapter 1

Introduction

This chapter introduces the concept of process modelling. Then it draws a state of the art in the field. It lists a large series of research projects and classifies them according to several criteria which draw the main orientation of this thesis. The concept of process modelling in the large will then be restricted to the database realm, and to the DB-MAIN environment in particular for prototyping.

1.1. Process modelling presentation

Every day, every living being performs a series of processes. Some of these processes are innate, such as breathing. Most must be learned though. Some processes are learned early in the life, naturally, without help, because they are vital, such as walking. All beings of the same species generally agree on the way of performing the process. More complex and less vital processes are learned by everybody. Talking is such a process for human beings. But all men and women do not speak the same language, some even communicate with hands. In this case, some references (specialist, book,...) detailing the way of performing the processes, or simply tips, can be useful. This is why most widespread languages have dictionaries. Finally, there are much more complex processes which are only practised by restricted groups only, like a specific kind of job. Each job can even have some specialities. In these cases, learning the process can be very long and the path can be littered with pitfalls because the way of working can be complex and various minds may apprehend it differently. In fact, we should write "ways of working", plural form, because each specialist can have his or her own one. In some cases, two specialists who should do the same work may do it differently enough for them to not understand each other during a discussion. For instance, two programmers can write applications that are compliant with the same requirements, that do the same things, but one writes the application in C++, while the other one writes it in Prolog.

The good health of a large company is based on three fundamental resources: money, people, and information. Money is necessary to pay people, to buy supplies, for buildings, heating, communications,... People are necessary to do the job. And information is necessary to manage people, to manage stocks, to manage customers, to manage suppliers, to manage production,... The lack of one of these three resources will inevitably lead to bankrupt the company. Managing these three resources are three different jobs that all need to be performed by specialists. Nowadays, for competitiveness reasons, information management must be supported by an information system. Since flaws in this information system can produce incorrect information, or no more information at all, the flaws can lead to bankruptcy. Furthermore, the information system has to evolve with the company and with the company environment (laws, markets, company size, users' wishes,...) in order to be trustworthy all along the company life. So the information system design is a vital process, as well as its maintenance and its evolution, so important that it cannot be performed by a single man or woman, but rather by teams. To do their job correctly, all these people need to understand each others, and to work in the same way. That is why, a few decades ago, some people tried to define good methods that would be followed by everybody. Merise is such a method example among many others. But if such general methods can be good starting points, they hardly suit the needs of companies that have specific problems. That is why researchers all over the world have been working on designing tools to help companies to model ways of working which are relevant to them, and to help these companies to make their ways of working available to their people and accepted by them.

Computers evolve, applications evolve, new information arrive and must be stored every day, but the archives, the memory of the company, do not evolve. Data stored in the information system have to survive to all changes in the computers or in the applications. Data have to survive during a very long time, possibly the whole life of the company. In other words, the databases around which an information system is build is a basic asset that must be handled as such. This is why we will concentrate, in this thesis on modelling the processes that can be performed by engineers who are in charge of designing, maintaining and making databases evolve.

Since a large database design activity has to be performed by several persons, it is important for them to share their knowledge. Since a database may have to survive during several dec-

ades, engineers who designed it at the beginning will probably retire before its death. So the engineers who maintain the database are not the same. It is thus very important that all the knowledge elaborated by the previous engineers be transmitted to their successors. For that reason, a good recording of all activities, all decisions taken, and all rationales that justify the decisions must be recorded in a reusable way. In other words, the complete history of the database design, maintenance and evolution must be kept.

In this thesis we will examine one particular way of seeing the modelling of database engineering processes and the recording of histories.

1.2. State of the art and related works

Process modelling is a rather general subject for which a lot of research have been conducted for several decades. The first researches recognised the necessity of well defined methods and models.

1.2.1. History of data and process engineering

Some first ways of structuring and representing data were introduced in the late sixties and in the seventies. For instance, [BACHMAN,69] gave birth to the entity-relationship (ER) model, and [CHEN,76] popularised this model. It then evolved to better suit the users needs, like [FOUCAUT,78] who adds the processing and their operational dynamics with the REMORA project, and [HAINAUT,89] who extended it in order to cover a broader range of data models. More recently, [OMG,01] presents the last release in date of the UML model, a graphical communication model for representing all the aspects of an application design, including the static (ER-like schemas) and dynamic aspects of data structures, as well as use cases and packaging.

Meanwhile, researchers noticed that software was becoming more and more unstable while becoming larger and more complex [DIJKSTRA,68]. So these researchers began to model ways of developing reliable software. This included researches on programming languages: structured programming with, among others, the birth of the ALGOL 58,60,68 language [DIJKSTRA,62], and the Pascal language [JENSEN,78], object oriented languages such as SIMULA 67 [DAHL,67], logical programming with languages such as Lisp [MCCARTHY,60][STEELE,90], or Prolog [CLOCKSIN,84]. It also included researches on the ways of designing software independently of the final programming language, that is to say ways to specify formally the different parts of the programs: The Jackson Structured Programming (JSP) [MCLEOD] is a method for modelling programs "in the small", which are programs manageable by a single programmer; the Jackson Structured Development (JSD) [MCLEOD] is aimed at larger projects; the waterfall model [ROYCE,70], the spiral model [BOEHM,88] and the fountain model [HENDERSON,90] are also well-known software engineering methods. [FICKAS,85] implements software automatically with a transformational development. More recently, researchers have explored ways to develop new software by reusing pre-existing software chunks [BENGHEZALA,01] instead of redesigning everything every time. A less technical, more human-oriented approach of improving software is proposed in XP programming [XP] which focus on team work and communication inside the team.

Other people also began to model other aspects of information systems engineering, such as the requirements in [SORENSON,88], [CHUNG,91], [ROLLAND,93], [DUBOIS, 94] or [POHL,96], or the human-machine interfaces [BODART,95], [VANDERDONCKT,97].

People then noticed that simply designing clean software does not solve entirely the problem of instability because the software has to be maintained and to evolve. These activities

deserve great care too. All the rationales behind the changes have to be stored correctly in order to not redo the same errors several times along the whole life of the applications [HAUMER,99]. In particular, since the information systems are build on top of a database system, this one deserves great attention too [HICK,98].

Eventually comes a time when the applications, or simply the hardware onto which the applications are run, become obsolete. It is then necessary to build new applications. But the content of the database, which is the memory without which the organisation cannot live, has to be kept. It is thus necessary to re-engineer the databases, as presented in [HAINAUT,95] and [HAINAUT,96a].

1.2.2. Process modelling in the large

For such activities as requirements engineering, human-machine interface design, maintenance and evolution, re-engineering, as well as for forthcoming activities, the software process models defined previously are not adapted. Furthermore, those methods also proved to be poorly adapted to the particular needs of each organisation, even for the jobs for which they were originally conceived. So a new trend is born: to give means to each organisation to define its own methods. [CURTIS,92] is a general paper which presents various aspects of process modelling in the large, including business process modelling and software process modelling. [FEILER,93] and [JAMART,94] define various terms commonly used in the process modelling domain. [KRASNER,92] shows the usefulness of process modelling with a particular case study. Along the years, several research labs published various project results. [FINKELSTEIN,94] and [GARG,96] present in details several process modelling projects. Among these project and others, we can enumerate:

- HFSP [KATAYAMA,89]: a process-centred software engineering model with a mathematical functional representation of processes.
- APPL/A [SUTTON,90]: a process-centred software engineering model with a programming (extension of the Ada language) way of representing processes. It is implemented in the Arcadia environment [TAYLOR,88].
- MELMAC [DEITERS,90]: A process centred software engineering process modelling approach using FUNSOFT nets (high level Petri nets).
- DAIDA [JARKE,92]: a knowledge-based process-centred environment for database applications.
- TAME [OIVO,92]: a goal-oriented approach to software engineering with a rule-based mechanism for constructing methods.
- KBMS [ZEROUAL,92]: a knowledge-based system for modelling software engineering methods with rule-based techniques.
- Marvel [BARGHOUTI,90][FINKELSTEIN,92]: a rule-based software engineering environment centred on reuse.
- Process Weaver [FERNSTROM,93][CAPGEMINI,95]: a process-centred environment for managing team-based activities with a Petri-net-like representation of the processes.
- SPADE [BANDINELLI,93]: a software engineering environment with an object-oriented process model based on Petri nets (using the SLANG language).
- TAP [YONESAKI,93]: the Task-Agent-Products approach is a process-centred environment for software process modelling with agents and Petri-net representation of the methods.
- EPOS [CONRADI,93][CONRADI,94b][EPOS,95]: A process centred approach for defining software engineering process models with an object-oriented specification lan-

guage (SPELL).

- Merlin [JUNKERMANN,94]: a process-centred software development environment with a Prolog-like process representation.
- SOCCA [ENGELS,94]: a process-centred software engineering environment with object-oriented and data flow diagram representation of the processes.
- Adele [BELKHATIR,94][ESTUBLIER,94]: a process-centred software engineering modelling environment with object-oriented and trigger representation of the processes.
- Sentinel+Latin [CUGOLA,95]: a process-centred software engineering environment (Sentinel) with a rule-based temporal constraint language (Latin) to represent processes.
- M_{CASE} [BRUNO,95]: a process-centred software engineering environment with data flow based processes.
- Metaview [SORENSON,88][FROEHLICH,95]: a process-centred software engineering environment with a rule-based description of processes.
- MetaEdit+/GOPPR [KELLY,96]: MetaEdit+ is both a CASE and a CAME (computer aided method engineering) environment; the method engineering part is process-centred and uses a graph and object oriented representation of the processes (GOPPR).
- Nature, Crews [NATURE,96][ROLLAND,97][TAWBI,99]: context-and-decision-oriented meta-models for defining requirements engineering processes with a rule-based representation of processes. It is implemented in the Mentor CARE (Computer-Aided Requirement Engineering) environment [SISAID,96].
- APEL [DAMI,97]: a graphical (using data flow, control flow and state transition graphs) representation of software engineering processes.
- E^3 [JACCHERI,98]: An object-oriented language with graphical representation for process-centred software engineering.
- Prime [POHL,99]: A process-centred environment for requirements engineering which uses the process representation of Nature and extends it to allow the use of third party tools.
- PROSYT [CUGOLA,99]: A process-centred distributed business process modelling tool using an artifact-based approach which allows deviations in enactment.
- [DITTRICH,00] presents a roadmap to using database technology for software engineering.

[MARTTIIN,98] and [SAEKI,94] are also nice papers that complete the list above with many other projects. [TOLVANEN,98] presents method engineering approaches in its third chapter too.

A lot of the tools above allow the interoperability of several third party tools (editor, compiler,...). More recent works [ESTUBLIER,96] [DAMI,97] [KELLY,96] [POHL,99] go further by investigating ways to make several process engines communicate with each other, and/or with third party tools.

The use of process models has proved its usefulness in various other domains as well. For example:

- [BOGUSCH,99] shows the use of a process model for chemistry practices.
- [MUETZELFELDT,01] uses a process model for ecology activities.

One of the most widespread use of process modelling outside software engineering is certainly business process modelling which is useful for three main purposes, namely Total

Quality Management, Business Process Reengineering, and Workflow Management:

- Catalysis [DSOUZA,98]: a graphical representation of business processes using UML.
- Artemis [CASTANO,99]: a business process reengineering environment.
- IDEF [DEWITTE,97]: a standard modelling and analysis method for business engineering.
- ProVision [PROFORMA,99]: an environment for business modelling and system design.
- Other business modelling works include: [BARROS,97], [BRATAAS,97], [MAYER, 98], [JORGENSEN,99], [GREEN,00], and [VONDRAK,01](workflow automation).

To summarise the previous enumeration of research project, six process modelling paradigms can be found in the literature:

1. Rule based: each process type is a set of rules. Some rules are preconditions that must be fulfilled for the process to be enactable. Some rules are postconditions that are guarantied to be fulfilled when the process terminates. Other rules describe the behaviour of the process type (the equivalent of the strategy in our model). This model is one of the most widespread (DAIDA, TAME, KBMS, Marvel, Merlin, Sentinel+Latin, Metaview, Nature, Prime,...).
2. Functional: preconditions, postconditions and behaviour are all stated with mathematical functions. This model is seldom used (HFSP for instance).
3. Petri nets: the strategy of process types are described with Petri nets, coloured Petri nets or any other variant of Petri nets. Also an often used technique (MELMAC, Process Weaver, SPADE, TAP among others).
4. Graph based: the strategy is represented with dataflow diagrams or state transition graphs. This technique is less used (M_{CASE}, MetaEdit+ or APEL for example).
5. Procedural: the strategy is expressed in a procedural language. This is the technique we will use in this thesis. It is also used by a few other research projects (such as APPL/A).
6. Object oriented: a variant of the procedural technique with the in fashion technique of object encapsulation (EPOS, SOCCA, Adele,E^3,...).

The research projects can also be divided in two categories: process-centred and goal-oriented. Process centred techniques put the accent on the method itself, while goal-oriented techniques stress their attention on the product that are to be produced and on the search for a way to reach that goal.

All the projects mentioned above present a way to describe a method and to use it. All of them also tell how to create such a method, but generally very briefly: the environment integrates a tool that provides this capability, nothing more. A few of them go further by giving some methodological guidelines. [SCHLENOFF,96] presents some requirements for modelling processes, with the hope to define a general framework that would suit every process modelling needs. [HUMPHREY,95] presents a way to make a process model (using the Personal Software Process model) evolve in the way of greater efficiency. [ROLLAND,97] presents a complete framework for engineering methods for requirement engineering: a method is build like requirements, with a context-and-decision approach, possibly with the reuse of method chunks. [RALYTE,01a] and [RALYTE,01b] go further in the same direction, focusing on the reuse of method chunks. Still further, [JORGENSEN, 00a] shows how to define a particular process model by reuse of “general process models” and to harvest the latter with the knowledge gained by the use of the particular process model.

One of the main goals of all these researches is the quality of software. The SCOPE project [WELZEL,92] provides an assessment method to measure the quality of software design and generated products. But the quality of software also passes through the quality of the process model itself: [BROCKERS,93] verifies properties of a software process model, using FUNSOFT nets in the MELMAC environment; [SADIQ,00] proposes another technique to analyse workflow based process models using graph reduction, in search of deadlocks and lack of synchronisation.

All the process modelling tools presented so far use their own representation of their data. [SCHLENOFF,00] presents the Process Specification Language (PSL) which is aimed at allowing the previous tools to exchange information.

A roadmap to the future of software engineering which identifies the principal research challenges is presented in [FINKELSTEIN,00].

1.2.3. CASE tools and meta-CASE tools

Software engineering, in particular database engineering, not only needs good methods to be performed, but also good CASE tools. Concerning specifically database engineering, [ROSENTHAL,94] proposed a prototype CASE tool, DDEW, that can handle several database schema models using a unified underlying model called ER+ and transform a schema from one model to another using content-preserving schema transformations. [HAINAUT,94] and [ENGLEBERT,95] present another similar tool called DB-MAIN which evolved towards a mature CASE tool [DB-MAIN,02a], while incorporating more advanced features, such as reverse engineering facilities [HENRARD,98]. This is the CASE tool that will be used in this thesis, and that will be extended with a methodological engine. A more general software engineering CASE tool, which already supports method specification, namely Phedias, is presented in [WANG,95]. This tool is in fact a meta-CASE tool (called a CASE shell in the paper): it is general enough to be used in various situations, and it needs to be customised in order to be usable as a CASE tool for specific needs. Prime [POHL,99] (see above) is another metaCASE tool which is requirement engineering oriented.

1.2.4. History recording

The use of a CASE tool or of a metaCASE tool can help a software (or database) engineer not only to perform a particular job. If the analyst has to perform a second or a third job of the same kind, he or she can simply follow the same way of working. But it appears clearly that learning the lessons from the first job can improve the quality and the efficiency of the subsequent jobs. The best way recognised by the research community to learn the lessons of a job is to keep a whole trace of it, and to record all the rationales behind all the decisions taken by the analyst. It allows engineers to be reminded, during subsequent projects, how the discussion was conducted and why the final decision was taken as such; it allows engineers to take future decisions much faster, and in concordance with the first one. Even the first application, result of the first engineering job, may have to evolve. It is even more important in this case to remind exactly what was done the first time and why it was done that way. The matter of recording rationales is discussed in [POTTS,88]. Later on, several researchers followed the idea. [SOUQUIERES,93] presents a requirement engineering framework in which all the decisions and their rationales are perfectly documented. [POHL,97] and [DOMGES,98] note that traces of genuine engineering projects can be very huge and time consuming, and propose a way to capture the needed information only, which may vary according to the projects specific needs. In [HAUMER,99], the traces of the original engineering of a system are accompanied with traces of concrete system usage scenarios in order to make the information even more pertinent when making the system evolve. [ZAMFIROIU,98] (also [ZAMFIROIU,01]) studies software engineering traces

independently of any CASE tool. This work has three objectives: recording traces (possibly with version management), synthesise them into operation flow (to enhance readability and usability), and a measure of the continuity of the flow (in order to detect subsequent changes, to evaluate the impact on the project, and possibly to assist the engineer in repairing breaks). This work proposes a trace model (KARMA), as well as tools to handle and to query traces.

Since the purpose of this thesis is to help database engineers perform their job, it is an important issue to produce usable tools. Lessons can be learned from [CATARCI,00] which relates the story of a database research team who already had to deal with similar problems of user acceptance of specific tools it designed.

1.3. Database specifics

A lot of process modelling projects were already conducted all over the world, as shown previously. They concern a very broad range of application domain: software engineering, requirements engineering, business processes, electrical engineering, ecology,... Within the framework of this thesis, we will concentrate on the database realm. Indeed, in a large organisation, the management of employees, customers, products, finances and other resources is nowadays always performed with one or several large information systems. All these information systems are a set of applications using a central database which contain all the memory of the organisation. Along the time, the applications evolve, sometimes rather deeply, and can even be completely replaced several times along the life of the organisation. The database management system may also evolve. But the data stored in the database are one of the main resources of the organisation and must be kept in perfect state without any loss along the whole life cycle of the organisation, even if their format and structure evolve. So databases really deserve a particular attention in their treatment.

Since so many projects were already conducted and since so many (meta)CASE tools already exist, one should wonder why we do not use one of these tools. The answer holds in two main points. Firstly, this thesis is conducted in the framework of the DB-MAIN project, so one prerequisite is to use the DB-MAIN CASE tool (see Section 1.6), either by developing new functions in it, or by integrating it with other tools. Secondly, the database realm has several specific aspects that cannot be handled by non-database-specific CASE tools:

- Database engineering theory is much more advanced than software engineering theory. Indeed, the transformation of a database schema compliant with one model to another semantically equivalent schema compliant with another model (for instance, the transformation of an ER schema into a relational schema) can be described very precisely with a series of semantics preserving elementary transformations that are all published and proved to be correct [HAINAUT,96c]. In the design of a program, the gap between the requirements and the code is much larger. Some formal requirements expressed with formal languages can be translated into source code of functional or logical languages, but seldom in much more popular procedural source code. And non-functional requirements expressed with the natural language have a semantic that cannot be grasped by machines. Nothing more than analysis tools (for instance searching, pattern matching and program slicing tools), prototyping tools, and simple text editors can help the programmers, not even to prove that the result is the one expected. In other words, most database engineering works can be performed through a set of dedicated elementary tools which do not exist within other disciplines. The need to take into account the particularities of the actors needs is underlined in [NUSEIBEH,93].
- As a corollary of the first point, in most disciplines a text (a source file, a requirement description, a scenario of a task,...) is often the smallest elementary concept that can be

handled by tools: a text can be edited, a source file compiled,... Within the database engineering paradigm, a database schema can be decomposed in all its components and transformations applied to some specific components directly. So a CASE tool that supports database engineering activities has to be able to handle a fine-grained decomposition of the products, which is seldom the case with other CASE tools.

- Elementary tools used in database engineering activities are often simpler than in software engineering. Indeed, schema editing functions are often simpler to implement than a compiler or a debugger. As a consequence, a software engineering CASE tool seldom offers all the tools which are necessary to perform a complete project, it often requires third party tools (such as an advanced text editor, a compiler,...). A database engineering CASE tool can more easily integrate all the necessary tools or provide means to easily add them (such as an advanced macro language or a 4GL like the Voyager language included within the DB-MAIN CASE tool, as presented in Section 1.6). Hence database engineering allows a better integration of tools.

The fact that database engineering is the target of this thesis does not mean that the model and the language developed in this work must be confined to databases. Indeed, only a few updates to the model should be necessary to extend it to other domains of interest. These extensions will be presented in chapter 14. In a way, [DOWSON,94] summarises the content of Chapters 3, 4, 5, 6, 7, 9, 13 applied to software engineering.

1.4. Goals

This thesis will pursue one main goal which is to bring the most useful possible help to database engineers. This goal has to be seen with various angles according to the different aspects of the database engineer's job:

- The way of working he or she should follow can be imposed to him or her, strongly or loosely. And he or she can be guided to correctly follow this path.
- The job the database engineer actually performs can be recorded. The fact that database engineering tools can be integrated can make the recorded history very useful for various tasks. This usability will be proved by proposing a structure for history recording and providing a series of operators to handle this structure.

A lot of research projects, some of them presented in Section 1.2, already tackled the guiding problem. Most of them use either a declarative language, an object-oriented language, or Petri net-like representations to define a method. Only a few projects use a functional model (HFSP for example) or a more traditional procedural language (APPL/A for instance).

About programming languages, it often appears that declarative languages, functional languages and Petri-net like representations are stuck in universities or research laboratories, and are poorly adopted by industrials. Object oriented programming languages have a better acceptance in the industrial world, but are often badly used, with a few objects encapsulating large chunks of procedural code. Procedural languages are still the most widespread languages.

Of course, as [OSTERWEIL,97] underlines, "Software Processes are Software too" is a false assumption. So the choice of a programming paradigm cannot be extended to process modelling as easily. According to [BOBILLIER,99], activities such as requirements engineering, which is more decision centric and which has to deal with non-functional requirements, are mental activities. Indeed, the problem is loosely and badly defined from the beginning and must be refined while solving it. It seems that declarative languages are better suited for modelling such processes. But database problems are different. Indeed, along

the advancement of the project, the work is more and more technical, more and more transformation oriented. When a design project begins, the database engineer receive requirements which were already specified during a previous requirements engineering project, and he or she draws a first conceptual schema using a graphical editor. Then he or she goes forward by normalising the schema or optimising it, possibly integrating several schemas. These operations can already be performed using some transformations, but some decisions still need to be taken to apply the transformations correctly. In a later step, the schema is transformed more automatically in order to produce the logical schema, the physical schema, and finally to generate the code. A reverse engineering job begins with legacy programs and data which are in use for a long time. The jobs mainly consist in analysing and transforming these sources. So these job are more technical ones, more transformation oriented. That is why this thesis supports the idea that a procedural language with an algorithmic graphical representation of a method is the way of working that should be preferred for this kind of database engineering activities. Advantages and disadvantages of the different paradigms will be discussed and it will be proved that this choice naturally leads to a real help for the database engineers.

This thesis does not only define one more framework and one more method description language to the research community, but it shows its usefulness with the implementation of the language and of a methodological engine in a CASE tool of professional quality. This implementation does not simply show the feasibility of the theory presented in the first chapter of the thesis, but it also shows that the technique is industrially viable, although industrial users still need to be converted to it for a wider use.

1.5. Structure of the thesis

In this introduction, process modelling was described informally, related works and the state of the art in this domain were examined in the large. Then the specificity of the database realm was shown, the remaining of this thesis being concentrated in it. When the framework was drawn, the goals of this work were stated. This chapter will be terminated by a short description of the DB-MAIN CASE tool, which is the concrete framework for the evaluation of the results of this work.

In the three following chapters, all the concepts and components that are necessary for modelling database engineering processes are precisely defined: Chapter 2 gives a definition of all the concepts, Chapter 3 gives a complete description of product models and product types, and Chapter 4 is about the description of process types. A language (MDL) for coding all these concepts is defined in Chapter 5.

Chapter 6 is devoted to a full description of histories and Chapter 7 to their handling and transformation.

The MDL language is procedural but aimed at being executed by human being rather than by machines. Since human beings and machines act differently, a few methodological guidelines deserve to be followed to correctly define a method. Chapter 8 is devoted to these methodological aspects.

Chapters 9, 10 and 11 address experimentation. Chapter 9 studies the human-machine interface aspects, while Chapter 10 is devoted to the internal architecture. Chapter 11 presents two case studies. Chapter 12 presents a few real projects using the implementation.

Chapter 13 underlines an important aspect of methods which is not taken into account previously but which deserves a full attention (maybe another thesis): the problem of the method evolution. Chapter 14 traces paths for future works and concludes this work.

1.6. DB-MAIN

DB-MAIN is a database oriented CASE environment developed at the university of Namur, in the Database Engineering Laboratory (LIBD[2]). The purpose of this CASE environment is to assist a database engineer in every database engineering activity he or she can face, including database design, database reverse engineering, database evolution, database re-engineering, databases integration,... In this section, we will describe its main characteristics.

- It is based on the GER model presented in Chapter 3 which is general enough to allow a database engineer to represent a very broad range of concepts from a very broad range of data models at all abstraction levels.
- It is transformation-based. A database schema which is compliant with a given schema model can be transformed into a semantically equivalent schema which is compliant with another schema model. This transformation can be performed step by step with a set of basic transformations by the analyst who can control the whole process and understand what happens. This is rather different from most commercial CASE environments where schema conversion from one model to another is just a black box.
- It allows different usage levels. Schema transformations can be performed in several ways; step by step with full control by the engineer, in an automated way with an advanced configurable assistant, in an automated way with a simple assistant working on a problem-solution basis, or as a fully automated black box.
- It is methodology neutral. An engineer using this CASE environment is allowed to do whatever he or she wants. He or she can either follow a well-known method, or his or her own method, or simply use the CASE environment as a white board on which he or she can draw freely. It is this aspect of the CASE environment that is addressed throughout this thesis.
- Users can personalise the GER model by defining new meta-properties for its different concepts. For example, it is possible to add an *owner* meta-property to the entity type concept, so that the *owners* of each entity type can be specified.
- It embeds a 4GL (namely *Voyager 2*, [ENGLEBERT,99]) which allows database engineers to develop their own schema transformations, or more complex tools such as report generators or specific DBMS DDL generators.
- It allows data structure extraction and data structure generation for several DBMS and computer languages. Some of the generators and extractors are written in the *Voyager 2* language and their sources are provided to allow engineers to adapt them to their own needs.
- It is repository-based. All the schemas and other texts are kept in a built-in object oriented repository. The structure of this repository is described in the manuals [ENGLEBERT,99]. The repository is accessible through the *Voyager 2* language, and through C++ and Java classes.

The theoretical aspects of this thesis will be implemented in the DB-MAIN CASE environment for evaluation. The repository will be extended to store the new concepts we will define in Chapter 2. And the user interface will have to be updated in order to help the engineers to use all the new capabilities.

A more comprehensive description of the CASE tool can be found in [ENGLEBERT,95] and in [DB-MAIN,02A].

2 Laboratoire d'Ingénierie de Bases de Données

Part 1

Models and Methods

Chapter 2

Basics

This chapter defines the building blocks that will be used throughout this thesis. Firstly, it defines the basic concepts and terms on which we will build our proposal: actor, analyst, database engineer, method engineer, process, engineering process, primitive process, process type, strategy, toolbox, product, schema, text, product type, product model, hypothesis, decision, product version,... Secondly, the basic concepts will be assembled in a three level engineering process model that will guide us all along this thesis like a map.

2.1. Basic definitions

This thesis aims to develop concepts, models and tools to help software engineers in their database design projects. The processes we are considering are perceived as product transformation activities. It is thus necessary to begin by defining more precisely the kind of products we are talking about, as well as the transformation processes, and who will have to do every job.

- *Actors*

An **actor** is a person, or a machine, that can perform actions and conduct processes. A human actor is an intelligent being capable of thinking and taking decisions. He or she can look for a non-predefined solution when facing a new problem. Human actors can also get slow and lazy when facing repetitive and tedious works. Machines can only apply predefined recipes, but they can do it quickly and without getting tired. In this thesis, we will develop principles about transformation processes, from their design to their use. So we can define two main classes of actors:

- The first class is made up of the people who design the transformation processes. They are human beings only, because their job is mainly based on decision taking and requires database engineering technical knowledge, as well as a good knowledge of the organisation and the people working for it. We will call them the **method engineers**. They decide how the actors of the second class have to work, and how they will be helped.
- The second class comprises the people and computer programs who will apply the methods as a series of transformation processes. We will call the people **database engineers**, **analysts**, or simply **users**. We will call the computer programs **function**, **procedure**, **operation** or **assistant** depending on the context.

Though we will be interested in the distinction between human actors and machines, we will not address some important project management problems, such as human resource management (studying dependencies between people and machines, affecting particular persons to particular tasks,...) which is a complex problem studied in [SUTCLIFFE,00]. In particular, actors modelling will be ignored in this thesis.

- *Products*

A **product** is a document used, modified or produced during the design life cycle of an information system. As we focus specifically on database engineering, we will describe mainly database **schemas** and database-related **texts**. *Database schemas* can be any data structure description that can be of interest during the whole life cycle of the database engineering project, in any phase, at every abstraction level, ranging from conceptual entity-relationship, object oriented or UML schemas, to physical Oracle or COBOL schemas. We examine this in more detail in Chapter 3. A *text* is any relevant character-based document that is not a schema. This concept encompasses program source files, SQL-DDL scripts, help files, word processing files, forms, etc.

- *Processes*

A **process** is an activity that is carried out by an actor in order to transform products. A *goal-driven* process, i.e. a process that tries to make its output products comply with specific *design requirements* [MYLOPOULOS,92], will be called an **engineering process**. Most generally, a process is made up of a series of operations, that are processes. Atomic processes, that is to say, processes that comprise a single operation, are called **primitive processes**. A primitive process is simple enough to be considered basic. It can be performed automatically using the correct tool. A primitive process is a single step on the path towards the goals of an engineering process. For instance, producing the SQL-DDL script of a data-

base is an engineering process. During this process, defining the type and length of a single column is a primitive process.

- *Histories*

For several reasons developed in Chapter 6, it is interesting to store a trace of every operation performed during each process. A **history** is the recording of everything that happens during the life cycle of an engineering project. We will see later on that this trace needs to be readable, formal, correct and complete. The history also includes all the products that are used or produced during the whole project. Finally, all the rationales according to which the processes have been carried out are part of the history too.

- *Methods*

When a process is performed, it follows a predefined commonly agreed upon *way of working*, called a **method**. From the seventies to the beginning of the nineties, a lot of method were developed (Merise [COLLONGUES,89] for instance) and published in the literature. More and more companies tried to adopt such methods, but a predefined method is generally perceived academic and not well adapted to the industrial world that often requires customised methods. To adapt a company way-of-working, or culture, to a particular method generally leads to failure. It is much better to attempt to adapt the method to the specific needs of the company. This thesis will show how one can define or adapt a customised method. To define a method, we have to precisely define the properties of two categories of components: its products and its processes. More precisely, we will define a method by an arrangement of *product types* and *process types.*

- *Product type*

A **product type** describes a class of products that play a definite role in the system life cycle. A product is an instance of a product type. For example, the Library Personnel Interview Reports is a product type. Every single interview report is an instance of this type.

- *Process type*

A **process type** will describe the general properties of a class of processes that have the same purpose, and that use, update or generate products of the same types. These general properties will have to include the list of product types to transform and the list of expected resulting product types, as well as a *strategy* to follow. A process is an instance of a process type. Engineering processes will be described by **engineering process types,** and primitive processes will be described by **primitive process types**. For instance, the SQL-DLL code production for the library management database design is an instance of the general *SQL Script Design* engineering process type which tells what type of product have to be generated and how. The specification of each column data type is an instance of the *Column Data Type Definition* primitive process type which proposes a list of valid data types.

- *Process strategies*

The **strategy** of an engineering process type specifies how any process of this type must be, or can be, carried out in order to solve the problems it is intended to, and to make it produce output products that meet its requirements. In particular, a strategy mentions what processes, in what order, are to be carried out, and following what reasoning. For example, the strategy for our SQL Script Design process can state that the database engineer must (1) create the database itself, (2) create all the tables, (3) declare all the columns in every table, (4) specify each column's data type, (5) declare primary identifiers, (6) declare foreign keys and (7) declare other constraints.

Primitive process types are basic types of operations that will be performed by the analyst, or by a CASE tool. They have no associated strategy. They can be classified into four categories according to the level of automation and user involvement:

1. **Basic automatic process types**. Such a process is context-free and does not require any parameters nor configuration settings. The *new* entry in the *file* menu of any application is such a process type.
2. **Configurable automatic process types**. The effect of such a process depends on general settings defined at method definition time. It is specific to a definite design environment and can be considered a part of the *culture* of the organisation. For example, the spelling checking facility of every word processor does its job automatically when the right dictionaries are installed.
3. **User configurable automatic process types**. These automatic processes need to be user configured before each activation. Process types that can still be executed automatically but which needs to be configured before each use, by the user himself or herself. For instance, each document photocopying session requires to manually set the correct number of copies, the contrast, the zoom factor, and the paper size before proceeding.
4. **Manual process types**. A manual process is carried out by the user, possibly with some ancillary help from the tool. The interpretation of interview reports when drawing a raw conceptual schema is an example of such a process type. Most generally these processes encompasses the knowledge-based user activities that cannot be carried out by tools. However, in order to perform a manual process, the database engineer needs some basic tools.

• *Basic tools and toolboxes*

A **basic tool** is a primitive function of the supporting CASE tool. Tools can be grouped to form a **toolbox**. To each process type of the fourth group, a toolbox is associated by the method engineer. To perform a process of one of these types, the database engineer can use any tool from the associated toolbox. For instance, the drawing toolbox can contain a pencil, a ruler and an eraser.

• *Product models*

We have defined the notion of product type which allows us to define a class of products that plays a definite role in the current method. Two product types can appear in a method though their model can be the same. In the *Conceptual Analysis* process, for instance, several conceptual schema types can be identified: the partial raw conceptual sub-schemas, the normalised conceptual sub-schemas, the integrated conceptual schema, the sub-system views, etc. All of them are made up of the same set of building blocks and assembly rules, namely some variant of the Entity-relationship model or of the UML class model (e.g., through a *conceptual profile*). Hence the concept of **product model**. A model defines a general class of products by stating the basic components they are allowed to be included and the assembly constraints that must be satisfied. A product type is expressed into a product model. For instance, all the interview reports of the Library Personnel Interview Reports type have to be written using the same Interview Report Form model which states that interview reports must have a date, references to the project, the interviewer, the interviewee, and a series of sections having a subject and the comments of the interviewee about the subject.

• *Hypotheses, versions and decisions*

User configurable and Manual process types need some human interaction to be performed. This is generally due to the need for some intelligence or some knowledge that supporting tools do not have. But, even database engineers can lack knowledge, so that they may face a problem they cannot solve straight away with certainty. They have different ideas of solution they want to explore. Each idea is developed into a design branch, leading to a definite solution. These solutions can then be evaluated and compared, one of them generally being chosen as the best fitted. Each solution results from a restriction of

the problem domain through **hypotheses**. By stating different hypotheses, an engineer can define several contexts and solve the problem in each of them. Each resulting product is in fact a different **version** of the final product. By comparing all these versions, an engineer can take the **decision** of keeping or rejecting each of them. The hypotheses, the product versions obtained from these hypotheses and the final decisions all have to be kept in the history. For example, an analyst trying to draw a conceptual schema on the basis of interview reports can have problems with the interpretation of some sentences. It is not clear whether the keywords characterising a document of the library have to be stored separately or not. The analyst can make both hypotheses independently and solve the problem twice. When the job is finished two schemas versions are produced: “library/several keywords” and “library/one keyword line”. The analyst can see the interviewee again with both solutions printed on paper to discuss of the best choice, then store in the history the decision to keep the “library/several keywords” version and the rationale of this decision: “These keywords will serve to search for the documents.” During the remaining of the project, the other version of the schema will no longer be used, but it is not discarded and is kept in the history.

Another kind of **decision** can be imposed by the strategy of a process type. This kind of decision can be necessary to decide of doing one action or another, or to decide how many times something has to be done, as it will be shown in Chapter 4.

2.2. Architecture

The concepts described in the previous section are shown in Figure 2.1, together with their relationships. This concept architecture comprises three levels, namely: the instance level, the type level and the model level.

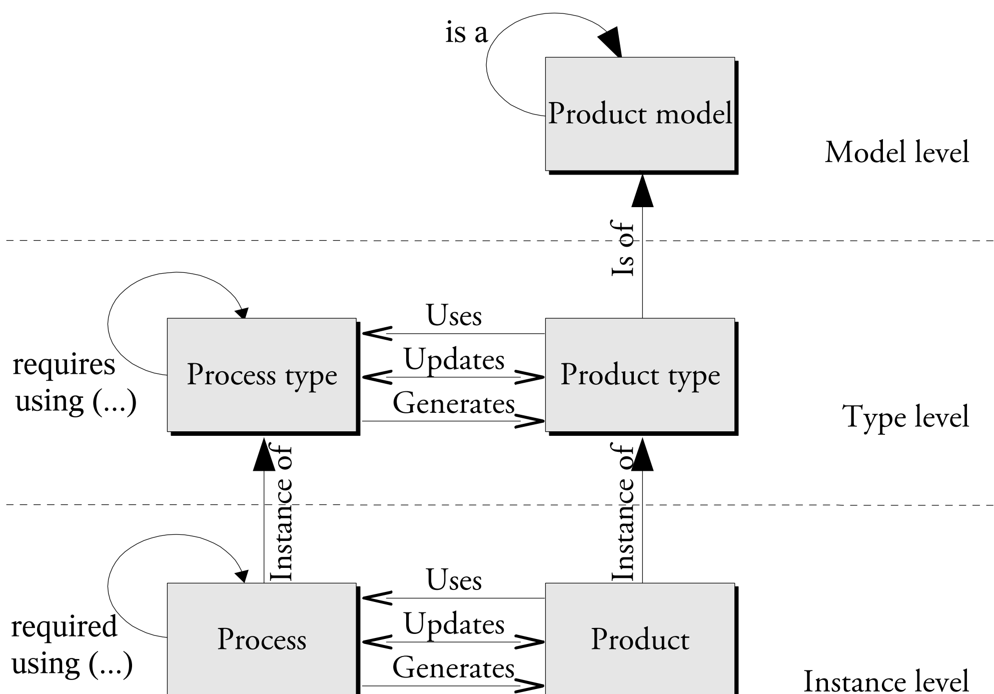


Figure 2.1 The architecture

The **instance level** contains the processes and the products used, generated and updated by the processes. The level comprises the objects of the history. A process is either an engineering process or a primitive process. The products are used, generated, or modified by these processes. To be performed, some processes required that other processes be performed.

These sub-processes use or update some products given by the calling process. The sub-processes can also generate some products and give them back to the calling process. The hypotheses formulated by an engineer are attached to the processes performed in the context they define. All the processes leading to different versions of a same product use the same input products. These different product versions form a series of products of a same type; they are given the same name with different version names. A decision to reject to product versions is a special kind of process: instead of generating or updating a product, it merely *designates* products among a collection of input product (generally different versions of a product). In standard terms, a decision "*uses*" the input products from which it will select a subset. We consider that the process "*updates*" the selected products. A decision imposed by the strategy of a process type is a special kind of process too, which only "*uses*" some products to evaluate an expression whose result determines what processes will be performed later.

The **type level** describes process types and product types that form a method. The description of each process type comprises its interface and its strategy. The interface is made up of the types of the products that are used, updated or generated by the processes of this type. Each process of the instance level is an instance of a process type. In the same way, each product is an instance of a product type. So, an integrity constraint of this architecture is that each product that is used, updated or generated by a process is an instance of a product type linked to the process type of which the process is an instance. This link must play the same function (*uses*, *updates* or *generates*). The performance of a process of one type often requires that some processes of other types are performed using some products of specified types. This process type composition must be compliant with the process composition at the instance level. A decision imposed by the strategy is itself a special kind of primitive process type that updates, or generates, no product type. Since a decision of keeping or rejecting some product versions is taken whenever at performance time, these decisions cannot be prescribed by the strategy, so they cannot appear at the type level.

Figure 2.2 shows the meta-model of the type level (using the ER model presented in Chapter 3). It shows that a product type is either a global product type (its instances can be used by any process) or declared locally to a process type. It also shows that some of the local product types are the formal parameter of the process type. According to the "mode" attribute, these are the product types used, updated or generated by the process type in Figure 2.1. When a process type requires that a process of another type is performed, some parameters are transmitted. These parameter are product types (either local product types or global ones) which must match two by two with the formal parameters of the required process type.

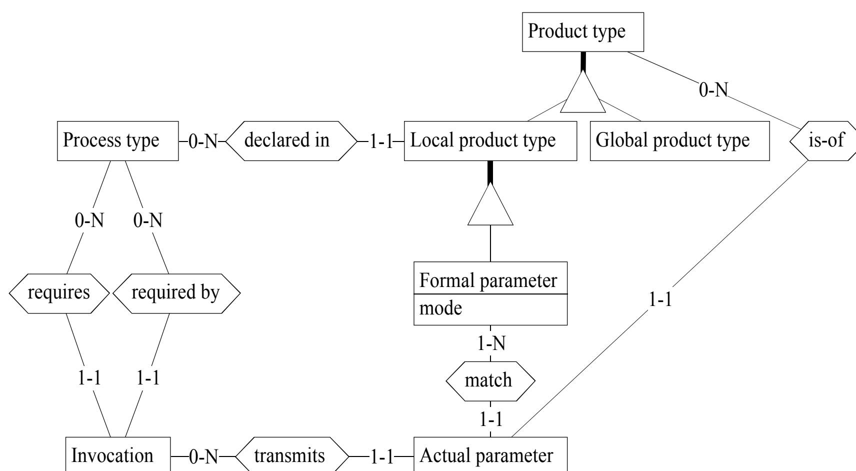


Figure 2.2 The type level meta-model

The **model level** describes product models. Each product type *is of* a product model, i.e., each product of that type, of that class, is built with components pertaining to its associated product model and must be compliant with that model.

This architecture obviously is asymmetrical: *there is no process model*. Such a model could have been defined and certainly would be useful. It would allow a method engineer to define, e.g., a *Logical design* process model which could be specialised into *Relational logical design* or *Object logical design* process types. We think that the effort would not be justified by an increased productivity of the method engineer, at least in the database realm. Therefore, in the limited context of this thesis, we have left it for further research. [JAMART, 94], [ROLLAND,95], [DOMINGUEZ,97] and [HARANI,98] present an architecture with process models (called process meta-models) that allow the design of methods for various domains of activities, the process (meta-)model allowing to define the concepts (dependent on the domain of activities) that can be used by the process types.

The Figure 2.3 suggests an illustration of the this concept architecture. The *C++ programs* model is a text model that specifies the syntax of C++ program files. *Main* and *GUI* are particular types of C++ files. The first type contains the core source files of an application, while *GUI* contains all the GUI-related source files of the same application. *Management/2.0* is a particular C++ program source file that contains the main procedure of a management module. *Management screen* is a file including all the procedures required for displaying the management module main screen. In the same way, *General Ledger* and *Personnel* are two instances of the *Conceptual schema* product type, which is expressed in the *ERA model* product model.

In the same way, Figure 2.4 shows two process hierarchy examples. The *C++ program design* process type has a strategy that was followed by the *Management GUI functions design*. *General Ledger schema design* and *Personnel schema design* are two conceptual schema designs performed with the same pattern described by the *Conceptual schema design* type.

Figure 2.5 shows an example of a very simple project combining the product and the process hierarchies, compliant with the architecture shown in Figure 2.1.

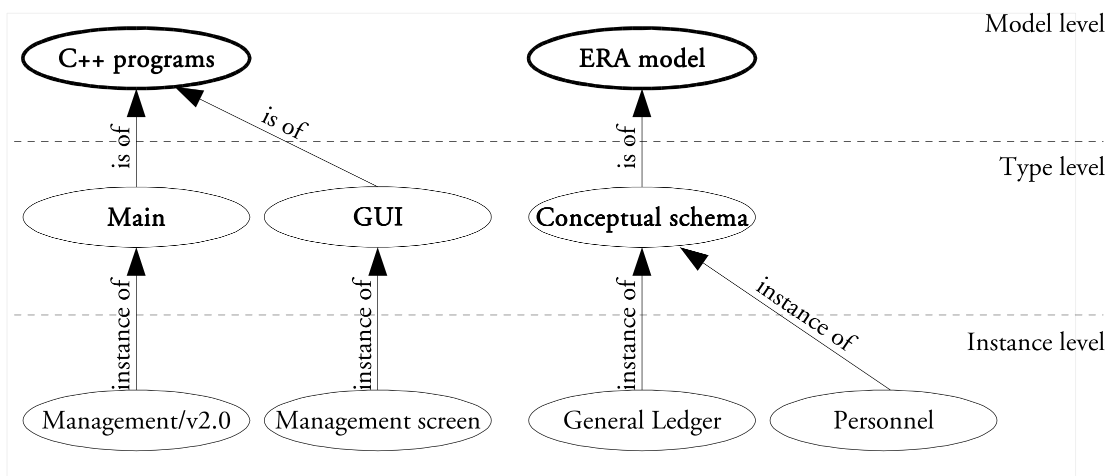


Figure 2.3 Two examples of product hierarchies

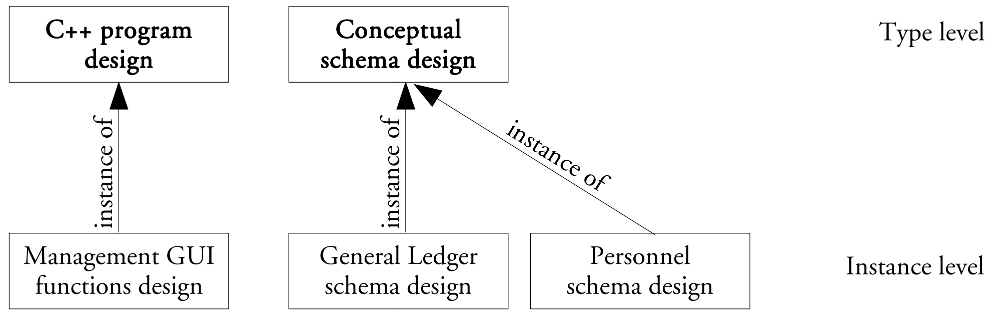


Figure 2.4 Examples of process hierarchies.

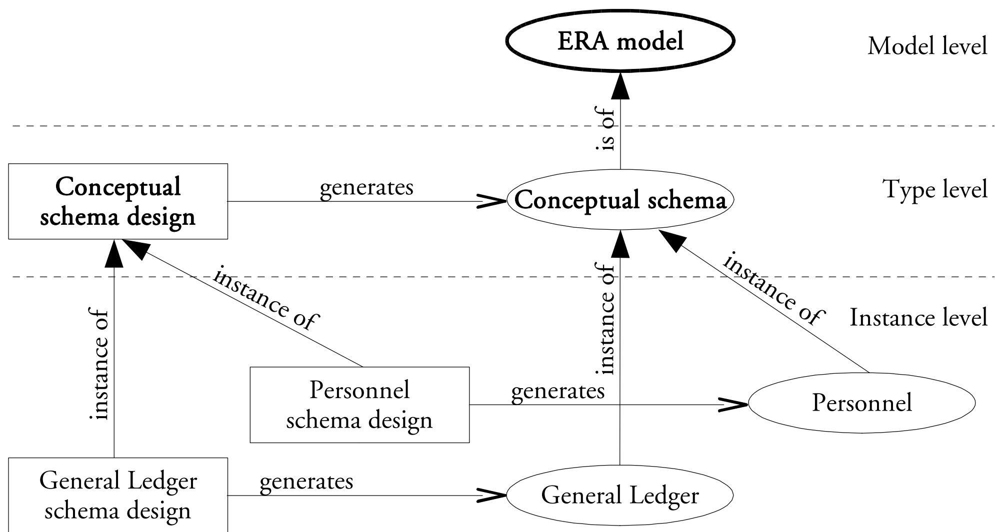


Figure 2.5 A complete example

Chapter 3

Product models

In this chapter, the notion of product model will be precisely detailed, as well as a way to define some of them. The GER (generic entity relationship schema) model will be presented as a reference schema model. It will be used to define particular database schema models by renaming its concepts and stating a series of constraints on these concepts. Text models will also be defined with a regular grammar. Finally, an inheritance mechanism to express product models as sub-models of other product models will be developed.

3.1. Basic considerations

An in-depth analysis of several database engineering methods exhibits both strong similarities and many specific aspects. What makes them similar, among others, is that, at each level of abstraction, they rely on some variant of popular specification models. However, instead of adopting such models as off-the-shelves components, most methods redefine and customise them according to the needs, culture and available technology of the business environment. In some sense, there are as many ERA, NIAM and UML models as there are organisations that use them. Product models are to be considered as a way to precisely define what is exactly intended by each model used by the organisation. In particular, they define the concepts, the names to denote them and the rules to be used to build any compliant product.

In the context of database engineering, we define two main kinds of models, namely *schema models* and *text models*.

A **schema model** allows designers to specify data/information structures. The ER model proposed by Bachman in the late sixties, inspired by the pioneer DBMS IDS [BACHMAN,69] and popularised by Chen [CHEN,76] is such a model. The general ER model (GER) developed in the LIBD[3] and implemented in the DB-MAIN CASE tool is an extension of the ER model. This wide-spectrum GER model is intended to describe data/information structures at different abstraction levels and according to the most popular paradigms [KIM,95] (Figure 3.1).

A personalised schema model will be defined as a *specialisation* of a the GER model. The GER model will be described in detail in Section 3.2, and some tools to specialise it will be built in Section 3.3.

Abstraction levels	Representation paradigms
Conceptual	ERA, Merise Merise-OO, Chen, NIAM, OMT, Booch, Fusion, UML, etc.
Logical	Relational, network, hierarchical, standard files, OO, XML Schema, etc.
Physical	ORACLE 9i, SYBASE, IMS, IDS2, O2, Microfocus COBOL, Java, XML, etc.

Figure 3.1 The 2-dimension scope of the GER model

A **text model** allows designers to specify every other kinds of information. Indeed, text files appear in many forms ranging from computer language source files with a very strict syntax to filled forms, and to natural language texts. Some of these text can be rapidly examined in order to analyse their structure:

- A C++ source file is made up of function declarations. A function is prefixed by a header and an opening curly bracket, it is made of statements, and it is terminated by a closing curly bracket. A header is made of a name and parameters, the parameters being put between parentheses and separated by commas. A statement is made of keywords, variables, constants and other symbols, and is terminated by a semi-colon. Keywords, function names, variables, constants, punctuation marks and other symbols are all made of characters which are classifiable in different sets: figures, letters, punctuation marks, mathematical symbols,...

3 LIBD: Laboratoire d'ingénierie de bases de données, database engineering laboratory, computer science department, university of Namur.

- An XML file is a text containing mark-ups and character data. A mark-up is a string enclosed between angle brackets <...>, and character data are the strings not surround by < and >. An XML file is made up of elements. An element starts with a start tag which is a mark-up and ends with an end tag which is another mark-up whose content is prefixed by a slash /. An element has a name, which appears in both the start and the end tag, and possibly attributes, which can be given a value in the start tag. All the character data and elements between the start tag and the end tag is the content of the element. XML being a kind of text descriptor, the result of the interpretation of an XML file is itself a text file with any other syntax.
- A form is made of sections. A section has a title and is made of questions and answers. A question and an answer are made of words, numbers or items, and punctuation marks. Items are made of words and numbers, and are prefixed by check marks. Words are made of letters.
- A text written in natural language is made of paragraphs. A paragraph is made of words and punctuation marks. A word is made of letters.

There are obvious similarities among all these text variants. Their structures can be described in a hierarchical way, each element being made of a sequence of sub-elements. In fact, all these texts are written according to a particular grammar. So, a text model can be described by describing the grammar with which it complies. As described above, XML itself could be used to describe the grammar of all the texts.

In most computing environments such as DOS-based or Windows-based, file names have an extension. This extension is content-based: it specifies the family of programs that are allowed to process the file. In other words, each file extension is associated with a particular grammar and with the processors that understand it. For instance, the "RTF" extension refers to word processors that understand the RTF grammar (e.g., Star Office, FrameMaker, MS Word, etc.)

Section 3.4 will show how a text model can be described by defining its grammar or simply by giving a list of associated file extensions.

3.2. The GER model

The GER model is a Generic Entity/Object-Relationship model that has been defined in [HAINAUT,89], and that has been implemented in the DB-MAIN repository. Its most important components are presented in this section.

3.2.1. Schema

A **schema** is a description of a collection of data or information structures. It is made up of the specification of entity types, relationship types, attributes, roles, is-a relations, processing units, collections and constraints. The full name of a schema comprises its name and its version. The schemas of a project have distinct full names. The graphical representation of a schema is shown in Figure 3.2: *TRANSPORT/Conceptual* is the full name of a conceptual schema of a Transport management system. The name of the schema is "TRANSPORT" while its version is "Conceptual".

Figure 3.2 A schema

3.2.2. Entity types

An **entity type** is used to denote objects in two different contexts:

- It can be the representation of a concept of the real world: a person, an invoice, a vehicle,... Such an entity type should be characterised by some properties and links with other entity types.
- It can be a technical data structure which has the same syntactical needs.

Figure 3.3 shows an example of an entity type named *PERSON*. The top compartment contains the name. The second compartment contains some attributes that characterise the entity type. The third compartment contains various constraints. The bottom compartment contains some processing units applicable to the entity type. Only the first compartment is mandatory, while the others are independently optional. These attributes, constraints and processing units are examined hereafter.

PERSON

PersID
Name
First name
Address
  Street
  Number
  Box[0-1]
  Zip code
  Town
  Country
Phone[0-5]

id: PersID
Validate()

Figure 3.3 An example of entity type

3.2.3. Relationship types (rel-types)

A **relationship type**, also called a **rel-type**, is a link between two (binary rel-type) or more (**n-ary** rel-type) entity types. An entity type plays a *role* in the rel-type as explained later. For example, in Figure 3.4, *owner* is a rel-type that establishes a link of ownership between a person and a vehicle. Like entity types, rel-types can have attributes, constraints and treatments.

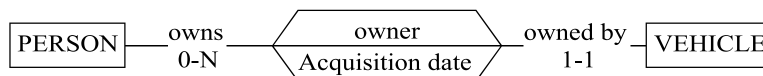


Figure 3.4 An example of rel-type

3.2.4. Attributes

Attributes are some properties that characterise an entity type or a rel-type. For instance, in Figure 3.3, *ID*, *name*, *first name*, *address* and *phone* are attributes of entity type *PERSON*. *Street*, *Number*, *Box*, *Zip code*, *Town*, *Country* are all sub-attributes of the attribute *Address*. In the same way, in Figure 3.4, *Acquisition date* is an attribute of rel-type *owner*. Being made up of meaningful components, *Address* is a **compound attribute**. All the other attributes are called **simple** or **atomic**.

Each simple attribute has a **type** or **domain**. There are three kinds of types, namely basic, object and user-defined.

A *basic domain* is any technical semantic-less value type that is available in most data management system. Some common examples: *numeric*, *char*, *varchar*, *boolean*, *date*, *float,* etc.

An *object domain* is defined as an entity type of the schema. An object attribute takes its values from the instances of another entity type. For example, a *SHAPE* entity type can have an object attribute, named *Colour*, the domain of which is *COLOUR*, itself an entity type of the schema.

A *user-defined* domain (UDD) is given a meaningful name and a definition by the database engineer. The definition of a UDD can be a basic domain, an object domain or even another UDD. In addition a UDD can be compound, just like an attribute. In this case, the list of its components has to be defined. In particular, a component of a compound UDD can be multivalued or optional.

An attribute is given a **cardinality constraint**. It is noted in the form $[i..j]$, where $0 \leq i <$ **N**, $1 \leq j \leq$ **N**, $i \leq j$, where the symbol **N** stands for *infinity*. It means that each instance of the parent (entity, rel-type or attribute) that contains this attribute must have at least i and at most j different values of this attribute. For instance, in Figure 3.3, a *PERSON* should have between 0 and 5 *phone* numbers. When $i = 0$, the attribute is **optional**, otherwise it is **mandatory.** When $j > 1$, the attribute is **multivalued**, otherwise it is single-valued. A multivalued attribute can be organised in several ways: *set*, *bag*, *list*, *unique list*, *array*, or *unique array*.

In a *set*, elements are distinct and un-ordered. For instance, {a,b} = {b,a} is a set. {a,a} is not a set.

In a *bag*, elements are un-ordered, but they can appear several times. {a,a,b} = {a,b,a} = {b,a, a} is a bag.

In a *list*, elements are ordered. (a,b,a) is a list, (b,a,a) is another list.

A *unique list* is a list into which each element appears only once. (a,b) is a unique list, (a,b, a) is not a unique list.

An *array* is a structure made of a given number of cells, each one possibly containing an element, possibly none.

a	b		b

is an array.

A *unique array* is an array into which each element appears only once. For example,

a		b	

is a unique array.

3.2.5. Roles

A **role** is the partnership of an entity type in a relationship type. In Figure 3.4, the role *owns* is played by the entity type *PERSON* in the rel-type *owner*.

The name of the role is optional. The default value is the name of the entity type playing the role. The roles of a rel-type have distinct names. So, when an entity type plays several roles in a rel-type, which is called a **cyclic** rel-type, at most one of the roles can get the default value, all others must be given an explicit name.

The roles have a **cardinality constraint**[4]. It is noted on the form *i-j*, where $0 \leq i <$ **N**, $1 \leq j \leq$ **N**, $i \leq j$, where the symbol **N** stands for *infinity*. The cardinality measures the number of relationships in which an entity participates in this role: this number must be, for any entity, between i and j. For instance, in Figure 3.4, a person may own any number of cars (0-N: minimum 0, maximum an infinity), and a car is the property of exactly one person.

4 The GER uses the *participation* semantics of the cardinality constraint (Merise style), as opposed to the *look-accross* semantics (UML style).

A role can be played by several entity types. Figure 3.5 states that an employee can travel by car, either with his or her own car or with a car of the society, never both. In the first case, the information system only needs to have few information about the car and it does not need to be identified. In the second case, all the cars of the society must be correctly identified, and registered with a series of information for their management.

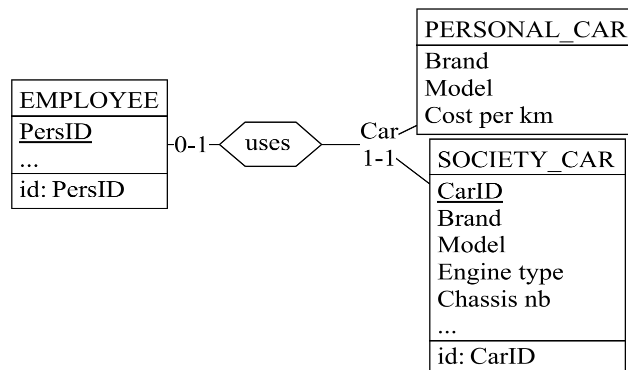


Figure 3.5 A multi-ET-role example

3.2.6. Constraints

In most cases, a constraint is expressed as a relationship holding on a group of components of the object (entity type, relationship type or attribute) on which it applies. A group comprises one or several attributes and roles. It is used either to declare a function played by its component, to state constraints between its components or to describe constraints between its components and the components of another group, possibly belonging to another entity-type or rel-type.

A. Functions of a group

The most common functions of a single group are the following ones:

- **Primary identifier**. Any tuple of component values identifies an entity (or relationship) among all entities (respectively relationships) of the same type. An entity type has at most one primary identifier. It is made up of mandatory attributes and/or roles. In Figure 3.6, each entity type has a primary identifier group, tagged with “id” and composed of a single attribute (*PersID*, *VehiID* and *Name* for entity types *PERSON*, *VEHICLE* and *BRAND* respectively).
- **Secondary identifier**. Like primary identifiers, secondary identifiers identify an entity (or a relationship) among all entities (respectively relationships) of the same type. An entity type (relationship type) can have zero, one or many secondary identifiers. They are made up of mandatory or optional attributes and/or roles. In Figure 3.6, the entity type *VEHICLE* has a secondary identifier made up of the role played by the entity type *PERSON* in the rel-type *OWNER*, and the attributes *brand* and *model*. It is tagged with “id’ ”.
- **Attribute identifier**. In a compound multi-valued attribute, any tuple of sub-attribute values identifies a value of the compound attribute among all its values for a single parent (entity type, rel-type or compound attribute) value. In Figure 3.6, the attribute *Importer* as a primary identifier made of its sub-attribute *ImpID*. It shows that each *Importer* value of a *BRAND* value is identified among all *Importer* values for the same *BRAND* value.

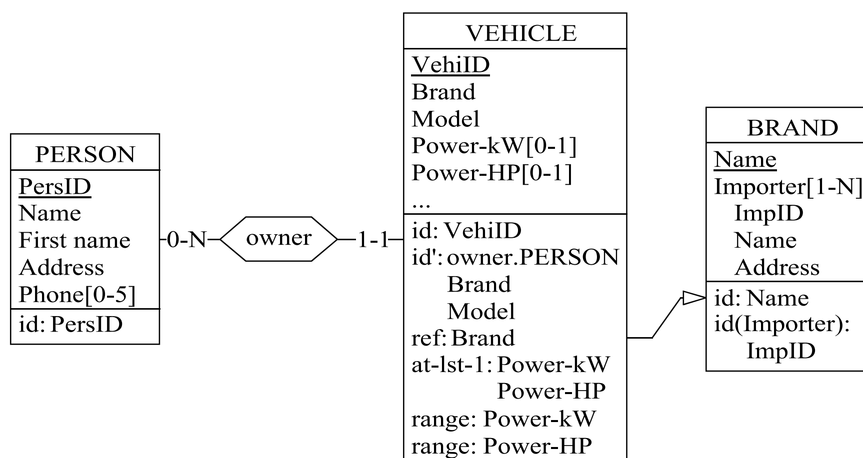


Figure 3.6 Examples of constraints

- **At least one**. Holding in a group of optional attributes and roles only, this constraint states that any entity must have a value in at least one of the components. In Figure 3.6, the attributes *power (kW)* and *power (HP)* of entity type *VEHICLE* are both optional, but the group tagged with “at-lst-1” shows that, for any vehicle, at least one of those two attributes must have a value. It is nevertheless permitted to give a value to both attributes.
- **Exclusive**. Holding in a group of optional attributes and roles only, this constraint specifies that no more than one of the optional components can have a value.
- **Exactly one**. Holding in a group of optional attributes and roles only, this constraint specifies that one and only one of the optional components must have a value.
- **Coexistence**. Holding in a group of optional attributes and roles only, this constraint specifies that any entity must have a value for every components of the group, or for none of them.
- **Access key**. Though it does not relate to integrity, this function simply expresses a technical property: an *access key* is a group that specifies that the tuple of components can be used to get fast access to the entities with these values. Access keys are meaningful in physical schemas, where they represent such constructs as *indexes*.
- Besides these predefined constraints, user-defined constraints can be defined as well. In Figure 3.6, groups tagged “range” (user defined tag) define valid ranges of values.
- Any group that has been assigned no function (so far) is tagged with symbol “gr”.

B. Constraints between groups

The second class of constraints defines relationships between two groups that can belong to different objects. The most common inter-group constraints are the following.

- **Reference constraint**. The referencing group references the referenced group. The referenced group (the target of the constraint) has to be an identifier (primary or secondary) of its parent entity type. The referencing group (the origin of the constraint) should have the same structure (same length and same type for all the corresponding components) as the referenced group. The values of the components of the referencing group in an entity identify an entity of the referenced entity type. In Figure 3.6, the entity type *VEHICLE* contains a reference group, tagged with “ref”, made up of one attribute which references the entity type *BRAND*. The attributes *brand* of *VEHICLE* and *name* which identifies *BRAND* are both strings of the same length. The entity type

BRAND is a dictionary of all known vehicle brands with their importers, and each value of the *brand* attribute must match an entry in that dictionary.

- **Equality constraint** It is a special kind of reference constraint in which every entity of the referenced type must be referenced as well. In Figure 3.6, if the reference constraint is replaced by an equality constraint, there must be at least one vehicle of every brand in the database. Graphically, the sole difference is the tag that becomes "equ".
- **Inclusion constraint.** It is a generalisation of the reference constraint in which the target group does not need to be an identifier; it shows that every instance value of the origin group must be an instance value of the target group. For instance, Figure 3.7 shows two entity types *GEAR* and *VEHICLE* which both have a *Chassis type* attribute. We can note that several gears can exist for the same chassis type and that several vehicles can have the same chassis type too. The inclusion constraint between the group of *GEAR* tagged with "inc" and the group of *VEHICLE* tagged with "gr" shows that the stock can only contain gears suitable for some of the vehicles of the company.

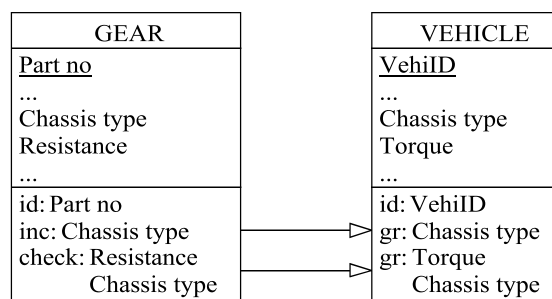


Figure 3.7 Examples of constraints between groups

- **Inverse constraint.** In an object oriented schema, if an entity type *A* (let us call it an object type), has an object attribute of domain *B*, *B* being itself an object type containing an object attribute of domain *A*, then either:
  - If both object attributes are single valued and identifiers of their respective object type, then an inverse constraint between these attributes shows that if *a* is an object of type *A* and if *b* is an object of type B such that the value of the object attribute of *a* is *b* then the value of the object attribute of *b* must be *a*, and reciprocally. This is a kind of one-to-one relationship between *A* and *B*.
  - If the object attribute of *A* is single-valued and non-identifying, and the object attribute of *B* is multi-valued and identifier of *B*, then an inverse constraint between these attributes shows that if *a* is an object of type *A* whose object attribute has the value *b*, then *b* must be an object of type *B* whose object attribute has a set of values containing *a*. Moreover, if *b* is an object of type *B*, each value a_i of its object attribute is an object of type *A* whose object attribute has the value *b*. This is a kind of one-to-many relationship between *A* and *B*.
  - If both object attributes are multi-valued and non-identifying, then an inverse constraint between them shows that if *a* is an object of type *A*, each value b_i of its object attribute is an object of type *B* whose object attribute value is a set containing *a*, and reciprocally. This is a kind of many-to-many relationship.

In Figure 3.8 two object types *EMPLOYEE* and *OFFICE* are referencing each other. An *EMPLOYEE* has, among other attributes, an *Office* object attribute of domain *OFFICE* which is single-valued. An *OFFICE* object has an *Occupier* multi-valued object attribute of domain *EMPLOYEE* which is a secondary identifier of *OFFICE*, showing that an

employee occupies no more than one office. The inverse constraint (tagged in both intervening groups with “inv”) shows that if two *EMPLOYEE* instances with *PersID* values of 522 and 635 have the same *Office* value CS,216, then the value of *Occupier* attribute of the *OFFICE* instance CS,216 must be the set of *EMPLOYEE* instances {522,635}, and no other *OFFICE* instance can have *EMPLOYEE* instances 522 or 635 as an *Occupier* value. And if an *OFFICE* instance identified by MD,312 has an *Occupier* value of {128,265}, then both *EMPLOYEE* instances with *PersID* values of 128 and 265 must have *EMPLOYEE* instance MD,312 has their value of *Office*.

- **Generic constraint**. A user-defined inter-group constraint can be defined, bearing a user-defined semantics. In Figure 3.7 a generic constraint tagged with symbol “check” is intended to assert that the *Resistance* of a gear is sufficient for the *Torque* of at least one of the vehicle which has the same *Chassis type*.

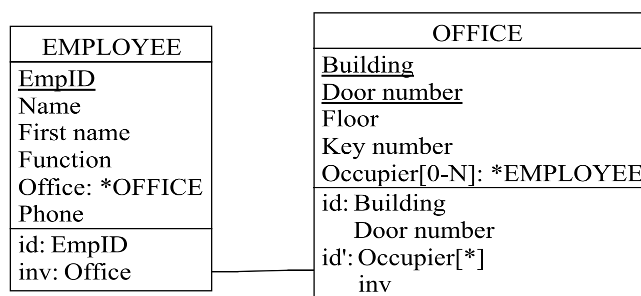


Figure 3.8 An inverse constraint example

3.2.7. Is-a relations

An **is-a relation** is a direct generalisation/specialisation structure between entity types. In Figure 3.9, the is-a relation between *EMPLOYEE*, *CUSTOMER* and *PERSON* expresses the fact that an *EMPLOYEE* (or *CUSTOMER*) entity is also a *PERSON* entity; an employee has an *id*, a *name* and a *salary*. Since the attribute *id* identifies a person, it also identifies an employee or a customer. *EMPLOYEE* and *CUSTOMER* **inherit** the properties (attributes, groups and processing units) from *PERSON*. *PERSON* is called a **super-type** of *EMPLOYEE* and *CUSTOMER*, while *EMPLOYEE* and *CUSTOMER* are called the **sub-types** of *PERSON*.

In Figure 3.9, there is no constraints on the is-a relation. So, a person can be either an employee, or a customer, or both, or none of them. But some constraints can be stated; an is-a relation can be:

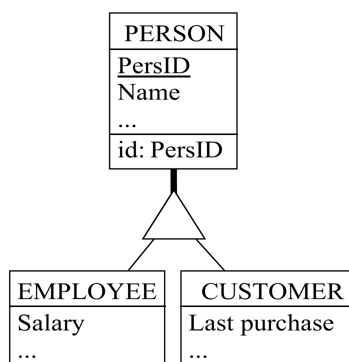


Figure 3.9 An is-a relation

- **total**, every person is at least an employee or a customer (possibly both). It is represented by the letter "T" in the triangle
- **disjunction**, a person can be an employee or can be a customer, but never both (possibly none of them); a letter "D" will represent the disjunction
- **partition**, a person is either an employee or a customer, but not both of them; a partition is a total disjunction. It is represented with the letter "P".

3.2.8. Processing units

A **processing unit** is anything that handles the data stored in entities, relationships, or in a whole schema. It can be a function, a procedure, a method, a script, a "to do" list,... For instance, it can be an SQL *check* that validates the data before inserting or updating a row in a table. It can also be a reporting function that converts data to make them more readable: the *ReportAuto* function of the *VEHICLE* entity type can convert the value of the boolean attribute *Auto* to the more readable string "Automatic gear" or "Manual gear". In the graphical representations, a processing unit appears in the fourth compartment of an entity type or a rel-type, as in Figure 3.3. It can also be associated with a schema by adding a rectangular compartment under the schema ellipse.

3.2.9. Collections

A **collection** is a structure that allows designers to group entity types together for whatever reason he or she may want. For instance, at the physical level, a collection can represent the notion of file. The designer can then use collections to dispatch SQL tables among files. In Figure 3.10, two collections represent files in a library management system. The designer chose to store the tables *AUTHOR*, *BOOK*, *COPY*, *KEYWORD*, *REFERENCE* and *WRITTEN* which, altogether, represent the inventory of all the books in the single *LIBRARY* file, and the remaining tables which represent the dynamic aspect of the book movements in a second file named *BORROWING*.

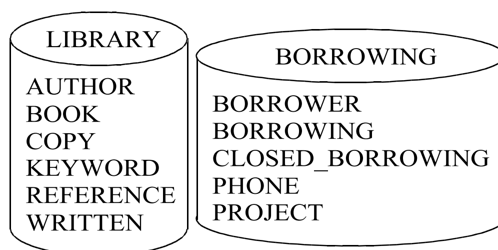


Figure 3.10 A collection

3.2.10. Dynamic properties

Each concept presented above has a series of properties, such as the cardinalities of an attribute or a role, the total and disjoint property of an is-a relation, etc. A **dynamic property** is a user-defined property that can be appended to every concept in each project. This property can be of various types: integer, character, boolean, real or string. It can also be mono-valued or multi-valued.

For instance, one can add two dynamic properties to entity types: *security-level* and *authorisation*. The *security-level* property is an integer whose value can be 0 (accessible and updatable by everybody), 1 (accessible by everybody in read-only mode), 2 (accessible and updatable by authorised persons only), 3 (accessible by authorised persons only in read-only mode) and 4 (accessible and updatable by system administrators only). *Authorisation* is

a multi-valued string property whose values are the ids of the people who can access entities of each type. In Figure 3.6, we can give values to these new properties for all entity types:

PERSON: *security-level*: 0, *authorisation*: ()
VEHICLE: *security-level*: 2, *authorisation*: (Smith,Johnson,Jones)
BRAND: *security-level*: 3, *authorisation*: (Jones)

3.3. Schema model

Let M be a schema model. M is a specific model used in a particular context, such as the data model of a target DBMS or the proprietary conceptual model of a particular company. In the same way as the GER model was described, M can be defined by a set of concepts and their assembly rules. Since the GER has been designed to encompass the main constructs of the practical models used in data engineering, M can be defined as a *subset* of the GER.

More precisely, M will be defined by:

1. Selecting the subset of the concepts of the GER that are relevant in the modelling domain of M
2. Renaming the selected concept according to the modelling domain of M
3. Defining the specific assembling rules of M. In other words, some constraints can be specified on the way the selected concepts can or cannot be used, by themselves or in their association with other concepts.

For example, a logical relational schema comprises tables, columns, primary keys, foreign keys and triggers. So, for expressing relational schemas, we define a *Relational model* as follows.

The most straightforward representation of a table is the GER entity type. A column will be represented by an attribute, a primary key by a primary identifier, a foreign key by a reference group. A unique constraint is best expressed by a secondary identifier while a trigger is a special kind of processing unit attached to the entity type of its table.

The following table describes these mapping rules: all the selected concepts of the GER in the left column, and their relational name at right.

Concept	*Name*
entity type	table
simple attribute	column
primary identifier	primary key
secondary identifier	unique
reference constraint	foreign key
processing unit	trigger

Then we specify the assembling rules that define valid relational schemas, including the following:

- A schema includes at least one entity type.
- A schema includes no relationship types.
- A schema includes no is-a relations.
- An entity type comprises at least one attribute.

- Attributes are simple (atomic).
- Attributes are single-valued.
- An entity type has at most one primary identifier.
- A primary identifier is made up of mandatory (i.e., with cardinality [1-1]) attributes only.
- A reference group and its target identifier have the same composition (their components have same type and length, considered pairwise).

It must be noted that these rules express *restrictions*, in that they state properties that cannot be violated. In other words, any schema obeys model M if,

- it comprises no GER objects but those that have been explicitly selected
- it comprises all the possible GER assembly, but those that are prohibited by the rules.

Therefore, these rules will be called constraints from now on. In this section, a subset of these constraints will be described, classified by object types. The constraint will be written in a predicative form to define **structural predicates**. Each structural predicate will be described with its name, its parameters and a short description. The complete set of structural predicates is proposed in Appendix A. Finally, the predicates will be assembled to form more complex constraints.

A. Constraints on a schema

The first set of constraints concern the nature and the number of the components of the current schema. The first constraints will be commented in detail. Many other constraints are built on the same pattern, and have to be interpreted in the same way.

A first constraint concerns the number of entity types that can be used in a schema. In the example above, every relational schema should have at least one entity type. But an upper limit can also be set on the size of a schema, for example because a particular DBMS cannot handle more than a given number of tables. So a constraint can be defined, let us call it ET_per_SCHEMA, to specify the number of entity types that can/must appear in a schema. It can be written in a predicative form:

ET_per_SCHEMA (*min max*)	where *min* is an non-negative integer, and *max* is either an integer not less than *min* or **N** standing for infinity.

This first constraint must be read: The number of entity types *(ET) per schema* must fall in the range [*min-max*].

In the same way, two additional constraints concerning the number of relationship types and collections in a schema can be defined:

RT_per_SCHEMA (*min max*)	The number of rel-type per schema must fall in the range [*min-max*].
COLL_per_SCHEMA (*min max*)	The number of collection per schema must fall in the range [*min-max*].

For example, a relational schema must include at least one table but no relationship types. In addition, the target DBMS imposes a limit of 1,000 tables. Therefore, the model describing the valid schemas for this DBMS will include the constraints,

ET_per_SCHEMA(1 1000)

RT_per_SCHEMA(0 0)

B. Constraints on an entity type

Similar constraints can be used to define valid entity types according to their components, i.e., their attributes, their groups, their processing units and the roles they play in rel-types:

ATT_per_ET (*min max*)	The number of attributes per entity type must fall in the range [*min-max*].
GROUP_per_ET (*min max*)	The number of groups per entity type must fall in the range [*min-max*].
PROCUNIT_per_ET (*min max*)	The number of processing units per entity type must fall in the range [*min-max*].
ROLE_per_ET (*min max*)	The number of roles per entity type must fall in the range [*min-max*].

The richness of the concept of group requires some specialisation of the constraint GROUP_per_ET. Hence the following constraints concerning, respectively, the primary identifiers, all the identifiers, the access keys, the reference groups (foreign keys), the coexistence groups, the exclusivity groups, the "at least one" groups, the inclusion constraints, the inverse constraints, and the generic constraints.

ID_per_ET (*min max*)	The number of identifiers per entity type must fall in the range [*min-max*].
PID_per_ET (*min max*)	The number of primary identifiers per entity type must fall in the range [*min-max*].
KEY_per_ET (*min max*)	The number of access keys per entity type must fall in the range [*min-max*].
REF_per_ET (*min max*)	The number of reference groups per entity type must fall in the range [*min-max*].
COEXIST_per_ET (*min max*)	The number of coexistence constraints per entity type must fall in the range [*min-max*].
EXCLUSIVE_per_ET (*min max*)	The number of exclusivity constraints per entity type must fall in the range [*min-max*].
ATLEASTONE_per_ET (*min max*)	The number of at-least-one constraints per entity type must fall in the range [*min-max*].
INCLUDE_per_ET (*min max*)	The number of inclusion constraints per entity type must fall in the range [*min-max*].
INVERSE_per_ET (*min max*)	The number of inverse constraints per entity type must fall in the range [*min-max*].
GENERIC_per_ET (*min max*)	The number of generic constraints per entity type must fall in the range [*min-max*].

Roles played by an entity type can also be categorised into optional ([0-j]), mandatory ([1-j]), "one" ([i-1]) and "many" ([i-j], j > 1). These categories induce specific constraints similar to those concerning groups.

For example, the definition of relational models could include the following constraints:

ATT_per_ET(1 N)

PID_per_ET(1 1)

INCLUDE_per_ET(0 0)

INVERSE_per_ET(0 0)

GENERIC_per_ET(0 0)

C. Constraints on a relationship type

Like entity types, rel-types can be made of attributes, groups, processing units and roles. So similar basic predicates can be defined:

ATT_per_RT (*min max*)	The number of attributes per rel-type must fall in the range [*min-max*].
GROUP_per_RT (*min max*)	The number of groups per rel-type must fall in the range [*min-max*].
PROCUNIT_per_RT (*min max*)	The number of processing units per rel-type must fall in the range [*min-max*].
ROLE_per_RT (*min max*)	The number of roles per rel-type must fall in the range [*min-max*].

For example, the last constraint applies on the degree of the rel-type, so rel-types can be forced to be binary:

ROLE_per_RT (2 2)

Since rel-types can have groups too, constraints similar to those defined on entity types are available as well:

ID_per_RT (*min max*)	The number of identifiers per rel-type must fall in the range [*min-max*].
PID_per_RT (*min max*)	The number of primary identifiers per rel-type must fall in the range [*min-max*].
KEY_per_RT (*min max*)	The number of access keys per rel-type must fall in the range [*min-max*].
COEXIST_per_RT (*min max*)	The number of coexistence constraintss per rel-type must fall in the range [*min-max*].
EXCLUSIVE_per_RT (*min max*)	The number of exclusivity constraints per rel-type must fall in the range [*min-max*].
ATLEASTONE_per_RT (*min max*)	The number of at-least-one constraints per rel-type must fall in the range [*min-max*].
INCLUDE_per_RT (*min max*)	The number of inclusion constraints per rel-type must fall in the range [*min-max*].
GENERIC_per_RT (*min max*)	The number of generic constraints per rel-type must fall in the range [*min-max*].

D. Constraints on an attribute

The constraints on the schema, entity types and rel-types concern the relations these concepts have with their environment. These are *relationship constraints.* Before defining such constraints on attributes, they can be examined for their intrinsic properties, namely their cardinality and type:

MIN_CARD_of_ATT (*min max*)	The minimum cardinality of an attribute must fall in the range [*min-max*].

MAX_CARD_of_ATT (*min max*) — The maximum cardinality of an attribute must fall in the range [*min-max*].

TYPES_ALLOWED_for_ATT (*type-list*)
The type of an attribute must belong to the list *type-list.*

TYPES_NOT_ALLOWED_for_ATT (*type-list*)
The type of an attribute cannot appear in the list *type-list.*

TYPE_DEF_for_ATT (CHAR *min max*)
The length of a *character* attribute must fall in the range [*min-max*].

TYPE_DEF_for_ATT (NUMERIC *min-len max-len min-dec max-dec)*
The length of a *numeric* attribute and its decimal part must fall in the ranges [*minlen-maxlen*] and [*mindec-maxdec*].

The other constraints describe the relationships attributes have with their environment:

SUB_ATT_per_ATT (*min max*) — The number of subattributes of the attribute must fall in the range [*min-max*]. For example, [*2 N*] means that compound attributes must comprise at least 2 subattributes.

DEPTH_of_ATT (*min max*) — The level (depth) of the attribute must fall in the range [*min-max*]. Attributes directly attached to their entity type or rel-type are of level 1. For example, [*1 2*] means that only two-level hierarchies of attributes are allowed.

Other constraints specify the groups an attribute can be part of: it can appear in a given number of general groups, primary identifiers, reference groups, etc.

For example, the definition of relational models could include the following constraints:

MAX_CARD_of_ATT (1 1)

TYPES_ALLOWED_for_ATT ('CHAR','NUMERIC','FLOAT','DATE')

TYPE_DEF_for_ATT (CHAR *1 255*)

TYPE_DEF_for_ATT (VARCHAR *1 65000*)

DEPTH_of_ATT(1 1)

E. Constraints on a role

A role has an intrinsic property: its cardinalities. Both the minimum and the maximum cardinality of the role can be constrained:

MIN_CARD_of_ROLE (*min max*) — The minimum cardinality of a role must fall in the range [*min-max*].

MAX_CARD_of_ROLE (*min max*) — The maximum cardinality of a role must fall in the range [*min-max*].

The number of entity types that can appear in a role is defined as follows:

ET_per_ROLE (*min max*) — The number of entity types playing the role must fall in the range [*min-max*].

For example, the definition of the Bachman Data Structure Diagram model must include the following constraints, that describe the valid rel-type patterns:

MIN_CARD_of_ROLE (0 1)

MAX_CARD_of_ROLE (1 N)

ET_per_ROLE(1 1)

F. Constraints on groups

The group is a complex and polymorph concept, so that it can be assigned a large set of constraints. The groups will be analysed in their general form first, then all their specialisations will be examined as well.

The only intrinsic property of a group is the function(s) it is allowed to play. The parameter *yn* takes two values, namely **yes** and **no**.

ID_in_GROUP (*yn*)	A group can/cannot be an identifier.
PID_in_GROUP (*yn*)	A group can/cannot be a primary identifier.
KEY_in_GROUP (*yn*)	A group can/cannot be an access key.
REF_in_GROUP (*yn*)	A group can/cannot be a reference group.
COEXIST_in_GROUP (*yn*)	A group can/cannot be a coexistence group.
EXCLUSIVE_in_GROUP (*yn*)	A group can/cannot be an exclusive group.
ATLEASTONE_in_GROUP (*yn*)	A group can/cannot be an *at-least-one group*.
INCLUDE_in_GROUP (*yn*)	A group can/cannot be the origin of an inclusion constraint.
INVERSE_in_GROUP (*yn*)	A group can/cannot be the origin of an inverse constraint.
GENERIC_in_GROUP (*yn*)	A group can/cannot be the origin of a generic constraint.

The relationship properties of the groups that can be constrained concern their components (relationship constraints with the owners of the groups are already defined for the parents). So the global number of components or the number of components of each type can be counted:

COMP_per_GROUP (*min max*)	The number of component of a group must fall in the range [*min-max*].
ATT_per_GROUP (*min max*)	The number of attribute components of a group must fall in the range [*min-max*].
ROLE_per_GROUP (*min max*)	The number of role components of a group must fall in the range [*min-max*].

For example, in a COBOL file, an index (unique or not) can contain only one field:

COMP_per_GROUP (1 1)

The group constraints can be specialised according to the roles the group plays. Identifiers are among the groups deserving the greatest attention. Indeed, the identifier definition can itself differ from one model to another. Furthermore, DBMSs may impose their own constraints on identifiers. For instance, one model could accept identifiers made of multi-valued attributes, while another could refuse them; or one DBMS could refuse identifiers

longer than 128 characters. In some models, the definition of identifiers can vary depending on their parents. For example, a model can accept that an entity type has an identifier made up of compound attributes, while identifiers of multi-valued compound attributes must be made of simple attributes only.

a. Constraints for entity type identifiers

COMP_per_EID (*min max*)	The number of components of an ET identifier must fall in the range [*min-max*].
ATT_per_EID (*min max*)	The number of attribute components of an ET identifier must fall in the range [*min-max*].
OPT_ATT_per_EID (*min max*)	The number of optional attribute components of an ET identifier must fall in the range [*min-max*].
MAND_ATT_per_EID (*min max*)	The number of mandatory attribute components of an ET identifier must fall in the range [*min-max*].
SINGLE_ATT_per_EID (*min max*)	The number of single-valued attribute components of an ET identifier must fall in the range [*min-max*].
MULT_ATT_per_EID (*min max*)	The number of multivalued attribute components of an ET identifier must fall in the range [*min-max*].
COMP_ATT_per_EID (*min max*)	The number of compound attribute components of an ET identifier must fall in the range [*min-max*].
ROLE_per_EID (*min max*)	The number of role components of an ET identifier must fall in the range [*min-max*].
OPT_ROLE_per_EID (*min max*)	The number of optional role (minimum cardinality is 0) components of an ET identifier must fall in the range [*min-max*].
MAND_ROLE_per_EID (*min max*)	The number of mandatory role (minimum cardinality is strictly positive) of the components of an ET identifier must fall in the range [*min-max*].
ONE_ROLE_per_EID (*min max*)	The number of "one" role (maximum cardinality is 1) components of an ET identifier must fall in the range [*min-max*].
N_ROLE_per_EID (*min max*)	The number of "many" role (maximum cardinality is strictly greater than 1) components of an ET identifier must fall in the range [*min-max*].

b. Constraints for relationship type identifiers

A similar list of constraints exists for rel-type groups. The constraint names are suffixed with _RID.

c. Constraints for attribute identifiers

The third list for groups defined on multi-valued compound attributes will be shorter because they can never be made up of roles:

COMP_per_AID (*min max*)	The number of components of an attribute identifier must fall in the range [*min-max*].
ATT_per_AID (*min max*)	The number of attribute components of an identifier must fall in the range [*min-max*].
OPT_ATT_per_AID (*min max*)	The number of optional attribute components of an attribute identifier must fall in the range [*min-max*].
MAND_ATT_per_AID (*min max*)	The number of mandatory attribute components of an attribute identifier must fall in the range [*min-max*].
SINGLE_ATT_per_AID (*min max*)	The number of single-valued attribute components of an attribute identifier must fall in the range [*min-max*].
MULT_ATT_per_AID (*min max*)	The number of multivalued attribute components of an attribute identifier must fall in the range [*min-max*].
COMP_ATT_per_AID (*min max*)	The number of compound attribute components of an attribute identifier must fall in the range [*min-max*].

d. Constraints for primary identifiers

Though primary identifiers form a subset of the identifiers, they may, in some models be assigned specific constraints. For instance, a candidate key in a relational schema can be made up of optional columns, but a primary key comprises mandatory columns only.

The constraints are similar to those described here above, with suffix _EPID for entity type primary identifiers, _RPID for rel-type primary identifiers and _APID for attribute primary identifiers.

e. Constraints for reference groups

Reference groups reference identifiers. So it is logical to want to define reference keys in the same way identifiers were defined. In fact, since reference keys can only be defined on entity types and never on rel-types, nor on attributes, the new list of predicates for reference keys will be defined in the same way as for entity type identifiers:

COMP_per_REF (*min max*)	The number of components of a reference group must fall in the range [*min-max*].
ATT_per_REF (*min max*)	The number of attribute components of a reference group must fall in the range [*min-max*].
OPT_ATT_per_REF (*min max*)	The number of optional attribute components of a reference group must fall in the range [*min-max*].
MAND_ATT_per_REF (*min max*)	The number of mandatory attribute components of a reference group must fall in the range [*min-max*].

SINGLE_ATT_per_REF (*min max*)	The number of single-valued attribute components of a reference group must fall in the range [*min-max*].
MULT_ATT_per_REF (*min max*)	The number of multivalued attribute components of a reference group must fall in the range [*min-max*].
COMP_ATT_per_REF (*min max*)	The number of compound attribute components of a reference group must fall in the range [*min-max*].
ROLE_per_REF (*min max*)	The number of role components of a reference group must fall in the range [*min-max*].
OPT_ROLE_per_REF (*min max*)	The number of optional role (minimum cardinality = 0) components of a reference group must fall in the range [*min-max*].
MAND_ROLE_per_REF (*min max*)	The number of mandatory role (minimum cardinality > 0) of the components of a reference group must fall in the range [*min-max*].
ONE_ROLE_per_REF (*min max*)	The number of "one" role (maximum cardinality = 1) components of a reference group must fall in the range [*min-max*].
N_ROLE_per_REF (*min max*)	The number of "many" role (maximum cardinality > 1) components of a reference group must fall in the range [*min-max*].

f. Constraints for access keys

An access key is a technical property often attached to identifiers and to reference groups, so constraints similar to those in the previous groups can be defined with the suffix _KEY.

g. Constraints for existence constraints

Coexistence, *exclusive* and *at-least-one* groups are simpler properties. Their definition is context independent, so they do not need special refinement.

h. Constraints for inverse groups and user-defined constraints

Inverse groups can only be made up of a single object attribute, so they need no specific constraints. Generic constraints are user-defined. Since their semantics is user-defined as well, and due to the variety of their interpretation, no specific constraints exist for them. A personalised way to do it anyway will be presented later on.

G. Constraints on is-a relations

Is-a relation have two intrinsic properties, namely totality and disjunction:

TOTAL_in_ISA (*yn*)	Totality property is allowed or not.
DISJOINT_in_ISA (*yn*)	Disjoint property is allowed or not.

Relations between their members can be seen as generalisation or specialisation:

SUPER_TYPES_per_ISA (*min max*)	The number of supertypes of an entity type must fall in the range [*min-max*].

SUB_TYPES_per_ISA (*min max*)	The number of subtypes of an entity type must fall in the range [*min-max*].

H. Constraints on names

The name of the components of a schema can be constrained by syntactic rules. This is particularly true for physical schemas, where name formation rules of the DBMS must be strictly enforced.

a. Valid characters and length

ALL_CHARS_in_LIST_NAMES (*list*)	The names must comprise characters from the list *list*.
NO_CHARS_in_LIST_NAMES (*list*)	The names must comprise characters that do not appear in the list *list*.
LENGTH_of_NAMES (*min max*)	The length of a name must fall in the range [*min-ma*x].

b. Reserved and valid words

DBMSs generally impose that special words of the DDL cannot be used to name schema constructs (reserved words) and impose some naming conventions (restricted set of characters for instance).

NONE_in_LIST_NAMES (*list*)	The name of a construct cannot belong in the list of words *list*.
NONE_in_FILE_NAMES (*file*)	The name of a construct cannot belong in the list of words stored in the file *file*.
ALL_in_LIST_NAMES (*list*)	The name of a construct must belong in the list of words *list*.
ALL_in_FILE_NAMES (*file*)	The name of a construct must belong in the list of words stored in the file *file*.

The names in *list* and *file* can be constants (exact words) or expressions in the regular grammar used by the name processing assistant of the supporting CASE tool [DB-MAIN,02b]. For example: “address”, or “?ddr*”. The last example will find all the names with a “d” in second and third places and a “r” in fourth place, whatever the length of these strings.

I. User-defined constraints

Developing a complete predicate list would be unrealistic. Rather, this chapter proposed a list of the main constraints that are relevant in the most widespread models, in legacy, current and future (at least as foreseeable) systems. This pragmatic approach obviously cannot meet all the requirements that could emerge in all possible situations. Hence the need for a more general expression mean to define ad hoc constraints. For that reason the analyst should be able to develop his/her own predicates in the form of functions of boolean type.

It is to be noted that user-defined concepts and user-defined constraints are closely linked to a CASE tool. In the continuity of the predicative syntax used so far to describe the predefined constraints, a logical language, such as OCL (Object Constraint language, part of UML [OMG,01]), could be used to define the boolean functions. But since the use of the DB-MAIN CASE tool is required as a foundation for this thesis, the tools it offers have to be used, namely the Voyager 2 4GL (see Chapter 1), which has proved along the years to be robust and efficient. So, The boolean functions are expressed in the procedural language

Voyager 2. This approach strongly reduces the complexity of the method engine. The main drawback is that no automatic reasoning, for instance about global consistency, can be applied on a set of constraints that includes such functions.

Technically, a new generic constraint is added to each group of concepts:

V2_CONSTRAINT_on_SCHEMA (*voyager-file voyager-function parameters...*)
V2_CONSTRAINT_on_ET (*voyager-file voyager-function parameters...*)
V2_CONSTRAINT_on_RT (*voyager-file voyager-function parameters...*)

and so on with all suffixes: _ATT, _ROLE, _EID, _RID, _AID, _EPID, _RPID, _APID, _REF, _KEY, _ISA, _NAMES. In these constraints, *voyager-file* is the name of the Voyager 2 executable file containing the function *voyager-function* to execute; *parameters* is a single string containing all the parameters to pass to the function, its format being dependant on the function. Since both the file and the function are passed as parameters, a database engineer can build libraries of functions, and to use only the constraint(s) he or she needs for the current model, possibly several for a same concept. The syntax of this constraint is detailed in Appendix A.17 with an example.

For example, in an IMS hierarchical schema, relationship types cannot form cycles. This cannot be expressed with the predefined constraints, but it can be checked by a Voyager 2 function, let us call it *IsThereCycles*, which can be placed in a library called *IMS.OXO*[5]. It does not need a parameter. Moreover, the number of levels in a hierarchy can be measured with a function *HierarchyDepth*, placed in the same library, with two parameters: *min* and *max* to specify that the number of levels in a hierarchy must fall in the range [*min-max*].

V2_CONSTRAINT_on_RT (IMS.OXO IsThereCycles)

V2_CONSTRAINT_on_RT (IMS.OXO HierarchyDepth 1 8)

Furthermore, the user can extend the GER model by defining dynamic properties on every concept. Another group of constraints has been defined on dynamic properties:

DYN_PROP_of_SCHEMA (*dynamic-property parameters*)

DYN_PROP_of_ET (*dynamic-property parameters*)

DYN_PROP_of_RT (*dynamic-property parameters*)

and so on with every other suffix. *Dynamic-property* is the name of a dynamic property defined on the concept corresponding to the constraint suffix, and *parameters* are the parameters whose syntax depends on the property definition. The syntax is detailed in Appendix A.16 with several examples.

For example, let us suppose an integer dynamic property named *security-level* is defined on entity types. We need a constraint to ensure that its value is comprised between 0 and 4 which are the only meaningful values:

DYN_PROP_of_ET (security-level 0 4)

J. Complex constraints

The structural predicates presented so far can be assembled to form complex constraints through the use of the standard *not*, *and* and *or* logical operators. Such a logical expression will be called a **structural rule**. In the same way a structural predicate is a constraint that must be satisfied by each concerned component of a schema, the structural rule is also a constraint that must be satisfied by each component of the schema. The two following examples show two structural rules:

5 .OXO is the standard extension for Voyager executable files.

COMP_per_EID (1 N) and ROLE_per_EID (0 0) or COMP_per_EID (2 N) and ROLE_per_EID (1 N)	for each entity type identifier ID: *either* ID comprises one or several components and comprises no roles, *or*, if ID comprises roles, it must comprise two or more components.
ROLE_per_RT (2 2) or ROLE_per_RT (3 4) and ATT_per_RT (1 N) or ROLE_per_RT (3 4) and ATT_per_RT (0 0) and ONE_ROLE_per_RT (0 0)	for each relationship type R: *either* R comprises two roles, *or* R is N-ary and has attributes *or* R is N-ary, has no attributes and has no *one* (i.e. [0-1] or [1-1]) roles

A complex constraint must satisfy the following rules:

1. all its predicates apply on the same concept. For example, the following rule is valid:

   ATT_per_RT (0 0) and role_per_RT (2 N)

   while the next one is not:

   ATT_per_ET (1 N) and ATT_per_RT (0 0)

   Guessing what the author probably meant, this constraint should be rewritten as:

   ATT_per_ET (1 N)

   ATT_per_RT (0 0)

2. The logical operators have their traditional priority rules. So, *not* operators are executed first, then the *and* operators, and finally the *or* operators. Parenthesis are not supported so every logical formula can be expressed in its *disjunctive normal form* [CHANG,73], that is to say as a disjunction of conjunctions, with the use of distributive laws. For instance, if P, Q and R are predicates,

   P and (Q or R) = (P and Q) or (P and R) = P and Q or P and R

Now, a more comprehensive definition of the relational model can be build. In other words, the set of constraints any RDBMS-compliant schema must meet is:

ET_per_SCHEMA (1 N)	A schema includes at least one entity type.
RT_per_SCHEMA (0 0)	A schema includes no relationship types.
SUB_TYPES_per_ISA(0 0)	A schema includes no is-a relations.
ATT_per_ET (1 N)	An entity type comprises at least one attribute.
SUB_ATT_per_ATT (0 0)	Attributes are simple (atomic). In other words, the number of sub-attribute per attribute is exactly 0.
MAX_CARD_of_ATT (1 1)	Attributes are single-valued. In other words, their maximum cardinality is exactly 1.
PID_per_ET (0 1)	An entity type has at most one primary identifier.
OPT_ATT_per_EPID (0 0)	A primary identifier is made up of mandatory (i.e., with cardinality [1-1]) attributes only.
V2_CONSTRAINT_on_REF (REL.OXO RefConsistency)	A reference group and its target identifier have the same composition (their components have

same type and length, considered pairwise). This complex constraint is checked by a user-defined function RefConsistency.

ALL_CHARS_in_LIST_NAMES (ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz0123456789$_)
and NONE_in_LIST_NAMES(_$,$$)
and LENGTH_of_NAMES(0 31)
and NONE_in_FILE_CI_NAMES
(ResWords.nam)

The names of the components of the schema must be valid:
1. They must be made of letters and figures and symbols $ and _ only
2. They cannot end by the symbols $ and _
3. They cannot be longer than 31 characters
4. They cannot be reserved words of the language, the complete list of these words being in the file ResWords.nam.

3.4. Text model

Like a schema model, a text model can be defined by a selection and renaming of concepts form a general text model, and by a series of constraints on the selected concepts.

In the beginning of this chapter, four typical examples of texts were presented. Their hierarchical structure can be described as follows: each element is made of sub-elements and so on until the smallest elements which are always characters. On this basis, the following *general text model* (GTM) can be build:

A text is a series of *text elements*. Each text element is either a *character* or itself a series of text elements.

To define the structure of a text is to define its grammar. That is to say, each text element has to be defined by specifying its name and its structure. For instance, most programming languages (C, C++, COBOL, Pascal,...) can have their syntax described in BNF format, like the MDL language in Appendix D of this thesis. Each line of a BNF description defines a new element by giving it a name and its decomposition in sub-elements.

Since the most basic elements are characters, the grammar defines elements as a series of characters. These definition are generally a restriction of the character set that can be used. For instance, an integer is only made of figures (0, 1,..., 9). Assembling elements is assembling series of characters. Everywhere an integer must appear, only series of figures can appear. Hence, the grammar is a series of constraints on the use of characters.

Since the number and the structure of elements are dependant on the text format, it is not possible to dissociate the naming conventions from the constraints as we did with schema models. Hence, The whole definition of a text model holds in its grammar.

There are several well known ways to define a grammar. The most common are the BNF notation, regular grammars [AHO,89], or XML [MARCHAL,01]. Since the scope of this thesis is limited to database engineering and to the use of DB-MAIN, and since DB-MAIN offers a *pattern definition language* (PDL) defined by Jean Henrard [HENRARD,98] for text analysis and program slicing purposes that suits all the DB-MAIN text analysis needs, this language can be used for the text modelling needs too. Indeed, choosing another grammar definition language would force us to have a conversion tool from that language

towards PDL.

In PDL, a text element is named a pattern. The grammar is expressed by a series of patterns close to a BNF notation with variables. A pattern is of the form:

> *pattern_name* ::= *expression*

where *pattern_name* is any word beginning by a letter and made of no more than 100 letters and figures, and *expression* describes the syntax of the pattern. The expression can be made of strings, of other patterns, of variables, and of some operators.

OpeningSymbol ::= "begin"

> The expression is a simple string.

Figure ::= range(0-9)

> The expression is a string made of a single character whose ASCII code is comprised between the ASCII code of "0" and the ASCII code of "9".

Space ::= \g"[\t\n]*"

> The expression is string defined by a grep expression, grep being a well-known unix originated tool aimed at searching for a string in a file.

Sequence ::= OpeningSymbol Instructions ClosingSymbol

> The expression is made of a sequence of other pattern names.

Sequence ::= OpeningSymbol @Instructions ClosingSymbol

> The second element of the expression is a variable. It is a pattern name prefixed by "@". A text element which matches this pattern during an analysis is stored for future reuse by some functions of the CASE environment.

Sequence ::= OpeningSymbol [Instructions] ClosingSymbol

> The second element of the expression is optional.

Instructions ::= Instruction*

> The element of the expression can be repeated several times.

Instructions ::= (Instruction Space ";" Space)*

> The series of elements between parentheses can be repeated several times.

ArithmeticOperator ::= "+" | "-" | "*" | "/"

> The expression offers an alternative between several sub-expressions.

The complete language syntax is presented in [DBMAIN,02b] and in Appendix B.

The following is a small complete example of a grammar for writing simple calculus:

```
Figure ::= range(0-9)
Number ::= Figure Figure*
Operator ::= "+" | "-" | "*" | "/"
Calculus ::= Number (Operator Number)* "=" @Number
```

This simple grammar expresses the syntax of a file containing a simple arithmetic calculus with integer numbers. A file containing the following single line is correct with respect to this grammar:

```
12*5+35=95
```

When the syntax of this file is checked with the grammar, the @Number variable is initial-

ised with the value 95, which can be used by the CASE environment.

On the contrary, the following files are not valid:

12*5+35=95 15/5+6=9	*the grammar does not allow several calculus*
1.2 * (5 + 35) = 48	*floating numbers, parenthesis and spaces are not allowed*
95=12*5+35	*operators are only allowed at left side of =*

But the following file is correct because only the syntax is checked, not the semantics:

1=2

In practice, a more realistic grammar is the one of SQL or Cobol. These are much more complicated grammars which lead to much longer PDL descriptions. When reverse engineering a Cobol application, it is necessary to write this grammar in order to allow the CASE environment and engineers who use it to analyse correctly the source files. But, for some different tasks, such as generating an SQL DDL from an SQL compliant schema, which is done automatically by a generator, detailing this precisely the grammar is useless. In fact, it suffices to express the fact that the generated file contains an SQL DDL. In the DOS/Windows based environments, it suffices to know the extension of a file to loosely know what it contains. So, in some circumstances, a text model grammar can be defined by a list of possible file extensions. For instance, it is well known that a ".txt" file contains free text, a ".cpp" file is a C++ file, and a ".sql" file contains an SQL DDL.

3.5. Product model hierarchies

Quite often, product models have concepts and constraints in common. If we consider the class of products associated with each product model, we can define class inclusion relations, that can be modelled by **inheritance**: the fact that any product of model B is also a valid product of model A can be described by stating that model B inherits from model A. If a product model B inherits from a product model A, then B includes all the concepts and constraints of A *plus* its own concepts and constraints. It is also possible for B to redefine a concept of A and to give it another name.

For instance, logical COBOL and logical SQL schema models have some common properties and can be defined with inheritance mechanism. For instance, they both belong to the *record-based* family: all the information is represented by field values collected into records, themselves stored into files or tables.

Onee can define a *LOGICAL-RECORD-MODEL* which specifies that entity types and attributes will be used without renaming and that rel-types will be discarded. Every entity type should have at least one attribute, and attributes can be decomposed up to 49 levels. Then one can define the *LOGICAL-COBOL-MODEL* which inherits from the *LOGICAL-COMMON-MODEL* and that simply renames entity types as *record types* and attributes as *fields*, and which adds a constraint stating that all the records of the same type must be stored in the same file. The *LOGICAL-SQL-MODEL* also inherits from the same model. It renames entity types as *tables* and attributes as *columns*, and it redefines the inherited rule about the number of attribute decomposition levels by limiting this number to 1 (i.e., a column cannot be decomposed).

Practically, these models could be defined in the following way:

LOGICAL-RECORD-MODEL	
entity type attribute	entity type attribute
RT_per_SCH (0 0)	*No rel-types allowed*
ATT_per_ET (1 N)	*At least one attribute per entity type*
DEPTH_of_ATT (1 49)	*Maximum 49 levels of attribute decomposition*

LOGICAL-COBOL-MODEL	is-a LOGICAL-RECORD-MODEL
entity type attribute	record field
COLL_per_ET (1 1)	*All the records of the same type go entirely in a single file*

LOGICAL-SQL-MODEL	is-a LOGICAL-RECORD-MODEL
entity type attribute	table column
DEPTH_of_ATT (1 1)	*No compound column allowed*

In the database realm various kinds of schemas can be encountered:

- Purely data oriented schemas, such as the SQL model defined above, every relational schemas, network schemas, or hierarchical schemas which all use data oriented concepts only (entity types, rel-types, attributes, roles, is-a relations, groups) with their own constraints.
- Object-oriented database schemas, such as Java classes, which also include some treatment aspects: each class having a series of methods for handling its attributes, entity types and rel-types can receive processing units to represent the method descriptions (its name and a few comments about its parameters, its pre- and post-conditions,...)
- If the object oriented schema naturally integrates the concept of treatment, this concept can also be added to the more traditional models of the first kind (like the *check* concept in an SQL model). To do so, the processing unit concept can be added to their models. In this case, processing units can be attached to entity types, to a rel-type, or to the schema (in order to represent global treatments that can concern several entity types and rel-types).

Chapter 4

Product types and process types

This chapter will detail concepts of the type level: product types and process types. Product types can be defined globally, at the method level, or locally to each process type. In the latter case they can serve to interface engineering process types with other process types. Engineering process types have a strategy based on a procedural paradigm which includes traditional constructs such as the sequence, loops or conditional action selection, as well as unusual non-deterministic user-oriented constructs. This chapter analyses the interfacing elements and the strategy building elements in detail.

4.1. Defining product types

A product type describes a class of products that play a definite role in the system life cycle, and that is expressed in a product model. This section shows how a product type can be specified.

With this definition, a product type is the use of a product model in a given context. For example, assuming the *physical SQL* schema model has already been described, the *Oracle 9 physical schema* product type can be defined on the basis of this model whenever this kind of schema needs to be introduced in a definite methodology. Since product types are defined in a particular context (either global to the whole method or local to a process type), some practical information about the instances of this type in this context have to be specified. In particular, the number of products of that type which are necessary and allowed at a precise moment. This precise moment depends on the role (input, output,...) played by the product type in the context. The various roles will be presented in Section 4.2.

Let us consider, for instance, the methodology fragment of Figure 4.1. It tells us that each process *Integrate* (i.e, each execution of the process type *Integrate*) requires some input schemas which are given the schema type *to_integrate* and produces some instances of the schema type *integrated*. In other contexts (not during the execution of a process of type *Integration*), these schemas can be of various other types, all compliant with the same product model. The process can be activated only if it is provided with at least 2 schemas to integrate (2 to N instances of product type *to_integrate*) and will produce exactly one integrated schema (from 1 to 1 instance of schema type *integrated*). Of course, the constraint on the number of products of the output type can only be checked when the process ends. Quite naturally, both *to_integrate* and *integrated* schema types are expressed in the same schema model *ERA Model*. Figure 4.2 shows the instance, type and model specifications.

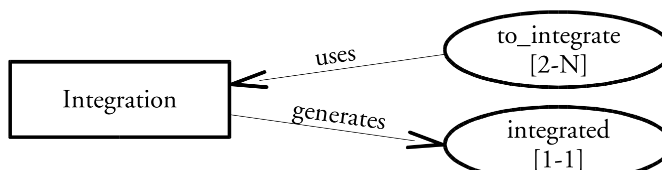


Figure 4.1 A complete example

Hence, both product types can be defined in the following way:

Product type:	to_integrate
Is of:	ERA model
Minimum number of products:	2
Maximum number of products:	N

Product type:	integrated
Is of:	ERA model
Minimum number of products:	1
Maximum number of products:	1

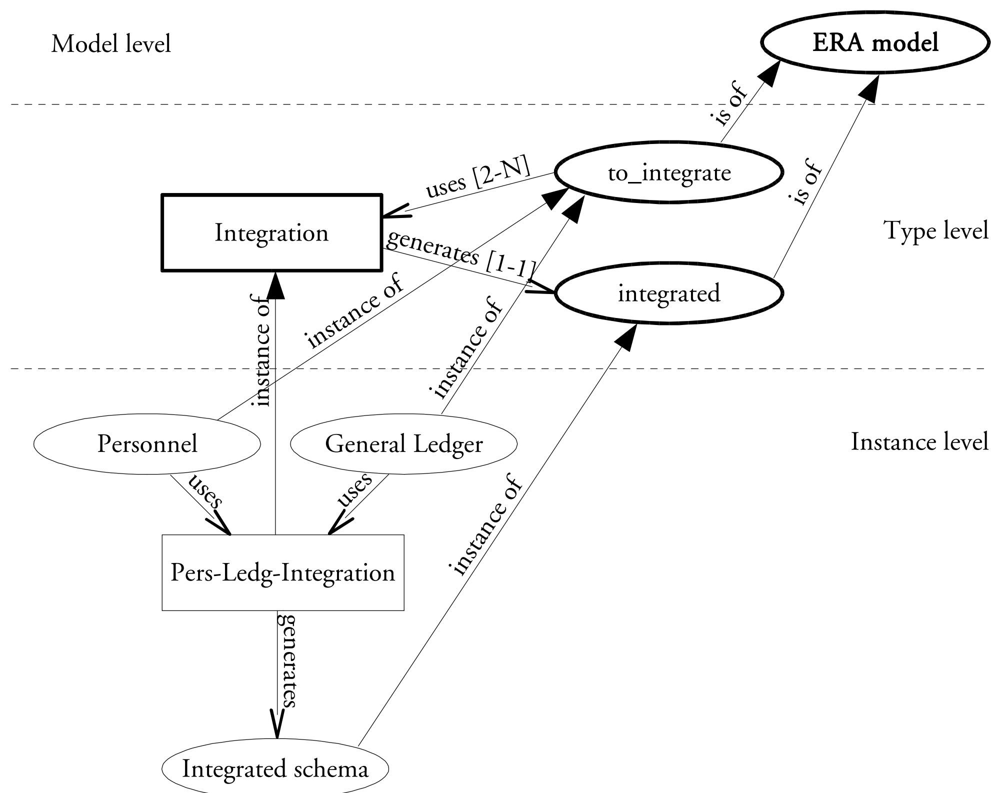


Figure 4.2 A complete example

4.2. Modelling engineering process types

Though a process type can describe a non-procedural behaviour, it is fairly close to the concept of procedure in standard programming languages. In particular, a process type has an *external description*, which states its activation condition and environment as well as its effect (its specification in software engineering terms) and an *internal description*, which states how the effect can be achieved. The external description of a process type will be called its *interface* and its internal description will be called its *strategy*.

Only engineering process types are provided with an internal description. Indeed, primitive process types being built-in functions of the supporting CASE tool, they have to be taken them as they are, i.e., as black boxes with immutable specifications.

4.2.1. Engineering process type decomposition

An engineering process type is the description of a class of processes, which are themselves activities performed in order to reach a given goal. The internal description is often simplified when expressed in terms of sub-process types, each of these sub-processes having its own description. When working with large problems, it is generally recommended to divide them into smaller sub-problems and to solve each of them independently. When designing a method, each sub-problem will be solved by a process type. All these process types will be assembled with control structures to solve the larger problem. Hence, a complex engineering process type can be decomposed in a hierarchy of process types.

For instance, a simple forward engineering database design (*FEDD*) can be decomposed in four main phases (a complete case study using this method is shown in Chapter 11):

1. Conceptual analysis.
2. Logical design.
3. Physical design.

4. Coding.

Then each of these phases can also be decomposed in several steps:

1. Conceptual analysis: problem analysis – conceptual normalisation.
2. Logical design: relational design – name processing.
3. Physical design: index setting – storage allocation.
4. Coding: coding parameters setting – SQL generation.

To go further, the *relational design* process type can be refined in several simpler steps too:

Relational design: is-a relations transformation – non-functional rel-types transformation – attributes flattening – resolving missing identifiers – transformation of rel-types into reference keys.

In this decomposition, *FEDD*, *conceptual analysis*, *logical design*, *physical design*, *coding*, and *relational design* are engineering process types. Others are primitive process types.

We will say that the execution of a process *requires* the execution of sub-processes. Or that a process type *uses* a sub-process type.

Each engineering process type in a decomposition defines its own context into which specific product types are defined. Some of them are defined in the interface, others are part of the internal description. When a process *p* of type *P* requires the execution of a sub-process *q* of type *Q*, products must be passed between them: a product *x* being of a given type T_1 in the context of *P* must be affected another type T_2 in the context of *Q*. So, during the execution of *q*, the same product is of two different types at the same time, in two different contexts, as it can be seen in Figure 4.3. Section 4.2.2 shows how the interface of *Q* can be specified precisely and how *P* can use *Q*. Then Section 4.2.3 shows how the use of several sub-processes can be organised by the strategy.

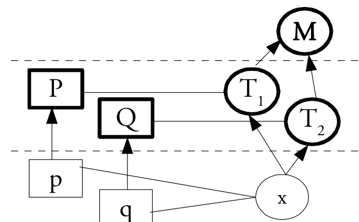


Figure 4.3 A product with two types (compliant with the same model M) in two different contexts

4.2.2. Engineering process type interface

The interface of an engineering process type is made up of:

- the *name* of the process type
- the *formal parameters* which are the product types used by the process type to exchange products with its environment
- an informal description, generally in natural language, of the goal and the way of usage of the instances of the process type.

Like in traditional programming languages, the name of a process type must identify it among all others and the parameters must be declared precisely in order to avoid confusion. But traditional programming languages are used (interpreted or compiled) entirely by computers, at the contrary of the system defined in this work which is used by human

beings who will have to choose to execute one process among several of them. This is why the informal description is an important new aspect.

A. Formal parameter declaration

Most generally, a process of a given type uses some products to produce and/or modify other products. A product type can play three roles in the interface of a process type: input, output and update.

- **Input product type**: a class of products that can be used during the execution of a process. These products can be referenced, consulted, analysed or copied, but cannot be modified nor created. When a process starts, the class is initialised with a series of products. The number of these products must match the minimum and maximum constraints of the product type.
- **Output product type**: a class of products generated by a process. When the process starts the output type has no instances. They have to be created or copied from other product types and modified. The number of products of that type has to match the minimum and maximum constraints when the process ends.
- **Update product type**: a class of products that can be modified during a process. When a process starts, the class is initialised with a series of products. The number of these products has to match the minimum and maximum constraints of the product type. During the process, products can be referenced, copied, modified but cannot be created.

New products can also be added to a non-initially-empty class using these three roles only. This will be shown later on.

Let P be an engineering process type and Q be a process type, such that Q is used by P. Let us denote by I a product type declared as input of Q, O a product type declared as output of Q and U a product type declared in update by Q. Let us examine what can be passed to I, O and U. In other words, let us examine what product type T declared in the context of P can have its products passed to I or U, or can receive products from O.

A product type T of P used in input of Q must be compatible with I. T is **I-compatible** with I if and only if one of the following propositions holds:

- T and I are of the same model
- the model of T inherits from the model of I.

Indeed, since products of type T exist before the use of Q and since the product type I is simply a product type aimed at seeing these products inside Q, the model of I has to be the same or to be more general than the model of T; the model of T must be a sub-model of the model of I in order to avoid unmanageable structures.

A product type T of P used in output of Q must be compatible with O. T is **O-compatible** with O if and only if one of the following propositions holds:

- T and O are of the same model
- the model of O inherits from the model of T.

Indeed, since O is the type of new products inside Q and since these products have to be mapped to type T, products of type O cannot contain structures that could not be valid in type T. So O has to be of a more restrictive model than T, at best of the same model as T.

If there already exists some products of type T before P uses Q, none of these products will be considered as instances of O, but all instances of type O will be mapped to T when Q ends without affecting the pre-existing products of type T.

A product type *T* of *P* used in update by *Q* must be compatible with *U*. *T* is **U-compatible** with *U* if and only if T and *U* are of the same model. Indeed, *U* cannot be of a more restrictive model than the model of *T* for the mapping when the process starts, and *U* cannot be of a more general model than the model of *T* since the products modified by the instance of *Q* still have to be of type *T* in the context of *P*.

When a process type calls a sub-process type and passes a product type, it means all the products of that type. This has to be possible according to the type cardinalities. Indeed, when a product type *T* is passed by *P* to an input product type *I* of a *Q* or to an update product type *U* of *Q*, the number of instances of *T* must fall in the range of the cardinalities of *I* or *U*; when a product type *T* of *P* receives products from a product type *O* of *Q*, the number of instances of *O* must fall in the range of the cardinalities of *T*. This constraint could be checked at method definition time by comparing the cardinalities of *I*, *U*, or *O* with the cardinalities of *T*, but this can lead to unnecessarily too much constraining situations, so it will actually be checked at execution time.

It may happen that just passing a subset of these products suffices. For this purpose, the notion of *product set* must be introduced, as well as set operations and product selection operations. But these are technical considerations which are relevant to the strategy. They will be presented in Section 4.2.3.

B. Using parameters

Let *P* be a process type and *Q* be a sub-process type required by *P*. *P* uses a product type, say *T*, whose instances will be passed to *Q* (for consultation or for modification) or produced by *Q* and passed back to *P*. Let us classify our possible needs along three independent axes:

1. *Q* can (1a) or cannot (1b) create (and modify) new products of type *T*.
2. *Q* can (2a) or cannot (2b) modify existing (before the use of *Q*) products of type *T*.
3. *Q* can (3a) or cannot (3b) access existing products of type *T* for consultation only.

This leads to eight parameter passing patterns (see Figure 4.4 for their illustration):

- 1b-2b-3a: existing products of type *T* are accessible, thought non modifiable, inside *Q* and no new products can be created. It suffices to declare an input product type *I* in *Q* and pass *T* to *I*.
- 1b-2a-3a: existing products of type *T* are accessible and modifiable inside *Q*, but new products cannot be created. It suffices to declare an update product type *U* in *Q* and pass *T* to *U*.
- 1a-2ab-3b: *Q* can create new products but cannot access old products of type *T* (note that since old products are not accessible, cases 2a and 2b do not need to be distinguished). This is the role of an output product type *O* declared in *Q* to which product type *T* can be passed.
- 1a-2b-3a: existing products of type *T* are accessible, thought not modifiable by *Q* and new products of type *T* can be created. The solution is simply to declare two product types *I* in input and *O* in output and to pass *T* to both of them.
- 1a-2a-3a: existing products of type *T* are accessible and modifiable inside *Q* and new products of type *T* can be created. The solution is simply to declare two product types *U* in update and *O* in output and to pass *T* to both of them.
- 1b-2ab-3b: existing products of type *T* are not accessible and none can be created. This is absolutely useless and distinguishing cases 2a and 2b do not change the situation.

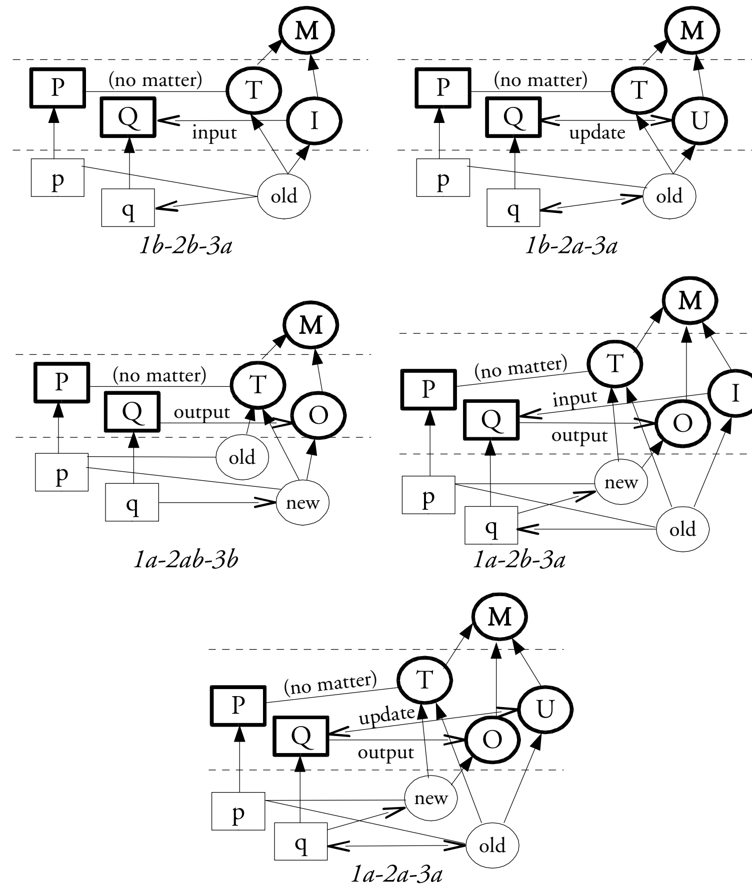


Figure 4.4 The various parameter usage patterns.

M *is a product model,* P *and* Q *are process types,* T, I, O, U *are product types,* p *and* q *are processes,* old *and* new *are products,* old *is a product that existed before* q, new *is a product generated by* q.
Lines without arrows reflect the fact that any kind of arrow would fit.

4.2.3. Engineering process type strategy

The strategy of an engineering process type is the way of working an engineer has to follow when performing a process of that type. This way of working will be expressed in a semi-procedural formalism. As opposed to procedures in traditional procedural programming languages, a way of working of a process type most often will be non deterministic, since it describes how to solve a class of problems for which no procedural solution has been discovered so far, and which be tackled by human beings with their non-deterministic behaviour, their education and their free will.

For better readability, strategies will be drawn graphically in an algorithmic way.

First, the graphical conventions will be defined, as well as some notations that will be used to express strategies and their interpretation. Then, the concept of internal product type will be defined too. In a third section, the various categories of deterministic and non deterministic control structures will be described. Finally, the various kinds of sub-processes and primitive processes that can be used in a strategy will be examined.

A. Graphical conventions and notations

a. Graphical conventions

The basic elements of every strategy are the sub-process types that have to be performed during the execution of instances of the process type, the product types that are used,

modified or generated, as well as the control flow (in what order the sub-processes are performed) and the data flow (how the products are used by the sub-processes).

A process type will be shown by a rectangle enclosing its name. A product type will be shown as an ellipse containing the product type name.

The control flow will be shown with bold arrows linking process types: an arrow from a process type to another one means that an instance of the former must be completed before an instance of the latter can start. The control flow starts with symbol ◡ and ends with symbol ◠.

The data flow will be shown with thin arrows linking process types and product types: an arrow from a product type toward a process type means that the instances of the process type use instances of the product type (input); an arrow in the reverse direction means that the instances of the process type create instances of the product type (output); a double headed arrow indicates that the instances of the process type both use and modify instances of the product type (update).

The external description of the process type (its interface) is described within a grey box. It shows graphically the name of the process type as well as the name and the role (input, output and update) of its product types.

For the ease of understanding of the various control flows, their use will be illustrated with a sample history[6]. These histories will be shown graphically too. Processes will be represented with rectangles, and products will be ellipses. Only the data flow will be represented, with thin arrows. Indeed the processes will be drawn top down in the order of their sequential execution (and from left to right on a same level if they are several versions of the performance of a same process type), making the drawing of the instance control flow useless. All the histories shown in this chapter will be easy to understand with these few tips and are shown for illustration only. A more complete definition and description is given in Chapter 6 which is entirely devoted to histories.

Figure 4.5 illustrate the interface and the strategy of a simple process type, as well as a sample history of a process following the strategy.

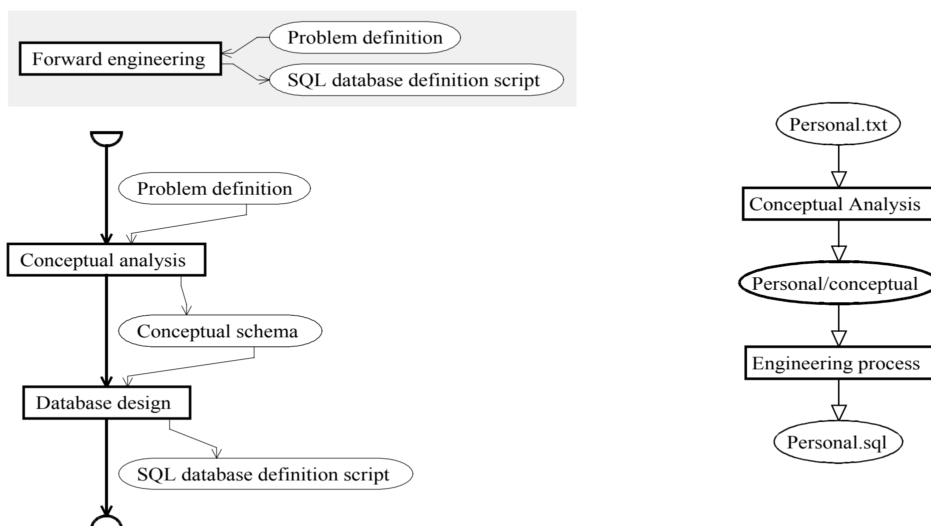


Figure 4.5 A sample method and history

6 As defined in Chapter 2.

b. Notations

The following notations will be employed:

- process types will be denoted *A*, *B*
- product types and product sets (defined in Section B) will be denoted *R*, *S*, *T*
- processes of type *A* will be denoted *a*, *a*1, *a2*,...; processes of type *B* will be denoted *b*, *b1*, *b2*,...
- products of type *R* will be denoted *r*, *r1*, *r2*,..., products of type *S* will be denoted *s*, *s1*, *s2*,..., products of type *T* will be denoted *t*, *t1*, *t2*,...
- *#R* will denote the number of products of type *R*,...

Furthermore, each control structure will be presented more formally with process algebra [BAETEN,95]. The alphabet is made of process types. The regular expression of a control structure expresses all the possible sentences (process sequences) that can be generated with this control structure, i.e. all the valid process sequences or histories.

The regular expression grammar is the following:

- *A* is the most simple expression, made of a single process type.
- $E_1.E_2$, or simply E_1E_2, is a sequence of sub-expressions, expression E_2 has to be performed after expression E_1 is terminated.
- E_1+E_2 expresses the fact that either E_1 or E_2 has to be executed.
- E^n expresses that *E* must be repeated exactly n times. This is equivalent to *EEE...E*, where E appears n times.
- $E_1//E_2$ expresses that both E_1 and E_2 have to be executed, in arbitrary order, and possibly in parallel.
- In an expression, the pattern E^n has the greatest priority, then come the sequences, and finally, E_1+E_2, and $E_1//E_2$. The four binary operators have to be evaluated from left to right. These priority rules can be changed by using parentheses.

In any sentence that derives from an expression, each process type denotation represents one instance of this process type. This constraint will be released in Chapter 6.

For example, expression $(AB+BC)^3$ must be read E_1^3 where E_1 *is* (E_2), E_2 is E_3+E_4, E_3 is E_5E_6, E_4 is E_7E_8, E_5 is *A*, E_6 and E_7 are both *B*, and E_8 is *C*. It expresses that three sequences have to be performed, each sequence being either *AB* or *BC*. So, all possible sentences that can be generated from that expression are: *ABABAB*, *ABABBC*, *ABBCAB*, *ABBCBC*, *BCBAB*, *BCABBC*, *BCBCAB*, *BCBCBC*. So, valid histories are respectively the process sequences: $a_1b_1a_2b_2a_3b_3$, $a_1b_1a_2b_2b_3c_1$, $a_1b_1b_2c_1a_2b_3$, $a_1b_1b_2c_1b_3c_2$, $b_1c_1a_1b_2a_2b_3$, $b_1c_1a_1b_2b_3c_2$, $b_1c_1b_2c_2a_1b_3$, $b_1c_1b_2c_2b_3c_3$. Obviously, the expression *A//B* cannot be expressed as one or several such sequences. To do so, it is necessary to distinguish the starting (denoted $\underline{A}$ or $\underline{a}$) and the ending (denoted $\underline{\underline{A}}$ or $\underline{\underline{a}}$) of each process type *A* or process *a*. So, the expression *A//B* can be interpreted as: $\underline{A}\underline{\underline{A}}\underline{B}\underline{\underline{B}}$, $\underline{A}\underline{B}\underline{\underline{B}}\underline{\underline{A}}$, $\underline{B}\underline{\underline{B}}\underline{A}\underline{\underline{A}}$, $\underline{A}\underline{B}\underline{\underline{A}}\underline{\underline{B}}$, $\underline{B}\underline{A}\underline{\underline{B}}\underline{\underline{A}}$, or $\underline{B}\underline{A}\underline{\underline{A}}\underline{\underline{B}}$, and valid histories are: $\underline{a}\underline{\underline{a}}\underline{b}\underline{\underline{b}}$, $\underline{a}\underline{b}\underline{\underline{b}}\underline{\underline{a}}$, $\underline{b}\underline{\underline{b}}\underline{a}\underline{\underline{a}}$, $\underline{a}\underline{b}\underline{\underline{a}}\underline{\underline{b}}$, $\underline{b}\underline{a}\underline{\underline{b}}\underline{\underline{a}}$, $\underline{b}\underline{a}\underline{\underline{a}}\underline{\underline{b}}$. With this notation, the sentences generated from expression $(AB+BC)^3$ become: $\underline{A}\underline{\underline{A}}\underline{B}\underline{\underline{B}}\underline{A}\underline{\underline{A}}\underline{B}\underline{\underline{B}}\underline{A}\underline{\underline{A}}\underline{B}\underline{\underline{B}}$, $\underline{A}\underline{\underline{A}}\underline{B}\underline{\underline{B}}\underline{A}\underline{\underline{A}}\underline{B}\underline{\underline{B}}\underline{B}\underline{\underline{B}}\underline{C}\underline{\underline{C}}$, $\underline{A}\underline{\underline{A}}\underline{B}\underline{\underline{B}}\underline{B}\underline{\underline{B}}\underline{C}\underline{\underline{C}}\ \underline{A}\underline{\underline{A}}\underline{B}\underline{\underline{B}}$, $\underline{A}\underline{\underline{A}}\underline{B}\underline{\underline{B}}\underline{B}\underline{\underline{B}}\underline{C}\underline{\underline{C}}\underline{B}\underline{\underline{B}}\underline{C}\underline{\underline{C}}$, $\underline{B}\underline{\underline{B}}\underline{C}\underline{\underline{C}}\underline{A}\underline{\underline{A}}\underline{B}\underline{\underline{B}}\underline{A}\underline{\underline{A}}\underline{B}\underline{\underline{B}}$, $\underline{B}\underline{\underline{B}}\underline{C}\underline{\underline{C}}\underline{A}\underline{\underline{A}}\underline{B}\underline{\underline{B}}\underline{B}\underline{\underline{B}}\underline{C}\underline{\underline{C}}$, $\underline{B}\underline{\underline{B}}\underline{C}\underline{\underline{C}}\underline{B}\underline{\underline{B}}\underline{C}\underline{\underline{C}}\underline{A}\underline{\underline{A}}\underline{B}\underline{\underline{B}}$, $\underline{B}\underline{\underline{B}}\underline{C}\underline{\underline{C}}\underline{B}\underline{\underline{B}}\underline{C}\underline{\underline{C}}\underline{B}\underline{\underline{B}}\underline{C}\underline{\underline{C}}$, and the corresponding valid histories: $\underline{a}_1\underline{\underline{a}}_1\underline{b}_1\underline{\underline{b}}_1\underline{a}_2\underline{\underline{a}}_2\underline{b}_2\underline{\underline{b}}_2\underline{a}_3\underline{\underline{a}}_3\underline{b}_3\underline{\underline{b}}_3$, $\underline{a}_1\underline{\underline{a}}_1\underline{b}_1\underline{\underline{b}}_1\underline{a}_2\underline{\underline{a}}_2\underline{b}_2\underline{\underline{b}}_2\underline{b}_3\underline{\underline{b}}_3\underline{c}_1\underline{\underline{c}}_1$, $\underline{a}_1\underline{\underline{a}}_1\underline{b}_1\underline{\underline{b}}_1\underline{b}_2\underline{\underline{b}}_2\underline{c}_1\underline{\underline{c}}_1\underline{a}_2\underline{\underline{a}}_2\underline{b}_3\underline{\underline{b}}_3$, $\underline{a}_1\underline{\underline{a}}_1\underline{b}_1\underline{\underline{b}}_1\underline{b}_2\underline{\underline{b}}_2\underline{c}_1\underline{\underline{c}}_1\underline{b}_3\underline{\underline{b}}_3\underline{c}_2\underline{\underline{c}}_2$, $\underline{b}_1\underline{\underline{b}}_1\underline{c}_1\underline{\underline{c}}_1\underline{a}_1\underline{\underline{a}}_1\underline{b}_2\underline{\underline{b}}_2\underline{a}_2\underline{\underline{a}}_2\underline{b}_3\underline{\underline{b}}_3$, $\underline{b}_1\underline{\underline{b}}_1\underline{c}_1\underline{\underline{c}}_1\underline{a}_1\underline{\underline{a}}_1\underline{b}_2\underline{\underline{b}}_2\underline{b}_3\underline{\underline{b}}_3\underline{c}_2\underline{\underline{c}}_2$, $\underline{b}_1\underline{\underline{b}}_1\underline{c}_1\underline{\underline{c}}_1\underline{b}_2\underline{\underline{b}}_2\underline{c}_2\underline{\underline{c}}_2\underline{a}_1\underline{\underline{a}}_1\underline{b}_3\underline{\underline{b}}_3$, $\underline{b}_1\underline{\underline{b}}_1\underline{c}_1\underline{\underline{c}}_1\underline{b}_2\underline{\underline{b}}_2\underline{c}_2\underline{\underline{c}}_2\underline{b}_3\underline{\underline{b}}_3\underline{c}_3\underline{\underline{c}}_3$.

B. Internal product types and product sets

An **internal product type** is a product type whose instances are temporarily created and used during the execution of a process of this type, and that disappear at completion of the process. It is declared locally to a process type and has no existence outside of it. When a process starts, its internal product types have no instance. Some instances can be created from scratch, can be copies of products of other types, or can be generated by a sub-process. These internal products can then be modified. Before terminating the process in which it has been created, an internal product, or part of it, can be copied into an output product. Since there is no product of this type at the beginning of a process, the minimal cardinality of the type cannot be checked permanently. But it can be checked when the process ends as a control tool. The maximum cardinality can be checked permanently.

A **product set** is a container that can accommodate any number of products. It allows products to be collected in order to be handled all at once. The products can be of different types. Sets can be used in set operations (union, intersection,...). They can also be used everywhere a product type is needed in input or update; in that case, all products of the set having the correct type are used, the others being simply left aside. For instance, an ORACLE-SQL and a DB2-SQL schema types can be defined compliant with a SQL-MODEL, and an integration process type can be defined with an SQL product type, compliant with the SQL-MODEL, in input. To integrate all the schemas, a product set can be defined as the union of the set of products of ORACLE-SQL type and the set of products of DB2-SQL type. This new set can be passed to a new integration process. Since the set is empty when a process starts and since the content of the set is always the result of a set operation (like the union) or product selection (the user has to choose the products to put in the set), the cardinality constraint of the set can be checked after each operation or selection.

From now on, for homogeneity and clarity reasons, product types will be considered special product sets. Indeed, since a product type is a class of products that play a definite role in the system life cycle, it can be considered to be a product set that cannot be modified by set operations. Each time the word *product set* is used, the reader should understand *product type or product set*, except when explicitly stated.

C. Control structures

A strategy has to specify how the different actions have to be performed. A control structure is a mechanism that is aimed at ordering actions. This section describes a series of control structures ranging from traditional sequence, alternatives or loops that can be found in any procedural language to particular non-deterministic alternatives or loops which are typical to human decisions.

a. Sequence

The sequence is the most traditional control structure that decomposes a task in simpler tasks that have to be performed in the specified order, one after the other. In a traditional programming language like Pascal, sequences are represented by a list of statements separated by semi-colons. In software engineering, including database engineering, performing sequences of actions is a common pattern.

Figure 4.6 defines a sequence of two process types: A and B. The history (right) complies with this sequence: using a product *r* of type *R*, the process *a* of type *A* generated a product *s* of type *S*, which was reused by the process *b* of type *B*, which, in its turn, generated the product *t* of type *T*.

A sequence is a series of process types and control structures that have to be performed one at a time in the specified order. Let E_1, E_2,...,E_n be *n* expressions (that can represent control

structures or process types), then a sequence of these *n* expressions is represented by the following expression: $E = E_1E_2...E_n$

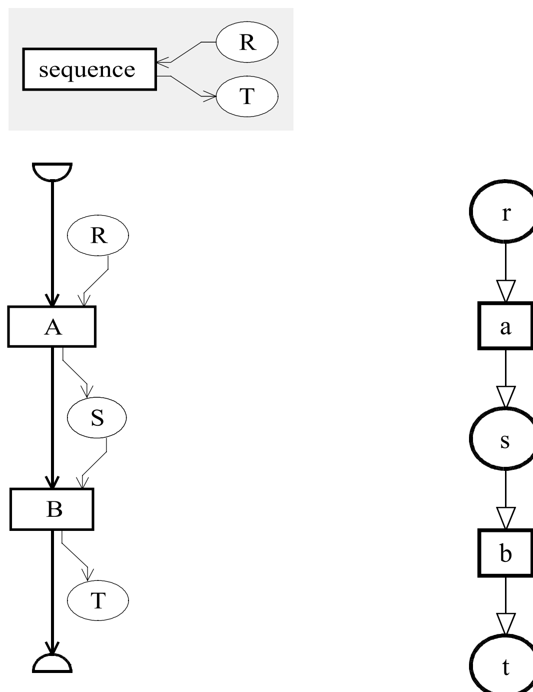


Figure 4.6 A sequence

The grey rectangle is the process type interface with one input product type and one output product type. The left graph is the strategy made of a simple sequence. The right graph is a small history of a process performed according to the process type.

b. Standard alternatives

A standard alternative is a possibility to perform one action (possibly a sequence or another control structure) or another depending on a given deterministic condition. In traditional programming languages, it is the *if...then...else* structures; *if* the condition is true, *then* do a first action, *else*, do another one; the condition is either true or false, never undefined.

Figure 4.7 shows an abstract example. The condition *cond* represented by the diamond has to be evaluated. This condition is a boolean expression, it results in either *true* or *false*. If the result is *true*, the control flow continues through the side branch and a process of type *A* has to be executed. On the other hand, if the condition is evaluated to *false*, the control flow goes on through the bottom branch and a process of type *B* has to be performed. The history example shows that only one process *a* of type *A* was performed, the condition being evaluated to *true*.

Let E_1 and E_2 represent two process expressions. Then, a standard alternative is the process expression: E_1+E_2. In practice, the choice of one alternative or the other is guided by the evaluation of a boolean expression. Let us note that one of the two expressions maybe an empty structure, so $E_1+()$ may denote that E_1 is an optional statement.

c. Standard loops

Traditionally, the loop is the third and last basic control structure. It allows an action (or other control structure) to be performed several times while or until a deterministic condition is satisfied. The condition can be checked after or before the first loop, forcing it to be performed at least once or not, respectively. In traditional programming languages, the *for*,

while, *repeat*, *do...until* structures are used for these loops. In database engineering, it is also necessary to be able to perform some actions several times, for instance while or until some products are in a specific state.

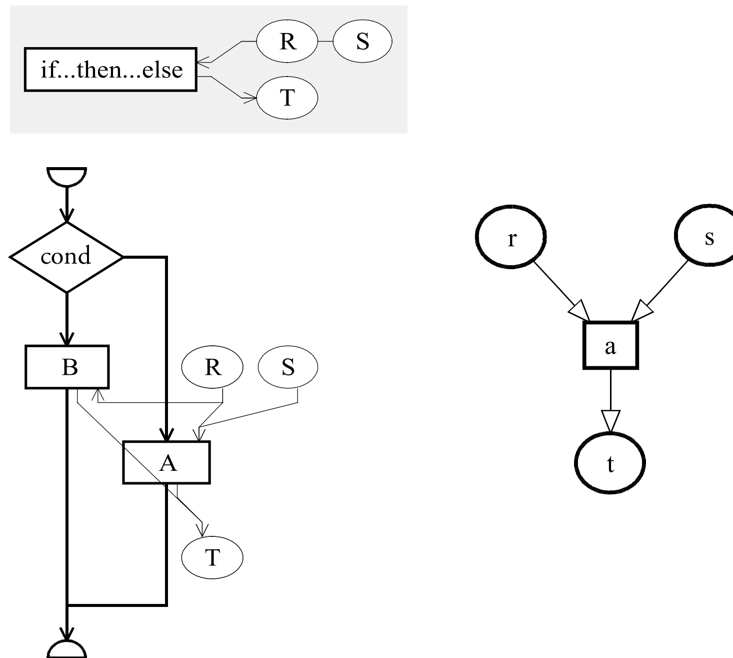


Figure 4.7 A standard alternative

If the cond *condition is true, the execution follows the side branch and A must be performed, else the bottom branch has to be followed and B is the next step. The history, at right, shows that only a process a of type A was performed.*

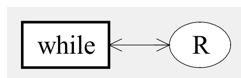
Figure 4.8 shows a simple loop: while the condition is evaluated to *true*, processes of type *A* have to be performed. The history shows that two processes (*a1* and *a2)* were actually performed, each of them updating the same product *r*.

Let E_1 denote a process expression, and *R* a product set containing *#R* products. Then a deterministic loop can be represented by either of the following regular expressions:

- $E = E_1^n$, n≥0 In practice, the evaluation of a boolean expression before each appearance of E_1 will indicate when to stop the repetition, as shown in Figure 4.8.
- $E = E_1^n$, n≥1 In practice, the evaluation of a boolean expression after each appearance of E_1 will indicate when to stop the repetition, as shown in Figure 4.9.
- $E = E_1^i$, $1 \leq i \leq \#R$ In practice, each appearance of E_1 should concern one different product of *R*. This is done using a product set with cardinalities [1-1]. At each iteration, the product set is filled with one different product of type *R* and the set itself is passed to the sub-process type, as shown in Figure 4.10.

d. Non-deterministic alternatives

The deterministic condition of standard alternatives is necessary for procedural programming languages designed for deterministic machines, but this can be a stronger constraint for conducting processes in which human expertise is required, as it is usual in software engineering in general, and in database engineering in particular. In such cases, process types strategies must include non-deterministic alternatives. Five kinds of such non-deterministic alternatives will be considered:



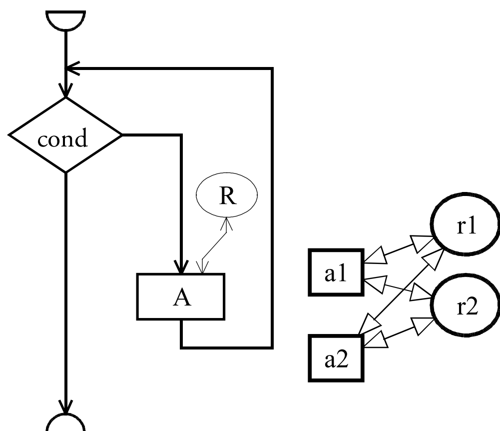
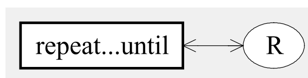


Figure 4.8 A standard while loop

Processes of type A have to be performed while the condition cond *is true. The history shows that two processes of type A were performed with every products of type R.*



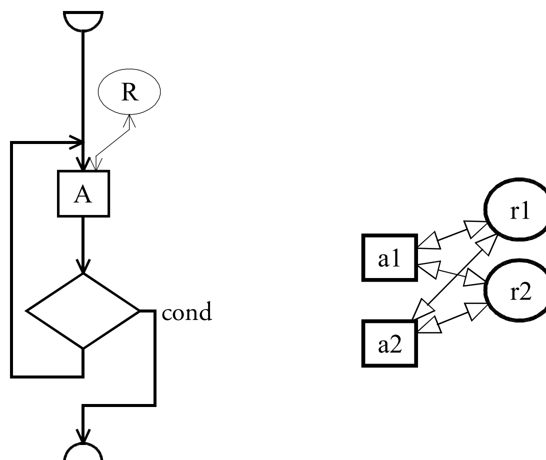
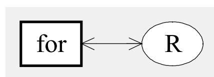


Figure 4.9 A standard repeat...until *loop*

Processes of type A have to be performed until the condition cond *is true. The history shows that two processes of type A were performed with every products of type R.*



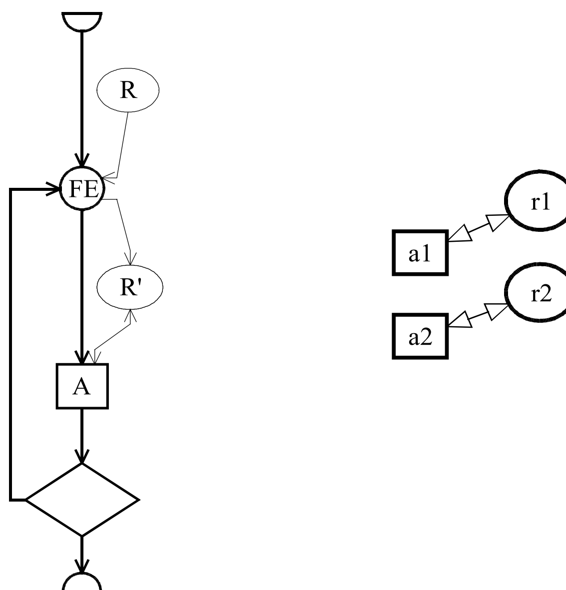


Figure 4.10 A standard loop

Processes of type A have to be performed while products of type R have to be treated. The selection process puts each product at its turn in the set R' and that this set is passed to the process type. The letters FE *show that each product of type R will have its turn. Letters* FS *(for some) should allow the user to select a few products only, and letters* F1 *would show that the user should select only one product to treat. The history shows that two processes of type A were performed with every products of type R.*

1. The **informal alternative**: a two-case alternative with a non-deterministic condition; the condition simply is a question that is asked to the engineer who will have to answer by *yes* or *no*.
2. The **weak alternative**: a two-case alternative with a weak deterministic condition; the deterministic condition is evaluated and its result is shown to the engineer who can accept the result or force another one.
3. The **one alternative**: a multi-case alternative with no condition in which the engineer has to choose one branch.
4. The **some alternative**: a multi-case alternative with no condition in which the engineer can choose at least one branch (possibly several) and perform them in any order.
5. The **each alternative**: a special multi-case alternative with no condition in which the engineer must choose every branch, but he or she can choose in what order. This is in fact also a special case of sequence with no pre-defined order on the actions.

Note that if a multi-case alternative can theoretically be easily replaced by a series of two-cases alternatives for a computer, it cannot for human beings, independently of the deterministic or non-deterministic characteristic of the conditions. Indeed, asking a human being to "Choose a letter between 'a' and 'j'." cannot be expressed by the sequence "Do you want to choose letter 'a'?", "Do you want to choose letter 'b'?", and so on until "Do you want to choose letter 'j'?"; forcing him or her to answer ten times a similar question takes time and could make him or her feel nervous. So multi-cases alternatives are simply a shortcut in traditional programming languages, but they take a real sense when dealing with human beings.

Figure 4.11 shows a simple example of a *one alternative*. An analyst has to choose to perform either a process of type *A* or of type *B*. The history shows that a process *b* of type *B* was performed, but no process of type *A*.

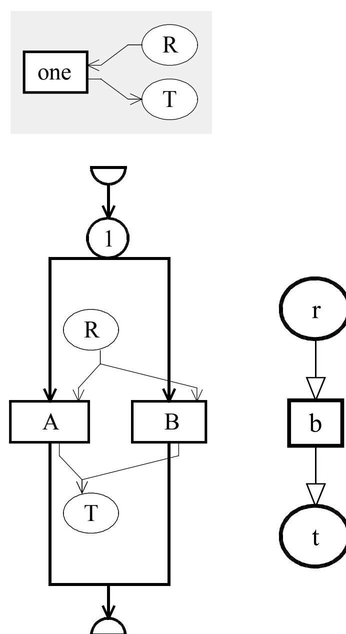


Figure 4.11 A non-deterministic multi-case alternative.

In this one alternative*, the user has to choose to perform either A or B.*
The history shows that a process b of type B was performed

Let E_1, E_2,..., E_n be n process expressions. Then non-deterministic alternatives of the five types above can be defined as:

1. $E = E_1+E_2$ for the *informal alternative*.
2. $E = E_1+E_2$ for the *weak alternative*.
3. $E = E_1+E_2+...+E_n$ for the *one alternative*.
4. $E=E_1+E_2+...+E_n+E_1//E_2+E_1//E_3+...+E_1//E_n+E_2//E_3+...+E_2//E_n+...+E_{n-1}//E_n+...+E_1//E_2//...//E_n$ for the *some alternative*.
5. $E = E_1//E_2//...//E_n$ for the *each alternative*.

e. Non-deterministic loops

The reasons to define non-deterministic loops are the same as the reasons to define non-deterministic alternatives. Three kinds of non-deterministic loops will be considered:

1. **Informal loops**: loops with a non-deterministic condition; a question is asked to the engineer who can answer either by “yes, let us do the loop one more time”, or “no, let us stop looping”; this question can be asked before or after the first loop.
2. **Weak loops**: loops with a weak deterministic condition; before or after each loop, the condition is evaluated and shown to the engineer with its result; the engineer can either accept or reject the result, forcing the looping process to stop or to go on.
3. **Free loops**: loops without condition at all; the engineer can do the loop as many times as he or she wants, but at least once. A non-deterministic alternative can be added to allow for no loop at all.

Figure 4.12 shows a free loop in which processes of type *A* can be performed as many times as the user would like to, but at least once. The history is made up of two processes.

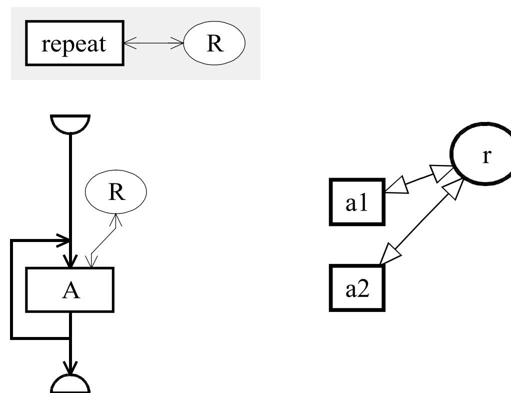


Figure 4.12 A non-deterministic loop.

This free loop *can be performed as many times as the engineer wishes. The history shows that he or she did it twice.*

Let E_1 be a regular expression. Then non-deterministic loops of the first and second kind can be defined in the same way as the deterministic loops, since these are the same structures with only a difference in the condition. So, regular expressions denoting sentences that can be generated by non-deterministic loops are:

- $E = E_1^n$, $n \geq 0$ if the loop can be executed any number of times.
- $E = E_1^n$, $n \geq 1$ if the loop has to be performed at least once.

Free loops can only be expressed as:

- $E = E_1^n$, n≥1

D. Sub-processes

The decomposition of a large problem in smaller problems was presented in Section 4.2.1. The smaller problems can be either engineering process types or primitive process types. The way of calling an engineering process type was described in Section 4.2.2. Attention will now be paid to the use of primitive processes. The concept of product transformation will be introduced and the primitive process will be examined according to the four groups defined in Chapter 2.

a. Product transformations

A **transformation** is a simple action that replaces a construct of a product by another construct. By applying a series of transformations on a product, it is possible to make it evolve. Most of the time, a product of a given type, which is compliant with a given product model, has to evolve in order to be of another type, compliant with another product model. For example, an ER conceptual schema can be transformed into a semantically equivalent relational logical schema. Within the scope of database engineering and the DB-MAIN CASE tool, the only products to transform are schemas. A more complete description of transformation theory will be presented in Chapter 6.

Transformations can be classified in three categories: semantics preserving, semantics augmenting, and semantics decreasing transformations.

A large set of database schema transformations was been studied in [HAINAUT,96c]. They are implemented in the DB-MAIN CASE environment.

In database schemas a transformation generally has to be applied to all the constructs that meet a definite condition. A **global transformation** is a couple <C,T> where C is a structural predicate on constructs of type O and T a transformation applicable to constructs of type O (see Chapter 6).

A toolbox of useful general global transformations can be defined in the same way the structural predicates were defined.

- The GER model includes compound attributes, but some models such as the SQL2 model reject them. So compound attributes have to be converted into equivalent constructs that are compliant with the SQL2 model. They could be disaggregated through a DISAGGREGATE global transformation that replaces every compound single-valued attribute by all its sub-attributes, whose name receive a prefix reminding the compound attribute name. Figure 4.13 shows an example of such a transformation.

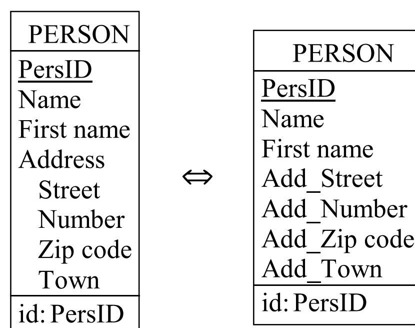


Figure 4.13 A disaggregation transformation

- Most commercial logical and physical models require that relationships (or referential constraints) being functional, i.e. many-to-one (possibly one-to-one) rel-types without attributes. So, every rel-type with attributes, N-ary or many-to-many have to be transformed into an entity type together with purely functional rel-types. So, a global transformation that replaces all the non-functional rel-types with entity types can be defined.

  More generally, a global transformation can transform all rel-types into entity types. It can be controlled with a structural rule that filters only rel-types of interest. This global transformation will be called *RT_into_ET* and the conversion of non-functional rel-types will be expressed as follows (the concrete syntax T(C) is used to express the global transformation <C,T>):

  ```
  RT_into_ET( ROLE_per_RT(3 N)
              or not ATT_per_RT(0 0)
              or N_ROLE_per_RT(2 N))
  ```

- *ISA_into_RT* transforms all is-a relations into binary one-to-one rel-types.
- *RT_into_REF* transforms all functional rel-types into referential attributes when the entity type at the "many" side of the rel-type has an all-attribute primary identifier.
- Multi-valued attributes can be transformed into entity types in two different ways: *ATT_into_ET_VAL* do the transformation using the value representation of the attributes, i.e. by creating an entity type whose entities are unique and possibly in relation with several entities of the entity type containing the original multi-valued attribute; *ATT_into_ET_INST* do the transformation using the instance representation, i.e. by creating an entity type whose instances are all the value instances of the original multi-valued attributes and whose instances are identified by their value and the relation to the entity originally containing the multi-valued attribute value.
- *REMOVE_PREFIX_KEY* removes access keys which are a prefix of another access key, i.e. whose components are the first components of another access key, in the same order.

The previous global transformation are useful for the transformation of a ER conceptual schema into a logical relational schema as it will be shown in a case study in Chapter 11.

In order to reverse engineer a relational schema, the reverse of these global transformations are also useful global transformations. Among them:

- *ET_into_RT* transforms all entity types which "look like" a rel-type, i.e. all entity types that play at least two "one" roles in mandatory functional rel-types, that play no role in other rel-types and whose instances are identified by all the roles they play, into rel-types. It is the reverse of the *RT_into_ET* global transformation.
- *REF_into_RT* transforms reference groups into functional rel-types. This is the reverse of the *RT_into_REF* transformation.

The complete listing of global transformations defined in DB-MAIN can be found in Appendix C and in [DBMAIN,02b].

A **global transformation script** is a deterministic strategy made of a series of global transformations, ordered with some particular adapted control structures. While the control structures defined before concern the handling of products in a project, global transformations carry out a deterministic task with constructs of a single schema. Three elementary control structures are proposed.

- In a sequence, all the global transformations are performed in the specified order. For example, if $<SC_1,T_1>,\ldots,<SC_n,T_n>$ are n transformations, a sequence looks like:

$T_1(SC_1)$
$T_2(SC_2)$
...
$T_n(SC_n)$

- In a loop, the body of the loop is a global transformation script which is repeated while some of its transformations actually fired on some constructs. It is always performed one time more than necessary, the last time during which the structural rule of each global transformation is evaluated to find out that nothing matches it. For instance, the following loop:

  loop
  $T_1(SC_1)$
  $T_2(SC_2)$
  endloop

  can be performed a first time during which SC_1 will find a series of constructs that T_1 will transform and SC_2 will find another set of constructs that T_2 will transform, then a second time during which only SC_1 will find some constructs that will be modified by T_1, and a third time during which neither SC_1 nor SC_2 is matched by constructs of the schema and nothing happens. The loop will stop after that third run.

- The *on(scope)...endon* structure allows the analyst to restrict the scope of the body to a particular set of constructs. The scope is a structural rule. For example, let us consider:

  on (scope)
  $T_1(SC_1)$
  $T_2(SC_2)$
  ...
  endon

  When the *on(scope)* line is encountered, the scope expression is evaluated and gives a set of constructs, called S. SC_1 is evaluated and T_1 will transform constructs from $SC_1 \cap S$. Then SC_2 is evaluated and T_2 will transform constructs from $SC_2 \cap S$. The interest of this structure lies in the fact that the scope is evaluated just once before all transformations. This allows SC_2 to select only constructs that already existed before T_1 and to exclude those created by T_1. Note that T_1 can also destroy some constructs of S; those constructs will not be selected by SC_2. During the following example:

  on (DEPTH_of_ATT(1 1))
  DISAGGREGATE(ALL_ATT())
  NAME_PROCESSING (P^;#;,ALL_ATT())
  endon

  all the level-1 attributes of the schema go in S. Then the disaggregation global transformation (T_1) takes place. SC_1 contains all the attributes of the schema, and $S \cap SC_1 = S$, so only level-1 attributes are disaggregated. Then the name processing global transformation (T_2) is executed, trying to prefix all attributes of the schema (SC_2) with the symbol #. But since T_2 is in the *on* structure, only attributes in $S \cap SC_2$ are transformed, in other words, attributes which were at the first level before the disaggregation, excluding the level-2 attributes that became level-1 attributes through the disaggregation.

  Note that the following script does not give the same result:

  NAME_PROCESSING (P^;#;,DEPTH_of_ATT(1 1))
  DISAGGREGATE(DEPTH_of_ATT(1 1))

  Indeed, in this version, the level-1 compound attributes will get the symbol in the first

transformation and this symbol will be transferred to the sub-attributes with the prefix.

More complex control structures than those three ones could have been introduced as well, such as a conditional structure (if-then-else), or loops with a condition (while, repeat-until), but these scripts should remain simple enough to be able to solve most problems without hassle. Several years of experience in using this CASE tool proved it works. In the DB-MAIN CASE tool, new transformations can be written using the internal Voyager 2 language to solve more complicated problems. These Voyager 2 programs can be included in a method as well.

b. Other primitive processes

The primitive process types are elementary actions that need not be given an explicit strategy. This does not mean that a primitive process type always carries out a simple task. On the contrary, some of them are highly complex, and are based on sophisticated strategies. However, since the latter are deterministic, they can be ignored by the analyst, who can execute these processes as if they were mere atomic actions.

An inventory of the primitive process types of the CASE tool and a classification in the four categories defined in Chapter 2 has to be performed as follows:.

- Basic automatic process types are, among others, simple DDL[7] script generators, DDL script extractors, copying products, creating a new blank schema,... all actions that simply need to launch the correct tool, the execution demanding no configuration and no interaction with the user.
- Configurable automatic process types are:
  - Complex DDL generators (more generic generators that can be used for many DBMS but that need to be configured properly, once for all, by the method engineer) or extractors.
  - External procedures. In some particular situations, the working environment may not provide the needed tools. It is then necessary to use external tools. For this purpose the method engineer will use the internal programming language of the working environment: Voyager 2 [ENGLEBERT,99] with DB-MAIN. This internal language can be used either to write directly the missing tool, or simply to write an interface with a third-party tool.
  - The global transformation scripts defined above. Indeed, the method engineer cannot simply specify that a global transformation script has to be executed, he or she has to write the script.
- User configurable automatic process types are process types that can be configured by the database engineer. This includes external procedures or complex DDL generators or extractors that need some user interaction to be executed, i.e. programs that require that the user answers a few questions or sets a few parameters in order to perform their job. For instance, a COBOL data structure extractor may require that the user specifies which COBOL syntax is used in the source files if the method engineer has designed a method general enough for reverse engineering programs written with various COBOL syntax.
- Manual process types are simply the use of a toolbox[8], that is to say, the manual use of the supporting CASE tool, limited to a subset of all its functions. The list of tools in the toolbox has to be defined by the method engineer. But the way to use these tools is up to the final user who can decide what to do only when the product to transform is in

7 DDL = data definition language, like SQL DDL, COBOL data definition section,...
8 The concept of *toolbox* was defined in Chapter 2 and will be described in detail in Chapter 5.

front of his or her eyes and with his or her knowledge of the problem. The list of tools that can be put in a toolbox depends on the supporting CASE environment but should include all editing facilities as well as the transformations defined above.

It is to be noted that the method engineer can decide to put some processes of the first category (for instance an SQL DDL script generator) in a toolbox (fourth category). When used alone, it is directly executed once when the method requires it. When inserted in a toolbox, the method requires the use of the toolbox with a given set of products and makes the tool available to the user. The user can himself or herself start the tool, possibly several times, alternately with other tools of the toolbox.

E. Assembling elements

a. Assembling control structures

The basic control structures presented above can be assembled to build complex strategies. To understand these assemblings, the process expressions can be written, and they can be transformed using the BPA process algebra defined in [BAETEN,95].

Using the symbols w, x, y, z to represent processes, the basic properties of this BPA algebra are (from [BAETEN,95],table 1):

$$x + y = y + x$$
$$(x + y) + z = x + (y + z)$$
$$x + x = x$$
$$(x + y)z = xz + yz$$
$$(xy)z = x(yz)$$

From these properties, the following expression transformations can be deduced:

$$(x + y)(w + z) = x(w + z) + y(w + z) = xw + xz + yw + yz$$
$$(x + y)^2 = (x + y)(x + y) = xx + xy + yx + yy$$
$$(x + y)^n = (x + y)(x + y)^{n-1} = x(x + y)^{n-1} + y(x + y)^{n-1} =$$
$$xx(x + y)^{n-2} + xy(x + y)^{n-2} + yx(x + y)^{n-2} + yy(x + y)^{n-2} = \ldots \quad \forall\, n \geq 2$$

More generally, $\forall$ m $\geq$ 1:

$$\begin{cases} (\sum_{i=1..m} x_i)^n = \sum_{j=1..m}(x_j(\sum_{i=1..m} x_i)^{n-1}), & \forall\, n \geq 1 \\ (\sum_{i=1..m} x_i)^1 = \sum_{i=1..m} x_i \end{cases}$$

[BAETEN,95] also introduces the PA algebra for parallel processes. It uses a *left merge* symbol $\llcorner\!\llcorner$. $x \llcorner\!\llcorner\, y$ means that x starts before y. It also uses a to symbolise an atomic process, that is to say an automatic primitive process type in this thesis. PA algebra is based on the previous axioms, as well as the following ones (from [BAETEN,95], table 42):

$$x \,/\!/\, y = x \llcorner\!\llcorner\, y + y \llcorner\!\llcorner\, x$$
$$a \llcorner\!\llcorner\, x = ax$$
$$ax \llcorner\!\llcorner\, y = a(x \,/\!/\, y)$$
$$(x + y) \llcorner\!\llcorner\, z = x \llcorner\!\llcorner\, z + y \llcorner\!\llcorner\, z$$

For instance, Figure 4.14 shows an example of a complex strategy made of several control structures and sub-process calls. It is an excerpt of the second case study in Chapter 11. Using the process expressions, this example can be written in the following way:

E_1 = COPY
E_2 = Physical schema enrichment expert
E_3 = ET-ID search
E_4 = Long fields refinement

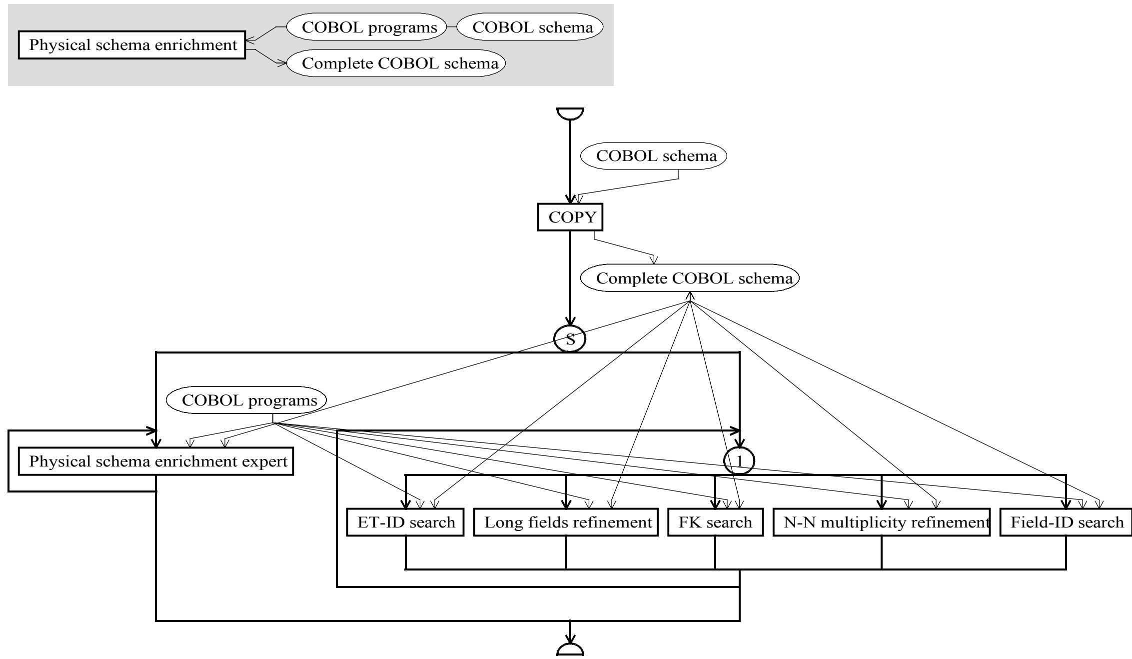


Figure 4.14 A complex strategy made of a sequence whose second element is a some *structure. Its two components are repeat loops. One of them contains a* one *structure. "Physical schema enrichment expert" is an engineering process type, "COPY" is an automatic process type and the five elements of the* one *structure are toolboxes. This strategy shows that the analyst can either do the schema enrichment job by himself or herself using the required toolboxes, or use the guidelines offered by a more expert strategy, or combine both methods.*

E_5 = FK search
E_6 = N-N multiplicity refinement
E_7 = Field-ID search
$E_8 = E_2^n$, $n \geq 1$
$E_9 = E_3+E_4+E_5+E_6+E_7$
$E_{10} = E_9^n$, $n \geq 1$
$E_{11} = E_8+E_{10}+E_8//E_{10}$
$E_{12} = E_1E_{11}$

By combining these expressions (and distinguishing the various n), we can write:

$$E_1(E_2^{n_1}+(E_3+E_4+E_5+E_6+E_7)^{n_2}+E_2^{n_3}//(E_3+E_4+E_5+E_6+E_7)^{n_4}),\ n_1,n_2,n_3,n_4 \geq 1$$

Using the process algebra, this expression can be transformed:

$$E_1(E_2^{n_1}+(E_3+E_4+E_5+E_6+E_7)^{n_2}+E_2^{n_3}//(E_3+E_4+E_5+E_6+E_7)^{n_4}) =$$
$$E_1E_2^{n_1} + E_1(E_3+E_4+E_5+E_6+E_7)^{n_2} + E_1(E_2^{n_3}//(E_3+E_4+E_5+E_6+E_7)^{n_4}),\ n_1,n_2,n_3,n_4 \geq 1$$

The last expression is made of three terms, showing the three possible ways of using this strategy:

1. After copying the schemas, the analyst uses the expert process as often as needed
2. After copying the schemas, the analyst can use all the toolboxes, as often as desired and in any order. Indeed,

$$(E_3+E_4+E_5+E_6+E_7)^1 = E_3+E_4+E_5+E_6+E_7$$
$$(E_3+E_4+E_5+E_6+E_7)^2 = E_3E_3+E_3E_4+E_3E_5+E_3E_6+E_3E_7+E_4E_3+E_4E_4+E_4E_5+E_4E_6+E_4E_7+ E_5E_3+E_5E_4+E_5E_5+E_5E_6+E_5E_7+E_6E_3+E_6E_4+E_6E_5+E_6E_6+E_6E_7+ E_7E_3+E_7E_4+E_7E_5+E_7E_6+E_7E_7$$
$$(E_3+E_4+E_5+E_6+E_7)^3 = E_3E_3E_3+E_3E_3E_4+E_3E_3E_5+E_3E_3E_6+E_3E_3E_7+E_3E_4E_3+\ldots+E_7E_7E_7$$
...

$(E_3+E_4+E_5+E_6+E_7)^7 = ...+E_5E_3E_4E_5E_7E_6E_5+...$ for example

...

3. After copying the schemas, the analyst can combine the two previous methods, performing them in parallel, starting by the one he or she prefers.

b. Performing processes in parallel

A question remains about starting two processes in parallel: Can they interfere with each other? Several cases may appear:

- the two processes work with different products: no interference
- the two processes use the same products in input: no interference
- the two processes generate products of the same type: no interference
- the two processes must modify the same products: interferences occur
- one process uses a product in input and the second process updates the same product: inteferences occur too.

In fact, interferences occur only when two processes use the same product at the same time and at least one process has to update that product, whatever the types of the processes and of the products. Three ways of working can be seen in this situation:

- The two processes work on the same product. This solution needs a concurrence mechanism that works at the component level of the products.
- The two processes work on copies of the products, then an integration mechanism must be used to produce a common result. The integration is not necessary if only one process modifies the product.
- A process cannot update a product that is already in use by another process, either in input or in update. The second process must wait the completion of the first one before it can start.

The third solution is by far the most simple and is the one that will be chosen in this thesis. It may seem to be a bit limitating to be forced to finish one process before the second one can be started, but an operational technique that allows to cope with this limitation will be presented in Chapter 9.

4.3. Comparison with other modelling techniques

In this section, the different process modelling techniques that can be found in the literature (see Chapter 1, Section 1.2) will be compared with respect to the process type description. Indeed, comparing the way to describe products (product types and product models) is non-sense because products depend much more on the application domain than on the technique itself; by example, a business object is not a database schema, nor a program. They will be compared according to the criteria of interest in the scope of this thesis: the ease of use for both the method engineer and for the database engineer.

Rule based and functional techniques are both declarative techniques that share a lot of characteristics. The main one is certainly the fact that a method is a collection of process type declarations, each one being independent from the others. The fact that the performance of a process type P can be followed by the execution of another process type Q can only be known at run time. Indeed, let us denote *Post*(P) the postconditions of P and *Pre*(Q) the preconditions of Q, supposing they concern the same product models. If *Post*(P) and *Pre*(Q) are the same rules, then the possibility of a sequence is obvious. If *Pre*(Q) is a set of rules which concerns the same concepts the set of rules *Post*(P) while being more

restrictive, this "inclusion" has to be detected but the possibility of a sequence is still obvious. But, if *Post*(*P*) and *Pre*(*Q*) are made of rules concerning different aspects of a same product model (for instance, *Post*(*P*) insures that every rel-type is binary and *Pre*(*Q*) requests nothing special about rel-types), only the evaluation of the rules on an actual product at run time can make the possibility of a sequence appear.

These declarative paradigms share a few properties:

- They allow the method engineer to concentrate on one process type at a time. There is no need to update every other process type required or used by it when modifying it.
- They are build on firm mathematical foundations for the help of the method engineer.
- The validation of a method needs an inference engine that may be unable to give a result in some particular cases as cited above.
- During a project using a method, the database engineer can only know at a precise time what can or cannot be done, but a general view of the method is difficult, sometimes impossible, to obtain, according to the same reasons.
- An important learning period may be required for both the method engineer and the database engineer to correctly grasp all the mathematical concepts.

Petri nets and graph based models answer several of the problems cited above. Indeed, they all allow a graphical representation of the process types. These modelling techniques share the following properties:

- They are supported by firm mathematical foundations for the help of the method engineer.
- The validation of a method can rather easily be done visually or with the use of a simulation engine.
- The adjunction of a new node to a Petri net or to a graph can be a very complex task which may require a transformation of the net or the graph.
- During the use of the method, the database engineer can easily both know what to do next and have a global view of the method.
- An important learning period may be required for both the method engineer and the database engineer to correctly grasp all the concepts.

Procedural (including most object oriented) methods can easily be presented graphically too, either with algorithms, call graphs, sequence diagrams,... These procedural techniques share the following properties:

- The validation of a method can rather easily be done visually on the graphical representations.
- The adjunction of a new process type ("procedure" or object) is rather heavy because all the other process types from which the new one could be enacted must be modified too.
- During the use of the method, the database engineer can easily know what to do next and have a global view of the method.
- Since most computer scientists or engineers learn procedural principles at the beginning of their computing education, learning to use these methods is rather simple.
- Procedural languages are more based on experience of life (most actions we do in every day life are described procedurally, like a cooking recipe) than on mathematical theories, but even this category of languages has been thoroughly studied since its creation and has now solid foundations.

In conclusion, all the techniques are equally solid, and they all require more or less the same amount of work in order to build a correct method (the techniques which need the less work to build a method need the more work to validate the result). But procedural techniques seem to be easier to use by non-scientists (personal background and graphical presentation). That is why we can think that the algorithmic method description defined in this chapter goes in the right way.

Chapter 13 will compare the techniques according to another criteria, namely the adaptability of the method to the current project.

Chapter 5

The MDL language

In the previous chapters, a meta-model for designing database engineering methods was defined. It is a complete set of concepts with a graphical representation. The aim of this meta-model being to guide an analyst during the use of a CASE environment, it is necessary to allow a method engineer to implement his or her methods; it is necessary to have a means of designing formally the method. For that purpose, this chapter will define MDL, a Method Description Language.

In a first time, some requirements for the language will be enumerated. Then, a complete definition of the language will be described. In a third time, this language will be analysed a little bit in order to understand its main characteristics and check the fulfilment of the requirements.

This chapter will present MDL, a Method Description Language, to allow method engineers to implement a method compliant with the two previous chapters in a CASE tool.

MEL [BRINKKEMPER,01] is such a kind of language too, although more oriented towards software engineering.

5.1. Requirements

These are the main characteristics we would like to give to the language:

- Since product models and product types are defined in the meta-model in a declarative fashion, the most natural way to declare them is in a declarative way too.
- Since the process types of the meta-model are defined in an algorithmic fashion, the most natural language to declare them is traditionally a procedural language.
- Since we want to use this language to help an analyst, the methods designed with it have to be easily readable and understandable, they have to use natural language to communicate with the database engineers.
- Since a method written with this language is to be followed by human beings, it is to be clear, ambiguities should be avoided, even if they could easily be resolved with some priority rules; analysts should not have to learn such rules.
- The language has to handle all the concepts defined in Chapters 2, 3 and 4: product models, product types, and process types.

5.2. Language definition

The language is built mainly as a transcription of the concepts presented in chapter 4. This chapter simply details the syntax and the semantics of this new language. In a first time, the main characteristics of the language will be defined. Secondly, the language will allow the method engineer to give a description to the method. Then, successively, a way to define product models, global product types, toolboxes and external functions will be presented. Finally, process models will be defined altogether with their local product types and their strategy. In this chapter, the language syntax and semantic are presented in a "programmer's manual" way. A full syntax description of the language is given with a BNF grammar in Appendix D.

5.2.1. Generalities

The meta-model is made up of several concepts. To declare a method, a method engineer needs to define elements of each kind of concepts: product models, products types, process types. Since all these concepts are independent from each other (they just reference each other) they will be defined in separate blocks:

- One block for the method itself.
- One block for each product model.
- One block for each global process type.
- One block for each engineering process type, with its own local product types.
- One block for each primitive process type that need a particular description.

In Chapter 2 product types were defined in the context of the process type into which they are used. In other words, these product types are defined locally to process types. The context of a product type can be the whole project too. For example, a product type whose instances are special annotations may have to be available at any time during the project.

At the most basic level, engineering process types use primitive process types. In Chapter 4, the different kinds of primitive process types that can be encountered are enumerated. The requirements for the four primitive process type categories are the following:

- Basic automatic process types are simple built-in functions of the CASE environment that do not need any configuration, their name is sufficient to use them.
- Configurable automatic process types are of three categories. Firstly, global transformations have to be declared entirely. Secondly, external procedures simply need to be referenced. To do so, they can be declared and given a name that identifies them inside the whole process. Finally, configurable DDL generators can be referenced through their own name and configured either by a few parameters or by their own means (setup function if they have one).
- User configurable automatic process types can be referenced by their own name. Their configuration should be performed by their own means directly before their actual use.
- Manual process types being the use of a toolbox, it is necessary to define these toolboxes and to give them an identifying name.

So, special blocks are needed to declare external procedures and to define toolboxes before they can be used. All other primitive process types will be declared directly when needed.

Finally, for the ease of reading a method and to avoid recursion, the language will not accept forward references. In other words, a block can only reference a block which was defined before. For instance, if the method looks like the following:

Block A

Block B

Block C

Then block C can make use of blocks A and B, while block B can only use block A, and block A cannot use B nor C.

5.2.2. Method

The syntax of method identification block is the following:

> **method**
>   **title** "*title*"
>   **version** "*version*"
>   [**description**
>   *description-text*
>   **end-description**]
>   **author** "*author*"
>   **date** "*day-month-year*"
>   [**help-file** "*help-file-name*"]
>   **perform** *process-type*
> **end-method**

where:

- *title* is the name of the method. It can be made of any character (max. 100).
- *version* is a version number. It can be made of any character (max. 16).
- *description-text* is an optional small description of the method that will appear in dialogue boxes in the supporting CASE environment. This text can hold on multiple lines. The first character of a line will go far left. The left margin can be symbolised with "|"

(ASCII code 124). In that case, this character will not appear in the dialogue boxes, but spaces between it and the text will. For instance, the following description:

```
description This is a
|       sample
      description
end-description
```

will be shown as:

```
This is a
      sample
description
```

- *author* is the name of the author. It can be made of any character (max. 100).
- *day-month-year* is the release date of the method. *day*, *month* and *year* are three integer numbers. The *year* must be coded with four digits.
- *help-file-name* is a filename containing on-line help about the method. This file should be the detailed handbook of the method. It can be a *.hlp file in a Windows environment or a man page in a Unix environment, for instance.
- *process-type* is the identifier of the process type by which the method begins. This process type must be already defined.

Example:

```
method
   title "Reverse engineering"
   version "1.2"
   description
      This method is aimed at reverse engineering COBOL files in order to retrieve
      the conceptual schema of its data structures and the way they were designed.
   end-description
   author "John Smith"
   date "28-07-2002"
   help "rev_eng_meth.hlp"
   perform REVERSE_ENG
end-method
```

Semantically, this block only indicates what help file must be used to guide the user and what process type block is the main one.

5.2.3. Product Models

A. Schema model description

The definition of a schema model follows the following pattern:

```
schema-model name [is inherited-schema-model]
   title "title"
   [description
      description-text
   end-description]
   concepts
      concept-name "local-name"
      concept-name "local-name"
      ...
```

constraints
rule
diagnosis "*diagnosis-string*"
rule
diagnosis "*diagnosis-string*"
...
end-model

where:

- *name* is an identifier that will be used to reference the model throughout the method description. This name must be made of maximum 100 letters (lower case or upper case, but no accents), figures, “-” or “_”.
- *inherited-schema-model* is another schema model from which the current schema model can inherit its definition (concepts and constraints); this is optional.
- *title* is a more readable name of the model that will be used by the supporting CASE environment user interface. It can be made of any character (max. 100). It does not need to be identifying.
- *description-text* is an optional small description of the model that will appear in dialogue boxes in the supporting CASE environment. The syntax is the same as for the method.
- *concept-name* is one of the concepts of the GER model the declared model is made up of. For instance, a relational model has the concept of entity type (renamed *table*, see below) but not the concept of relationship type. So *entity_type* will appear in the list, but not *rel_type*. The allowed concept names are the following:

schema	entity_type
is_a_relation	is_a
sub_type	super_type
rel_type	attribute
atomic_attribute	compound_attribute
referential_attribute	object
processing_unit	group
role	collection
identifier	primary_identifier
secondary_identifier	access_key
coexistence_constraint	exclusive_constraint
at_least_one_constraint	exactly_one_constraint
user_constraint	referential_constraint
inverse_constraint	generic_constraint
in_out_relation	call_relation
decomposition_relation	

- *local-name* is the renaming of a concept into the local model. For instance, the GER concept of entity type will be renamed in an OO model with name *Object class* and in a relational model with name *Table*.
- *rule* is a constraint that each schema expressed into the new model must satisfy. A rule applies on a class of GER concepts. It defined valid configurations. These are the rules defined in Chapter 4. Since the notation and naming used in Chapter 4 is formal and precise, the MDL language can use the same syntax.
- *diagnosis-string* is associated with a rule. It contains a message to be printed on screen when the rule is violated. This message can be made of any character. It can contain the special word ’&NAME’ to include the name of the object that violates the rule.

The following example illustrates the MDL definition of a simple binary model close to the historical Bachman model.

```
schema-model BACHMAN-MODEL
   title "Bachman binary model"
   description
      |Simple Bachman model:
      |           no supertype/subtypes structures,
      |           binary one-to-many rel-types without attributes,
      |           no compound attributes,
      |           no multivalued attributes,
   end-description
   concepts
      project      "project"
      schema       "schema"
      entity_type  "record type"
      rel_type     "set type"
      role         "role"
      attribute    "field"
   constraints
      ISA_per_SCHEMA (0 0)      % No is-a relations allowed
         diagnosis "Is-a relations are not allowed. Transform them."
      ROLE_per_RT (2 2)         % Maximum degree of a rel-type = 2
         diagnosis "Rel-type &NAME must be binary. Transform it."
      ONE_ROLE_per_RT (1 1)   % Only one "one" role (with card [i-1])
         diagnosis "Rel-type &NAME must have one 1 role. Transform it."
      ATT_per_RT (0 0)          % Rel-types cannot have attributes
         diagnosis "Rel-type &NAME cannot have attributes. Transform it."
      SUB_ATT_per_ATT (0 0)     % Attributes must be atomic
         diagnosis "Attribute &NAME cannot have sub-att. Transform it."
      MAX_CARD_of_ATT (1 1)  % Attributes must be single-valued
         diagnosis "Attribute &NAME must be single-valued. Transform it."
end-model
```

The semantics of such a schema model is depicted in details in Chapter 3.

B. Text model description

The specification of a **text model** can be simple when no syntax is enforced. Otherwise, the file including the grammar of the contents of the texts is mentioned.

text-model *name* **is** [*inherited-text-model*]
   title "*title*"
   [**description**
      description-text
   end-description]
   extensions "*extension*", "*extension*",...
   [**grammar** "*grammar*"]
end-model

where:

- *name* is an identifier that will be used to reference the model throughout the method description. This name must be made of maximum 100 letters (lower case or upper case, but no accents), figures, “-” or “_”.

- *inherited-text-model* is another text model from which the current text model can inherit its definition extensions; this is optional.
- *title* is a more readable name of the model that will be used by the supporting CASE environment user interface. It can be made of any character.
- *description-text* is an optional small description of the model that will appear in dialogue boxes in the supporting CASE environment. See the method *description-text* for the syntax.
- *extension* is a possible file extension for a file containing a text of this model. As file extensions are usually associated with the same kind of files, they suffice for describing the content of a file. For instance, **extension** "cob" means that texts of this model are all COBOL files, therefore they are texts with a COBOL syntax. An extension can be made of any character (max. 100).
- *grammar* is the name of a file that contains the grammar description as presented in Chapter 3, Section 3.4. This is optional.

Text models do not have a concept selection and renaming list like schema models. This is due to the fact that the DB-MAIN CASE environment is mainly oriented towards schema manipulation and only treats a text as a single indivisible element, except in some dedicated process. DB-MAIN is not capable of distinguishing text parts like a word processor, making text concept selection and naming useless.

The following are two examples of text models:

```
text-model PLAIN-TEXT
    title "Plain ASCII text"
    description
        ASCII file that can be read by text editors
    end-description
    extensions "rpt", "txt"
end-model

text-model COBOL-PROGS
    title "COBOL programs"
    extensions "cob"
    grammar "COBOL.PDL"
end-model
```

The semantics of such a text model is depicted in details in Chapter 3.

5.2.4. Global product types

Global product types are defined in their own paragraph. products of these types are accessible by all process types.

The syntax of global product type description is the following:

```
product name
    title "title"
    [description
        description-text
    end-description]
    model [weak] model-name
    [multiplicity [min-max]]
end-product
```

where:

- *name* identifies the product type throughout the method. This name must be made of maximum 100 letters (lower case or upper case, but no accents), figures, “-” or “_”.
- *title* is a second name for representing the product type in the supporting CASE environment in a more readable way then the identifier. It can be made of any character.
- *description-text* is an optional free text describing the product type in a natural language. This description is to be used by the supporting CASE environment user interface. Its syntax is the same as the *description-text* of the method.
- *model-name* is the name of the product model the current product type is a type of. It must be the identifier of a previously defined product model (schema model or text model). If the **weak** keyword is specified, products of this type should preferably respect all the constraints declared in the product model, but some transgressions are bearable.
- *min* is the minimum number of products that must be defined with the type along the life of the project. *min* is an integer value.
- *max* is the maximum number of products that can be defined with the type. It is an integer value or **N** to represent infinity.

Note that the **multiplicity** line is optional. When it is not specified, *min* is assumed to be equal to 0 and *max* is assumed to be equal to **N**.

Here is an example of a product type.

```
product Optimized Schema
    title "Logical Optimized Schema"
    description
        Logical binary schema including optimization constructs
    end-description
    model BACHMAN-MODEL
    multiplicity [0-1]
end-product
```

The semantics of such a product type is depicted in details in Chapter 4.

5.2.5. Toolboxes

A toolbox is a subset of the supporting CASE environment tool kit that can be used at a particular time. It is aimed at being used by manual primitive process to let the analysts work by themselves and to prevent them to do mistakes by allowing them to use some particular tools only. Several toolboxes can be defined by the language. The process types defined afterward will allow the use of the toolboxes when needed. A toolbox has an identifying name, a readable title, possibly a textual description and the list of tools. Toolboxes can be defined hierarchically. If a toolbox is defined on the basis of another toolbox, it inherits all its tools. The new toolbox is then defined by adding or removing tools from the original toolbox. The syntax of a toolbox description is the following:

```
toolbox name [is inherited-toolbox]
    title "title"
    [description
        description-text
    end-description]
    add|remove tool-name
    add|remove tool-name
    ...
end-toolbox
```

where:

- *name* identifies the toolbox in the method. This name must be made of maximum 100 letters (lower case or upper case, but no accents), figures, "-" or "_".
- *inherited-toolbox* is the name of another toolbox from which the new one inherits its definition. This is optional.
- *title* is a second, more readable, name that will be used in the supporting CASE environment user-interface. It can be made of any character.
- *description-text* is an optional free text describing the toolbox in a natural language. This description is to be used by the supporting CASE environment user interface. Its syntax is the same as the *description-text* of the method.
- *tool-name* is the name of a tool to add to or to remove from the toolbox. This name is a predefined name provided by the supporting CASE environment. Appendix D lists all the tools provided by DB-MAIN. The number of tools that can be added is unlimited.

The following shows an example of a toolbox description.

```
toolbox TB_BINARY_INTEGRATION
   title "Binary schema integration"
   description
      This toolbox allows you to integrate a slave schema into a master schema.
   end-description
   add SCHEMA_INTEGRATION
end-toolbox
```

Semantically, a toolbox definition is purely static, it only describes the content of the toolbox. Information about its use will be given later, in process type definitions.

5.2.6. External function declarations

External functions are primitive process types that have to be performed by third-party tools. In order for them to be accessible, they have to be declared with their signature. These special functions will be developed in a 4GL. Voyager 2 is the 4GL of DB-MAIN that can be used for that purpose. The syntax of such a declaration is:

extern name **"***voyager-file***"**.*voyager-function***(***param-type* [*param-name*]**,...)**

where:

- *name* is the name by which the function will be identified throughout the method.
- *voyager-file* is the compiled Voyager 2 file name (*.oxo) that contains the function.
- *voyager-function* is the name of a Voyager 2 function that is defined in *voyager-file*. It must be declared exportable and return an integer value. The semantic of this integer value depends on the intended use of the function:
  - If the function is a boolean expression, a value of 0 means false and all other non-null value means true.
  - If the function is a primitive process type, it should return 1 if it performs correctly and 0 if an error occurs; other values are undefined and cannot be returned. The function has to handle error messages by itself.
- *param-type* is a formal parameter of the function. It can take various values according to the actual function which has to be written with respect to the method requirements:

- To pass an integer value in input of the actual function, it must be defined with an integer parameter and *param-type* must be **integer**.
- To pass a string in input of the actual function, it must be defined with a string parameter and *param-type* must be **string**.
- To pass a product type in input or in update of the actual function, it must be defined with a parameter of type *list* and *param-type* must be **list**. When the function is called, the list is initialised with all the products of the type passed. The function cannot modify the list (add or remove products) but the products can be modified.

  To pass a product type in output so that the function can create new products of the passed type, the function has to be defined with a product type parameter and *param-type* must be **type**. The Voyager 2 function has to create the new product with the *create* instruction; for instance, to create a schema of type "st" (passed in parameter), the Voyager 2 function should contain the following line:
  create(SCHEMA,...,SCHEMA_TYPE:st)

- *param-name* is the name of the parameter. It is optional. This name is only used for readability of the source code; it is simply skipped by the compiler.

For instance, a Voyager 2 function can be defined in file c:\functions\lib.oxo as:

export function integer F(list L, integer I, product_type T) {...}

So it needs to be declared with the following line:

extern extf "c:\functions\lib.oxo".F (**list**, **integer**, **type**)

In the method, this function is known as *extf* and needs a product type whose instances will be passed in input or update, an integer value, and a product type for the products that will be generated in output.

An external function declaration is only a reference definition. Its use is defined later.

5.2.7. Process types

Besides general practical information (its name, its title, a short description, a help text), a process type is defined by its input and output product types, its internal product types and sets and by a strategy.

A. The process description

The MDL specification of a process type states the input/output flows of the process, as well as the way it must be carried out. It has the following syntax:

process *name*
  title "*title*"
  [**description**
    description-text
  end-description]
  [**input** *input-product-type*, *input-product-type*,...]
  [**output** *output-product-type*, *output-product-type*,...]
  [**update** *update-product-type*, *update-product-type*,...]
  [**intern** *intern-product-type*, *intern-product-type*,...]
  [**set** *product-set*, *product-set*,...]
  [**explain** "*explain-section*"]
  strategy
    strategy
end-process

where:

- *name* identifies the process type in the method. This name must be made of maximum 100 letters (lower case or upper case, but no accents), figures, "-" or "_".
- *title* is a second, more readable, name of the process type that will be used in the supporting CASE environment user-interface. It can be made of any character.
- *description-text* is an optional free text describing the toolbox in a natural language. This description is to be used by the supporting CASE environment user interface. Its syntax is the same as the *description-text* of the method.
- *input-product-type* is a local product type used as a formal parameter for input products. Products of this type are renamed copies of actual arguments that are produced at the enactment of a process of type *name*. Modifications done on these products are lost at the end of the process.
- *output-product-type* is a local product type used as a formal parameter for output products. Products of this type must be created during a process of type *name*. At the end of the process, products of this type are copied into the actual arguments.
- *update-product-type* is a local product type used as a formal parameter for updated products. Products of this type are the actual arguments themselves. Hence, every modification done to a product of this type is done on the corresponding actual argument too.
- *intern-product-type* is a local product type which is not a formal parameter. Hence, products of this type have no existence outside processes of type *name*.
- *product-set* is a local product set that can be used for handling large quantities of products by using set operators. Product sets are described below.
- *explain-section* is the section of a help file that explains the goal and the way of working of any process of type *name*. This section has a name that can be made of any character allowed by the help system (help or man files).
- *strategy* is the way of carrying out the processes of type *name*, as described below.

B. Local product types

Global product type declaration was presented in 5.2.4. The semantics of global and local product types is the same, the only difference is in the scope: global product types can be used anywhere in the method, while local product types can only be referenced in the strategy of the process type in which they are declared.

Properties of global and local product types are the same. They all have a name (identifier), a title, a minimum and maximum multiplicity and they are all of a product model. But, local product types do not have a description. Their definitions hold in a single line:

name [[*min-max*]] ["*title*"] **:** [**weak**] *model-name*

where:

- *name* identifies the product type inside de process type. This name must be made of maximum 100 letters (lower case or upper case, but no accents), figures, "-" or "_".
- *min* is the minimum number of products of this type that must be used (or created) during a work that follows the method. It is an integer value.
- *max* is the maximum number of products of this type that can be used (or created) during a work that follows the method. It is an integer value or **N** to represent infinity.
- *title* is a second name that is aimed at representing the product type in the supporting

CASE environment in a more readable way then the identifier. It can be made of any character. It is optional. If omitted, it is assumed to be the same as *name*.

- *model-name* is the name of the product model the current product type is a type of. It must be the identifier of a previously defined product model (schema or text model).
- If the **weak** keyword precedes the *model-name*, products of this type should preferably respect all the constraints declared in the product model, but some transgressions are bearable.

Note that the multiplicity is optional. By default, *min* = 1 and *max* = **N**.

For instance, the declaration of a conceptual schema integration process may comprise two input product types *master* and *secondary* both compliant with a conceptual model. The first one, with multiplicity [1-1], represents the master schema, and the second one, with multiplicity [1-*N*], represents all the secondary schemas that will be integrated into the first one.

C. Product sets

Product sets have a name (identifier), a title and a minimum and maximum cardinality. Their definitions hold in a single line:

name [[*min*-*max*]] ["*title*"]

where:

- *name* is a name for the product set, unique inside the process type. *name* must be made of maximum 100 letters (lower case or upper case, but no accents), figures, “-” or “_”.
- *min* is the minimum number of products in this set. It is an integer value.
- *max* is the maximum number of products of this set. It is an integer value or **N** to represent infinity.
- *title* is a second name that is aimed at representing the product set in the supporting CASE environment in a more readable way then the identifier. It can be made of any character (max. 100). It is optional. If omitted, it is assumed to be the same as *name*.

Note that the multiplicity is optional. By default, *min* = 1 and *max* = **N**.

D. The explain section

It is very important for a database engineer to understand very well the ins and the outs of the problem to be solved by an engineering process, to be aware of all the the related facts and information,... The small description section sketches the main line to follow, but it is sometimes necessary to be more precise. Furthermore, small drawings may greatly help to improve explanations. Since the method already contains a link to the detailed handbook of the method which can be written using all the capabilities offered by the supporting operating system, this *explain* section is a simple link to a section in that help file.

E. The strategy

The strategy is declared in a procedural way with the control structures described in Chapter 4. The syntax of their translation in MDL is defined here. Their semantics was described precisely in Chapter 4 and will be presented in an operational way in Chapter 9.

a. The sequence

A sequence of process types means that all the process types must be done each at its turn, in the specified order. The syntax of a sequence is the following:

[**sequence**]
sub-structure;
sub-structure;
...
[**end-sequence**]

where:

- *sub-structure* is one of the substructures or sub-process use defined in this chapter.

Note that the **sequence** and ***end-sequence*** keywords are optional. They are normally not used, except when necessary, for instance when a sequence is an alternative in a *one*, *some* or *each* structure defined below.

The following example shows a sequence made of process types *Conceptual_analysis* and *Logical_design*:

do Conceptual_analysis(Interview_report,Conceptual_schema);
do Logical_design(Conceptual_schema,Logical_schema)

b. The *while* structure

The *while* structure is a standard loop which indicates that the encompassed structure must be done again and again while the condition is satisfied. If the condition is not satisfied the first time it is evaluated, then the sub-structure will never have to be performed. It can also be an informal loop or a weak loop, depending on the condition. The syntax of the structure is the following:

while *condition* **repeat**
sub-structure
end-repeat

where:

- *condition* is an expression the syntax and semantics of which is discussed in Section j.
- *sub-structure* is any structure or sub-process use as described in this chapter.

In the following example, the structure show that a process of type *Import* can be done several times until condition *ask "Do you want to import a source file?"* is satisfied.

while (ask "Do you want to import a source file?") **repeat**
do import(Source_file)
end-repeat

c. The *repeat...until* structure

The *repeat...until* structure is a second kind of standard loop which indicates that the encompassed structure must be done again and again until the condition is satisfied. The sub-structure must be done at least once. It can also be an informal or a weak non-deterministic loop. The syntax of the structure is the following:

repeat
sub-structure
end-repeat until (*condition*)

where:

- *condition* is an expression the syntax and semantics of which is discussed in Section j.
- *sub-structure* is any structure or sub-process use as described in this chapter.

The following example shows that processes of type *Import* should import products of type *Source_file* until condition *ask "One more source file?"* is satisfied.

repeat
  do import(Source_file)
end-repeat until (ask "One more source file?")

d. The repeat structure

The *repeat* structure is the informal non-deterministic loop. It looks similar to the *repeat...until* structure except that no condition is specified. During a process, the analyst is the one who decides if the sub-structure has to be performed one more time. The syntax is:

repeat
  sub-structure
end-repeat

where:

- *sub-structure* is any structure or sub-process use as described in this chapter.

The following example shows a sample *repeat* structure.

repeat
  Import(source_file)
end-repeat

e. The if...then...else structure

The standard alternative, the informal and the weak non-deterministic alternative can be translated with an *if...then...else* structure. According to a specified condition the methodological engine or the analyst can decide whether an action has or has not (*if...then*) to be performed, or which of two alternatives (*if...then...else*) comes next. The syntax is:

if *condition* **then**
  sub-structure-1
[**else**
  sub-structure-2]
end-if

where:

- *condition* is an expression the syntax and semantics of which is discussed in Section j.
- *sub-structure-1* is any structure or sub-process use as described in this chapter. It is executed when *condition* is satisfied.
- *sub-structure-2* is any other structure or sub-process use as described in this chapter. It is optional. If it is present, it is executed when *condition* is not satisfied.

The following example shows the graphical representation of an *if...then...else* structure where a process of type *Two_schemas* is executed if condition *count-equal(SCH,2)* is satisfied, and a sequence of processes of types *Several_schemas* and *Selected_schemas* otherwise.

if (**count-equal**(SCH,2))
  do Two_schemas(SCH)
else
  do Several_schemas(SCH,SEL)
  do Selected_schemas(SEL)
end-if

f. The one, some, each structures

The *one*, *some* and *each* structures are the non-deterministic one, some and each alternatives. They are user driven structures. The *one* structure means that the user has to choose one structure among all those that are presented and to execute it and no other one. The *some* structure means that the user can choose several (or just one or none or all) sub-processes and execute them. He or she can do them in any order. Finally, the *each* structure means that the user must execute each sub-structure but, on the contrary of a sequence, in any order he or she wants. The syntax of those substructures is the following:

one	**some**	**each**
sub-structure;	*sub-structure;*	*sub-structure;*
sub-structure;	*sub-structure*;	*sub-structure*;
...	...	...
end-one	**end-some**	**end-each**

where:

- *sub-structure* is any other structure or sub-process use as described in this chapter.

The following example allows a database engineer to generate either an *Oracle_script* or a *DB2_script* DDL, or both.

> **some**
>   **generate** Oracle_script(Physical_schema,DDL_file);
>   **generate** DB2_script(Physical_schema,DDL_file);
> **end-some**

g. The for structure

A product type can have several instances. But some process types can need to work on one product only. The *for* structure allows a process type to be executed once for every instance of a product type or product set. The syntax of the *for* structure is the following:

> **for one** *product-set* **in** *product-type-or-set* **do**
>   *sub-structure*
> **end-for**

> **for some** *product-set* **in** *product-type-or-set* **do**
>   *sub-structure*
> **end-for**

> **for each** *product-set* **in** *product-type-or-set* **do**
>   *sub-structure*
> **end-for**

where:

- *product-set* is a product set that must be declared with multiplicity [1-1]. At each iteration, the set is filled with one element of the *product-type-or-set*. The element is the product type whose instance is one, some or each instance or *product-type-or-set* at its turn.
- *product-type-or-set* is the product type the instance of which have to be used one at a time. In the **for one** form, one instance of product-type must be used. In the **for some** form, the user has to choose a set of products of type product-type to use. Finally, in the **for each** form, every product of product-type has to be used.
- *sub-structure* is any other structure or sub-process use as described in this chapter.

In the following example, each instance of *All_schemas* at its turn is used as the only element of the *One_schema* set and used as an input for *Integrate*.

for each One_schema **in** All-schemas **do**
  do Integrate(One_schema, Integrated)
end-for

h. Sub-process use

The previous sections showed how to specify a strategy, a way of combining several sub-processes. But they do not show how to declare a sub-process. This section will have a look to every available sub-process types.

i. To use a sub-process

A process-type can be refined into sub-process types, each one being a complete engineering process type with their product definitions and strategies. The **do** keyword allows a process to use its engineering sub-processes.

do *sub-process* (*parameter*, *parameter*,...)

where:

- *sub-process* is the identifier of the engineering process to use.
- *parameter* is an integer, a string, a product type or a product set (they will be distinguished in this paragraph) passed to the sub-process. The parameters must be in the same order as declared in the sub-process. Product types and product sets need to be I-compatible, O-compatible or U-compatible with the formal parameters declared in the sub-process, as defined in Chapter 4. A product set can only be passed to a **list** argument. If a parameter is a product type passed to a **list** argument, all the products of that type will be passed to the sub-process. If the parameter is a product set, the set itself will be passed, but only the products it contains that are I-compatible or U-compatible with the formal product type will be in the set inside the sub-process. But the product set parameter can be prefixed by "**content:**" in order to pass only the products it contains rather than itself. A Product type passed to a **type** argument will be used by *sub-process* to build new products of that type.

The following example shows a process use example. Process Q uses process P passing W, X, Y and Z in parameters. When the use is required, every products of type W are passed and cast to product type A, the set Y is passed and all its products I-compatible with X are cast to X, and every products of the set Z which are I-compatible with C are passed and cast to type C. When process P ends, all products of type D are cast to type Z and the control is passed back to process Q that goes on.

process P
  ...
  input A,B,C
  output D
  ...
end-process

process Q
  ...
  intern W,X
  set Y,Z
  strategy
    ...

P(W,X,**content:**Y,Z)
...
end-process

ii. To allow the use of a toolbox

Toolboxes have already been defined previously. The strategy simply shows what toolbox can be used and on what product types. The syntax is the following:

toolbox *toolbox*[[**log** *log-level*]]**(***product-type-or-set*,*product-type-or-set*,...**)**

where:

- *toolbox* is the identifier of a previously defined toolbox.
- *log-level* is an optional configuration parameter which specifies how the actions performed by the analyst should be performed. It can be one of the following values:
  - *off*: turns off the logging facility
  - *replay*: concise recording facility: the log will contain only the information that are necessary to replay the actions performed. This includes only the identifier of the components that are transformed, the transformations performed and the data entered by the analyst.
  - *All*: extended recording facility: the log file contains all the same information as in the *replay* log plus the state before transformation of all the components that are modified by the transformation. For instance, the transformation of an entity-type into entity rel-type will log the name before transformation of the entity-type, and the name of all rel-types connected to that entity-type, as well as the name of all roles played in the rel-types. This is usefull to be able to reverse the transformation.
- If the [**log** ...] configuration parameter is not present, the default log state of the supporting CASE environment will be used.
- *product-type* is the identifier of a product type or of a product set. *toolbox* can work on every instances of *product-type-or-set*. The number of *product-type-or-set* used as actual parameters of a toolbox is unlimited.

The following example shows an example of a toolbox use: *A* can be updated freely by the analyst using toolbox *TB*.

toolbox TB
...
end-toolbox

process P
...
update A
strategy
...
toolbox TB(A);
...
end-process

iii. To perform a global transformation

The usage of global transformations, automatic configurable primitive process types, is the following:

glbtrsf ["*title*"][[**log** *log-level*]] (*schema-type-or-set*,
global-transfo[(*scope*)],
global-transfo[(*scope*)],...)

where:

- *title* is an optional readable string to name the transformation on screen for the user.
- *log-level* is an optional configuration parameter which specifies how the actions performed by the analyst should be performed. It can be one of the values defined in the toolbox section. If the [**log** ...] configuration parameter is not present, the default log state of the supporting CASE environment will be used.
- *schema-type-or-set* is a group of schema to work on; all the schemas of that type or set will be transformed.
- *global-transfo* is the identifier of a global transformation as defined in Chapter 4. All these identifiers are listed in Appendix C.
- *scope* is a schema analysis structural rule (see Chapter 3) that defines the scope of the transformation. It is optional. If it is not present, the default scope is used, according to the transformation. If it is present the rule will reduce the default scope.

For instance, the following global transformation will transform all the rel-types of schema S into entity types:

glbtrsf "All rel-types into entity types" (S,**RT_into_ET**)

while the following one will only transform non-binary rel-types into entity types:

glbtrsf(S,**RT_into_ET**(**ROLE_per_RT**(3 N)))

iv. To use an external function

The external functions declared previously can be used the in following way:

external *extern-function* [[**log** *log-level*]] (*parameter*,*parameter*,...)

where:

- *extern-function* is the name of a Voyager 2 function that was previously declared.
- *log-level* is an optional configuration parameter which specifies how the actions performed by the analyst should be performed. It can be one of the values defined in the toolbox section. If the [**log** ...] configuration parameter is not present, the default log state of the supporting CASE environment will be used.
- *parameter* is an actual argument to pass to the function. It must match the upper declaration. A parameter declared as **integer** must receive an integer number. A parameter declared as **string** must receive a double-quoted string. A parameter declared as **list** can receive any product type or product set; all the products of a product type will be passed in a list to the function that can use or modify them; a product set will be passed itself and the external function has to handle the set; and all the products of a product set prefixed by the **content:** keyword will be passed like the products of a product type. Finally, a parameter declared as **type** can receive any output or intern product type. Products of these types will not be accessible inside the function, but the function will be able to create new products of that type. To allow an external function to both use the existing products of a given type P and create new products of the same type P, the function has to be defined with two parameters, one being a list and the other being a product type, and P has to be passed to both parameters.

The following example shows an external function use: products of type *A* can be updated

by function *F* using other parameters.

```
extern F "c:\library\lib.oxo".f(list,string,integer)
...
process P
    ...
    update A
    strategy
        ...
        external F (A,"string",10);
        ...
end-process
```

v. To use a data extractor

The supporting CASE environment should be able to import data structures from a text into a schema (For example, COBOL data division into a COBOL compliant schema). The procedure that allows this extraction is the following:

extract *extractor*(*source-text*,*destination-schema*)

where:

- *extractor* is the identifier of the data extractor to use. It depends on the supporting CASE environment (DB-MAIN recognise SQL, COBOL, IDS_II and IMS).
- *source-text* is a text-type or a set that should only contain texts. All the texts of this type or set will be analysed.
- *destination-schema* is a schema type. All schemas generated by the process will be of this type.

Example:

extract COBOL(COBOL_FILE,COBOL_SCHEMA)

allows the CASE environment to extract COBOL data structures from COBOL source files into COBOL compliant schemas.

vi. To use a generator

The supporting CASE environment should be able to generate database creation scripts from schemas. The following process does the job:

generate *generator*(*source-schema*,*destination-text*)

where:

- *generator* is the identifier of the generator. It depends upon the supporting CASE environment (DB-MAIN versions 3 and more recognise STD_SQL, VAX_SQL, ACA_SQL, STD_SQL_CHK, VAX_SQL_CHK, ACA_SQL_CHK, COBOL, IDS).
- *source-schema* is a schema type or a set that should only contain schemas. All schemas of this type or set will be used to generate the new text files.
- *destination-text* is a text type: the type of all the texts that will be generated.

Example:

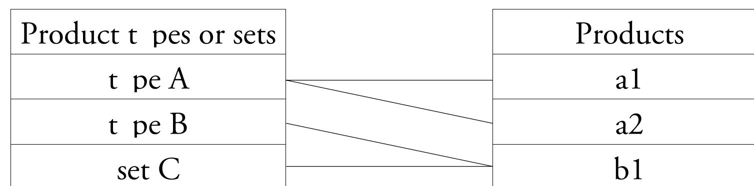
generate COBOL(COBOL_SCHEMA,COBOL_FILE)

allows the CASE environment to generate files containing COBOL data divisions from COBOL-compliant schemas.

i. Built-in procedures

The MDL language also contains a few built-in procedures that can be used in the same ways as sub-processes. All these built-in functions are aimed at handling product sets.

Each built-in procedure will be applied to the following example. It shows two product types and one product set: product type A has two instances (products a1 and a2), product type B has one instance (product b1), and product set C contains the product b1.



i. To create a new product of a given type

When a process type has to produce an output product, it is sometimes necessary to build it completely. The **new** keyword allows a process to generate a blank product, the name of which will be asked to the analyst. This command needs one argument which is a product type. The syntax of the command is the following:

new (*product-type*)

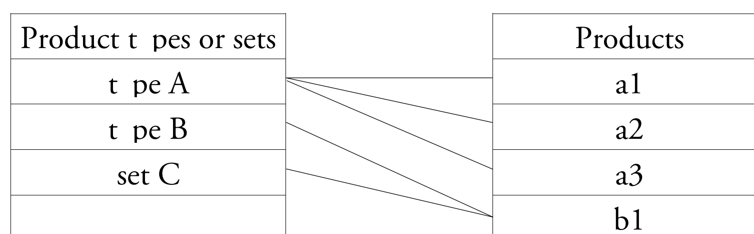
where:

- *product-type* is the type of the new product to generate. At run-time, the product type will have one more instance. If the product type is a schema type, the new instance will be a blank schema; if the product type is a text type, the user will be prompted for the name of an existing file, and the new instance will be a reference to that file.

In the example above, the command

new (A)

gives:



ii. To import a schema from another project

When a schema already exists in another project, it is sometimes more interesting to import it in the new project then to redraw it. Import can also be useful with big projects: several analysts work on separate sub-projects, and, in a phase of importation and integration, all these sub-projects are assembled in a master one. This command needs one argument which is a product type. The syntax of the command is the following:

import (*product-type*)

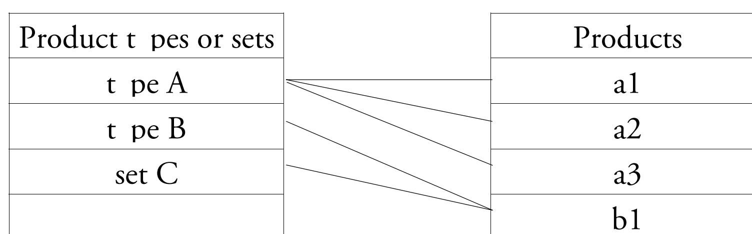
where:

- *product-type* is the type of the schema that will be imported. At run-time, the schema type will have one more instance, which is the imported schema.

In the example of Figure 18, the command

import (A)

gives:



iii. To make a copy of a product

When a process type has to generate output products, it is sometimes possible to make a copy of other products and to modify the copies. The **copy** procedure allows a process to duplicate each product of a set and to cast it to the specified type. The new products have the same name as the original ones, but they have a different version number which is requested to the analyst. The syntax of the **copy** command is the following:

copy (*source-product-type-or-set*,*destination-product-type*)

where:

- *source-product-type-or-set* is the product set to copy.
- *destination-product-type* is the product type that will receive the copies.

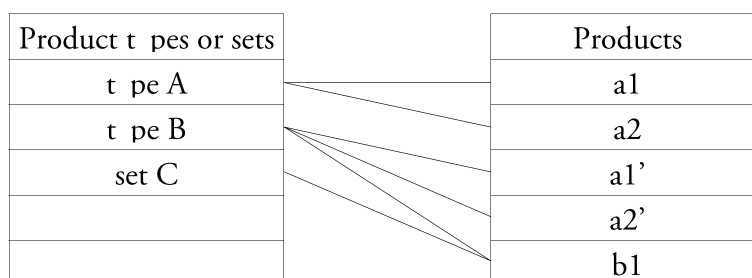
Note that the *source-product-type-or-set* and the *destination-product-type*, if they are both product types, must be of the same model, or the model of the *source-product-type* must be a sub-model of the model of the *destination-product-type*.

If the source is a product type, all the products of that type will be copied. If the source is a product set, all its products will be copied and the set will contain all the new products and only them. If the source is a product set prefixed by "**content:**", all its products will be copied, but the set will not be modified, it will still contain the original products.

In the example above, the command

copy (A,B)

gives:



where a1 is identical to a1′ and a2 is identical to a2′.

iv. To define a product set as the result of a computation

A new set can be built on the basis of other sets or product types. For instance, standard set operators (union, intersection, subtraction) can be used to combine sets. The syntax of the **define** command is the following:

define (*product-set*,*set-expression*)

where:

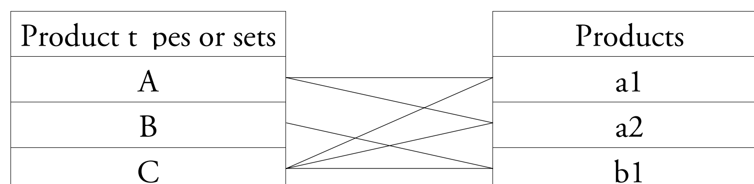
- *product-set* is the new product set, result of the *set-expression*.

- *set-expression* is one of those below. The first seven are deterministic, computer driven and the two last are user driven. In these definitions, *set* is either a product type, a product set or the result of another set expression.
  - *set*, the set expression simply is a product set or a product type.
  - **union** (*set-expr1*,set-*expr2*), the standard union operator (*set1* ∪ *set2*) where *set1* is the result of *set-expr1* and *set2* is the result of *set-expr2*, two set expressions.
  - **inter** (*set-expr1*,*set-expr2*), the standard intersection operator (*set1* ∩ *set2*) where *set1* is the result of *set-expr1* and *set2* is the result of *set-expr2*, two set expressions.
  - **minus** (*set-expr1*,*set-expr2*), the standard difference operator (*set1* \ *set2*) where set1 is the result of *set-expr1* and *set2* is the result of *set-expr2*, two set expressions.
  - **subset** (*set-expr*,*rule*) to extract a sub-set of products out of a product set (result of set expression *set-expr*); the rule is a structural rule; the resulting subset is made up of all the products of the *set* that satisfy the rule.
  - **origin** (*set-expr*) defines a set of products made up of the origin of the products in the result of *set-expr*. The origin of a product, according to the history, is the set of products that were used to generated the given product.
  - **target** (*set-expr*) defines a set of products made up of the target of the products in the result of *set-expr*. The target of a product, according to the history, is the set of the products that are produced by using the given product.
  - **choose-one** (*set*-expr) asks the user to choose one product in the resulting set of *set-expr* and defines a new set with it.
  - **choose-many** (*set-expr*) asks the user to choose one or many products in the resulting set of *set-expr* and defines a new set with them.
  - **first** (*set-expr*) defines a new set containing one product from *set-expr*. The product that will be chosen is the first one in insertion order.
  - **last** (*set-expr*) defines a new set containing one product from *set-expr*. The product that will be chosen is the last one in insertion order.
  - **remaining** (*set-expr*) defines a new set containing all elements from *set-expr* except one. This one is the result of **first**(*set-expr*).
    Hence, *set-expr* = **union**(**first**(*set-expr*),**remaining**(*set-expr*))
    and **inter**(**first**(*set-expr*),**remaining**(*set-expr*)) is empty.

In the example above, the command

define (C,**union**(A,B))

gives:



j. Expressions

Some control structures (**if**...**then**...**else**, **while**, **repeat**...**until**) need an expression. This section will examine every possible form of expression. They can be formal and strict, formal

but not strict, or even not formal at all, making alternatives and loops to be standard, weak or informal.

An expression is made of boolean functions which can be combined with standard boolean operators (*and*, *or*, *not*). There are two kinds of functions: product evaluation functions that concern the syntax or semantics of products and product set evaluation functions that concern the content of a product set without looking at the products themselves.

i. The exists function

Does it exist some objects in the given schema for which the schema analysis constraints are satisfied?

exists (*schema-type-or-set*,*schema-analysis-constraints*)

where:

- *schema-type-or-set* is the group of schemas to analyse. Every schema of this set or type is analysed. The answer of the **exist** function is *yes* if the result is *yes* for at least one schema.
- *schema-analysis-constraints* is a list of comma-separated schema analysis constraints such as presented in Chapter 3.

This is a strong condition which must be satisfied, except if the **weak** keyword is appended in front of it:

weak exists (*schema-type-or-set*,*schema-analysis-assistant*)

This is a weak condition: it is better if it is satisfied, but it is not mandatory. At runtime, the result of the evaluation will be presented to the user and he or she will be the one who decides whether to keep the result (*yes* or *no*) or force the opposite.

ii. User oriented textual condition

A message in clear text can be printed on the screen for the user to take a decision:

ask "*string*"

This is always a weak condition, the user being the only actor who can take the decision.

iii. The model function

Are the products of the given set conform to the given model ?

model (*product set*,*product model*)

- *product set* is the set of products to analyse. Every product of this set is analysed. The answer of the **model** function is *yes* if the result is *yes* for every product.
- *product model* is one of the product models defined in a *schema-model* or *text-model* section of the method.

This is a strong condition. But, like for the **exists** function, the **weak** keyword can be appended in front of it:

weak model (*product set*,*product model*)

iv. External Voyager 2 function

Schema analysis functions allow the user to specify formal expressions, but they are limited. More complex functions can be written in the Voyager 2 language and used with the **external** keyword:

external *function* (*parameter*,*parameter*,...)

where:

- *function* is the name of the Voyager 2 function declared previously.
- *parameter* is a parameter to be passed to the function. All the comments concerning the parameters that were made above about external functions as primitive process types are still valid.

This is a strong condition. But, like for the **exists** function, the **weak** keyword can be appended in front of it:

weak external *function* (*parameter*,*parameter*,...)

v. Product set evaluation functions

Is the number of products in the given set greater, equal or less than the given number?

count-greater (*product-type-or-set*,*nb*)

count-equal (*product-type-or-set*,*nb*)

count-less (*product-type-or-set*,*nb*)

count-greater-equal (*product-type-or-set*,*nb*)

count-less-equal (*product-type-or-set*,*nb*)

count-different (*product-type-or-set*,*nb*)

where:

- *product-type-or-set* is the group of products to be analysed.
- *nb* is the reference number, an integer value.

These are strong conditions. But, like for the **exists** function, the **weak** keyword can be append in front of them:

vi. Operators

Complex conditions can be built by linking the simple expressions defined above by the following operators:

- **and**
  This is the standard logical binary operator. Its result is *yes* when, and only when, both its operands are *yes*.
- **or**
  This is the standard logical binary operator. Its result is *yes* when, and only when, at least one of its operands is *yes*.
- **not**
  This is the standard logical unary operator. Its result is *yes* when its operand is *no* and *no* when its operand is *yes*.

5.3. Language analysis

In order to be usable, the MDL language must satisfy a set of properties:

1. Its syntax must be unambiguous: each symbol must have its own function. If, in some cases, the language permits some ambiguous situations, it must provide a means to resolve the ambiguities.

2. It must be possible to write a program that reads and understands MDL methods, that is to say a lexical analyser that is able to recognise each symbol of an MDL listing, that can understand the precise function of each symbol, and that can translate a listing in a format usable by a CASE environment.
3. Its semantic must be unambiguous.
4. It must be compliant with its requirements (Section 5.1).

The compiler principles presented in [AHO,89] will be used to verify these properties. So the reader will refer to this book for a correct definition of the terms used in this section.

5.3.1. The syntax is unambiguous

In this chapter the syntax of the MDL language is described in a more or less formal way, with a good description of the syntax and a rather good explanation of the semantics in natural language. A full, formal description of the syntax of a language can be done with a context-free grammar such as BNF. The full BNF description of the MDL language is listed in Appendix D. The BNF language used there is rather rich and allows us to write the full grammar rather shortly.

Using various techniques (BNF grammar transformation, BNF grammar analysis) presented in [AHO,89] only one ambiguity appears in this language. Let us examine the following strategy:

```
one
   do P1(S);
   do P2(S);
   do P3(S)
end-one
```

According to the syntax, it could be interpreted either as in Figure 5.1 (interpretation *Pa*: the three sub-processes play the same role), as in Figure 5.2 (interpretation *Pb*: the *one* structure has only two components, the second one being a sequence) or as in Figure 5.3 (interpretation *Pc*: the *one* structure has two components, the first one being a sequence).

It was decided to solve the ambiguity in the most intuitive way: *Pa* ; all the components of the **one** structure play the same role. To allow a method engineer to write strategy chunks such as in *Pb* or in *Pc*, the keywords **sequence** and **end-sequence** were added to the language in order to encompass a sequence when needed. So, strategy chunks that express the situations *Pb* and *Pc* can be written, respectively:

```
one
   do P1(S);
   sequence
      do P2(S);
      do P3(S)
   end-sequence
end-one
```

and

```
one
   sequence
      do P1(S);
      do P2(S)
   end-sequence;
   do P3(S)
end-one
```

Obviously, the same ambiguous situation exists with the **some** and **each** control structures.

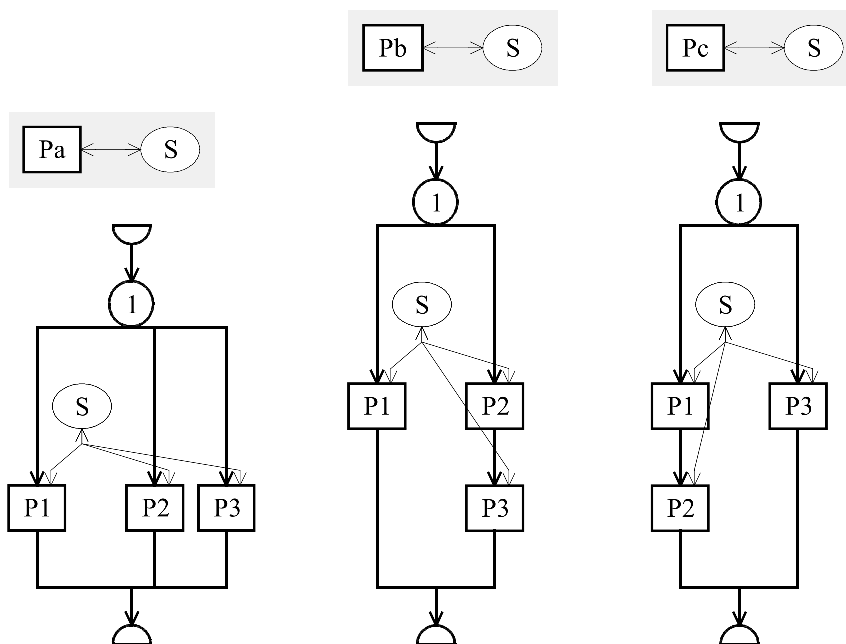


Figure 5.1 A one *structure with three components*

Figure 5.2 A one *structure with two components*

Figure 5.3 Another one *structure with two components*

5.3.2. Syntactical analysis

According to [AHO,89], the MDL language is both an LL(1) and an LR(1) language:

- It is an LL(1) language because it can be parsed and syntactically analysed in a top-down fashion with at most one symbol (a word, a number, a string, a special character,...) read in advance. In other words, at every moment, knowing what has already been read and analysed, it is always possible to predict what symbol can come next. If several possibilities exist, reading only one symbol will determine what possibility is the right one. If a non-predicted symbol is read, then an error is detected. For instance, when the analysis begins or when the analysis of a paragraph ends, what will be read next is known: if the end of the listing is not reached, it is either a new paragraph beginning by "schema-model", "text-model", "product", "extern", "process", "method". The simple fact of reading this single symbol completely determines the type of the paragraph being read and the symbol that must come afterward.
- It is an LR(1) language because it can be parsed and syntactically analysed in a bottom-up fashion by reading at most one symbol in advance. That is to say that, knowing what has been read and analysed, one or several rules of the grammar can be matched. When the analysis has to cope with several possibilities, the only symbol read in advance suffices to determine the right possibility. For example, the MDL grammar (in Appendix D) contains the following rules:
  <action-list> ::= <action>{;<action>}
  which can be rewritten without repetitive part as two separate rules:
  <action-list> ::= <action>
  <action-list> ::= <action> ; <action-list>
  Let us suppose the following characters have been read in an MDL source file:
  ... do conceptual_analysis ; do logical_design ; do physical_design
  The analysis made "do conceptual_analysis", "do logical_design" and "do

pysical_design" match with "<action>" through other rules. Should the <action> "do physical_design>" be matched with the right member of the first rule or with the left part of the right member of the second rule? Looking at the next symbol to be analysed in the input source file will allow us to push the analysis forward. If it is not a semi-colon, the analysed text cannot match with the right member of the second rule. So it must match with the first rule. Hence, "do physical_design", which is an <action>, is also an <action-list> (left member of the first rule). Then, since "do logical_design" is an action, since ";" matches with ";" and since "do physical_design" is an <action-list>, "do logical_design ; do physical design" matches with the right member of the second rule. It is an <action-list> too, and, in the same way, "do conceptual_analysis ; do logical_design ; do physical_design" is also an <action-list>. If, at the contrary, the next symbol is a semi-colon, then the reading of the source file has to be continued so that, later, maybe, an <action-list> will be found and the right member of the second rule will be matched. If this never happens, an error will be detected.

Hence it is possible to write a program that analyses an MDL source. A simple way to analyse an LR(1) language is to use the Lex and Yacc pair of tools. Within the scope of this thesis another technique has been used in order to avoid licensing problems: a new LL(1) analyser was designed from scratch. It reads an MDL source file and produces a syntactic tree of the analysed method which is stored in the repository presented in Chapter 10.

5.3.3. The semantics is unambiguous

A formal analysis of the semantics of a language is much more complex than its syntactical analysis. Several techniques exist, such as the operational and denotational semantics. Since programming languages exist, are designed and implemented, very few of them have been semantically analysed with such formal techniques. And most of the analysis performed are done with languages existing for a long time. In fact, it is the use of the language and pragmatic observations that let people think that a language is semantically unambiguous.

In the MDL language, each keyword, each construct has only one meaning which is not context dependent. During the design of various methods, either debugging tests, examples, cases studies or real methods, no ambiguity ever appeared. So we believe, without formal proof, that the semantic of the language is indeed exempt of ambiguity.

Nevertheless, the MDL language is not exempt of redundant constructs, that is to say different constructs which have the same meaning. They will be studied in Chapter 8.

5.3.4. Compliance with the requirements

In the beginning of this chapter, a few requirements that the MDL language should fulfil were stated. Their fulfilment can be checked.

- The language has to be procedural.

It was conceived that way. It definitely is.

- Methods designed with the MDL language have to be easily readable and understandable, they have to use natural language to communicate with the database engineers.

An MDL listing, like a Pascal listing or a C listing is only readable by specialists. But, once read by the syntactical analyser and stored in the repository (see Chapter 10), the method can be shown graphically, in an algorithmic way, as presented in Chapter 4. The reading of these algorithms still requires some learning, but this typically requires a few minutes (pragmatically observed).

The use of natural language to communicate with the database engineer is omnipresent:

- a readable *title* is attached to each component (product model, product type, process

type or toolbox) of the method in addition to the identifying name for the readability

- diagnosis messages, in the product model description, allow the method engine to show clear messages to the database engineers instead of complex schema analysis formulas
- a description in natural language can be (and should be) added to each component that can be shown to a database engineer upon request, that should give a few explanations about the component (for example, preconditions, postconditions and goals of a process type in free language, a brief translation of the algorithm in free text, a brief description of the constraints of a product model in free text,...)
- a help file with a more global description of the whole method, possibly a tutorial, can also be added to the method
- the only exceptions are the **glbtrsf** command and the **exists** function which use some structural rules as parameters; but they are automatically evaluated by the methodological engine, and an explanation can be included in the description of the engineering process type whose strategy encompass the **glbtrsf** command or **exist** function.

In other words, with a correctly documented method (all the necessary tools are provided for it), a database engineer should not face an unreadable acronym or complex formula without explanation in natural language.

• Since a method written with the MDL language is to be followed by human beings, it is to be clear, ambiguities should be avoided, even if they could easily be resolved with some priority rules; analysts should not have to learn such rules.

The only place in the language where priority rules could not be avoided is in the writing of schema analysis expressions (used in product model descriptions, in conditions for some control structures in strategies, or in **glbtrsf** parameters). Indeed, these expressions use the traditional boolean operators **and**, **or**, **not** which already have a well-known semantic which must obviously be kept. The method engineer has to master these operators. When such expressions are presented to the database engineers, the methodological engine will format them with indentations which make the priorities appear. For instance, in the following expression, the indentation shows that the correct reading is (P1 or(P2 and P3)):

```
P1
or P2
   and P3
```

• The MDL language has to handle all the concepts defined in Chapters 2, 3 and 4: product models, product types, and process types.

Definitely.

Part 2

Histories

Chapter 6

Histories

The history of a database engineering process contains the trace of all the activities that were performed, all the products involved, all the hypotheses that were made, all the versions of the products resulting of these hypotheses as well as all the decisions taken. Naturally, the result is a complex graph. This chapter examines more precisely what is in this graph, how it can be displayed and how it can be constructed. But first of all, the usefulness of the histories is presented through a few scenarios of use.

The goal of this chapter is to define precisely histories. They can be obtained by following a method defined in the MDL language, but they can also be the result of a well-organised methodology-free work. So this chapter will not refer to methodologies.

6.1. Usefulness of histories

A history can be reused in a great variety of ways, for different purposes. The main applications that can be performed on their basis will now be examined.

6.1.1. Documentation

The simplest use of a history surely is for documentation. It is always interesting to be able to remember what was done during the conception of a project. A history allows a database engineer to answer such questions as:

- What is the meaning of the *PVBFR* column?
- Why are there two tables, *product1* and *product2*?
- Why did we choose to use two fields to store telephone numbers, the *prefix* and the *number* fields, rather than a single field?
- Why does the *account* table include a *phone* field ?

Basically, the documentation will be processed in two ways:

- An analyst can simply look at it, statically.
- It can be replayed, like a movie, so that the analyst can see what happened in a more dynamic way.

In order to improve the usefulness of the history as a documentation, it can be cleaned. Cleaning a history means that all actions that do not participate directly in the development of the project are removed. This comprises processes performed according to some hypotheses that were rejected in later decisions, some simple tests (just to see what it would give), some actions followed by their inverse due to backtracking,... This cleaning can be useful in order to generate examples or tutorials to teach new analysts how to proceed.

Documentation is the most common use of histories in most projects [POTTS,88], [ZAMFIROIU,98], and most of the projects presented in Chapter 1.

6.1.2. Undo

A lot of computer applications possess an undo function. One way to implement it is to store the state of the product before each operation. This technique, due to the large memory consumption it requires, often limits the undo function to one or a few steps. Another way to implement it is to use a history of the performed actions. To undo the last operation, it suffices to perform one or several operations that do the reverse of the last operation in the history.

6.1.3. Database design recovery

The history of a reverse engineering job can be inverted in order to generate a possible forward engineering process that could have been followed at design time. Inverting a history means replacing each transformation with its reverse, that is one or several transformations that undo the original one, and to store these transformations in the new history in the opposite order. The new history can be reused for reengineering [HAINAUT 96b].

6.1.4. Database evolution

The history can be used to make the database project evolve. Traditionally, a database design is made up of three main phases: the conceptual analysis yields to a conceptual schema showing an abstract representation of the real world, the logical design transforms the conceptual schema into a semantically equivalent logical schema showing an implementation-suitable interpretation of the problem, and the physical design transforms the logical schema into a physical schema that is specifically oriented toward a given DBMS. Recording the history of the design insures the traceability of the constructs from the begin to the end of the design. Later on, when modifications are needed, the database engineer could be tempted, when dealing with small alterations, to work directly on the logical or even the physical level and going down. This results in breaking the traceability: the semantic equivalence between the physical and the conceptual schemas is lost, hence, the correctness of the conceptual schema as an interpretation of the database in the real world is lost too.

Using the history, the CASE environment can automatically (at least in a great part) update each schema when one of them is modified, as shown in [HAINAUT,94]. For instance, if the database engineer updates the logical schema, the CASE tool can replay the stored history on the new schema in order to propagate the modification to the physical schema. It can also propagate the modification backward to the conceptual schema by inverting the history.

6.1.5. History analysis

A history can be analysed in order to evaluate the underlying method and to improve it. Indeed, by analysing the history, for instance by finding places where the analyst had to make hypotheses, or places where the analyst had too much freedom and did things that should not have been done, or even places where the analyst was too much constrained and could not perform a task that should have been done, the method engineer can make the method evolve for a future project. By analysing histories coming from several analysts, it is also possible to understand how the method is interpreted by each one and to see where it could be refined in order to obtain a more uniform interpretation.

The history can also be analysed in order to evaluate the quality of the work. This can be useful to the project manager in order to distribute the work among the analysts according to their skills.

[ZAMIFIROIU,98] also supports this history analysis need by providing a history querying tools.

6.1.6. Method induction

A particular analysis of the history of a method-free project is the induction of an underlying method. Indeed, even if there was no explicit method to configure the CASE environment when a project was conducted, the engineer followed an implicit strategy. With an in-depth analysis of the history, it is possible to find some transformation patterns that give tips about the behaviour of the engineer. By assembling all the patterns we hope to discover the strategy implicitly followed by the engineer. This technique can be used as a method design technique, either on a learn-by-example basis, or as a way to conserve the traditional way of working while adopting new technology (it is better to adapt technology to human beings rather then human beings to technology).

[VANDERAALST,02] studies workflow mining, also a method induction by analyses of the workflow during software engineering projects.

6.2. Expectations for histories

Since histories are aimed at being reused, both by analysts and by the CASE environment itself, each history has to be:

- **Readable**: a human being should be able to read and understand it easily, even if he or she needs some training. This precludes binary coding, but textual keyword based coding or graphical coding is acceptable.
- **Formal**: every entry of the history must have a unique unambiguous interpretation.
- **Correct**: each entry of the history must represent a valid action in its context; the **context** of an action is the state of the product obtained by applying all the preceding actions in the history to the product in the state it was when the recording of the history began. For instance, an entry cannot show an action on an object that does not exist; the object had to exist in the original state of the product, or it had to be created by a previous entry of the history.
- **Complete**: All information that can be useful for reuse have to be stored. This definition is context-dependent because it depends on the intended reuse of the history as it will be shown later.

Such criteria as Readability, correctness and completeness, are widely recognised for a long time [POTTS,88],[LACAZE,02]. [ZAMFIROIU,98] goes further in the study by detecting breaks in the continuity.

6.3. Structure of histories

Histories are aimed at containing any kind of information used and produced during a project. So it is necessary to define a data structure that is able to keep all that information. This structure will now be defined, and every component of the history, namely products, primitive and engineering processes, hypotheses and decisions, will be examined.

6.3.1. Products

The first basic elements of a history are the products. At this level all kinds of products will be treated in the same way. For instance, schemas will not be distinguished from texts.

A product is identified by its name and its version ID. Since an analyst can generate different versions of a product when trying different hypotheses, the version ID has to be part of the product identifier. It must be noted that a product name has to be unique throughout the project, and not only in the scope of the current engineering process. Indeed:

- Histories can be handled and shown in different views, as presented in section 6.4, among which some global views show the flattened project structure.
- As it will be seen in section 6.3.6, the same product can be passed from process to process and so appears several times in a history; therefore it needs to be identifiable in any context without being renamed.

In order to document the work, an analyst will always have the possibility to add some descriptions or comments to products.

A product will evolve along its lifetime. It is generated by a process. Then it can be updated by several other processes. At some definite time, the product is finished. It has to be declared as such. From that moment, the product is locked. It cannot be modified anymore. Hence, each product must have a *locked-unlocked* state. Each product is created in the unlocked state, and it has to be put in the locked state manually by the analyst or automatically by a process (some automatic primitive processes) when its processing is finished.

From that moment, it cannot be set back in the unlocked state anymore (except to undo the locking while no other action is performed).

The symbol $\wp$ will denote the set of all possible products.

6.3.2. Processes

A history should contain all the processes that are performed during an engineering activity. The method being specified in a semi-procedural language, the resulting history is a tree of process calls. The root of the tree is the project which is performed by executing processes, each process performance being described by a branch. Each process is made up of sub-processes and so on. It is useful to know in what order the sub-processes have been performed, e.g., serially or in parallel, so each process will be stamped by its beginning date and time (mandatory) and end date and time (available only when the process ends). They will be identified by a name and the begin time stamp. In order to document his or her work, the analyst can add a description (some free text) to each process. This description can be used, among others, to store the hypotheses that have been stated to begin the process. In Chapter 2, two kinds of processes were defined: primitive processes at the operational level (these processes can be performed in a mechanical way, just by following a precise way-of-working) and engineering processes at the decisional level (some knowledge and decision taking are required in order to perform sub-processes). History structures for storing these processes will be defined in the two following sections.

6.3.3. Primitive processes

A primitive process is performed using only primitives, that is built-in functions of the CASE environment or external functions written in the built-in language of the CASE environment. During a method-driven project, a primitive process can be performed by an analyst when the method allows him or her to use a toolbox or by the CASE environment itself when the method uses built-in or external functions directly. During a method-free project, the analyst can use any tool of the CASE environment at any moment. The execution of primitives can be recorded in a **primitive process history**. The built-in functions of the CASE environment, which are product transformations, will be examined. Secondly, a way to formally represent their signature will be defined. In a third time external functions will be studied in the same way. Finally, log files will be used to record primitive process histories.

A. Transformations

In order to be able to keep a good trace of the built-in functions of the CASE environment, it is necessary to understand them. This section is dedicated to their formal analysis.

All the built-in functions of a CASE environment, which are basic product transformations, can be defined[9] formally, with their signature, their preconditions, and their postconditions.

A **transformation** Σ consists of two mappings T and t:

- T is the *structural mapping* that applies source construct C in product S (construct C in S is a collection of components of S) to construct C'. C' is the target of C through T, and is noted C' = T(C). In fact, C and C' are classes of constructs that can be defined by structural predicates. T is therefore defined by a *minimal precondition Pre* that any construct C must satisfy in order to be transformed by T, and a *maximal postcondition Post* that T(C) satisfies. T specifies the *syntax* of the transformation.

9 In [HAINAUT,96c], one can find more about database schema transformations, about their formal definition and their reversibility.

- t is the *instance mapping* that states how to produce the T(C) instance that corresponds to any instance of C. If c is an instance of C, then c' = t(c) is the corresponding instance of T(C). t specifies the *semantics* of the transformation. Its expression is through any algebraic, logic or procedural language.

According to the context, Σ can be noted either <T,t> or <Pre,Post,t>. In the following, Σ and T will be mixed up, and T will generally be used instead of Σ.

The construct C is part of product S in its initial state. The transformation replaces the construct C with construct C', to yield a new state of the product: S'. The effect of transformation can be clarified as follows. Let us consider the structural functions Δ_-, Δ_+ and Δ_0:

$\Delta_-(T) = S - S'$ returns the set of components of S that have disappeared.

$\Delta_+(T) = S' - S$ returns the set of new components that appear in state S'.

$\Delta_0(T)$ returns the set of components of S that are concerned by T, but that are preserved from S to S'.

We also have:

$C = \Delta_0(T) \cup \Delta_-(T)$

$C' = \Delta_0(T) \cup \Delta_+(T)$

$S' = (S - \Delta_-(T)) \cup \Delta_+(T)$

These concepts are illustrated in the scenario of Figure 6.1: the product S is a database schema in which an instance of the *rel-type into entity type* transformation is applied on rel-type R, and in which every object has been given a denotation.

The structural functions evaluate as follows:

S = {A,B,A1,B1,Q,qA,qB,R,rA,rB}

S' = {A,B,A1,B1,Q,qA,qB,R',RA,RB,rRA,rAR,rRB,rBR,id(R')}

C = {A,B,R,rA,rB}

C' = {A,B,R',RA,RB,rRA,rAR,rRB,rBR,id(R')}

$\Delta_-(T)$ = {R,rA,rB}

$\Delta_+(T)$ = {R',RA,RB,rRA,rAR,rRB,rBR,id(R')}

$\Delta_0(T)$ = {A,B}

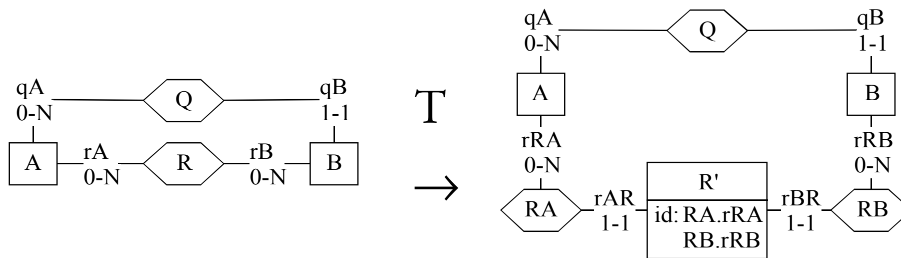


Figure 6.1 A basic transformation example

B. Transformation signature

In a primitive process history, a transformation will be specified through its **signature**, that states the name of the transformation, the name of the source product, the names of the concerned objects in the source product, the name of the target product (generally the

same as the source product) and the names of the new objects in the target product. For example, the following is the signature[10] of the schema transformation in Figure 6.1:

T : (S',R',{(A,RA),(B,RB)}) ← RT-to-ET(S,R)

It is interpreted as "*when applying RT-to-ET to rel-type R in schema S, the new entity type is called R' in the resulting schema state S', the rel-type involving A is called RA and the one involving B is called RB*".

A signature alone does not hold the Δ_-, Δ_+ and Δ_0 structural components, but it brings sufficient information to identify them in the source and target schemas. In addition, the format of a signature is not unique, it depends, among others, on the default naming conventions. For instance, the roles are given default names in transformation T described above.

In a CASE environment, every built-in transformation has such a signature. When it is used, it is instantiated. For example, transformation T above could be instantiated, in the actual schema shown in Figure 6.2, into

T:(S',WRITING,{(BOOK,written_by),(AUTHOR,writes)}) ← RT-to-ET(S,written)

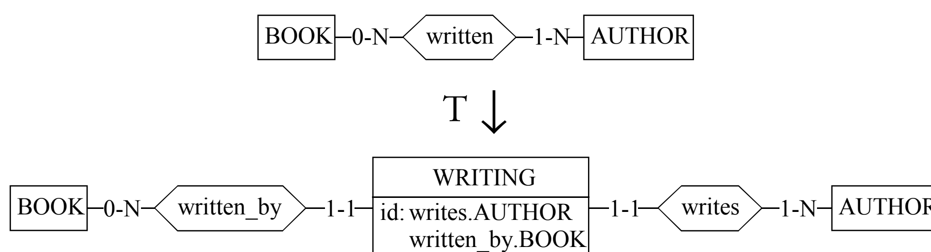


Figure 6.2 An instanciated transformation

An essential property of some signatures is their *reversibility*. Being provided with the right-side schema and the signature of T, the signature of a transformation T' which is the reverse of T, i.e. T'(T(C)) = C, can be defined:

T': (S',order) ← ET-to-RT(S,ORDER)

In other words, the signature provides enough information, not only for *redoing* the operation, but also to *undo* it. This property is less obvious for some non-reversible-transformations. Let us consider the example of the *del-ET* operator, which removes an entity type from a schema. It can be illustrated as shown in Figure 6.3.

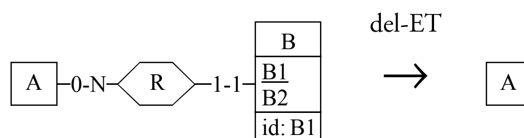


Figure 6.3The del-ET *transformation*

At first glance, it seems that the following signature could be quite right:

(S') ← del-ET(S,B)

Unfortunately, though the transformation can be performed again, it cannot be undone. The fact that entity type B was removed is kept, but information about its structure has been lost: what were its attributes, its roles, its constraints, etc ?

In this case, the signature must be augmented with those of the derived operations. In fact,

10 Fixed-length lists are enclosed into parentheses, while variable-length lists are enclosed into curly brackets.

removing B consists in removing its constraints (e.g. identifiers), then its attributes and its roles, then the inconsistent relationship types, and finally B itself. So the above signature has to be replaced by the following one:

$(S^1) \leftarrow$ del-ID(S,B,{B1},δ)
$(S^2) \leftarrow$ del-Att(S^1,B,B1,δ)
$(S^3) \leftarrow$ del-Att(S^2,B,B2,δ)
$(S^4) \leftarrow$ del-Role(S^3,R,B,δ)
$(S^5) \leftarrow$ del-Role(S^4,R,A,δ)
$(S^6) \leftarrow$ del-RT(S^5,R,δ)
$(S') \leftarrow$ del-ET(S^6,B,δ)

In these signatures, the symbol δ stands for any kind of additional information needed to create the object (value type and length, cardinality constraint, narrative description, etc).

Now, the complete signature of `del-ET` is reversible, though the operation itself is not.

C. External functions

External functions written in the built-in language of the CASE environment are simply complex functions that use the built-in functions described in the previous section. So the signature of such an external function is the concatenation of the signatures of the performed built-in functions.

D. Primitive process histories and log files

A primitive process history L is a list of instances of transformation signatures:

$$L = (P,\{S_1,\ldots,S_m\}),(T_1,T_2,\ldots,T_n))$$

where P is the performed primitive process, $S_1,\ldots,S_m$ are the used and modified products, and each $T_1,\ldots,T_n$ are the signatures of the transformations performed in P.

The symbol $\mathcal{L}$ will denote the set of all possible primitive process histories ($L \in \mathcal{L}$).

In practice, a log file, that is a text file listing sequentially the signatures of all the transformations performed, with a well defined syntax and the possibility to add comments and bookmarks, seems to be a good implementation. Indeed, the expectations for histories stated in section 6.2 are satisfied:

- The choice of the syntax (keywords, traditional notation conventions, structure, indentation,...) makes the history *readable*; the adjunction of comments can also improve this readability; the adjunction of bookmarks can help to mark some turning points or important steps.
- Since the history is made up of the formal signatures of the performed transformations stored sequentially in the exact order of performance, the history is *formal*.
- The fact that the history stores only the signatures of transformations that are actually performed correctly suffices for the history to be *correct*.
- The fact that *Pre* is minimum and that *Post* is maximum for each transformation, the fact that all transformation instances are stored, and the fact that they are ordered the same way they were performed make the history *complete for replay*.

A complete log syntax has been developed in the DB-MAIN CASE environment. Note that this syntax is not of mathematical nature as above; rather, it uses text based keywords. A DB-MAIN log file can be generated in either of two detail levels, *concise* or *extended*, according to the user needs:

```
*TRF rt_to_et                  Rel-type into entity type transformation
%BEG                           Beginning of the transformation signature
  %NAM "written"               The rel-type to transform is "written" from schema "LIBRARY/Conceptual"
  %OWN "LIBRARY"/"Conceptual"
  *CRE ENT                     Firstly, a new entity type, named "WRITING" is created
  %BEG
    %NAM "WRITING"
    %OWN "LIBRARY"/"Conceptual"
  %END
  *CRE REL                     Secondly, two new rel-types are created: "written by"...
  %BEG
    %NAM "written_by"
    %OWN "LIBRARY"/"Conceptual"
  %END
  *CRE REL                     ... and "writes"
  %BEG
    %NAM "writes"
    %OWN "LIBRARY"/"Conceptual"
  %END
  &CRE ROL                     A new role is created to link the new entity type with the first new rel-type
  %BEG
    %OWN "LIBRARY"/"Conceptual"."written_by"
    %ETR "LIBRARY"/"Conceptual"."WRITING"
    %CAR 1-1
  %END
  &MOD ROL                     The old role linking "BOOK" to "written" is moved to link
  %BEG                         "BOOK" to "written_by"
    *OLD ROL
    %BEG
      %OWN "LIBRARY"/"Conceptual"."written"
      %ETR "LIBRARY"/"Conceptual"."BOOK"
    %END
    %OWN "LIBRARY"/"Conceptual"."written_by"
    %ETR "LIBRARY"/"Conceptual"."BOOK"
  %END
  &CRE ROL                     A new role is created to link the new entity type with the second new rel-type
  %BEG
    %OWN "LIBRARY"/"Conceptual"."writes"
    %ETR "LIBRARY"/"Conceptual"."WRITING"
    %CAR 1-1
  %END
  &MOD ROL                     The old role linking "AUTHOR" to "written" is moved to link
  %BEG                         "AUTHOR" to "writes"
    *OLD ROL
    %BEG
      %OWN "LIBRARY"/"Conceptual"."written"
      %ETR "LIBRARY"/"Conceptual"."AUTHOR"
    %END
    %OWN "LIBRARY"/"Conceptual"."writes"
    %ETR "LIBRARY"/"Conceptual"."AUTHOR"
  %END
  &CRE GRP                     A new group is added to the new entity type to define its primary
  %BEG                         identifier, made of roles "writes.AUTHOR" and "written_by.BOOK".
    %NAM "IDWRITING"
    %OWN "LIBRARY"/"Conceptual"."WRITING"
    %COM "LIBRARY"/"Conceptual"."writes"."AUTHOR"
    %COM "LIBRARY"/"Conceptual"."written_by"."BOOK"
    %FLA "P"
  %END
  &DEL REL                     Finally, the old rel-type is deleted
  %BEG
    %NAM "written"
    %OWN "LIBRARY"/"Conceptual"
  %END
%END                           End of the transformation signature
```

Figure 6.4 A log fragment of a primitive process: transformation of the written *rel-type into an entity type. Text in italics are added comments which are not part of the log file.*

- the **concise log file** is the strictly minimum log file which contains the minimum signature instances as defined above;
- the **extended log file** is the concise one completed with all the information that can be needed for all the purposes of the CASE environment.

The complete syntax depends on the needs of the supporting CASE environment, so it will not be detailed precisely here. A short example of log file produced in the DB-MAIN CASE environment is presented in Figure 6.4. It shows how the rel-type *written* in schema *LIBRARY/Conceptual* is transformed into an entity type as shown graphically in Figure 6.2.

6.3.4. Engineering processes

An engineering process follows a strategy, either given by the current method or in the analyst's mind. As the analyst can make hypotheses, try various solutions and decide to abandon some of them, it is no longer possible to record actions in a linear way like in the primitive process history. The history of an engineering process has to be a graph G=(P,V, E) where P is the engineering process, V is a set of nodes, and E a set of edges. The symbol $\mathcal{G}$ will denote the set of all engineering process graphs ($G \in \mathcal{G}$). The nodes of the graph G are products, primitive process histories, engineering sub-process graphs and decisions: $V \subseteq \mathcal{P} \cup \mathcal{L} \cup \mathcal{G} \cup \mathcal{D}$ ($\mathcal{D}$ will be defined later as a set of decisions). The edges, possibly oriented, show the use of products in processes:

- an edge directed from a product to a process history shows that the product is used by the process as an input
- an edge directed from a process history to a product shows that the process generates the product and returns it in output
- a non-oriented edge between a process history and a product shows that the process modifies the product.

A node will be represented by the name of the process or the product it concerns. An edge will be represented by a pair of nodes (i,j) where i is the origin of the edge, and j is the target. In a non-oriented edge, the order of the elements does not matter (i,j) = (j,i). To distinguish non-oriented edges, they will be underlined: (i,j) is oriented, $(\underline{i,j})$ is non-oriented, $(\underline{i,j})=(\underline{j,i})$.

For example, Figure 6.5 shows a graph in which a process (A) generates two products (R, S), each of them being used by another process (B,C) that generates a new product (B generates T, C generates U), and these latter products being used by a fourth process (D) that generates a last product (V). So,

V = {A,B,C,D,R,S,T,U,V}
E = {(A,R),(R,B),(B,T),(T,D),(A,S),(S,C),(C,U),(U,D),(D,V)}

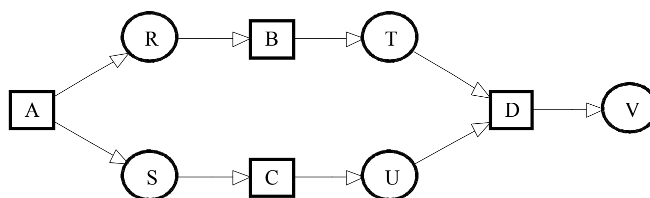


Figure 6.5 An example of engineering process graph

An engineering process graph is submitted to a few restrictions:

- It is finite because it includes a finite number of products and performs a finite number of transformations on them.

- There will never be parallel edges, i.e. two edges between the two same nodes.
- There will never be any self-loops since edges only go from product nodes to process nodes and from process nodes to product nodes.
- Since a product cannot be generated after being used, cycles cannot arise without non-oriented edges.

The engineering process histories satisfy the expectations stated in section 6.2:

- Graphs are *readable*; the adjunction of comments can also improve this readability.
- Graphs are *formal*.
- Graph theory is proved to be *correct*.
- Since every possible component of a history (products, primitive processes, engineering processes, decisions) appears in a graph and since all their possible links are represented by edges, the graphs are *complete*, and the engineering process histories too.

6.3.5. Decisions

The third basic elements of a history are the decisions. A decision is a special kind of process that does not alter nor generate products. It only adds a node to the graph and edges directed from the products in the scope of the decision to the decision itself, and edges directed from the decision to selected products, if any. There are two kinds of decisions:

- Decisions that must be taken according to the method followed. For instance, when the condition of an *if* or a *while* statement needs a response from the analyst. For instance:

  ```
  if ask("Do you want to optimise the relational schema ?")...)
  ```

  These decisions need a *yes/no* answer. They are only the target of oriented edges, no edge is directed from them to selected products. These edges show which products were consulted to take the decision. The description of the decision will contain the choice and a possible added comment. Figure 6.6 shows a simple graph with a decision.

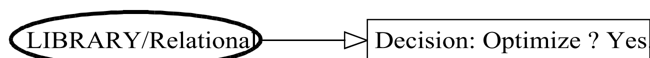


Figure 6.6 A yes/no *decision*

- Decisions that follow hypotheses, reflecting the choice of one or several product versions among all those obtained by performing the same process several times with different hypotheses in mind. The description of a decision process contains the rationales that lead to the analyst's choice. This second kind of decision is not linked to the method followed, and it can be made at any moment. Figure 6.7 shows such a decision.

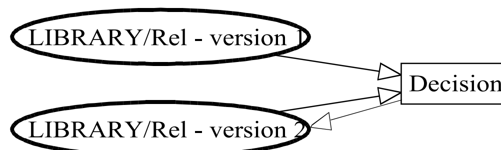


Figure 6.7 A best product version decision

The symbol $\mathscr{D}$ will denote the set of all possible decisions. A decision $D \in \mathscr{D}$ is either:

- a pair (S,b) where S is the set of products consulted for taking the decision and b is the boolean result

- a pair (S,C) where S is the scope of the decision and C is the set of selected products, $C \subseteq S$.

The expectations from section 6.2 are fulfilled with respect to decisions:

- A decision is *readable* because it is part of a readable graph and its description is readable free text.
- Since a decision of the first kind has a boolean answer, and since decisions of the two kinds are included in a graph with meaningful oriented edges, a decision is *formal*.
- While edges are correctly oriented and while its description correctly stores the *yes/no* choices, a decision is *correct*.
- Since the semantics of a decision is in its edges and *yes/no* choice, a decision is *complete*.

6.3.6. The history of a project

A project is an engineering process. It is generally decomposed into several phases which are themselves engineering processes. Each phase is, at its turn, decomposed into several steps, which are engineering processes too or simply primitive processes. The engineering processes can be further decomposed... It appears that the process histories can be organised in a tree structure: non-leaf nodes are engineering processes histories, and leaf nodes are either primitive process histories or unfinished engineering process histories; oriented edges show that the target node is used by the origin node.

Since a history tree (H) is a special kind of graph, it can be represented by H=(V,E) where V is a set of nodes and E is a set of edges between nodes of V. The symbol $\mathcal{H}$ will denote the set of all possible histories.

A history tree can be generated from the process histories. In a given database engineering project, let $L_i \in \mathcal{L}$, $1 \le i \le n_L$, be the primitive process histories of a project, and $G_i \in \mathcal{G}$, $1 \le i \le n_G$, the engineering process histories of the same project. The tree of the history of the project can be defined by defining V and E as:

$$V_L = \{L_i | 1 \le i \le n_L\} \subseteq \mathcal{L}$$
$$V_G = \{G_i | 1 \le i \le n_G\} \subseteq \mathcal{G}$$
$$V = V_L \cup V_G$$
$$E = \{(v_i,v_j) | \ v_i=(P_i,V_i,E_i) \in V_G \wedge v_j \in V \wedge v_j \in V_i\}$$

The fact that a tree has no cycle (even if the orientation of the edges is removed), the fact that every node of a tree is connected to all others by a path, and the fact that our graphs are finite allow us to use a more appropriate notation. Let us classify all the nodes in levels:

- The root node, origin of one or several edges and target of none, is at level 1.
- All the nodes that are target of edges originating from level n ($n \ge 1$) are at level n+1.

A tree H=(V,E) can now be represented as a n-uple of couples made up of the nodes of V and their level:

$$H = ((v_{i_1},l_{i_1}),(v_{i_2},l_{i_2}),\ldots,(v_{i_n},l_{i_n}))$$

where $(i_1,i_2,\ldots,i_n)$ is a permutation of $(1,2,\ldots,n)$ and l_{i_j} is the level of node v_{i_j}, $1 \le j \le n$.

The pairs of the n-uple H are ordered in a depth-first way on their node: the node of the first pair is the root of the tree, and each pair is immediately followed by the pairs the nodes of which are all the descendants of its node, then by the pairs the nodes of which are its brothers with their descendants. All the son nodes of a same father appear in the chronological order of their creation time.

Since the project history, like an engineering process history, is a graph, the verification that a project history satisfies the expectations stated in section 6.2 is straightforward.

Figure 6.8 shows the history of a library database design project. The main phase is the root of the tree. That graph shows that the project was conducted as a sequence of processes: *conceptual analysis*, *logical design*, *physical design* and *coding*. These are four engineering processes the graphs of which are the nodes of the second level of the tree. The *logical design* process itself was refined with one engineering sub-process: the *relational design*. All other sub-processes that can be found in all the graphs of the tree are primitive processes (*new schema*, *analysis*, *conceptual normalisation*, *schema copy*, *name conversion*, *ISA relations*, *non-functional rel-types*, *attributes*, *identifiers*, *references*, *setting indexes*, *storage allocation*, *setting coding parameters*, *generate SQL*); their histories are the leaves of the tree (they should have been drawn in this figure too, but it would take a lot of place). The tree will be noted:

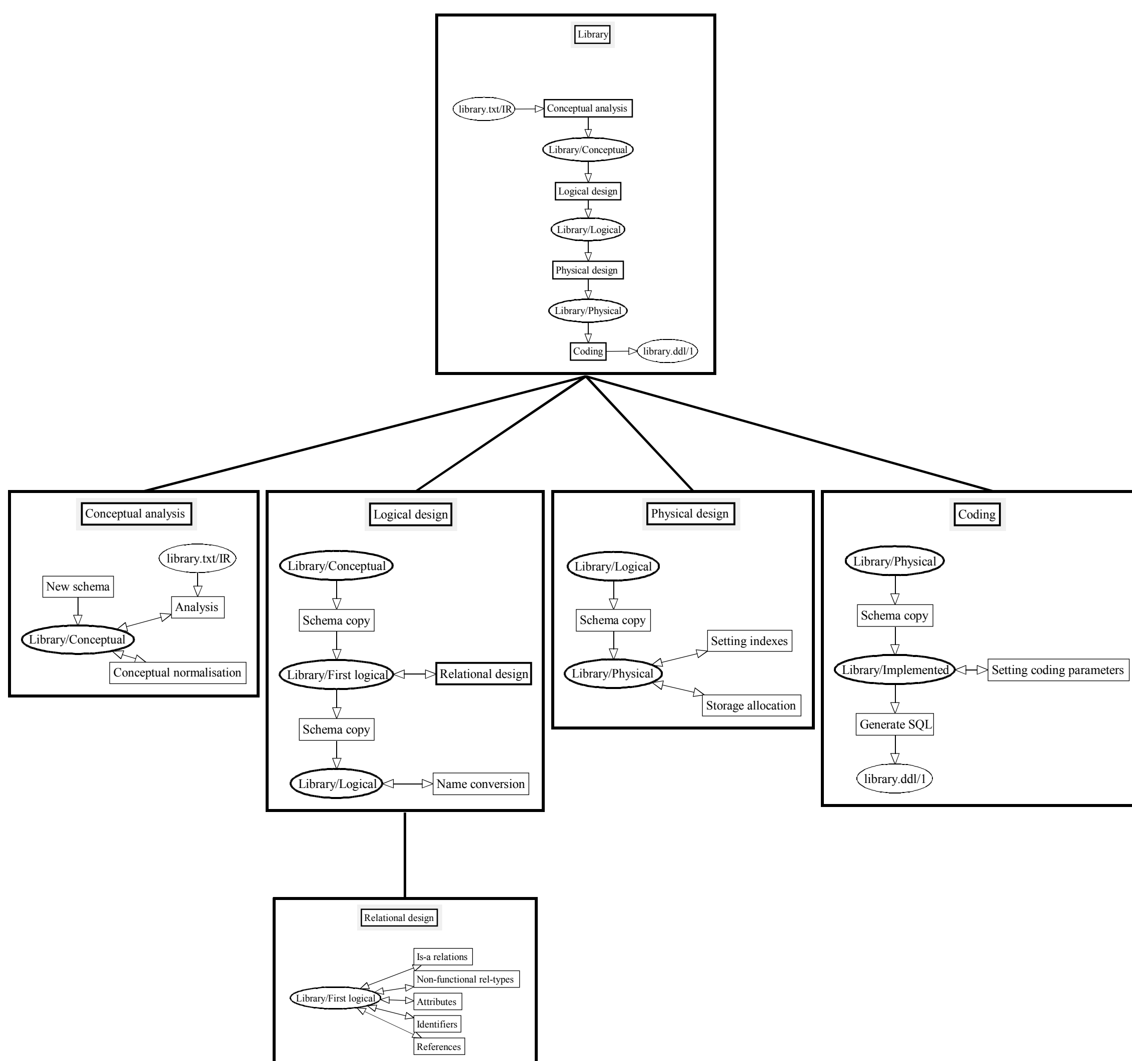


Figure 6.8 An engineering process tree

Note that non-oriented edges are represented with two arrows rather than with none because it makes the input/output role of these edges appear better. This representation will be adopted more formally later.

H = ((LIBRARY,1), (Conceptual Analysis,2), (New schema,3), (Analysis,3), (Conceptual normalisation,3), (Logical Design,2), (Schema copy,3), (Relational Design,3), (Is-a relations,4), (Non functional rel-types,4), (Attributes,4), (Identifiers, 4), (References,4), (Schema copy,3), (Name Conversion,3), (Physical Design,2), (Schema Copy,3), (Setting indexes,3), (Storage allocation,3), (Coding,2), (Schema copy,3), (Setting coding parameters,3), (Generate SQL,3)).

Commonly, in software process or business process modelling tools, engineering process histories are recorded, but they often use third party tools (editors, text processors, compilers, debuggers,...) for primitive processes which have their own logging facilities, and all the logs are generally independent one from the others. In this thesis we integrate them all together to reuse them as a whole.

6.4. History representation

A project history is a tree of process histories. This tree is a kind of table of contents of the project, the process histories being the real material. Different ways of representing these various histories will be examined in different situations, with different purposes in mind.

6.4.1. Representation of the tree structure

The simplest way of showing a table of contents is in a textual fashion, like in a book: the name of every process at the top most level are listed in the order the processes were performed. Under each of them, the list of sub-processes they are made up of, also sorted in performance order, and so on. To each process name in the list, a reference (an hyperlink) to the process history is added. Drawing this table of content from the tree notation is straightforward, each node in the n-uple being a line in the table of content. Figure 6.9 shows the table of content associated to the tree shown in Figure 6.8.

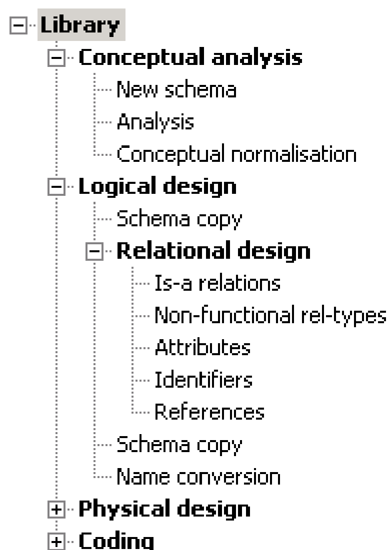


Figure 6.9 An example of history tree

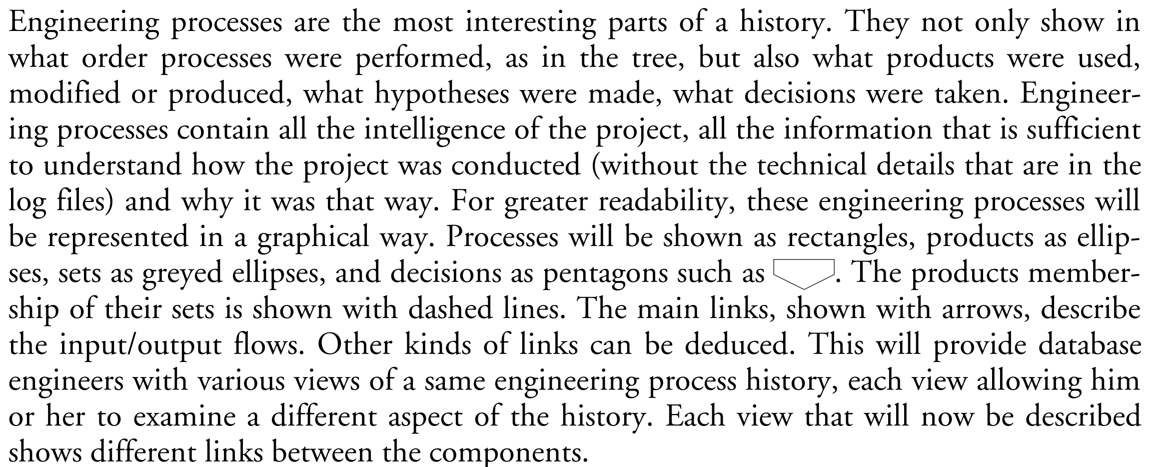
Bold characters show engineering processes and regular characters show primitive processes. A "-" character before an engineering process indicates that its content is shown, a "+" character indicates the content is hidden.

6.4.2. Representation of primitive process histories

Since a primitive process history is stored as a log file in a readable text file, it can be displayed and examined with a plain text editor. But such a file can be very long and tedious to read. Since those log files are also built with a formal syntax in order to be easily reused

by tools, the supporting CASE environment has to include such tools, and other tools can be written by analysts for their own special purposes too. For example, the person who browses through a history is not always interested by the very small details of the process execution. The log file syntax can include bookmarks and comments. So, if the analyst takes care at bookmarking and commenting correctly his or her job, a tool that only shows these bookmarks and comments can be of great help.

6.4.3. Representation of engineering process graphs

Engineering processes are the most interesting parts of a history. They not only show in what order processes were performed, as in the tree, but also what products were used, modified or produced, what hypotheses were made, what decisions were taken. Engineering processes contain all the intelligence of the project, all the information that is sufficient to understand how the project was conducted (without the technical details that are in the log files) and why it was that way. For greater readability, these engineering processes will be represented in a graphical way. Processes will be shown as rectangles, products as ellipses, sets as greyed ellipses, and decisions as pentagons such as . The products membership of their sets is shown with dashed lines. The main links, shown with arrows, describe the input/output flows. Other kinds of links can be deduced. This will provide database engineers with various views of a same engineering process history, each view allowing him or her to examine a different aspect of the history. Each view that will now be described shows different links between the components.

A. Basic view

The basic view of an engineering process graph G=(V,E) is the exact representation of its content: all its components (the nodes $v_i \in V$) with all the stored links between these components (the edges $e_j \in E$):

- an input link is shown with an arrow directed from a product to a process
- an output link is shown with an arrow directed from a process to a product
- an update link is shown with a double-headed arrow between a process and a product
- a product in the scope of a decision is linked to it with a single-headed arrow
- a product selected in a decision is linked to the decision with a double-headed arrow.

Figure 6.8 shows six examples of engineering process basic views. Figure 6.10 shows a much more complex example extracted from the second case study in chapter 11. It describes a reverse engineering process. *ORDER/Extracted* is a database schema generated by an SQL DDL extractor. This schema is member of the *cobsch* set of COBOL schemas. The file *Order.cob* is an application program that uses the data structures of the schema. The engineering process is aimed at enriching the COBOL schemas with information found in the COBOL programs. During the process, the *N-N refinement* sub-process was performed twice with different hypotheses, resulting in two versions of the schema: *ORDER/draft-2* and *ORDER/draft-3*. Later on, after the performance of the *Field-ID* sub-process, the decision to keep the second version was made, and the engineering process continued with it. *ORDER/completed* is the resulting product of the engineering process.

B. Dependency view

The basic view is ideal for working, but, for some reasons (specially the readability of bigger projects), simplified views could be prefered. A dependency view showing only products and their dependencies is one of them. Figure 6.11 shows the dependency view of the *Conceptual analysis* engineering process of Figure 6.8. The arrow show the dependencies

between products: *Library/Conceptual* is made on the basis of *library.txt/IR*. Figure 6.12 shows the dependency view of the more complex engineering process of Figure 6.10.

More formally, the dependency view of an engineering process is defined as follows.

Let p be an engineering process, and G=(p,V,E) be the graph of its history where:

> $V = V_{pd} \cup V_{pc} \cup V_d$,
> $V_{pd} = \{v_1,v_2,\ldots,v_{n_1}\} \subseteq \mathcal{P}$, $0 \le n_1$, be the set of products involved in p.
> $V_{pc} = \{v_{n_1+1},v_{n_1+2},\ldots,v_{n_2}\} \subseteq \mathcal{L} \cup \mathcal{G}$, $n_1 \le n_2$, be the set of sub-processes of p.
> $V_d = \{v_{n_2+1},v_{n_2+2},\ldots,v_{n_3}\} \subseteq \mathcal{D}$, $n_2 \le n_3$, be the set of decisions taken in p.
> E is the set of edges in the graph G.

The dependency view of p is the graph G' = (p,V_{pd},E') where E' is calculated from E by applying the following rules:

- if $\exists\ v_i, v_j \in V_{pd}$ and $\exists\ v_k \in V_{pc}$ such that $(v_i,v_k) \in E$ and $(v_k,v_j) \in E$, then $(v_i,v_j) \in E'$
- if $\exists\ v_i, v_j \in V_{pd}$ and $\exists\ v_k \in V_{pc}$ such that $(v_i,v_k) \in E$ and either $(\underline{v_k,v_j}) \in E$ or $(\underline{v_j,v_k}) \in E$, then $(v_i,v_j) \in E'$
- if $\exists\ v_i, v_j \in V_{pd}$ and $\exists\ v_k \in V_{pc}$ such that either $(\underline{v_i,v_k}) \in E$ or $(\underline{v_k,v_i}) \in E$, and $(v_k,v_j) \in E$, then $(v_i,v_j) \in E'$

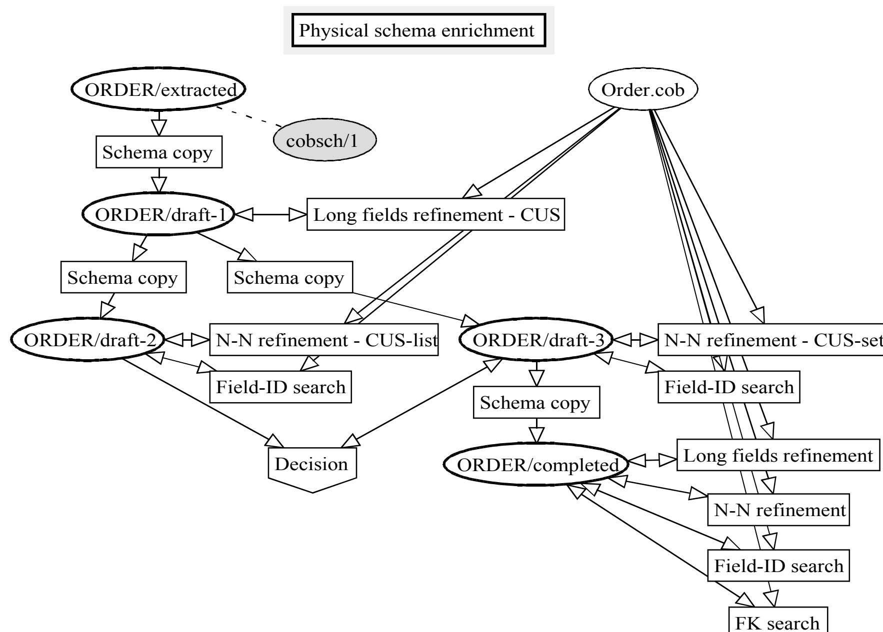


Figure 6.10 An example of engineering process basic view

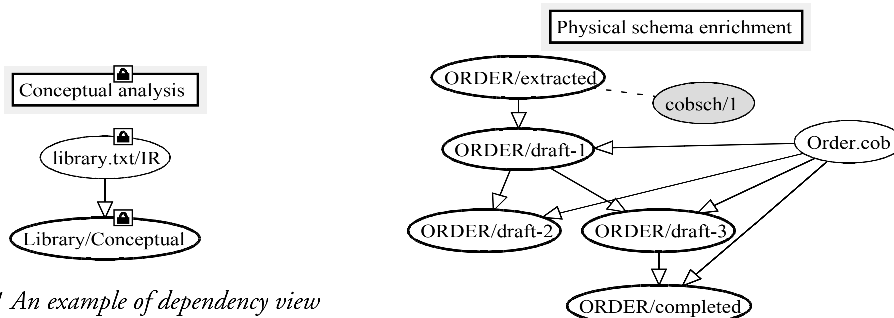


Figure 6.11 An example of dependency view

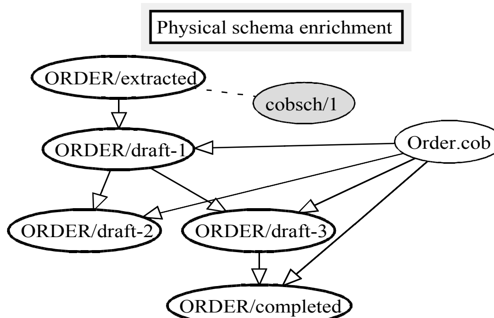


Figure 6.12 A second example of dependency view

- if $\exists$ v_i, $v_j \in V_{pd}$ and $\exists$ $v_k \in V_{pc}$ such that either $(\underline{v_i,v_k}) \in E$ or $(\underline{v_k,v_i}) \in E$, and either $(\underline{v_k,v_j}) \in E$ or $(\underline{v_j,v_k}) \in E$, then either $(\underline{v_i,v_j}) \in E'$ or $(\underline{v_j,v_i}) \in E'$ exclusively
- if $\exists$ v_i, $v_j \in V_{pd}$ such that $(v_i,v_j) \in E'$, and either $(\underline{v_i,v_j}) \in E'$ or $(\underline{v_j,v_i}) \in E'$, then $(\underline{v_i,v_j}) \notin E'$.

Note that decisions and edges connected to them do not give rise to dependency edges because decisions do not alter the products.

In more natural terms, a single-headed arrow between product *R* and product *S* means that *S* directly depends on *R*, i.e. there exists at least one primitive or engineering process for which either *R* is an input and *S* is an output, or *R* is an input and *S* is an update, or *R* is an update and *S* is an output. A double-headed arrow between two products means that they directly depend on each other, i.e. both products are updated by at least one process. In a special case where both kinds of arrows should be drawn, the double headed-arrow prevails. For instance if product *R* is updated by two processes *A* and *B*, and product *S* is an output of *A* and is updated by *B*, a single-headed arrow should be drawn because of *A* and a double-headed arrow should be drawn because of *B*. Only the second one will be drawn.

C. Global dependency view

Figure 6.11 shows a derived view of the current process. The whole tree of graphs can also be summarised in a single product dependency graph. For instance, Figure 6.13 shows the dependencies between all the products in the *LIBRARY* forward engineering project shown in Figure 6.8. To obtain this graph, the tree is flattened. The complete flattening process will be detailed in chapter 7. In summary, in the graph of the root process of the tree, all the engineering process nodes are replaced by their graphs, and so on recursively until there are no more engineering processes in the graph. The result is the graph of a pseudo engineering process doing the same job as the whole project. Calculating the dependency view of the resulting graph will give the result.

In formal and short terms, the global dependency view of a project is the dependency view of the flattened history as defined in chapter 7.

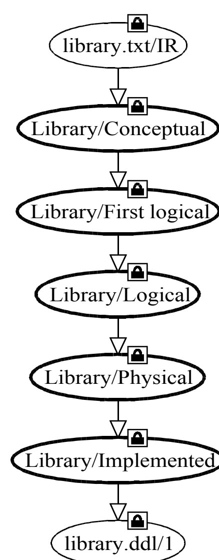


Figure 6.13 An example of summarised derived view

6.5. History construction

This section describes the way the supporting CASE environment builds a history. When a new project begins, a method must be chosen and followed. The CASE environment has to build the history automatically, documenting the job in a precise way, according to the actions of the engineer; these actions generally occur according to the method. This section examines how every action proposed by a method or self-decided by the engineer is recorded.

6.5.1. Primitive processes

When a primitive process is performed, two kinds of information need to be stored: the fact that the primitive process is performed and how it is performed.

Since a primitive process is always performed as a part of an engineering process, its execution is recorded by adding a node in the graph of the engineering process. Each time a primitive process is performed, a new node is created in this graph, and appears in the global tree of the history. This node is labelled with a name. It also has to be annotated with a reference to its primitive process type. Furthermore, the edges connecting the primitive process with the products it uses, modifies, or generates are created.

The recording of the way the primitive process was performed depends on the objective of the history (see section 6.3) and on the type of the process type (see Chapter 2). Let us now examine what to record in the history for primitive processes according to these two dimensions.

A. Concise history for replay only

1. A *basic automatic process type* is always performed in the same way, so a simple reference to it suffices to replay it.
2. A *configurable automatic process type* is stored in the method with its configuration parameters that do not evolve. So a reference to this definition in the method suffices to replay it.
3. A *user configurable automatic process type* needs to be configured at each execution. A simple log file with a reference to the process type and all the parameter values decided at runtime, and a reference to this log file suffices to replay the process.
4. A *manual process type* must be entirely performed by an analyst. So the primitive processes must be reflected by a history containing all the actions performed by the analyst, as depicted above. The node representing the primitive process in the graph has to have a reference to the log file of this history.

B. Extended history for more complex tasks

On the contrary, to build a history aimed at being reused for more complex tasks, such as reverse engineering, it can be useful to record every single action in a log file. Indeed, primitive processes of any kind that modifies a product will do it in several little steps which can all be recorded. It can also be useful to record the parts of the products that will be transformed just before the transformation, in particular before non semantic preserving transformations (see section 6.3).

6.5.2. Engineering processes

Like a primitive process, an engineering process needs to record two kinds of information, the fact it is performed and the way it is performed.

When the project begins, the history is created. A new engineering process is created with a blank graph that will grow all along the life cycle of the project, and the main tree of the history is initialised with that engineering process as the root and only node.

During the project, when an instance of an engineering process is performed, a new node has to be added in the graph of the current engineering process. In the same time, a new blank graph is created that will grow in size during the performance of the new engineering process. In the graph of the current engineering process, edges are created between the new node and all the products it uses or modifies.

When an engineering process ends, the product it generates must be added to the graph of the calling engineering process as well as an edge between the new products and the terminated sub-process.

For instance, in Figure 6.8, it is possible to continue the project with a new report generation phase: in the *LIBRARY* process, a new engineering process labelled *Report generation* can be added, to which the *library.txt* and the *library.dll* products should be linked in input. In the tree, an empty graph would appear, also labelled *Report generation.* When the process is over, the graph contains a node for the generated report. Another node representing this report is also added to the father process graph as an output of *Report generation*.

In the strategy of an engineering process, sub-process types appear within control structures. A control structure is a programming concept which has no equivalence at the instance level. In the history, only the effects of the control structures are stored, possibly with the decisions that have to be taken:

- The execution of a sequence of process types is translated into a sequence of processes in the history. An example is shown in Figure 4.6.
- When an *if* structure is encountered in the strategy, its condition has to be evaluated. The result of this evaluation, a decision, is stored in the history: a node is appended to the graph with edges linking the products on which the decision is based to the new node. Then one branch of the *if* structure is followed, and its trace is recorded in the history. Since the other branch has been ignored, it leaves no trace in the history. Figure 4.7 shows that only a process *a* of type *A* was performed.
- A *repeat*, a *while* or an *until* structure will lead to the fact that some sub-process types (possibly one or several organised with another control structure) will have to be performed several times. It will result in the appearance of several processes of the same type in the history. If a condition (*while* and *until* structures) has to be evaluated at each iteration of the loop, each decision will be stored in the history too. Figure 4.8 shows that two processes *a1 and a2* of type *A* are performed.
- The *one*, *some* and *each* structures, like the *if* structure, will also make some branches only to be performed, and only these branches will leave a trace in the history. If the engineer wants to store the rationales that conducted him or her to choose those branches, he or she can add, voluntarily, a decision to the history. In Figure 4.11, only the branch of process type *B* is performed, which makes process *b* appear in the history, the branch of *A* being ignored.
- A *for* structure works like a *repeat* structure with the difference that the user has to choose a new product in a given set at each iteration. This choice will be stored in the history through the edges which link the processes of each iteration.

6.5.3. Hypotheses, versions and decisions

Sometimes engineers face particular problems they cannot solve in a straightforward way:

- Time consuming tasks for which the engineer considers several ways of working but does not know which one will take the less efforts. It can be useful to start and perform a bit of the work in each way, to make an estimation of the effort, and to pursue in the best way.
- A complex problem for which several solutions are possible but one should be better than the others. It cannot be guessed a priori. In that case, it is necessary to develop these solutions and to compare them afterwards.
- A complex problem that has only one solution. Several ways of starting the reasoning exist but only one of them leads to the result. This is like in a labyrinth. The engineer has to try several ways until he or she finds the good one.
- A problem for which the requirements are not clear. The engineer sketches several ideas of solution as a basis for discussion with other people.

In such cases, the solution-finding pattern is always the same: trying different solutions, then choosing the best one. So, different processes of the same type are performed on the basis of the same products, but with different ideas and different hypotheses. The result of all these processes are various product versions. Then the engineer has to take a decision: choosing the best versions of the products.

If one or several versions of the products are given up, it is important to keep them in the history with the hypotheses and the reasoning that lead to them. Indeed, it may be useful, later on, to know why the final solution was chosen and, maybe more important, why the other solutions where rejected.

This situation is shown in the history by as many nodes as processes performed, each one annotated with its hypothesis, and one more for the decision, annotated with the (or the list of) chosen product and the rationales of the choice. Oriented edges are created from the different versions of the product to the decision for showing which versions have been taken into account in the decision process. The chosen versions are marked as such.

For instance, the short history sample in Figure 6.14 shows that a process was performed twice with different hypotheses *Hyp 1* and *Hyp 2*. Each execution (*Proc/Hyp1* and *Proc/Hyp2*) returned a different version of a product: *Output/1* and *Output/2*. Then both versions have been evaluated and the second one has been kept: the arrow going from *Output/1* to *Decision* shows that *Output/1* entered in the decision process but was not kept, the double headed arrow between *Output/2* and *Decision* shows that *Output/2* also entered in the decision process and was selected. Then the work went on with *Next process* using only the chosen version of the product.

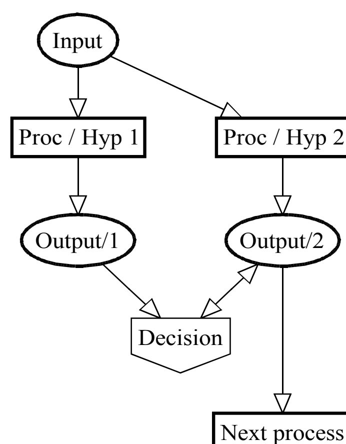


Figure 6.14 A decision example

It must be noted that making various hypotheses and performing several instances of the same process type can only be made without updating products. Indeed, an updated product is updated once and for all. In Figure 6.15, Proc 1 updates the product, then Proc 2 updates it again; it is clear that the result can only be a single schema that includes all the modifications, and that there cannot exist two different versions of the resulting schema. To solve this problem, it is necessary to copy the product to update and to perform each instance of the process type with the different copies. When the decision is taken, the history of the modifications of the selected copy should be replayed on the original product in order for it to be correctly updated.

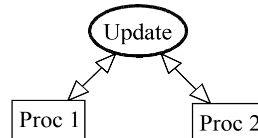


Figure 6.15 Two processes update the same product. These processes cannot be two instances of the same process type.

Chapter 7

History processing

In this chapter, histories will be used and transformed in order to fulfil the goals listed in the beginning of Chapter 6. Firstly, tools will be developed for replaying histories. Secondly, this chapter will show how histories can evolve while the database the design of which it represents evolves. Then a series of basic transformations that can be applied to histories will be examined. Finally, These basic transformations will be applied to clean and to revert histories.

7.1. Basic hypotheses

In this chapter, histories previously recorded will be reused. In order to simplify the reasoning, everything not directly concerned by this topic is supposed to be unchanged. That is to say the CASE environment, the method, and the external tools are unchanged since the history was recorded. At the end of this thesis, tracks towards a release of these limitations will be opened without bypassing the scope of this thesis.

7.2. History replay

To replay a history, in short, consists in doing again all the actions that were performed when it was recorded. A history can be replayed for several reasons. One of them is for documentation. This is a simple task. It suffices to take a copy of all the products that where used during the construction of the history, in the state they were when the recording of the history began, then to look at the history and redo every action, exactly in the same way, in the same order, and with the same parameters. This will be described in more details below.

A history can also be replayed in order to do a job twice, possibly with different (slightly or strongly) products. This is a more complicated job as shown below too.

7.2.1. Replaying primitive processes of automatic basic type

Replaying primitive processes of an automatic basic type is a very simple task since it suffices to apply the selected tool one more time. Processes of this type automatically work with all the constructs of a product within a defined range rather than on specifically specified constructs. This range is defined within the tools themselves. So, the fact that the products that are passed to the tools are the same or not than when the history was built does not matter since the list of constructs within the range will be automatically re-evaluated.

For example, if the history contains a COBOL data structure extraction process applied to all the texts of type COBOL_programs, replaying the same history will redo the data extraction to all the current texts of type COBOL_programs, no matter whether these texts are the same as during the recording of the history (replay for documentation) or if they are new ones (replay for doing a job twice).

7.2.2. Replaying primitive processes of automatic configurable type

The configuration of automatic configurable primitive process types being made at method-definition time, and being immutable, replaying a process of that type is similar to replaying a process of an automatic basic type.

7.2.3. Replaying primitive processes of automatic user configurable type

To replay a process of an automatic user configurable primitive type, it is necessary to know how the process type was configured when the process was recorded in the history. This configuration was stored in a log file. So it suffices to extract the parameters from this log file to configure the process type before replaying the process.

Here again, since the process is performed automatically, the fact that the input and update products are the same or not does not matter.

7.2.4. Replaying primitive processes of manual type

Replaying a process of a manual type can be very different if the input and update products are the same as when the history was built or not.

A. Same products

When the history was built, the process was performed by an analyst who had a toolbox at his or her disposal and who had to decide what tool to use, on what part of the product and in what order. All these actions where recorded sequentially in the history. It suffices to read it, entry by entry, and to treat them in the same order. Each entry identifies the constructs of the product that are concerned and the tools to use. replaying consists in identifying these constructs and in applying the tools on them.

It must be noted that when the process was first performed, only a human being could decide what action to perform and on what part of the product but, when replaying, since it suffices to read a log file and do the same again, there is no more decision to take; this can be done automatically by the CASE environment.

B. Different products

If the product on which the log file was recorded and the product on which we want to replay this log file are different, the process is much more complex. Let us suppose that these products are very similar with just one little difference. Let us examine different situations:

- Actions that were performed on parts of the product that are not concerned with this difference can still be redone in the same way.
- If an action involves a part of the product that has disappeared, the action cannot be performed anymore, so it is discarded.
- If an action involves a part of the product that is just slightly modified, the difference may either have no real impact on the transformation, or impeach the transformation. For instance, in a database schema, the transformation of a compound attribute to which a sub-attribute has been appended into an entity type will automatically transform this sub-attribute into one more attribute of the new entity type, without bothering about it (Figure 7.1). But the transformation into a foreign key of a functional rel-type to which an attribute has been appended is no longer possible (Figure 7.2).

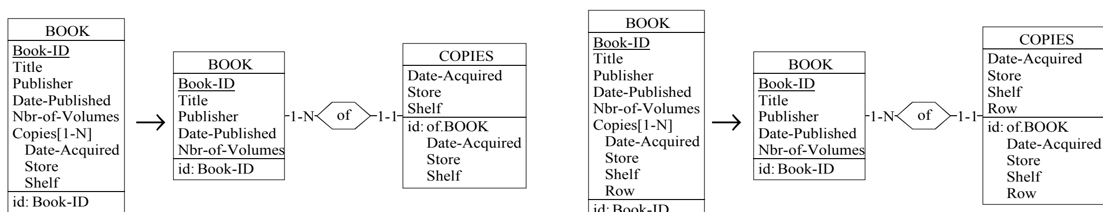


Figure 7.1 An attribute-into-entity-type transformation stored in the history (left) and replayed after a small modification (right, the "row" sub-attribute has been added)

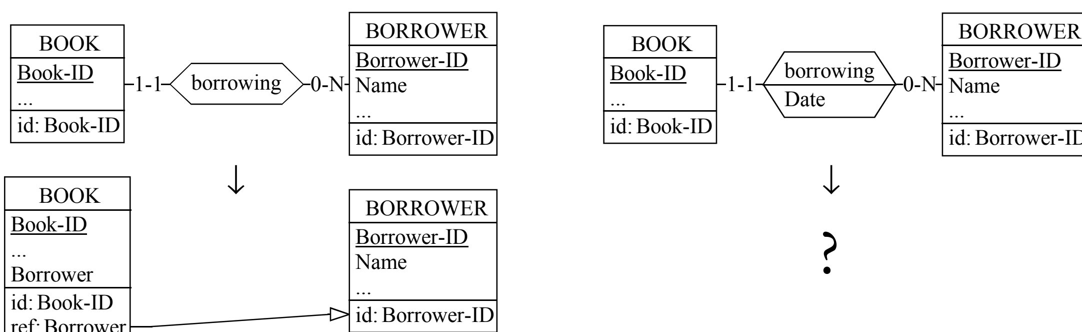


Figure 7.2 A rel-type-into-foreign-key transformation (left) cannot be replayed after an attribute has been added to the rel-type (right)

It is to be noted in this last example that since the rel-type cannot be transformed, the new foreign key is not created and other problems can arise in cascade.

Worse, let us imagine that the small difference is simply the renaming of an entity type, in that case, the transformation is still possible in theory, but in practice, since its identifier was changed, the entity type cannot be found anymore, even if it is still there.

A solution to these problems is the following: when a transformation is read from the history, its preconditions are verified; if they are fulfilled, the transformation can be executed, else, the replay engine stops and asks the user what to do, such as discarding the transformation, replacing it by another one (for instance, by the same transformation of a construct having another name), or performing one or more other transformations before executing the first one. But this is a manual job that cannot be automated.

7.2.5. Replaying every primitive processes

The previous four paragraphs showed how to replay every kind of primitive process. But it was done with a strong hypothesis not mentioned for simplicity. It is now time to remove this hypothesis. It was assumed that the products that where used during the construction of the history can be copied in the state they were when the recording of the history began, and that these products are effectively available. But it may happen that they are not.

For instance, the *Relational design* process type shown in Figure 7.3[11] specifies that a single product, of *Relational logical schema* type, has to be transformed by five consecutive primitive processes. In the history, the state of the product is stored before the first primitive process. All the five primitive processes are stored too. The third process cannot be replayed directly since the product is not stored in the history in the state it was after the second process. This problem can be solved in two ways:

- The first two primitive processes can be replayed before replaying the third one. More generally, this solution requires that an engineering process, or at least a part of it, is replayed before the process of interest can be replayed too.

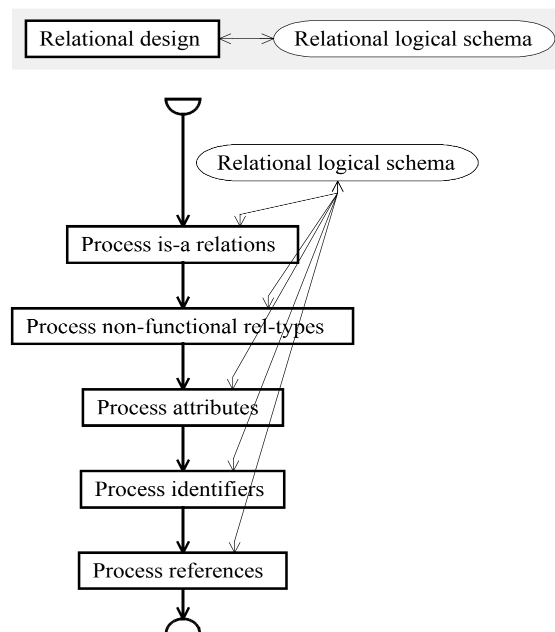


Figure 7.3 A simple process example

11 This example is an extract of the first case study in Chapter 11.

- When performing the engineering process the first time, the analyst can manually do a copy of the product after each primitive process in order to keep all these states in the history, even if it is not required by the method. The replay can then start with these copies. This solution is simpler than the first one, but can only be applied if the analyst foresees this replay scenario. However, recording more product states makes the history bigger.

7.2.6. Replaying engineering processes

Before trying to replay engineering processes, it is necessary to answer the question: what does replaying an engineering process mean? The technique used with primitive processes of automatic type cannot be applied. Indeed, reusing the input and update products and replaying the strategy of the engineering process type is impossible since decisions have to be taken, and it is necessary to remember what decisions were taken the first time. So we could imagine to replay an engineering process as primitive processes of manual type are replayed, by taking every sub-process in sequence and replaying them, taking the same decisions. But this raises a series of questions:

- What if the products are no longer the same?
- Can a sub-processes that can no longer be replayed be just skipped?
- Can any sub-process be skipped?
- If a sub-process is skipped, is it still possible to replay the remaining of the process?
- Can a sub-process be replaced by another sub-process? Or several?
- If the replay ends in a deadlock, can the strategy be used to put the replay back on track, possibly by performing some new intermediate sub-processes? Or is it better to acknowledge the failure?
- If several hypotheses were made and if some parts of the engineering process were performed several times with the hypotheses, do all the hypotheses need to be replayed again? Or just the ones the resulting products of which were selected in a decision?

It appears that the meaning of replaying an engineering process may vary according to the needs of the user. So, a universal replay method cannot be provided. It is better to provide a series of tools that allow the user to replay according to his or her needs. Section 7.4 will provide some tools. The implementation of a CASE environment may also provide other tools.

7.3. History evolution

The life of a database can span several decades. However, the world it represents evolves, so the database has to evolve too. Possible reasons of evolution are numerous:

- users' needs change
- the enterprise owning the database evolves in size, it expands geographically, it offers new products, new kinds of products, new services,...
- enterprises merge or buy other ones
- the economic world evolve with the internet, or with the arrival of new products, new competitors,...
- laws evolve
- ...

So a database has to evolve continuously along its lifetime. If it was originally designed with a CASE environment, the same CASE environment can be used to make it evolve. [HAINAUT,94] and [HICK,98] present the approach followed by the DB-MAIN CASE environment. And if the complete history of the original design is still available, it can be reused. The basic ideas that underlie the concept of database evolution will now be summarised. A complete study can be found in [HICK,01].

Let us start a new information system project for a company. In the first phase, which is not in the scope of this work, the requirements of the industry are collected and organised. Let us call R0 the resulting functional requirements. Then the database engineers start their job and, on the basis of these requirements, draw a conceptual schema[12] for the database that will support the new information system. Let us call this schema CS0. The database engineers go on with their job and transform CS0 into an equivalent logical schema which is compliant with the chosen DBMS. Let us call R0' the logical requirements which are partly imposed by the DBMS. And let us call the logical schema LS0. The whole complex history of the transformation job is recorded. Let us call this history H0. Finally, the LS0 schema is passed to programmers that will generate application programs and data structures (P0) that will be used by the employees of the company. This usage will fill the database with data (D0). The left half of Figure 7.4 shows the starting situation.

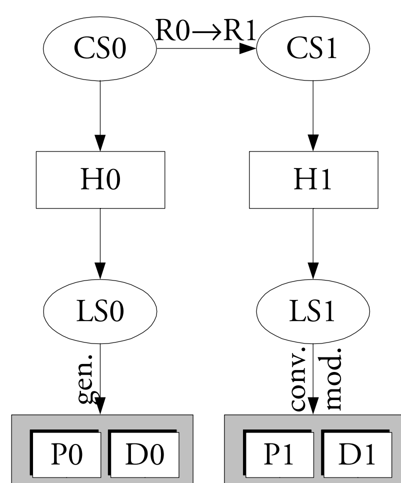


Figure 7.4 A simple project evolution example with an update of the requirements

Some time after the database was created, the database is naturally bound to evolve in order to meet new requirements: R1. R1 are supposed to be just slightly different from R0[13]. The history built during the creation, H0, will be reused. In fact, the history, or its components, cannot be modified anymore because the memory of the past should not be altered. So, when an engineer wants to make a project evolve, he or she has to start a new engineering branch in the history, called H1. The whole history is then made up of two branches: H0 and H1 (Figure 7.4). To start the new branch, the engineer will make a copy of CS0 and apply all the changes required by the changes (R0→R1). The result, CS1, is a new ver-

12 This example uses only one schema for simplicity, but this is done without loss of generality since several schemas could either be treated separately, as a single set, or even as several sets separately.

13 *Slightly different* means that just a few points of the requirements have changed. This definition does not affect the complexity of the changes. If the requirements are substantially changed, it is generally advised to treat different problems separately, to view all the changes as several sets of a few (possibly only one) changes and to treat each set at once. So, this limitation is recommended for the ease of use only and can be made without loss of generality. Note that the use of the knowledge acquired during the design of an information system to help to solve a new problem (no common point between R0 and R1) cannot be seen as an evolution problem.

sion of CS0. The major part of the work that has been done from that point in H0 to produce LS0 should be redone. To do so, the analyst will replay H0 as presented in the previous section: every action performed from that point in H0 will be performed again and recorded in H1, except for the constructs that have been updated or deleted. In the latter case, the analyst has to manually process the new components, which is recorded in H1 too. Moreover, the replayed history can be simplified. If, in H0, the analyst had to try several hypotheses, now that the best ones are known, he or she can discard the others. The result of this updated replay is a logical schema LS1 which is a new version of LS0. From there, application programs have to be modified (P1) and the data stored in the database have to be converted (D1) in order to be consistent with the new database definition. This last phase is beyond the scope of this thesis (see [HICK,01]).

If changes can appear in the functional requirements R0, they can also appear in the logical requirements R0'. For instance, if the company wants to upgrade its DBMS, the logical schema may have to be updated in order to fulfil the requirements of the new DBMS. In that case, database engineers will make a new version of the logical schema (LS2) and update it, as shown on Figure 7.5. The modification of the programs (P2) and the conversion of the data (D2) have to be done as above.

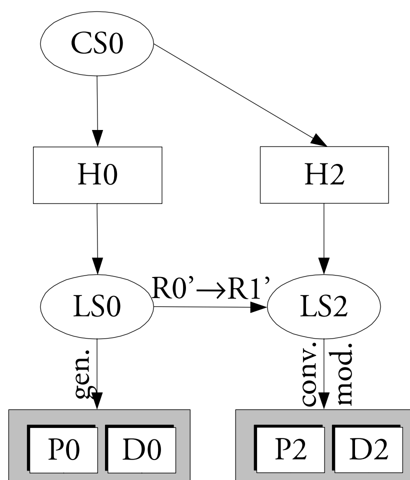


Figure 7.5 A second project evolution example with a modification at the logical level

Since the logical schema has changed (LS0→LS2) but not the conceptual schema (CS0), the history of the logical design (H0) is no more valid. It must be updated too. This can be done by replaying H0 with some exceptions: transformations that concern constructs of LS0 which are modified in LS2 are discarded. Constructs of CS0 which are not treated by the replay, are transformed with other tools in order to obtain LS2. The replay and the new transformations are recorded in the history, named H2 in its new state.

Let us examine a more concrete example. Figure 7.6 shows the state H2 of a history resulting from the evolution of the project in Chapter 6, Figure 6.14 (H0). With the new requirements in mind, he or she performed the process *Proc* a third time. Since all the hypotheses, the processes, the different resulting product versions, and the decision are still in the history, the engineer directly knew he or she had to perform the process with the new requirements in the same way as with the second hypothesis. In fact, the engineer replayed the *Proc / Hyp 2* process with just a few alterations due to the new requirements. This gave birth to a third version of the product: *Output/2'*. Then the work continued like during the first execution with the *Next process / New req* by replaying *Next process*.

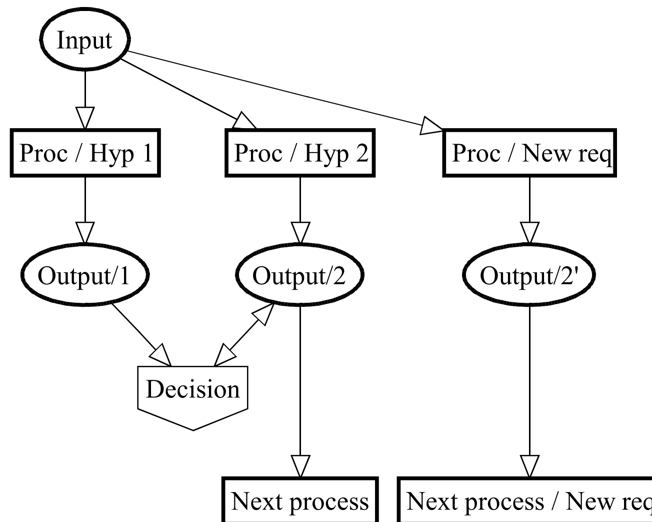


Figure 7.6 A project evolution example

7.4. History transformation

The previous sections sketched two possible uses of the history: replay and database evolution. But the history can be used for other tasks too. For some of them, the history has to be modified, transformed, before being usable; for instance, to recover a database design during a re-engineering activity as presented in [HAINAUT,96b]. In order to transform histories, some tools need to be defined. The structure of histories was already described. Histories can be handled as a whole, but, very often, an intermediate degree of granularity may be more appropriate. A few simple functions will be defined in order to better identify some parts of the histories. Then the notion of excerpt will be introduced. This section will end by showing how these excerpts can be organised and what relationships can be defined between them.

7.4.1. History characteristics

This section gives basic definitions which allow to identify some components of histories.

A. The scope of a history

Each history shows the transformation of products. They define the scope of the history:

- The **scope of a primitive process history** $L=(P,\{S_1,\dots,S_m\},(T_1,T_2,\dots,T_n)) \in \mathcal{L}$ is the set of all the products it uses, modifies or creates. It will be written *Prod*(L). More formally,

  $Prod(L) = \{S_1,\dots,S_m\}$

- The **scope of an engineering process history** $G=(P,V,E) \in \mathcal{G}$ is the set of all the products it uses, modifies or creates. It will be written *Prod*(G). More formally,

  $Prod(G) = \{r \mid r \in V \wedge r \in \mathcal{P}\}$

- The **scope of a decision** $D=(S,C) \in \mathcal{D}$, or a decision $D=(S,b) \in \mathcal{D}$, that is to say the set of products on the basis of which the decision is taken, is part of its definition:

  $Prod(D) = S$

- The **scope of a project history** $H=(V,E) \in \mathcal{H}$ is the set of the products it uses, modifies or creates:

  $Prod(H) = (\cup_{L \in V \cap \mathcal{L}} Prod(L)) \cup (\cup_{G \in V \cap \mathcal{G}} Prod(G)) \cup (\cup_{D \in V \cap \mathcal{D}} Prod(D))$

Applied to the LIBRARY design example in Chapter 6, Figure 6.8, these definitions give:

- Analysis $\in \mathcal{L}$, *Prod*(Analysis) = {Library/Conceptual, library.txt/IR}
- Logical Design $\in \mathcal{G}$. *Prod*(Logical Design) = {Library/Conceptual, Library/First Logical, Library/Logical}
- The history contains no decision.
- H = Library, *Prod*(Library) = {library.txt/IR, Library/Conceptual, Library/First Logical, Library/Logical, Library/Physical, library.ddl/1}

B. Identifying histories

Since a project history is made up of several smaller histories, it is necessary to be able to identify all of them:

- A primitive process history $L \in \mathcal{L}$, $L=(P,\{S_1,...,S_m\},(T_1,T_2,...,T_n))$ can be identified by the process to which it is attached: *Id*(L) = P.
- An engineering process history $G \in \mathcal{G}$, G=(P,E,V) can be identified in the same way: *Id*(G) = P.
- A decision $D \in \mathcal{D}$, D=(S,C) can be identified by its scope: *Id*(D) = S. Indeed, the fact that two decisions are taken about the same products can be forbidden: two decisions with the same result is useless and two decisions with different results is a contradiction.

C. Finer grained scope of a history

Due to the transformational interpretation of histories, they can be considered transformations. In chapter 6, the structural functions $\Delta_-(T)$, $\Delta_+(T)$ and $\Delta_0(T)$ were defined to express the changes due to transformation T on a single product. In order to state that the concerned product is r, an extended notation can be defined for these functions:

$\Delta_-(T,r)$ gives the set of components of product r destroyed by transformation T

$\Delta_+(T,r)$ gives the set of components created by transformation T in product r

$\Delta_0(T,r)$ gives the set of components of product r concerned but preserved by T

A primitive process history $L=(P,\{S_1,...,S_m\},(T_1,T_2,...,T_i,...,T_n)) \in \mathcal{L}$ is a sequence of transformations on various products. For each $r \in$ *Prod*(L), we can define:

$C_{r,L,pre}$ the set of all the constructs in r before execution of P.

$C_{r,L,post}$ the set of all the constructs in r after execution of P.

$\Delta_-(L,r) = \cup_{1\leq i\leq n}\Delta_-(T_i,r) \cap C_{r,L,pre} = C_{r,L,pre} \setminus C_{r,L,post}$
the set of constructs in product r destroyed by the process P whose history is L.

$\Delta_+(L,r) = \cup_{1\leq i\leq n}\Delta_+(T_i,r) \setminus \cup_{1\leq i\leq n}\Delta_-(T_i,r) = C_{r,L,post} \setminus C_{r,L,pre}$
the set of constructs in product r created by the process P whose history is L. The set substraction in the second member is due to the fact that one transformation T_j can delete some constructs created by a previous transformation T_i.

$\Delta_0(L,r) = \cap_{1\leq i\leq n}\Delta_0(T_i,r)$, $\Delta_0(L,r) \subseteq C_{r,L,pre} \cap C_{r,L,post}$
the set of constructs in product r concerned and preserved by process P.

Let us note that if one construct is destroyed, then created again, the second one is supposed to be another construct, no matter what its characteristics are, being the same or not. This simplifies expressions and it can be done with no loss of generality.

At the end of chapter 6, in a concise history, it was decided to record the signature of every single transformation (every T_i) for processes of manual types only. For processes of automatic types, these basic transformations exist but they are hidden, so the second member of these expressions can hardly be evaluated in practice. This can be a problem in the last expression because the third member is only an approximation of the true result. It can then be best to use only extended primitive process histories when such calculus will be needed. From a theoretical point of view, since the hidden actions exist anyway, the following reasoning remains valid whatever the type of primitive process history.

More globally, the following expressions can also be defined on L:

$$C_{L,pre} = \cup_{r_i \in Prod(L)} C_{r_i,L,pre}$$

$$C_{L,post} = \cup_{r_i \in Prod(L)} C_{r_i,L,post}$$

$$\Delta_-(L) = \cup_{r_i \in Prod(L)} \Delta_-(L,r_i)$$

$$\Delta_+(L) = \cup_{r_i \in Prod(L)} \Delta_+(L,r_i)$$

$$\Delta_0(L) = \cup_{r_i \in Prod(L)} \Delta_0(L,r_i)$$

An engineering process history $G=(P,V,E) \in \mathcal{G}$ is a graph whose nodes are either:

- primitive processes for which the previous expressions can be evaluated
- other engineering processes
- decisions that do not alter products
- products used, created, or modified by sub-processes; let us define

  $S_1 = \{r \mid (\exists\, L \in V, L \in \mathcal{L}, r \in prod(L)) \vee (\exists\, G \in V, G \in \mathcal{G}, r \in prod(G))\}$
- unused products:

  $S_2 = \{r \mid r \in V \wedge p \in \mathcal{P}\} \setminus S_1$

The previous definitions can be extended to engineering process histories in the following recursive way:

$\forall\, r \in Prod(G), C_{r,G,pre}$ the set of all the constructs in r before execution of P.

$\forall\, r \in Prod(G), C_{r,G,post}$ the set of all the constructs in r after execution of P.

$\forall\, r \in S_1, \Delta_-(G,r) = \cup_{p_i \in V, p_i \in L \cup G} \Delta_-(p_i,r) \cap C_{pre} = C_{r,G,pre} \setminus C_{r,G,post}$
the set of constructs in product r destroyed by the process P whose history is L.

$\forall\, r \in S_1, \Delta_+(G,r) = \cup_{p_i \in V, p_i \in L \cup G} \Delta_+(p_i,r) \setminus \cup_{p_i \in V, p_i \in L \cup G} \Delta_-(p_i,r) = C_{r,L,post} \setminus C_{r,L,pre}$
the set of constructs in product r created by the process P whose history is L.
The set substraction in the second member is due to the fact that one sub-process p_i can delete some constructs created by a previous sub-process.

$\forall\, r \in S_1, \Delta_0(G,r) = \cap_{p_i \in V, p_i \in L \cup G} \Delta_0(p_i,r) = C_{r,G,pre} \cap C_{r,G,post}$
the set of constructs in product r concerned and preserved by process P.

$$C_{G,pre}(G) = \cup_{r_i \in Prod(G)} C_{r_i,G,pre}$$

$$C_{G,post}(G) = \cup_{r_i \in Prod(G)} C_{r_i,G,post}$$

$$\Delta_-(G) = \cup_{r_i \in S_1} \Delta_-(G,r_i)$$

$$\Delta_+(G) = \cup_{r_i \in S_1} \Delta_+(G,r_i)$$

$$\Delta_0(G) = (\cup_{r_i \in S_1} \Delta_0(G,r_i)) \cup (\cup_{r_i \in S_2} C_{r_i,G,pre})$$

Since the effect of the complete history $H=(V,E) \in \mathcal{H}$ on its products is the same as the effect of its root engineering process, G_{root} ($G_{root} \in V$, $H=((G_{root},1),...)$ in its representation by levels), the following can be written:

$$\Delta_-(H) = \Delta_-(G_{root})$$

$$\Delta_+(H) = \Delta_+(G_{root})$$

$$\Delta_0(H) = \Delta_0(G_{root})$$

In fact, since the history of a project contains no products when it begins, $C_{G_{root},pre} = \emptyset$, and $\Delta_-(G_{root}) = \Delta_-(H) = \emptyset$.

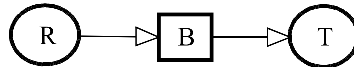
7.4.2. Excerpts

A **history excerpt** is a part of the history that can be isolated in order to concentrate on it only. Excerpts can be extracted from each kind of history:

- $L_e=(P,\{S_1,...,S_{m_e}\},(T_{i_1},T_{i_2},...,T_{i_e})) \in \mathcal{L}$ is an excerpt of the primitive process history $L=(P, \{S_1,...,S_m\},(T_1,T_2,...,T_n)) \in \mathcal{L}$ if, and only if, $\{S_1,...,S_{m_e}\} \subseteq \{S_1,...,S_m\}$ and $1 \le i_1 \le i_2 \le ... \le i_e \le n$, i.e. all the transformation signatures of L_e appear in L in the same order, not necessarily consecutively. This is denoted by the expression $L_e \subseteq L$. For instance, the following is an excerpt of the log file shown in Chapter 6, Figure 6.4:

```
&MOD ROL
%BEG
   *OLD ROL
   %BEG
      %OWN "LIBRARY"/"Conceptual"."written"
      %ETR "LIBRARY"/"Conceptual"."BOOK"
   %END
   %OWN "LIBRARY"/"Conceptual"."written_by"
   %ETR "LIBRARY"/"Conceptual"."BOOK"
%END
```

- $G_e=(P,V_e,E_e) \in \mathcal{G}$ is an excerpt of the engineering process history $G=(P,V,E) \in \mathcal{G}$ if, and only if, $V_e \subseteq V$ and $E_e \subseteq E$, i.e. all the nodes of G_e are excerpts of some nodes of G and edges of G_e are edges of G. This is denoted by the expression $G_e \subseteq G$. The following is an example of graph excerpt of the graph shown in Chapter 6, Figure 6.5:



- The graph $H_e=(V_e,E_e)$ is an excerpt of the project history $H=(V,E) \in \mathcal{H}$ if, and only if, $V_e \subseteq V$ and $E_e \subseteq E$, i.e. all the nodes of H_e are excerpts of some nodes of H, and all nodes of H_e are edges of H. This is denoted by the expression $H_e \subseteq H$. H_e is not necessarily a tree, it can be a forest. So, a history excerpt is not necessarily a history.

Let us assume $H=(V,E) \in \mathcal{H}$, $L \in \mathcal{L}$ and $L \in V$, $G \in \mathcal{G}$ and $G \in V$; it is straightforward that $(\{L\},\emptyset) \subseteq H$ and $(\{G\},\emptyset) \subseteq H$. For the simplicity of notations, $(\{L\},\emptyset)$ and L can be assimilated, as well as $(\{G\},\emptyset)$ and G. The following simplified notations can be defined too: $L \subseteq H$ and $G \subseteq H$, meaning that both L and G are excerpts of H.

The notion of scope of the history can be extended to history excerpts: *Prod*(L_e), *Prod*(G_e), *Prod*(H_e) denote the sets of all the products used, modified or created by L_e, G_e and H_e respectively. It is straightforward that *Prod*(L_e) $\subseteq$ *Prod*(L), *Prod*(G_e) $\subseteq$ *Prod*(G), and *Prod*(H_e) $\subseteq$ *Prod*(H).

Let H_e be history excerpt, $H_e \subseteq H$, $H \in \mathcal{H}$ which starts at a time point where a product $p_1 \in Prod(H_e)$ is known to be available. H_e can be seen as the history of a possibly fictive transformation process which produces product $p_2 \in Prod(H_e)$. We can write: $p_2 = H_e(p_1)$.

The structural functions $C_-(H)$, $C_+(H)$ and $C_0(H)$ can be applied to history excerpts too, if they are made of a single tree. If an excerpt H_e is a forest, it contains several root processes. Since engineering processes are just seen as a way to encapsulate products in the expression of the structural functions, a fictive engineering process graph G_{fic} whose sub-processes are the root processes of the forest can be added in order to transform the forest in a single tree, and the structural functions can be evaluated on this new fictive history excerpt:

$$C_-(H_e) = C_-(G_{fic})$$

$$C_+(H_e) = C_+(G_{fic})$$

$$C_0(H_e) = C_0(G_{fic})$$

7.4.3. Independent history excerpts

Let us consider a history H and two excerpts $H_1 \subseteq H$, $H_2 \subseteq H$. Does the execution of H_1 depend on the execution of H_2 or are they independent? If they are independent, replaying H_1 then H_2, or H_2 then H_1, or even both in parallel will give the same results.

Let us define a **partial order relation**:

$$H_1 < H_2 \iff C_0(H_1) \cap C_-(H_2) \neq \varnothing \vee C_+(H_1) \cap C_0(H_2) \neq \varnothing \vee C_+(H_1) \cap C_-(H_2) \neq \varnothing$$

In other words, $H_1 < H_2$ (H_1 must be performed before H_2) if, and only if, either H_2 deletes constructs concerned by H_1 or H_1 creates constructs concerned by H_2.

H_1 and H_2 are said to be **independent** if, and only if,

$$\neg(H_1 < H_2) \wedge \neg(H_2 < H_1)$$

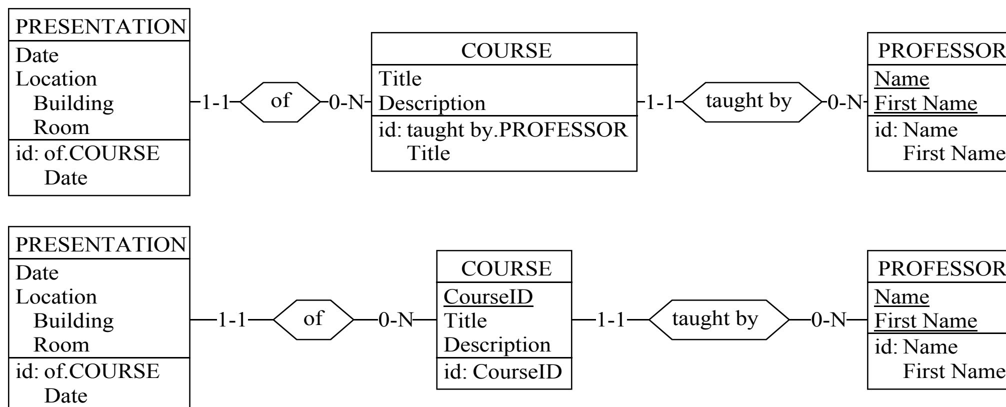


Figure 7.7 Two schemas to transform into relational schemas. In the first one, the rel-type "of" can only be transformed into a reference group of attributes if the rel-type "taught by" is transformed first. In the second schema, the transformation of each rel-type is independent from the transformation of the other.

For instance, Figure 7.7 shows two conceptual schemas bound to be transformed towards a relational model. The conversion process includes the transformation of the rel-types into reference groups of attributes. In the upper schema, since the primary identifier of “COURSE” includes a role, the rel-type “of” cannot be transformed directly (reference groups with roles are not permitted). But the rel-type “taught-by” can be transformed. This transformation modifies the primary identifier of “COURSE” which is then made up of the new reference attribute and of “Title”. So that “of” can now be transformed too. In the lower schema, both rel-types can be transformed directly. Doing the transformations in any

order always lead to the same result. If H_{11} is the excerpt containing the transformation of "taught by" in the upper schema, if H_{12} is the excerpt containing the transformation of "of" in the same schema, and if H_{21} and H_{22} are respectively the excerpts containing the transformations of the same rel-types in the lower schema, then H_{21} and H_{22} are independent, but H_{11} and H_{12} are not.

7.4.4. Equivalent history excerpts

Assuming $H \in \mathcal{H}$, let $H_1 \subseteq H$ and $H_2 \subseteq H$ be two excerpts.

The two history excerpts H_1 and H_2 will be said **equivalent with respect to a product** $p \in Prod(H_1) \cap Prod(H_2)$:

$$H_1 \equiv_p H_2 \Leftrightarrow H_1(p) = H_2(p).$$

Note that if $p \in Prod(H_1)$ and $p \notin Prod(H_2)$ (respectively $p \notin Prod(H_1)$ and $p \in Prod(H_2)$), then $H_1(p) = (E_1,V_1)$, $E_1 \neq \varnothing$ and $H_2(p) = (\varnothing,\varnothing)$ (respectively $H_1(p) = (\varnothing,\varnothing)$ and $H_2(p) = (E_2,V_2)$, $E_2 \neq \varnothing$), and the non-equivalence is straightforward. If $p \notin Prod(H_1) \cup Prod(H_2)$, then $H_1(p) = (\varnothing,\varnothing) = H_2(p)$ and the equivalence is straightforward, but is of no interest.

The two history excerpts H_1 and H_2 are said **equivalent**:

$$H_1 \equiv H_2 \Leftrightarrow (Prod(H_1) = Prod(H_2)) \wedge (\forall\, p \in Prod(H_1), H_1 \equiv_p H_2)$$

For instance, to disaggregate a compound attribute, or to delete it after having created new simple attributes with the same characteristics as the compound attribute components, are different but equivalent processes.

7.4.5. Minimal history excerpts

Let $H \in \mathcal{H}$ and $H_m \subseteq H$.

H_m is **minimal with respect to product** $p \in Prod(H_m)$ if, and only if,
$\forall\, H' \subset H_m, H'(p) \neq H_m(p)$.

H_m is a **minimal excerpt** if, and only if, $\forall\, p \in Prod(H_m)$, H_m is minimal with respect to p.

7.4.6. Operations on history excerpts

A. On primitive process history excerpts

Let us assume $L, L_1, L_2, L_3, L_4 \in \mathcal{L}$ and $L_1 \subseteq L$, $L_2 \subseteq L$, $L_3 \subseteq L$, $L_4 \subseteq L$.

Delete: $L_1 - L_2$ is a new primitive process history excerpt obtained by copying from L_1 all the transformations that do not appear in L_2. $L_1 - L_2 \in \mathcal{L}$.

Concatenation: $L_1 + L_2$ is the concatenation of both excerpts L_1 and L_2. In other words, $L_1 + L_2$ is a new primitive process history excerpt on the same product as L_1 and L_2 made up of all the transformations of L_1 followed by all the transformations of L_2. $L_1 + L_2 \in \mathcal{L}$, but $L_1 + L_2 \not\subseteq L$ because the transformations in $L_1 + L_2$ does not necessarily appear in the same order as in L.

If L can be decomposed into L_1, L_2, L_3, L_4 such that $L \stackrel{\text{def}}{=} L_1 + L_2 + L_3 + L_4$, L_1 and L_4 being possibly empty, and if L_2 and L_3 are independent, then L_2 and L_3 can be swapped and $L \equiv L_1 + L_3 + L_2 + L_4$.

Replace: If L can be decomposed as $L_1 + L_2 + L_3$, L_1, L_2 or L_3 being possibly empty, $L|L_2 \rightarrow L_5$, where $L_5 \in \mathcal{L}$ and $Prod(L_5) \subseteq Prod(L)$, is the transformation of the primitive pro-

cess history into a new one which is $L_1+L_5+L_3$. Furthermore, $L_2 \equiv L_5 \Rightarrow L \equiv L|L_2 \rightarrow L_5$. The reverse is not true. It can easily be proved by a counter-example in which L_1 creates an entity type A, L_3 deletes A, L_2 do several transformations that does not concern A, and L_5 is exactly the same as L_2, except that it contains one more transformation adding an attribute to A.

B. On engineering process history excerpts

Let us assume $G, G_1, G_2 \in \mathcal{G}$ and $G=(P,V,E)$, $G_1=(P,V_1,E_1) \subseteq G$, $G_2=(P,V_2,E_2) \subseteq G$.

Delete: $G_1 - G_2$ is a new engineering process history excerpt obtained by copying all the components from G_1 that do not appear in G_2; $G_1 - G_2 = (P,V_3, E_3)$ where:

$$V_3 = \{v \mid (v \in V_1 \wedge v \notin V_2) \vee (\exists\, v_1 \in V_1 \wedge \exists\, v_2 \in V_2, \mathit{Id}(v_1) = \mathit{Id}(v_2), v = v_1 - v_2)\}$$

$$E_3 = E_1 \setminus E_2$$

For $G_1 - G_2$ to be a graph, that is to say for $G_1 - G_2$ to be valid, the extremities of edges in E_3 must be in V_3. In other words,

$$\begin{aligned} G_1 - G_2 \in \mathcal{G} \Leftrightarrow \forall\, v_i \in V_2, \forall\, v_j \in V_1:\ & ((v_i,v_j) \in E_1 \Rightarrow (v_i,v_j) \in E_2) \\ & \wedge ((v_j,v_i) \in E_1 \Rightarrow (v_j,v_i) \in E_2) \\ & \wedge ((\underline{v_i,v_j}) \in E_1 \Rightarrow (\underline{v_i,v_j}) \in E_2) \\ & \wedge ((\underline{v_j,v_i}) \in E_1 \Rightarrow (\underline{v_j,v_i}) \in E_2). \end{aligned}$$

Replace: Let $N, N' \in \mathcal{L} \cup \mathcal{G}$ such that $\mathit{Prod}(N') = \mathit{Prod}(N)$ and let us assume $N \in V$. $G|N \rightarrow N'$ is an engineering process history excerpt obtained by copying the graph G, except the node N which is replaced by the node N'.

Merge: Let $G'=(P',V',E') \in \mathcal{G}$ be a node of G: $G' \in V$. $G \oplus G'$ will denote the merging of the graph G' into the graph G, an operation that replaces a single engineering process node of G by its content:

$$G \oplus G' = (P, (V \setminus \{G'\}) \cup V', (E \setminus \{(v_i,v_j) | (v_i,v_j) \in E \wedge (v_i=G' \vee v_j=G')\}) \cup E')$$

This expression shows that the set of nodes of the result is the set of nodes of G in which the node G' itself is replaced by all the nodes of G'. It also shows that the set of edges of the result is the original set of edges from which all the links with G' in G are removed and to which all the links of G' are added. Indeed, $V \cap V'$ is the set of all products of G that are input, update or output of G'. So, $V \cap V'$ is the set of all the nodes that are connected to G' in G, and all these connections have to disappear when G' disappears. Furthermore, $V \cap V'$ are all the products to which the content of G' is connected by links in E' and these links are not altered by the removal of the links above.

An interesting property is that $G \oplus G' \equiv G$. Indeed, the merge operator only changes the structure of the history, not its content, all the recorded transformations are still performed in the same order.

C. On project history excerpts

Let $H=(E,V)$ be a history and $H_1=(E_1,V_1) \subseteq H$, $H_2=(E_2,V_2) \subseteq H$ be two excerpts.

Delete: $H_1 - H_2$ is the new history excerpt obtained by copying all the components from H_1 that do not appear in H_2. $H_1 - H_2 = (V_3,E_3)$ where:

$$V_3 = \{v \mid (v \in V_1 \wedge v \notin V_2) \vee (\exists\, v_1 \in V_1 \wedge \exists\, v_2 \in V_2, \mathit{Id}(v_1) = \mathit{Id}(v_2), v = v_1 - v_2)\}$$

$$E_3 = E_1 \setminus E_2$$

For $H_1 - H_2$ to be a forest, the extremities of edges in E_3 must be in V_3. In other words, $H_1 - H_2$ is a forest if, and only if:

$$\forall v_i \in V_2, \forall v_j \in V_1: ((v_i,v_j) \in E_1 \Rightarrow (v_i,v_j) \in E_2) \wedge ((v_j,v_i) \in E_1 \Rightarrow (v_j,v_i) \in E_2).$$

Replace: Let N, N' $\in \mathcal{L} \cup \mathcal{G}$ such that *Prod*(N') = *Prod*(N) and let us assume N $\in$ V. H|N→N' is a history excerpt obtained by copying the graph H, except the node N which is replaced by the node N', and except the engineering process history $G=(P,V_G,E_G) \in V$ such that $N \in V_G$ which must be replaced by G|N→N'.

7.4.7. History transformation

If $N_1, N_2,..., N_n \in \mathcal{H} \cup \mathcal{L} \cup \mathcal{G}$, and if *f* is a function that transforms $N_1, N_2,..., N_n$ into N $\in \mathcal{H} \cup \mathcal{L} \cup \mathcal{G}$, this history transformation will be written in the following way:

$$N \leftarrow f(N_1, N_2,..., N_n)$$

Typically, *f* is a composition of the operators defined above. For instance,

$$H \leftarrow (H_1|L_1 \rightarrow L_2) - H_2$$

7.5. History cleaning

The idea of recording a trace for every single action, possibly completed with extra information, is interesting to keep a complete trace of a whole project, but it may lead to enormous data files. The fact is that all these data are not always relevant. For several reasons, the absolute completeness of the data is not always necessary. Data could be summarised so that they still look complete (even if they are not), without loss of precision or correctness, and often with a gain of readability. Let us examine a few situations:

- We are human, we make errors. When drawing a database schema, an analyst may select a wrong tool and draw an entity type instead of a rel-type. He simply deletes the faulty rel-type, and selects the right tool to draw the intended entity type. Globally, the schema has not changed, but the log file contains both actions. The log file is complete. But it would still look complete if the two above actions were not in it: their presence does not add nor remove some precision to the history. And, the context for the other actions of the history does not change according to their presence or absence. Those two entries are some noise that reduces the history readability.
- Some actions can sometimes be performed in several steps. For instance, the readability of a database schema can often be improved by re-arranging its components during a primitive process. During the process, an analyst can move a particular entity type several times: at the beginning of the process she moves it next to another entity type to which it is connected; two minutes later, she aligns it horizontally with the entity type placed above; finally, she aligns it vertically with the entity type at its right. Before the positioning process, the entity type was at one place, and after it is at another place; this single fact is important. So the three real moves can be replaced by a single one from the origin of the first real one to the destination of the third real one. The history does not lose or gain precision, moving an object only once is as correct as doing the same move in several steps, and the summary makes the history eventually gain in readability.
- When an information system evolves, some requirements become obsolete. So the analyst evolves the information system in order to remove the parts of the database that deal with the obsolete requirements. The history shows everything from the first analysis of the requirements to their evolution, and to their obsolescence and removal. If it is sure that all this will never be useful again, it can be discarded.

- When a computer program is used, doubts concerning the use of a particular functions may arise. So users usually browse through the help files, and make a little test in the margin of the schema thay are working on. It is surely not interesting to keep the trace of this test.

All these cases are good examples among a lot of possible situations that can arise daily in which cleaning the history can be interesting.

Cleaning a history means removing the trace of all the actions we do not want to see anymore while preserving the correctness and the global content of the history.

7.5.1. History cleaning

The result of an engineering process is a history H. Most generally, H is not minimal. The goal of history cleaning is to suppress some uninteresting constructs (possibly all of them) in order to find an excerpt $H_{clean} \subseteq H$ (possibly a minimal excerpt) which is equivalent to H with respect to some products of interest (generally all the sources of information such as requirement analysis reports,...).

This process can be performed in three steps:

1. cleaning primitive process histories
2. cleaning engineering process histories
3. compute the new project history as explained in Chapter 6, Section 6.3.6.

Cleaning a history must be performed by removing or modifying its components with respect to the four basic properties of histories (see chapter 6):

- A history has to keep its readability. But this is not a problem since the goal of the cleaning is to improve readability.
- A history has to remain formal. When some components are simply removed of the history, and if no other component is modified, the history surely remains formal. But if some components are modified, some attention is worth to be drawn to them.
- A history has to keep its correctness. This is the point that deserves the more attention.
- A history has to keep its completeness. In fact, this can hardly been achieved in the strict sense of the term since some action traces are removed, but the cleaned history must *look* complete in the sense that every possible reuse of the original history can still be performed with the new one with no difference in the final result; a log file H_1 is replaced by an equivalent one, H_2, with respect to the products of interest: $H_1 \equiv_{p_1,\ldots,p_n} H_2$.

7.5.2. Primitive process history cleaning

Attention must be paid to the correctness property. The correctness of a primitive process history entry was defined above in its context, and the context was defined with the preceding entries of the history. So, when an entry is removed or modified, the context of all the subsequent entries is modified. In order to keep the correctness of the whole history, it is necessary to check that all these subsequent entries are still valid in their new context. If it is not the case, either the removal or the modification cannot be performed or other ones must be performed too in order to recover a correct context for all entries.

Let $L_1 = (P,\{S_1,\ldots,S_{m_1}\},(T_1,T_2,\ldots,T_{t_1})) \in \mathscr{L}$ be a primitive process history, let $T_{i_1},T_{i_2},\ldots,T_{i_n}$, $1 \leq i_k \leq t_1$, $1 \leq k \leq n$ be history entries from L_1 to be modified or deleted, and let $L_2 = (P,\{S_1,\ldots, S_{m_2}\},(T_1,T_2,\ldots,T_{t_2}))$ be the result from the modifications and deletions. L_2 is a cleaning of L_1 if, and only if:

- $L_2 \in \mathcal{L}$
- $\forall$ j, $1<j\leq t_2$, the preconditions Pre_j of the transformation T_j are satisfied
- $\forall$ $p_1 \in$ *Prod*(L_1), $\exists!$ $p_2 \in$ *Prod*(L_2) : p_1 and p_2 are the same

These conditions can be checked by replaying L_2. If the replay reaches its end, then the two first conditions are satisfied. Comparing the resulting products will tell if the third condition is satisfied too.

In practice, a primitive process history can be cleaned in the following way:

If $L = (P,\{S_1,...,S_m\},(T_1,T_2,...,T_t)) \in \mathcal{L}$ contains two entries, T_i and T_j, $1\leq i<j\leq t$, such that there exists $T'=T_j o T_i$, and such that for all T_k, $i<k<j$, T_i and T_k are independent, then L can be modified by removing T_i and replacing T_j by T'.

Indeed, $L = L_1+L_2+...+L_t$ where $L_i = (P,\{S_{1,i},...,S_{m,i}\},(T_i))$, $1\leq i\leq t$, $S_{j,i}$ being the state of product S_j before transformation T_i. Furthermore, $\forall$ T_k, $i<k<j$, T_i and T_k are independent $\Leftrightarrow$ L_i and L_k are independent.

So, $L = L_1+...+L_i+L_{i+1}+...+L_j+...+L_t$
$= L_1+...+L_{i+1}+L_i+...+L_j+...+L_t$
$= ...$
$= L_1+...+L_{i+1}+...+L_{j-1}+L_i+L_j+...+L_t$
$= L_1+...+L_{i+1}+...+L_{j-1}+L'+...+L_t$

where $L' = (P,\{S_{1,i},...,S_{m,i}\},(T_i,T_j)) = (P,\{S_{1,i},...,S_{m,i}\},(T'))$.

For example, an entity type is created in a schema at a precise position, then it is moved a first time, a second time, and finally suppressed. Globally these actions bring nothing but noise to the history, so it can be cleaned. If the trace of the first move is deleted, different actions may need to be performed according to the purpose of the history. In a concise history, only the new positions of the entity type need to be recorded with the creation and movement actions: the entity type is created in position (x_0,y_0), then moved in position (x_1, y_1), and moved again in position (x_2,y_2); in that case, the history is still valid after the removal of the first move. In an extended history designed for undoing, more information is needed, it is necessary to know what the original position of the object was before the move to be able to put it back in that position, so the log file contains the creation at position (x_0,y_0), a move from (x_0,y_0) to (x_1,y_1), and a move from (x_1,y_1) to (x_2,y_2). In that case, the removal of the trace of the first move makes the log file invalid since the object placed in (x_0,y_0) is suddenly supposed to be placed in (x_1,y_1). To make it valid again, the trace of the creation must be modified in either of the following ways:

- merge the creation and the first move and suppose the entity type is created in (x_1,y_1)
- merge the first and the second moves, resulting in a single move from (x_0,y_0) to (x_2,y_2).

A particular case arise often where T' is the identical function ($T'=T_j o T_i =id$). That is to say that T_j undoes T_i. Since the identical function is neutral and useless, it can be suppressed.

More generally, the method above can be extended to the replacement of more than two functions by a single one:

If $L = (P,\{S_1,...,S_m\},(T_1,T_2,...,T_t)) \in \mathcal{L}$ contains several entries, T_{i_1} , T_{i_2},..., T_{i_j}, $1\leq i_1<i_2<...<i_j\leq t$, such that there exists $T'=T_{i_j} o...o T_{i_2} o T_{i_1}$, and such that for all T_k, $i_1<k<i_j$, $k\neq i_1,i_2,...,i_j$, T_{i_1} and T_k, T_{i_2} and T_k,..., T_{i_j} and T_k are independent, then L can be modified by removing T_{i_1}, T_{i_2},..., $T_{i_{j-1}}$ and replacing T_{i_j} by T'.

The particular case where $T'=T_{i_j} o...o T_{i_2} o T_{i_1}=id$ may happen as well.

7.5.3. Engineering process history cleaning

Removing products, processes, decisions and input/output/update relationships, is the way to clean engineering process histories. But it has to be done in a way that preserves the coherency of the remaining of the history. In other words, cleaning the history G of an engineering process means identifying a graph excerpt G' which is useless and calculating G – G' when the conditions of validity for this operation are satisfied.

Practically, what can be easily removed from an engineering process graph can be determined with the following algorithm:

1. *Initialisation*. Let $G=(P,V,E)$ be the graph to clean and let $G'=(P,V',E')=(P,\emptyset,\emptyset)$ be an empty graph excerpt. Let us define $I \subseteq V$ the set of input products of G, $O \subseteq V$ the set of output products of G, $U \subseteq V$ the set of products updated by G, and $N = (V \cap \mathcal{P}) \setminus (I \cup O \cup U)$ the set of internal products of G.
2. *Dead branches*. A **dead branch** is a branch of the graph that ends up in products of N that are neither used in input by another sub-process, nor rejected by a decision. they are useless and can be suppressed. So, all the processes, the products of N, the decisions, the edges between them all forming a dead branch, and the edges that link the branch to the remaining of the graph form a graph excerpt that can be copied to G'. To find a minimal history, all the dead branches are copied into G'.

   The detection of dead branches can be performed by applying the following rules:

   Removing unused products:

   $\forall\ r \in N \setminus G'$ such that $(\neg\exists\ p \in \mathcal{L} \cup \mathcal{G},\ p \notin G' : (r,p) \in E) \wedge (\neg\exists\ d \in \mathcal{D},\ d \notin G' : (d,r) \in E)$:
   $V' \leftarrow V' \cup \{r\}$,
   $\forall\ p \in \mathcal{L} \cup \mathcal{G} \cup \mathcal{D}$ such that $(\underline{p,r}) \in E$: $E' \leftarrow E' \cup \{(\underline{p,r})\}$,
   $\forall\ p \in \mathcal{L} \cup \mathcal{G} \cup \mathcal{D}$ such that $(p,r) \in E$: $E' \leftarrow E' \cup \{(p,r)\}$,
   $\forall\ d \in \mathcal{D}$ such that $(r,d) \in E$: $E' \leftarrow E' \cup \{(r,d)\}$

   Removing processes that generate no products and modify no products used afterward:

   $\forall\ p \in V \setminus G'$, $p \in \mathcal{L} \cup \mathcal{G}$ such that $\forall\ r \in \mathit{Prod}(p)$, $r \notin G'$ either (1) $r \notin V$, (2) $r \in V \wedge (r,p) \in E$, (3) $r \in N \wedge (\underline{r,p}) \in E$:
   $V' \leftarrow V' \cup \{p\}$,
   $\forall\ r \in V$, $r \in \mathcal{P}$ such that $(p,r) \in E$, $E' \leftarrow E' \cup \{(p,r)\}$,
   $\forall\ r \in V$, $r \in \mathcal{P}$ such that $(r,p) \in E$: $E' \leftarrow E' \cup \{(r,p)\}$,
   $\forall\ r \in V$, $r \in \mathcal{P}$ such that $(\underline{r,p}) \in E$: $E' \leftarrow E' \cup \{(\underline{r,p})\}$

   Removing decisions with no selected product:

   $\forall\ d \in V \setminus G'$, $d \in \mathcal{D}$ such that $\neg\exists\ r \in V$, $r \in \mathcal{P}$, $(d,r) \in E$:
   $V' \leftarrow V' \cup \{d\}$,
   $\forall\ r \in V$, $r \in \mathcal{P}$ such that $(r,d) \in E$, $E' \leftarrow E' \cup \{(r,d)\}$

   Do these three steps again until no constructs are added to G'.

   In practice, if some dead branches contain interesting information that should not be lost, some of the previous rules can be bypassed.
3. *Decisions*. Along the paths that were not copied into G', decisions can still be found. All the dead branches or some of them are copied into G'. Since these decisions simply show that all products remaining in the scope are selected, there is no use to keep them.

$\forall d \in V, d \in \mathcal{D}, \forall r \in V, r \in \mathcal{P}, (d,r) \in E$:
$V' \leftarrow V' \cup \{d\}$,
$E' \leftarrow E' \cup \{(r,d)\}$,
$E' \leftarrow E' \cup \{(d,r)\}$

1. *Termination*. The cleaned engineering process graph may be obtained by computing
$G \leftarrow G - G'$

7.6. History flattening

A **flat history**, by opposition to the structured histories studied so far, is made up of a single engineering process history whose graph only contains primitive process histories, decisions and products, all the other engineering processes being discarded. Flat histories are very interesting for several reasons:

- They are methodology neutral: they do not reflect any structured method, they are simply a series of elementary actions with no strategical structure.
- They are the simplest kind of histories, every CASE environment, even the simplest, can work with this kind of histories.
- Their reuse is much simpler than the reuse of a structured history because primitive process history transformations suffice to do the job.

Let us consider a structured history tree $H = (V_H,E_H) \in \mathcal{H}$. The root node of H is an engineering process graph $G_{root} = (P_{root},V_{root},E_{root})$. Let us assume H also contains *n* other engineering processes. The goal of the flattening transformation is to find a history

$H_F = (V_F,E_F)$

such that

1. $H_F \equiv H$
2. $V_F = \{G_{flat}\} \cup (V_H \cap \mathcal{L})$ where G_{flat} is the root node of H_F
3. $E_F = \{(G_{flat},v_j) | v_j \in V_H \cap \mathcal{L} \wedge \exists k, 1 \le k \le n, (G_k,v_j) \in E_H\}$

The principle is simple: initialise $G_{flat}=(V_{flat},E_{flat})$ as a copy of G_{root}, choose a graph node G_i in G_{flat}, merge G_{flat} and G_i, and do that again with all graph nodes in G_{flat}.

$G_{flat} = G_{root}$
while $V_{flat} \cap \mathcal{G} \neq \emptyset$ do
  choose one node $G=(P,V,E) \in V_{flat} \cap \mathcal{G}$
  $G_{flat} \leftarrow G_{flat} \oplus G$
compute H_F as in Chapter 6, Section 6.3.6 with $V_L = V_{flat} \cap \mathcal{L}$ and $V_G = \{G_{flat}\}$

It can be proven that this algorithm generates a history that fulfils the goals above:

Let $G_1, G_2,..., G_n$ denote the n engineering process graph nodes treated by the above algorithm in the order they are chosen, defined as $G_i=(P_i,V_i,E_i)$, $1 \le i \le n$. Let $G_0 = (V_0,E_0)$ denote the initial state of $G_{flat}=(V_{flat},E_{flat})$.

Before the first execution of the loop body,

$G_{flat} = G_0,\ V_{flat} = V_0,\ E_{flat} = E_0$

If we suppose that, after the kth execution of the loop body,

$G_{flat} = G_0 \oplus G_1 \oplus ... \oplus G_k$

$V_{flat} = \cup_{0 \le i \le k} V_i \setminus \{G_1, G_2, ..., G_k\}$

$E_{flat} = \cup_{1 \le i \le k} E_i \setminus \{(v_i, v_j) | \ v_i \in \{G_1, G_2, ..., G_k\} \vee v_j \in \{G_1, G_2, ..., G_k\}\}$

then, after the $(k+1)^{th}$ execution of the loop body,

$G_{flat} = G_0 \oplus G_1 \oplus ... \oplus G_{k+1}$

$V_{flat} = (\cup_{0 \le i \le k} V_i \setminus \{G_1, G_2, ..., G_k\}) \cup V_{k+1} \setminus \{G_{k+1}\} = \cup_{0 \le i \le k+1} V_i \setminus \{G_1, G_2, ..., G_{k+1}\}$

$$\begin{aligned} E_{flat} &= (\cup_{1 \le i \le k} E_i \setminus \{(v_i, v_j) | \ v_i \in \{G_1, G_2, ..., G_k\} \vee v_j \in \{G_1, G_2, ..., G_k\}\}) \\ &\quad \setminus \{(v_i, v_j) | (v_i, v_j) \in E_{flat} \wedge (v_i = G_{k+1} \vee v_j = G_{k+1})\}) \cup E_{k+1} \\ &= \cup_{1 \le i \le k+1} E_i \setminus \{(v_i, v_j) | \ v_i \in \{G_1, G_2, ..., G_{k+1}\} \vee v_j \in \{G_1, G_2, ..., G_{k+1}\}\} \end{aligned}$$

So, at the end of the loop,

$G_{flat} = G_0 \oplus G_1 \oplus ... \oplus G_n$

$V_{flat} = \cup_{0 \le i \le n} V_i \setminus \{G_1, G_2, ..., G_n\}$

$E_{flat} = \cup_{1 \le i \le n} E_i \setminus \{(v_i, v_j) | \ v_i \in \{G_1, G_2, ..., G_n\} \vee v_j \in \{G_1, G_2, ..., G_n\}\}$

and $V_{flat} \cap \mathcal{G} = \emptyset$

By a property of the $\oplus$ operator,

$G_{root} = G_0 \equiv G_0 \oplus G_1 \equiv ... \equiv G_0 \oplus G_1 \oplus ... \oplus G_n = G_{flat}$.

Since a history can be assimilated to its root engineering process,

$H \equiv G_{root} \equiv G_{flat} \equiv H_F \quad \Rightarrow$ (1) is proved.

When H_F is computed as in Chapter 6, Section 6.3.6 with $V_L = V_{flat} \cap \mathcal{L}$ and $V_G = \{G_{flat}\}$,

$$\begin{aligned} V_F &= V_L \cup V_G = (V_{flat} \cap \mathcal{L}) \cup \{G_{flat}\} \\ &= ((\cup_{i0 \le i \le n} V_i \setminus \{G_1, G_2, ..., G_n\}) \cap \mathcal{L}) \cup \{G_{flat}\} \\ &= ((\cup_{0 \le i \le n} V_i) \cap \mathcal{L}) \cup \{G_{flat}\} \end{aligned}$$

Since, by definition, $V_H = \{G_0\} \cup ((\cup_{0 \le i \le n} V_i) \cap (\mathcal{L} \cup \mathcal{G}))$, or, in other words, V_H contains the root engineering process node and all the nodes of all the engineering process graphs,

$V_H \cap \mathcal{L} = (\{G_0\} \cap \mathcal{L}) \cup ((\cup_{0 \le i \le n} V_i) \cap (\mathcal{L} \cup \mathcal{G}) \cap \mathcal{L}) = \emptyset \cup ((\cup_{0 \le i \le n} V_i) \cap \mathcal{L})$

and $V_F = (V_H \cap \mathcal{L}) \cup \{G_{flat}\}. \quad \Rightarrow$ (2) is proved.

Moreover,

$$\begin{aligned} E_F &= \{(v_i, v_j) | \ v_i = (P_i, V_i, E_i) \in V_G \wedge v_j \in V_F \wedge v_j \in V_i\} \\ &= \{(v_i, v_j) | \ v_i = (P_i, V_i, E_i) \in \{G_{flat}\} \wedge v_j \in ((V_H \cap \mathcal{L}) \cup \{G_{flat}\}) \wedge v_j \in V_i\} \\ &= \{(G_{flat}, v_j) | \ v_j \in V_H \cap \mathcal{L} \wedge v_j \in V_{flat}\} \end{aligned}$$

because there are no self-loops in a graph, hence $v_j \neq G_{flat}$.

Since $V_{flat} = \cup_{0 \le i \le n} V_i \setminus \{G_1, G_2, ..., G_n\}$, if $v_j \in V_{flat}$, then $\exists$ k, $0 \le k \le n$ such that $v_j \in V_k$.

So, there is a graph $G_k = (P_k, V_k, E_k)$ in H for which $v_j \in V_k$. During the construction of H, G_k and v_j gave rise to a couple (G_k, v_j) in E_H. So,

$E_F = \{(G_{flat}, v_j) | \ v_j \in V_H \cap \mathcal{L} \wedge \exists k, 1 \le k \le n, (G_k, v_j) \in E_H\} \quad \Rightarrow$ (3) is proved.

Finally, this algorithm surely ends because n is finite.

The algorithm is then proved to be correct.

For instance, Figure 7.8 results from flattening the history of Figure 6.8.

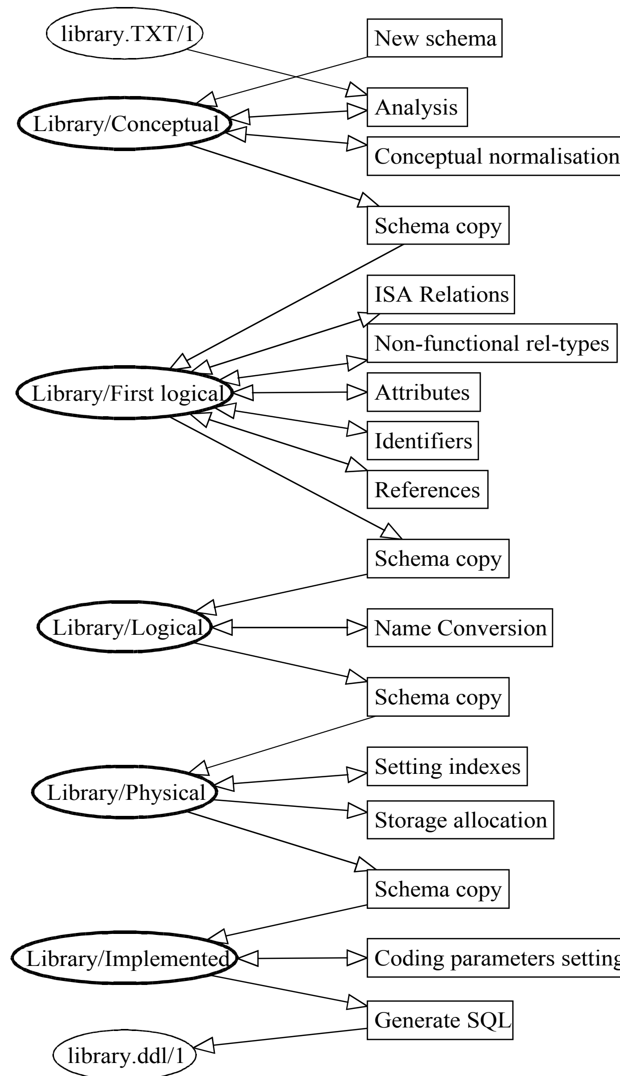


Figure 7.8 The flattening of the Library project

7.7. History inversion

Inverting a history H of a process P (whatever its kind) is generating a new history H' of a pseudo process that can be replayed in order to undo P. This is useful for many reasons, including undo, design process recovery,...

Chapter 6 showed how a product transformation can be inverted, assuming sufficient information are available in the history. This section shows how a primitive process history can be inverted using this principle. Inverting an engineering process history is a much more complex task. Indeed, it implies reverting strategical decisions. If this is only possible, it is out of the scope of this thesis. So, without loss of generality because a history can be flattened, this section only deals with extended primitive process histories.

Let $L=(P,\{S_1,...,S_m\},(T_1,...,T_n))\} \in \mathcal{L}$ be the extended primitive process history to invert, and let $\{S_1',...,S_{m'}'\}$ be the set of products resulting from the execution of P.

In order to simplify expressions, the following simplified notation will be adopted to show that each T_i, $1 \leq i \leq n$, is applied to some constructs of one of the products $S_1,...,S_m$:

$$T_i(S_1,...,S_m) = T_i(C),\ C \in S_j,\ 1 \leq j \leq m,\ 1 \leq i \leq n$$

Since the extended history was constructed in order for each T_i, $1 \leq i \leq n$, to be reversible,

$$\forall\ i,\ 1 \leq i \leq n,\ \exists\ T_i^{-1} \text{ such that } T_i^{-1}(T_i(C))=C$$

So,

$$T_1^{-1}(T_2^{-1}(\ldots T_{n-1}^{-1}(T_n^{-1}(T_n(T_{n-1}(\ldots T_2(T_1(S_1,\ldots,S_m))..))))\ldots)) = Id(S_1,\ldots,S_m)$$

where Id(...) is the identical function.

The history of this process is:

$$\begin{aligned} L_{Id} &= (Id,\{S_1,\ldots,S_m\},(T_1,\ldots,T_n,T_n^{-1},\ldots,T_1^{-1})) \\ &= (P,\{S_1,\ldots,S_m\},(T_1,\ldots,T_n)) + (P',\{S_1',\ldots,S_m'\},(T_n^{-1},\ldots,T_1^{-1})) \\ &= L + L' \end{aligned}$$

where L' is the history of a pseudo-process P' (that is to say a process that could exist but which was never really performed).

This new history L' is in fact the invert of history L. So, in practice, inverting a history is simply building a new history with the reverse of the transformations of the original history inserted in reverse order.

An example of history inversion is shown in the second case study in chapter 11, and in appendix F.

[HAINAUT,96b] shows in more detail how history inversion is applied in design recovery.

Part 3

In practice

Chapter 8

Method design: basic elements

MDL is a procedural, non-deterministic, language. Consequently, the traditional deterministic methodology for writing procedural programs does not apply to MDL. This chapter will introduce the main differences between deterministic and non-deterministic paradigms and present a few basic methodological elements for the new technology.

Writing a traditional imperative computer program is a complex task that requires a lot of knowledge, not only of the syntax and semantics of the chosen computer language, but also of algorithmic notions and programming paradigms. If a program is not well written, it might not provide the correct results, or it can simply end prematurely in a blocked state (endless loops for instance). To design a good method is even more complicated. Not only the result is important, but the way to reach it is important too. Of course, well-structured and clean traditional programs are preferable to dirty programs for the ease of maintenance, but the computer itself does not care about the programming style. At the contrary, a method is designed to be followed by human beings who need clear directives. If a method does not suit analysts way of working, it will simply be abandoned. So the method engineer must have one more goal in mind during his or her designs: the acceptability of the method, not only for the quality of its results and for the ease of use of its interface, but also for the ease of understanding and following the algorithms.

To write a good method is such a complex task that the subject deserves a separate thesis. This chapter will just examine a few basic elements and raise some problems that have to be taken into account by every method designer, focusing attention on the fact that a method is mainly non-deterministic.

In a first part, basic elements about how to structure product models will be presented. In a second time, a few facts about product types will be stated. Finally, process types will be studied, underlying what makes a method intrinsically different from a computer program.

8.1. Product model declarations

Product models are very important because the whole method is based on them. A good identification of the required product models and a correct declaration of them are the key-stone of the method.

The very first step the method engineer who designs a new method should perform is to model the products the database engineer will receive and the products that will have to be generated. All the intermediate products that will be useful during the project, even if they are not aimed at being divulged, also need some precise models for the help of database engineers. But these intermediate product models will only show their usefulness during the definition of the process types, so they can be defined only at that moment, during the definition of the needs for a sub-process definition.

System requirement reports, COBOL programs, Java programs, forms, screenshots and all other texts can be modelled very easily with a simple file extension.

A database schema model is made of two parts: the concepts and the constraints. The concepts, as defined in Chapter 3, is a simple glossary that establishes a correspondence between the terms used in the GER model and the terms which are particular to the model the method engineer is defining; this is a rather simple task. The definition of the constraints is more complex and deserves a good understanding of the model to define, a good understanding of the predicative constraint language, and the awareness of the level of help the method engineer wants to provide to the database engineer. The understanding of the model to define and of the predicative constraint language sounds natural, but the awareness of the database engineer needs is easily underestimated, leading to unusable model definitions.

The usability of a validation constraint lies in the fact that it can often be expressed in several ways:

- A same constraint can sometimes be expressed on different concepts. For instance, the constraint “MIN_CARD_of_ROLE(1 N)” stating that every role should be mandatory

means the same as the constraint “OPT_ROLE_per_ET(0 0)” that states that no entity type should play an optional role. They are equivalent in the sense that each time a role invalidates the first one, it also invalidates the second one and conversely. But, once they are violated, they report different information: the first rule provides the culprit role, while the second one only reports the name of the entity type that plays the incorrect role; if this entity type plays several roles, this information is less precise.

- Several constraints can be grouped in a single rule or they can remain separated in several rules. For example, to state that all attributes of a schema have to be atomic and single-valued, like in an SQL table, either the two simple constraints “SUB_ATT_per_ATT(0 0)” and “MAX_CARD_of_ATT(1 1)” can be used, or a combination of them in a single rule “SUB_ATT_per_ATT(0 0) and MAX_CARD_of_ATT(1 1)”. The first solution has the advantage that each rule returns its own list of problematic attributes, so it is clear that the attributes in the first list are compound and that attributes in the second list are multi-valued. The second solution returns a single list of problematic attributes without distinction. But the second solution can also be useful since the integrated list enumerates each problematic attribute only once, even those that cumulate both problems.
- The content of the diagnosis message is of great importance too. Indeed, if the rule itself is rather easily readable by the method engineer, it may prove to be hardly understandable by a database engineer not trained to it. The diagnosis messages should translate clearly the meaning of the rule in a human native language. It may also suggest a solution to solve the problem. For instance, the message “The attribute &NAME is compound, it should be disaggregated” is preferable to the message “rule SUB_ATT_per_ATT(0 0) violated by &NAME.”

8.2. Product type declarations

Product types can be declared locally to a process type or globally to all process types. Similarly, in traditional programming languages like Pascal, C or Fortran, variables can also be declared globally to the whole program or locally to a procedure[14]. But the comparison does not hold further.

In imperative programming languages, variables can either be of a given type or be a pointer (or a reference) to a memory location of a given type. When a procedure ends, its local variables are destroyed. This means that, if not copied to output parameters, the content of the non-pointer variables is lost and pointers to memory locations are lost too; non-freed memory locations become unreachable and unavailable.

When using an MDL method, the memory of the system is the history. Since the history keeps everything, products of the local types cannot be destroyed when a process ends. They simply will not be available anymore for the following processes of other types (except if they are of an output type) but they will still be accessible to who wants to read the history.

With imperative programming languages, it is often recommended to declare as much variables as possible locally, passing them from procedure to procedure using parameters, and to use global declarations for variables that are used by all procedures or which are so big, such as large arrays, that passing them in parameters costs too much in processing time or memory use. When using an MDL method, since only references to products in the history are passed, the problem of size does not exist, so global product types should only be used for products that must be accessible throughout the whole projects.

14 Or to a function.

When using local product types, the method engineer should pay great attention to the cardinality constraints. Indeed, a problematic situation can arise where product types cannot match, even if they are of the same model. Let us examine the pattern shown in Figure 8.1. In this method chunk, process type B requires the use of process type A with product of type Q. But according to the declaration, there can be 1, 2 or more products of type Q, while process type A only accepts one product in input. If the strategy of process type Q is designed so that only one product of type Q can exist at the time of performing a process of type P, there is no problem (the [1-N] cardinality constraint being justified by the fact that new products of type Q can be created later during B). But if several products of type Q exist when a process of type A has to be performed, no process of type A can be started. If this sub-process use is in the body of a *one*, a *some* or an *each* structure, the database engineer can simply follow another branch of the structure. But if there is no alternative to this sub-process, the performance of the process of type B simply ends in a deadlock. The method has to be corrected.

```
process A
    ...
    input P[1-1] : T
    ...
end-process
process B
    ...
    intern Q[1-N] : T
    ...
    strategy
        ...
        do A(Q)
        ...
end-process
```

Figure 8.1 A problematic sub-process use

8.3. Process type declarations

To write a process type could be seen as similar to writing a procedure in an imperative programming language since the MDL language is based on the same basic control structures. This should be true if a method was not aimed at being used by a human being because computers just execute what is ordered to them without trying to understand what they are doing, and without complaining that they would prefer to do the same thing another way that would need less efforts or that they already did the exactly same action several times before.

Without willing to be exhaustive, we will now examine several situations that should be seen with a different point of view by a traditional imperative language programmer and by a method engineer.

8.3.1. Loops

To design a strategy that begins by the collect of interview reports, using what was learned from traditional imperative languages programming, one would surely write one of the two following MDL strategy chunks containing a *while* and *a repeat-until* structures (graphically shown in Figure 8.2 and Figure 8.3 respectively):

1. while (ask "Do you want to collect a new interview report?") repeat
    new (InterviewReport)
   end-repeat

2. repeat
    new (InterviewReport)
   end-repeat until (ask "Have you finished collecting interview reports?")

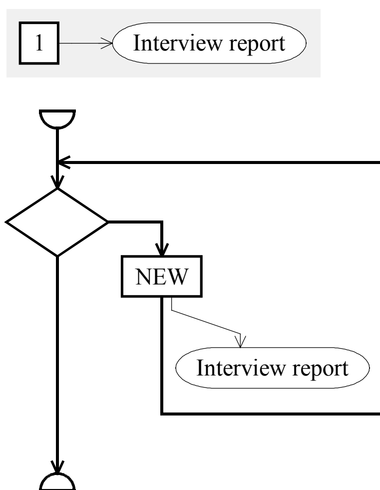


Figure 8.2 History chunk 1

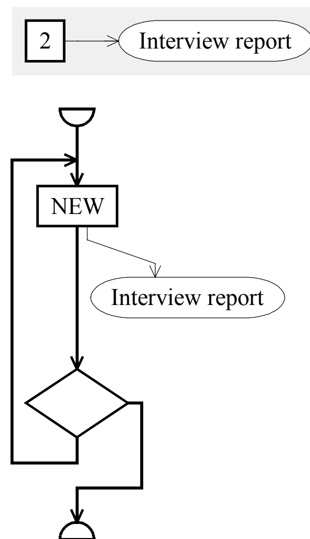


Figure 8.3 Strategy chunk 2

They both allow the users to collect as many interview reports as they want. In fact, the first one allows users not to collect any report at all, while the second one forces the users to collect at least one report. For them to be really equivalent, the first one can be modified as the chunk 1' below (see Figure 8.4) to force that at least one report is collected.

1'. while (count-less(InterviewReport,1) or
        ask "Do you want to collect a new interview report?") repeat
    new (InterviewReport)
   end-repeat

Or the second one can be changed as follows (strategy chunk 2', shown in Figure 8.5) for the users to be able not to collect a single report.

2'. if (ask "Do you want to read interview reports?") then
    repeat
      new (InterviewReport)
    end-repeat until (ask "Have you finished collecting interview reports?")
   end-if

But these strategy chunks are not aimed at being used by computers, but rather by human beings, which implies several differences:

- Human beings are able to have a glance at an algorithm before executing it. So they are able to understand what they have to do and what they need before doing it. Computers are only capable of starting to execute directly, step by step, and to stop when a problem occurs. Human beings are able to forecast, computers are not.
- Human beings are lazy, they do not like to work when it is not necessary, so they will not start a process if they can foresee a problem by looking at the algorithm. Computers do not care and will do the job until they reach the problem they could not foresee.

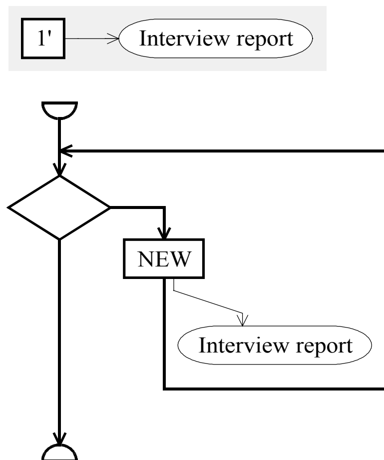


Figure 8.4 History chunk 1'

It looks like History chunk 1, only the condition of the loop differs.

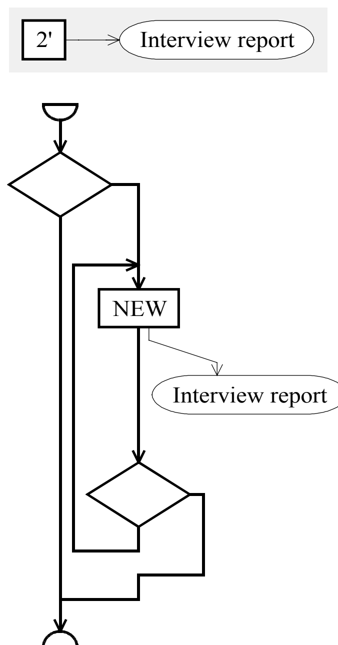


Figure 8.5 History chunk 2'

- Human beings like simplicity, computers do not care about that. Humans prefer simpler structures such as 1 and 2 above, rather than 1' or 2'. 1 and 2 are more readable, therefore easier to understand.
- Human beings are able to think and to take intelligent decisions by themselves.

So, if an analyst encounters a strategy containing the chunk 1, he or she will see that it will not be possible to go further than the collecting loop if no interview report is collected and he or she will certainly not begin to follow the strategy. So, in practice, even if it is not mathematically correct, it can be said that chunks 1 and 2 are equivalent. But, among them, it is difficult to tell which one is the best. In fact, some people will prefer the first one where the question is asked before collecting each report, others will prefer the second one where the question is asked after each report is collected. But, since human beings are lazy, They will find annoying, when a lot of interview reports have to be collected, to answer the same question again and again until the last report is collected. So, finally, the strategy chunk most people will prefer is the third one (shown in Figure 8.6):

```
3. repeat
       new (InterviewReport)
   end-repeat
```

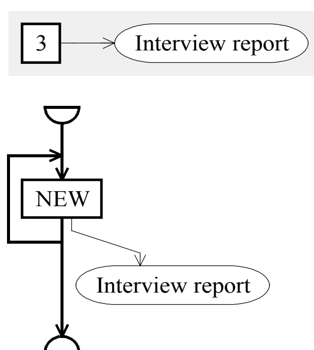


Figure 8.6 History chunk 3

It allows people to do exactly the same: to collect at least one interview report, and to stop whenever they want without the need to answer the same question several times.

8.3.2. Sequences and *each* structures

Computer programs as well as methods often require several actions to be performed just one time. Let *A1* and *A2* be either two program instructions or two process types. If *A1* and *A2* both modify the same resources (variables, memory location, products,...), or if only one of them modifies a resource used by the other, they have to be performed in the correct order, namely within a sequence. But both *A1* and *A2* may have to use or modify different resources. In this case, *A1* and *A2* will be said independent. They can still be used within a sequence, but they can be swapped without impact on the final result. Computers need a precise description of what they have to do. So it is the role of the programmer to decide which of *A1* or *A2* will come first in the sequence. But human beings are able to decide by themselves what they prefer to do first, so the method designer should leave to the final user the freedom of the choice. The *each* keyword can be used to force the analysts to perform all the processes in the order of their choice.

More generally, when processes of several types have to be performed, they can be grouped, all the non-independent process types in the same group, two non-independent process types in the same group. All the process types within each group can be ordered in sequence and all the sequences can be presented in parallel to the end-user within an *each* control structure. For example, if *P1*, *P2*, *P3* and *P4* are four process types, *P1* generating a product of a type used in input by *P2*, *P3* and *P4* updating a same product type, and *P1*, *P2* being individually independent from *P3* and *P4*, the following strategy chunk, graphically shown in Figure 8.7, is certainly the best way to model the situation:

```
each
    sequence
        P1;
        P2
    end-sequence;
    sequence
        P3;
        P4
    end-sequence
end-each
```

With a traditional programming language, one of the six following sequences would have been chosen for the exemple: *P1-P2-P3-P4* (Figure 8.8), *P1-P3-P2-P4*, *P1-P3-P4-P2*, *P3-P4-P1-P2*, *P3-P1-P4-P2*, or *P3-P1-P2-P4*. They give the same results, but the algorithm is less readable and they impose more constraints to the final database engineer.

8.3.3. Sub-process use

Figure 8.9[15] shows a strategy chunk that creates a new blank schema and that uses a sub-process, *Update*, which updates the new product, which fills it. In Figure 8.10, it is the *output* sub-process itself that creates the new product before filling it. The two situations, on a strictly theoretical point of view, will provide the same results. A machine would execute them indifferently. But they both bring a different perception of the problem to a human being: the first method gives a greater importance to the *New* primitive process, the fact of creating a new schema is strategically as important as filling it, while the *New* primitive process is a simple technical act in the second method.

15 An expanded style of drawing shows both a process and a sub-process on the same view.

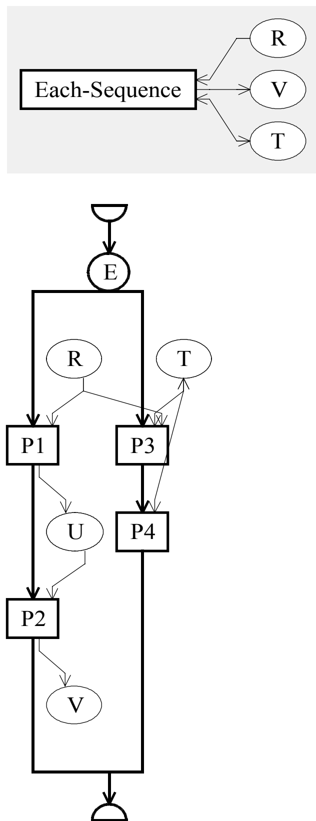


Figure 8.7 A combination of each and sequences

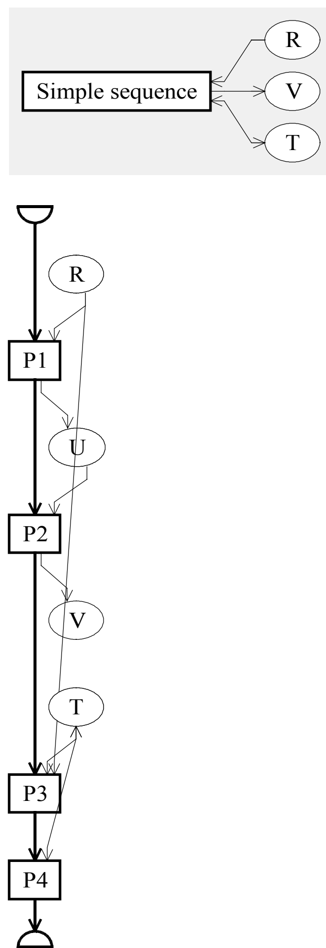


Figure 8.8 A simple sequence

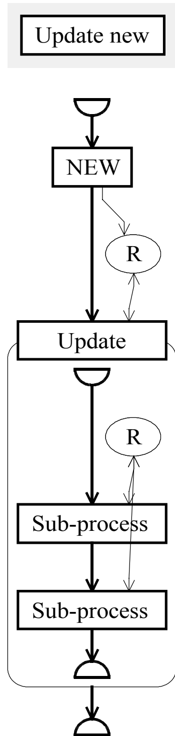


Figure 8.9 A method chunk updating a blank product

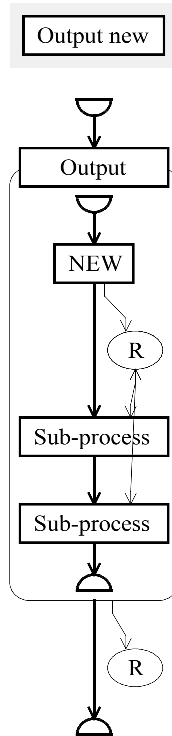


Figure 8.10 A method chunk creating a new product

When programming with traditional imperative languages, a similar situation is the initialisation of a pointer variable by allocating memory and the initialisation of the allocated memory. The choice between splitting both operations in a procedure and a sub-procedure or grouping both of them in a same procedure will generally be induced by the number of times they have to be performed, and the diversity of situations in which they have to be performed: if five procedures need to create exactly the same data structure, a situation similar to the second one is certainly the best choice; if the five procedures need similar data structures of personal size, a situation similar to the first one will certainly be a better choice. But this is simply a technical choice which has no impact on the final result, the persons who will use the program will not know and will even not bother to know how the program works.

When developing a method, technical details similar to those above can have to be taken into account, but the perception problem that does not exist with programming will generally have a great importance.

By writing two different strategies to obtain a same final result[16], the method engineer can also allow or disallow some possibilities. In Figure 8.11, the engineer can update the products of type *R* by performing the sub-process. In Figure 8.12, the engineer has to copy the products of *R* before updating the copies. In the second case, the engineer has the possibility to make several copies of each product before updating them according to various hypotheses, then to choose the best solution. In the first case, it is more complicated: the engineer can make several draft copies of the products by himself and update each draft copy according to an hypothesis, but, when he or she has taken the decision of the best solution, the updates must be performed again to the original products, possibly by replaying the history of the best draft.

Another difference between the two situations is the possibility, when browsing through the history for documentation, to watch at the original schema more easily in the second case since it appears unmodified in the history.

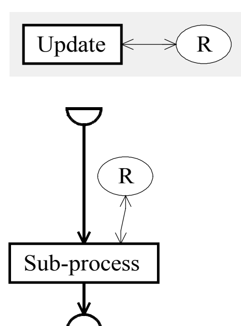


Figure 8.11 A method chunk updating a product

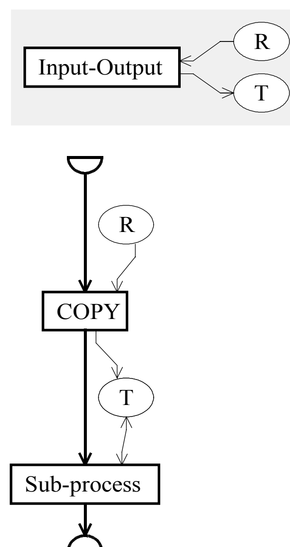


Figure 8.12 A method chunk generating a new product

16 Since the parameters are different in both situations, the process types that use these two ones have to be different, but they can be easily adapted for one or the other to reach the same result.

8.3.4. Degrees of freedom

One of the greatest differences between computers and human beings, as it already appeared above, is the ability for the human being to take decisions, to act freely. Without methodological help, a well-trained human being is capable to do a database engineering job entirely by himself or herself while a computer needs much more than a methodological help, it needs to be precisely guided step by step. Between these two extremes, a methodological help is aimed at guiding human beings while restraining their freedom of doing whatever they want. This section will show that the degree of freedom a method engineer can leave to database engineers is left to his or her own will; the MDL language contains a series of concepts that allow a great flexibility.

The same concern about freedom of action is also approached by [FAUSTMANN,99].

A. Primitive processes

Chapter 2 classifies primitive processes in four groups. The *basic automatic process types* are fully automated and leave absolutely no control to the method engineer nor to the analysts. The *configurable automatic process types* can only be configured by the method engineer developing a method. In fact, the first two kinds of primitive processes give no freedom of action to the database engineers because they are fully computer-oriented. The *user configurable automatic process types* allow the database engineers to act with a little bit more freedom at the initialisation of the process, but these engineers will still undergo its actual execution. Finally, the *manual process types* offer much more freedom to their users. When processes of this last kind are supported by a toolbox, the degree of freedom can even be regulated by the choice of the tools in the toolbox.

For instance, the following process of the second group automatically transforms all the functional rel-types of a schema S into referential attributes:

```
glbtrsf ( S, RT_into_REF ( ATT_per_RT (0 0)
                           and ROLE_per_RT (2 2)
                           and N_ROLE_per_RT (0 1))
```

Database engineers can also be allowed to do the same job manually, possibly leaving a few rel-types unchanged. By using the following toolbox, the database engineers have the freedom of choosing what they think needs to be transformed:

```
toolbox RT-to-REF
   title "RT-to-REF"
   add tf-RT-into-att
end-toolbox
```

Finally, the following extended toolbox also allows database engineers to perform the same transformations, but also to edit a little bit the schema in order to prepare it for the transformation when needed:

```
toolbox RT-to-REF
   title "RT-to-REF"
   add tf-RT-into-att
   add delete-attribute
   add delete-role
   add tf-RT-into-ET
end-toolbox
```

These three primitive process types allow database engineers to perform the same job, but give them different levels of responsibility and of freedom in their actions.

B. Sequence, each, one, some structures

It was shown above how the *each* structure can be used instead of a sequence, with independent sub-processes, to give more freedom to the user. The *one* structure has the same degree of freedom as the *each* structure since it imposes that one sub-process has to be performed too. But more freedom can be added to the *one* structure by adding an empty sequence alternative to allow the database engineer to choose one process or none:

```
one
    sub-process 1;
    sub-process 2;
    ...
    sequence  end-sequence
end-one
```

The *some* structure gives still more freedom to the database engineer who has the possibility to execute sub-processes of any number of enumerated types, from one to all, without regard to the selection order. By the adjunction of an empty sequence, the database engineer can be given the possibility to perform sub-processes from none to all enumerated types.

Finally, if the method engineer combines a *one*, a *some* or an *each* structure with a *repeat...end-repeat* loop, the database engineer will even be able to perform several processes of each type. For instance,

```
one
    repeat sub-process1 end-repeat;
    repeat sub-process2 end-repeat;
    ...
end-one
```

allows the database engineer to perform several times the same process, and

```
repeat
    one
        sub-process1;
        sub-process2;
        ...
        sequence  end-sequence
    end-one
end-repeat
```

allows him or her to perform processes of any number of types, any number of times (including none), in any order. The freedom of action is almost total in this last case.

C. Sets

When the method shown in Figure 8.7 is used, several products of type *R* can be passed to *P1* at the same time, the number of these products being in a range defined in the product type definition. By default, when a new process of a given type starts, all the products of the required types are passed to the new process. During the sub-process, the database engineer is allowed to actually work with all the products or with only some of them. The freedom of action is large. For example, let us suppose a database engineer has five text files of type *InterviewReport*, named *ir1*,...,*ir5*, and let us assume *Conceptual* is a schema type. If the engineer encounters the following strategy chunk he or she can perform a first analysis process with *ir1* to generate a first conceptual schema, then perform a second analysis process with *ir3*, *ir4* and *ir5* to generate a second conceptual schema:

```
do Analysis(InterviewReport,Conceptual)
```

The method engineer can reduce this freedom with the *for* control structure. A first restriction lies in the use of the *for some* structure which forces the user to perform actions on some products one by one. For instance, if a user has to follow the following strategy chunk in the same context as above, he or she can decide to perform four processes of type *Analysis*, the first time with *ir1*, the second time with *ir3*, the third time with *ir5* and the fourth time with *ir4*, giving a total of four conceptual schemas:

```
for some IR in InterviewReport do
   do Analysis(IR,Conceptual)
end-for
```

A further restriction is imposed by the *for one* and the *for each* structures because they impose that, respectively, exactly one of the products or all the products must be used one by one. For instance, still in the context above, the following strategy chunk forces the user to choose exactly one of the five interview reports and to treat this one only:

```
for one IR in InterviewReport do
   do Analysis(IR,Conceptual)
end-for
```

And the strategy chunk below makes mandatory the treatment of every interview report, one at a time:

```
for each IR in InterviewReport do
   do Analysis(IR,Conceptual)
end-for
```

D. Weak conditions

Even when using more standard structures, like *if...then...else*, *while*, *repeat...until*, a method engineer can give several degrees of freedom to the final user of the method. Those three control structures all need a condition. A condition is an expression as defined in Chapter 5. Three types of expressions were presented: formal and strict, formal non-strict, and non-formal.

Formal and strict expressions are the kind of expressions that can be found in every traditional procedural programming language. These expressions are expressed correctly and without ambiguities with a well-defined syntax and semantics and they can be evaluated in a deterministic way by a computer. Formal and strict conditions of the MDL language are formal expression based conditions that can be computed by the supporting CASE environment. Users of methods containing such conditions have no choice but to accept their result. They have no freedom.

Formal non-strict conditions are formal expressions too, so they can be computed by the supporting CASE environment, but the database engineers who are confronted to them have the possibility to accept the results or to refute them. In this case, the supporting CASE environment can be seen as a well-advised help that should wisely be followed. The freedom of the engineers to eventually accept or reject the advice is total.

Finally, non-formal conditions cannot be understood by the supporting CASE environment; only the engineers meeting them have the possibility and the total freedom to choose an answer. But they must answer anyway.

Chapter 9

CASE tool usage

In the previous chapters, the Method Description Language was presented as a means to bring methodological control to a CASE environment and a way of recording every action performed in the CASE environment was presented. This chapter will examine what is necessary and helpful to the method engineers to design methods and to database engineers to use the CASE environment using the method. Some requirements for the extension of the CASE tool will be presented, according to the GUI (graphical user interface) aspect, and a proposition of functions and dialogues will be suggested.

9.1. Requirements

The methodological engine intended to support method control in an existing CASE environment has to improve the usability of that CASE environment. To do this correctly, a good understanding of the requirements is necessary. Developing a method is itself a complex task that deserves its own environment, which must be specified and implemented too.

9.1.1. Method development environment requirements

The MDL language is an ASCII text based language. An MDL source has to be parsed and converted in a formalised proprietary file format to be reused by the supporting CASE tool.

Traditional procedural programming languages development environments can be classified in three main classes of tools that appeared along the computing history[17], with the idea of simplifying the developers' job:

- The simple command line interpreter tools. The programmer starts a text editor, types his or her program, saves it to a disk, quits the editor, starts a command line compiler, if errors occur, restarts the editor,...
- The integrated environments. A single tool contains the editor, the compiler, an error message window and a series of other helpful functions. A few mouse clicks suffice to perform most of the tasks instead of typing long commands at the terminal prompt as previously.
- The RAD (rapid application development) tools. These are integrated environments into which "intelligent" assistants are added. These assistants use easy-to-use dialogue boxes or graphical interfaces to help developers design some parts of their program by generating automatically tedious and error-prone chunks of program that correspond to the expressed desires of the developer.

To develop a method in the MDL language, a simple text editor and a simple MDL-to-CASE proprietary format translator, as in the first of the three cases above, suffice. But, for the ease of use of the method engineer, an integrated environment is much more interesting, specially for working with modern operating systems which are based on graphical interfaces.

The following main functions must be present in the development environment:

- a text editor for editing the MDL source texts
- an MDL-to-CASE environment proprietary format translator
- an error message window to help the engineer to "debug" the method
- a graphical browser to show the method designer the same algorithms as those that will be presented to the database engineer inside the CASE environment.

To make this environment evolve towards a RAD environment, the following functions can be added:

- an assistant to prepare the skeleton of a method
- an assistant to help the method engineer to design product models
- an assistant for the design of toolboxes
- a graphical assistant for the design of process types.

17 Without going back to the very first days of the computer age when programming was performed by soldering wires or by punching cards.

Method design surely deserves a RAD environment. Indeed, product models are lists of concepts and of constraints that can easily be chosen in a predefined list, and algorithms could easily be drawn graphically. But method design is a complex job that will only be performed by a few specialists rather than by as large a public as traditional programming. Even these specialists will only design a few methods, each method containing no more than a few tens of product models and process models. Designing such a complex tool is a costly activity for a rather small use. Moreover, it is of little interest in the framework of this thesis. So this direction will not be investigated any further.

Anyway, other traditional secondary functions can prove to be useful, either within a RAD or within a simple development environment:

- printing facilities for the MDL source and the graphical presentation of the method
- report generation
- copy function to the Windows clipboard or to other applications
- ...

9.1.2. CASE environment requirements

A method defined in the MDL language is aimed at supporting a CASE environment. Any methodology neutral database oriented CASE environment could be updated to support MDL methods. We will try to be general enough to cover any such existing or imaginable CASE environment, but we will particularly focus our attention to the DB-MAIN CASE environment presented in Chapter 1.

The improvement of the CASE environment is in fact a three parts goal:

- the CASE tool has to keep all its functions and their usability has to be unchanged
- database engineers using the CASE tool must not feel disappointed with the modified interface, so the methodological engine has to be the most transparent possible
- the methodological engine has to bring some help and some guidelines to database engineers.

To fulfil these goals, a few elements can be added to the CASE environment, some elements can be slightly updated, but nothing can be removed.

A. Method visualisation and browse

The first element to add to the CASE environment is the possibility to choose a method to follow in the project creation dialogue box. This possibility has to be optional because the user cannot be forced to follow a method, for small projects for instance.

When a new project is started, it is necessary for the database engineer to be able to see the method. A dedicated window will be added to the CASE environment. From now on, it will be called the **method window**. It will show engineering process strategies with the algorithmic presentation described in Chapter 4. This window will have to be dynamic, the user will be able to use it to browse through the whole method, and he or she will be able to select any element of the shown algorithm to examine it in more details. It must be possible to:

- see the properties of a product type, the model it is expressed in and the properties of this product model
- see the definition of a primitive process type; it can be a toolbox with the list of all its tools, a global transformation with its complete definition, the use of a built-in function, the use of an external process,...

- show the graphical representation of an engineering sub-process type
- go back to the previous view.

In all the cases, the descriptions that were included in the method definition should be easily readable.

B. Method driven activities

When a method is loaded into the CASE environment, it has to be used to guide database engineers and to allow them to do their job:

- The new method window should contain some distinctive signs that should clearly present the current state of the project to the user:
  - all process types which have instances currently running should be distinguishable
  - all process types which were already run should be distinguishable in another way
  - all process types that are ready to be performed should be signalled in a third way.
- The other parts of the CASE tool also have to be updated slightly to guide engineers:
  - product edition functions should only be enabled when they are part of a toolbox referenced by a primitive process type an instance of which is currently running, and with products concerned by this primitive process
  - when no primitive process is active, no product edition tool should be available
  - when a product is being edited, all the CASE environment interface elements (menus, dialogue boxes, messages,...) should use the correct terminology according to the *concepts* part of the model on which the product is based
  - products have to be validated when a process ends. Let us note that the products only need to be validated at the end of a process rather than in real time during the process, because the second solution could make the CASE environment unusable: for instance, if a schema model contains the constraint “ATT_per_ET (1 N)”[18]; the simple fact of creating an entity type will invalidate the schema, which will be valid again when attributes will be created, later.

On the contrary, when no method is selected during the project creation, the CASE tool should not be affected by the methodological engine which should be completely invisible. The CASE environment has to keep its methodology-neutral capability.

C. Several levels of constriction

It was shown in Chapter 8 that a method can be made very constricting, or, at the contrary, it can allow the database engineers a large degree of freedom of action. But a method engineer can make errors and produce a problematic method. This may lead to a blocked state during the use of the method. Database engineers cannot accept to be blocked in their work because of the bad method, they have to continue by themselves anyway. So the CASE environment itself needs several levels of constriction to the method:

- Strict use of the method. This is the preferred mode. By default, the CASE environment should automatically be in that mode when a new project is created using a method. The users should always use this mode and leave it only in case of problem.
- Permissive use of the method. This mode can be used to bypass some constraints imposed by a method, specially if these are blocking constraints. In this mode, the CASE environment will operate as if the method engineer had designed the method

18 This is a realistic constraint used in the first case study in Chapter 11.

with weak conditions only and every product types defined with a weak respect of their product model.

- No use of the method. The methodological engine is inactivated. Database engineers can still view the method in its window, but for documentation only. They are left to themselves to do the job and to organise the history manually. This mode should only be used in case of major problem in a method.

D. History recording

Recording the history is a major activity. It must be available either with or without a method and in all the constriction modes. So this function has to be designed independently of the methodological engine. Both the user interface and the methodological engine have to be adapted to be able to control the recording of histories.

The recording of primitive process log files should have the following characteristics:

- It can be enabled or disabled. Once enabled, the database engineers should be able to work without noticing the recording of every action they perform, or the methodological engine performs, on any product. This recording has to be automatic, complete and transparent.
- The recording has to be possible at different levels as defined in Chapter 6 (concise recording for replay, extended recording for undo or reverse engineering,...) according to the foreseen usage.
- Both the users and the methodological engine should be allowed to add bookmarks to facilitate the reuse.

Since there is no way, independently of the methodological engine, to record an engineering process graph automatically (taking design decisions is a manual activity), this task involves other requirements:

- It should offer the possibility to create new primitive processes and new engineering processes, to terminate these processes, and to take decisions, either conditions in control structures of the method or product version selections
- For the ease of use, the CASE environment can also allow the users to continue a process that would have been terminated prematurely.
- Since the whole intelligence of the project should be present in the engineering process graphs, it is important for the users to be able to browse through all the graphs, and it is important for them to be able to make a parallel with the method if one is present. So a history browser that works the same way as the method browser is a strong requirement.

Finally, the tree of all processes that can be computed automatically will be shown in its own read-only window.

E. History replay

The replay of a primitive process log file is a simple task database engineers like to perform in several ways:

- To replay a complete log file automatically on the provided schema.
- To replay in the same way a part of a log file comprised between two bookmarks.
- To replay step by step, in a controlled way, a log file or a part of it.

The replay of an engineering process is a much more difficult task since its meaning is user-dependent as explained in Chapter 7. So the CASE environment can only provide the tools

for recording and reading these process graphs. The user (either the method engineer or the database engineer) is the one who will have to design his or her own tools. The use of the built-in macro language or 4GL of the CASE environment (the Voyager 2 language in DB-MAIN) will be required. History evolution is such a task requiring some replay; the Ph. D. thesis [HICK,01] gives one vision of the problem and brings its own solution.

F. History transformation

In Chapter 6 a series of basic history transformations and some possible applications were defined. Database engineers can be interested at some times in some punctual transformations for a few improvements of the history, but they will surely be more interested by some particular applications that correspond to their particular needs.

The basic primitive process log file transformations (the delete, the concatenation and the replace operations) can easily be carried out by a simple text editor. But, since they involve the verification of several conditions which are tedious to check manually, an intelligent text editor or, better, a specific log file editor would be a better choice. This tool should:

- have basic edition functions
- recognise the log file syntax
- be able to recognise and to treat complete log file entries as atomic elements
- be able to automatically validate operations and ensure that the resulting log file is syntactically correct.

The delete, replace and merge operations on engineering process graphs are simple functions that can be implemented directly in the history window. The delete and the replace operations on history excerpts are more complex operations since a history excerpt can be a mix of log files and graphs, but these operations can be decomposed into simpler delete or replace operations on the components of the history excerpts, and be carried out as a series of delete and replace operations on log files and graphs.

The problem engineers will face when they try to use transformation operations on log files is their size. Indeed, a log file can be made up of several thousands of entries. So, applications such as history cleaning are not always as simple to perform as described in Chapter 6. For instance, two log entries which can be combined or removed can easily be detected if they are one next to the other among a few tenths of entries, but it will be much more arduous if they are separated by several tenths of other entries. Such applications really need some complex search and ordering assistants. But these assistants are application dependent and they would deserve a complete study which is out of the scope of this thesis.

9.2. HMI proposals

According to the requirements presented in the first part of this chapter a proposition of *graphical user interface* (GUI) for supporting the design and the use of a method can be examined. The design of a method being something new, a development environment needs to be build from scratch as presented in a first time. The method being parsed, it can be shown graphically and browsed by the users both in the method development environment and in the CASE environment, as shown in a second time. For the use of a method, since it was decided to adapt an existing CASE environment, a way to update the DB-MAIN CASE environment interface will be studied: tools to follow a method will be presented in third time and tools for recording the history will be shown in the fourth time. A few complementary tools for helping the analysts to use the CASE environment will then be presented, followed by tools to configure the CASE environment. Finally, tools for browsing and handling histories will be presented as well.

9.2.1. Method development environment

A method development environment, to respond to the requirements above, can be as simple as an elementary text editor with a compile function and a graphical viewer, or as complex as a complete RAD environment. A tool with an extensible architecture is surely the best solution. At the very first release, it will contain all the required basic functions. Along the time, it will be possible to add new functions as they become available.

Figure 9.1 shows a simple prototype of a basic environment. Two kinds of windows appear: the text editor window and the method viewer window. The main menu contains standard *File*, *Edit*, *Window* and *Help* menus plus a *Search* menu to help editing text files, a *View* menu for configuring the graphical viewer and an *MDL* menu with the compilation tool. This last tool parses the MDL text in the editor window, stores the method it represents in the internal repository[19], and opens a viewer window to show the graphical representation of the method. The *File* menu contains commands for loading and saving MDL texts, as well as a command for exporting the content of the internal repository to a DB-MAIN proprietary file format. The small toolbar contains standard shortcuts for loading and saving MDL texts, editing these texts and printing, as well as an icon (third from right) for executing the compiler. When compiling a method, if errors occur, a third kind of window should appear with the error messages.

To implement new functions in this basic environment, it suffices to create new windows or dialogue boxes for them and to add new menu items or even new sub-menus, possibly new shortcuts in the toolbar or new toolbars.

The editor window is a classical text editor. It can be the most simple one with just the few basic functions such as insert, delete, cut and paste. It can also be a more elaborated programming oriented editor with functions such as auto-indent or parenthesis match checking. It can even be a fully MDL-oriented editor with syntax highlighting.

The graphical viewer has to be specifically developed for the method algorithms described in Chapter 4. The following section is devoted to it.

The way of working of the MDL source parsing function will be presented in chapter 10 which is devoted to the internal aspects of the environments.

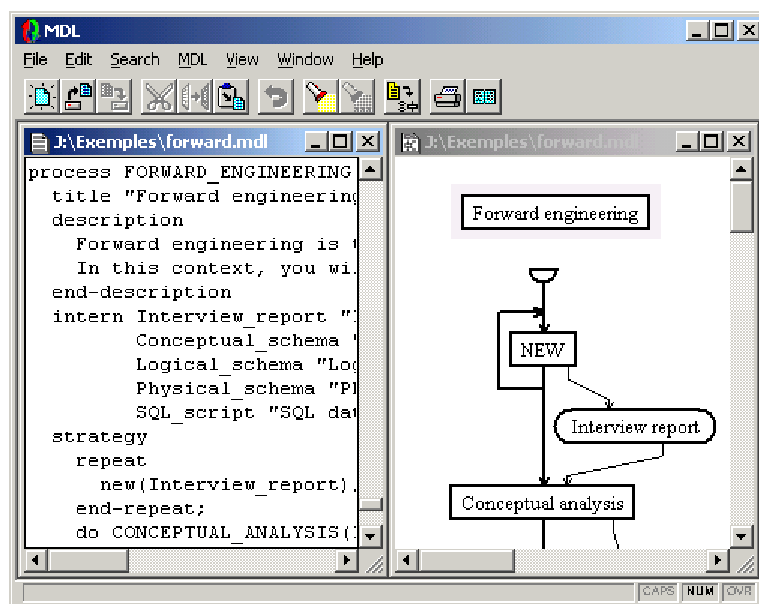


Figure 9.1 A simple method development environment

19 The internal repository will be presented in Chapter 10.

9.2.2. Method visualisation and browsing

The method visualisation window will contain the graphical representation of a method, or more precisely of one engineering process type strategy of the method, with hyperlink towards other engineering process strategies. This window can be implemented both in the method development environment for viewing the result of compilations and in the CASE environment itself for showing the current method.

When the window is created, it shows the strategy of the main process type of the method, that is the process type declared in the *perform* line of the *method* paragraph of the MDL method description. The strategy is presented with the algorithmic graphical formalism presented in Chapter 4, showing the sub-process types, products types, control structures, the control flow, the data flow, and the title, as in the example shown in Figure 9.1.

This window is active. Each element in it (process type, product type and diamond) is associated with a contextual menu which becomes visible when the element is clicked on with the right mouse button. These menus contain the following entries:

- A *properties* entry associated with a product type shows a dialogue box (Figure 9.2) with the name of the product type, the model it is based on, its *strong/weak* status, and its multiplicity. A *model* button allows the engineer to open a second dialogue box containing the properties of the product model: text model properties show the name, the list of associated file extensions, the grammar file, and the description as in the example shown in Figure 9.3; schema model properties show the name and list the concepts, the constraints and the description, as in Figure 9.4.
- A *properties* entry associated with a primitive process type opens a dialogue box as the one shown in Figure 9.5 when the primitive process type is of the manual type and guided by a toolbox. It shows the name and the list of all the tools included in the toolbox, as well as the toolbox description. A dialogue box like the one shown in Figure 9.6 is opened for all other kinds of primitive process types. The large text zone lists the complete primitive process type definition. This example shows a complete transformation script, a primitive process of a configurable automatic process type.

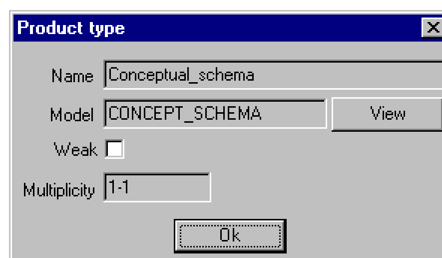


Figure 9.2 A product type properties dialogue box

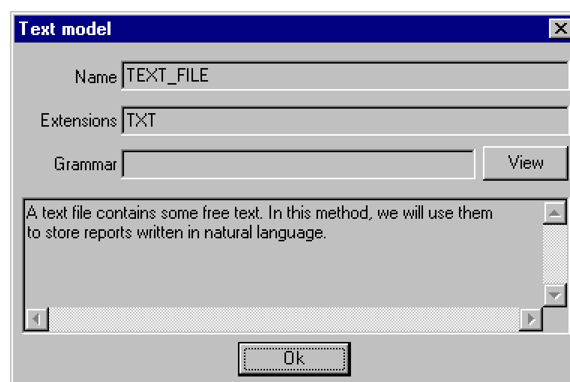


Figure 9.3 A text model properties dialogue box

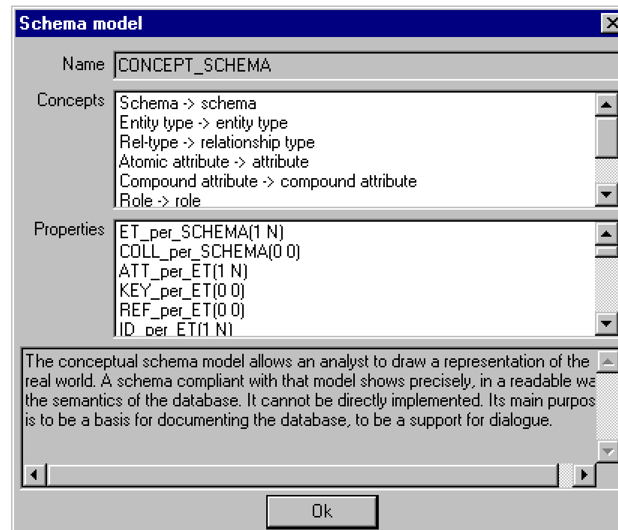


Figure 9.4 A schema model properties dialogue box

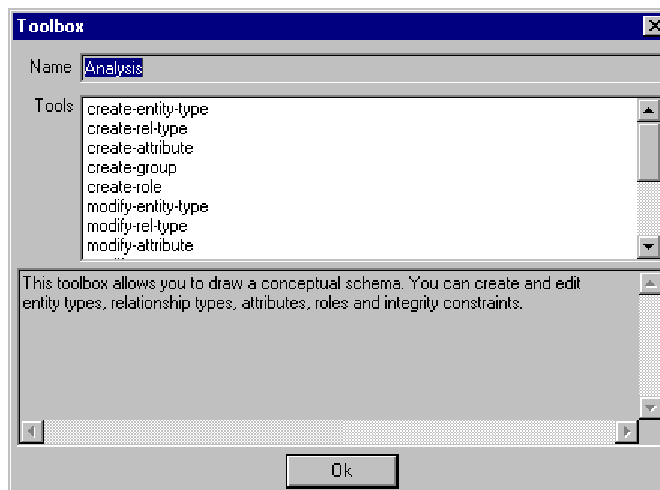


Figure 9.5 A toolbox dialogue box

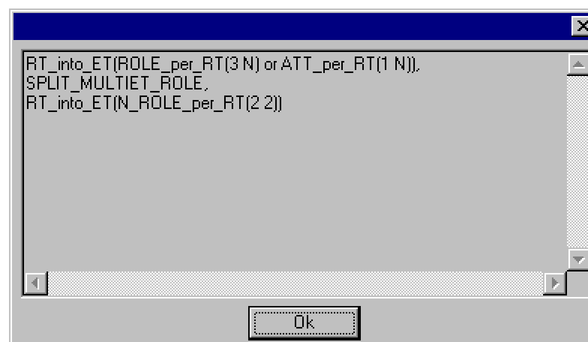


Figure 9.6 A primitive process type properties dialogue box

- A *properties* entry associated with an engineering process type, or with the title, opens a dialogue box like in Figure 9.7 showing the process type name, the lists of all product types in input, in output, in update, and the internal product types. It also shows the short process type description and a *help* button that can open the method help file to the engineering process section that can contain a detailed description of the strategy to follow.
- A *properties* entry associated with a diamond opens a dialogue box similar to the one for general primitive process types as shown in Figure 9.6, the difference being in the con-

tent of the large text zone which now contains the whole definition of the condition in clear text, as it appears in the MDL source listing.

- An *open* entry associated with an engineering process type is the first of two functions for navigating trough the method. When selected, this function replaces the content of the method window by the representation of the selected engineering process type strategy. In other words, this function goes down in the hierarchy of process types.
- A *back* entry associated with the title is the reverse of the *open* function. When selected, the content of the method window is replaced by the previous one, the strategy of the engineering process using the current one. This function goes upward in the hierarchy of process types towards the root one.

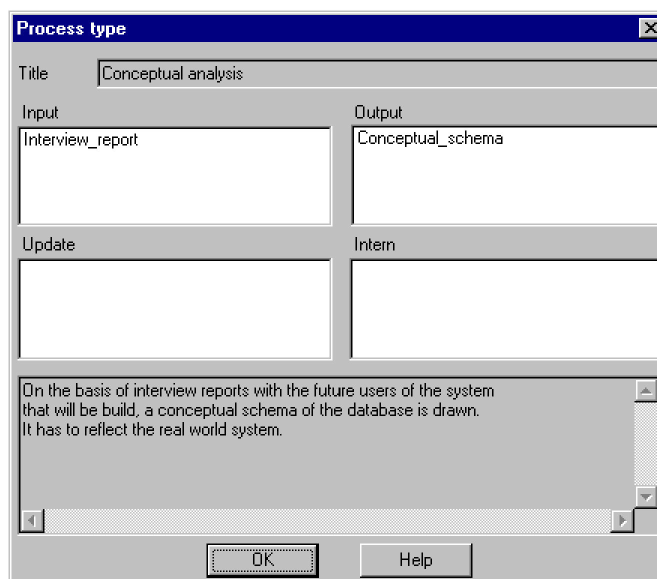


Figure 9.7 An engineering process type dialogue box

9.2.3. Following a method

In the CASE environment itself, the method window will not only be used to show the current method to database engineers, but also to guide them during their projects. This guiding must be compliant with the semantics presented in Chapter 4. It will be done by showing the following process type states with colour codes, as it can be seen in the method window in the left side of Figure 9.8 showing the CASE environment:

- the *allowed* state concerns the process types that can be performed at a given time; they are shown in a first colour, green borders by default, grey for the “Logical design” process type in the example
- the *running* state is for the process types for which some instances are being executed; they are shown in a second colour, red by default (none in the example)
- the *done* state is for already executed process types; they are in a third colour, white with black borders by default, like the “New” and “Conceptual analysis” process types
- the *unused* state is the original one, process types which have not yet been performed, and which still cannot be performed at the moment, are in that state; they are shown with a fourth colour, grey by default, white with grey borders for the “Physical design” process type in the example.

The CASE environment will also receive a new menu called *Engineering* (see Figure 9.8 and Figure 9.9) which will show a series a functions, that will be defined below and in the next section, to allow database engineers to follow the method and record histories.

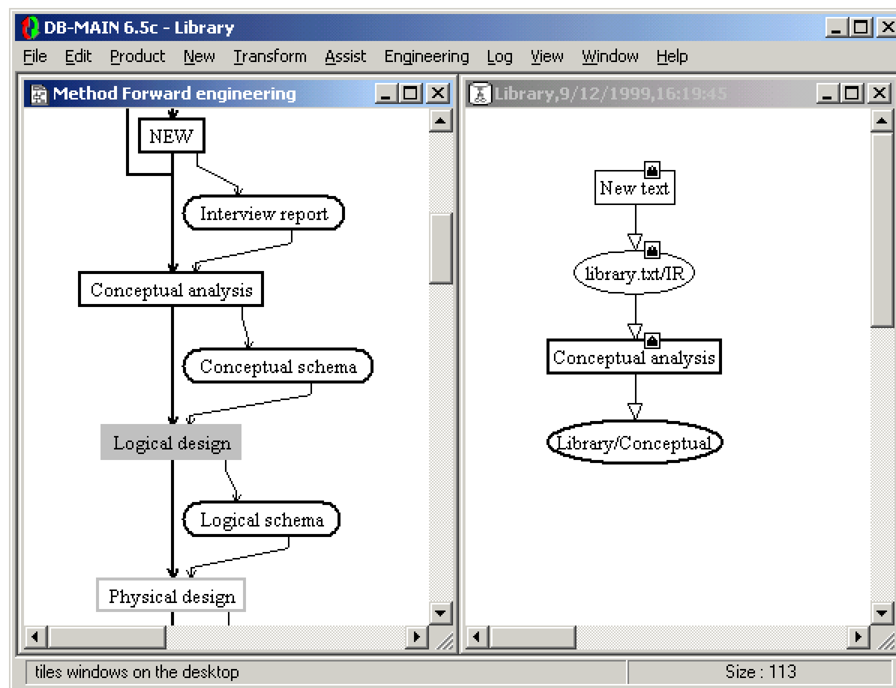


Figure 9.8 Different colours for different process type states and Engineering *menu*

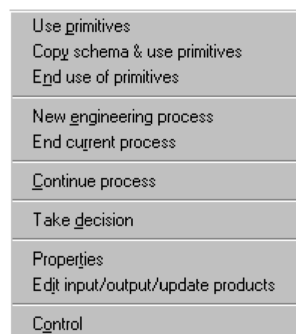


Figure 9.9 The engineering menu

To every process types in the method window, new contextual menu items will be appended too, depending on their state:

- an *execute* item is appended to all *allowed* process types
- a *terminate* item is added to all *executing* process types for which all the instances are finished, as explained below
- an *execute again* item is also appended to all *executing* process types and to all *done* process types. This function has many purposes:
  - A product type decalred in input or in update of a process type can have many instances when a process of that type starts. The analyst can either start the process with all the products of that type, or start a first process with only a few products, then start new processes with the other products, to make the history more readable.
  - As presented in Chapter 4, two processes cannot be executed in parallel if at least one of them updates at least one product used or modified by the other process. This can be a limitation that can be bypassed with this *execute again* function. Indeed, the analyst can do one process in part and terminate it, do the second process in part and terminate it, then execute again a process of the same type as the first one to do a second part of the job and terminate it again, and so on.
  - It is useful to allow versioning as it is explained later.

The following pages show how all this can be combined, firstly in a step by step execution, secondly in a more automated way.

A. Step by step method execution

When an engineering process is started, the method window shows the strategy of its type with all the sub-process types drawn in the *unused* state, except the one – or the ones – by which the engineering process must – respectively can – start which is – respectively are – drawn in the *allowed* state. The user can select one of them with the right mouse button to make its contextual menu appear. The following depends on the kind of the sub-process type: one of the four kinds of primitive process types or an engineering process type.

a. Automatic primitive process types

The contextual menu of an automatic – possibly configurable – primitive process type contains the *properties* and the *execute* items. The user can select *properties* to examine the full process type definition before executing the process. Since the process is automatic, when the user clicks on *execute*, he or she has nothing to do but to wait. During the execution, the content of the method window evolves. When the execution begins, the state of the type of the process is changed to *running* and other types that were in the *allowed* state are passed either to the *unused* state or to the *done* state, as they were before being put in the *allowed* state according to the semantics of the control structure that encompasses them as defined later in this chapter. When the execution ends, the primitive process type is put in the *done* state then, according to the control structures of the strategy, all the process types that can be performed after the current one are put in the *allowed* state, and the CASE tool gives the control back to the user.

An automatic user configurable primitive process type has the same behaviour, except, that one or more dialogue boxes are presented to the user to allow him or her to specify the value of a few parameters before the CASE tool does the job. These dialogue boxes will always appear while the process type is in the *running* state.

b. Manual primitive process types

In the DB-MAIN CASE environment, with the MDL language, the only manual primitive process types are toolbox uses. When the engineer wants to use a toolbox, the methodological engine only changes the toolbox state, as well as the state of other sub-process types as for automatic types, updates the CASE environment by enabling all the tools in the toolbox, then suspends itself. The user is then the one who has to work. During that time, the method window is updated with the colours associated to the new states. If the user selects again the same primitive process with the right mouse button, the contextual menu still appears, but it is different: the *execute* item has disappeared. But, according to the progress of the user's job, a *terminate* item and a *execute again* item can be present or not. When selected, the *execute again* item allows the user to perform several processes of the same type with various hypotheses. The *terminate* entry gives the control back to the methodological engine. It disables all the functions enabled earlier and ends the primitive process. This way of working seems to be simple, but the three main points are left unexplained: the meaning of enabling or disabling tools, the meaning of working for a user, and the way the CASE environment can judge of the user's job progress.

i. Enabling and disabling tools

In a graphical environment, every application has functions that can be executed by several events: clicks in menus, clicks on buttons in toolbars or buttons in dialogue boxes, clicks with the different mouse buttons in some parts of a window, even shortcut keys pressed. By default, if the CASE environment is used without a method, all the tools must be enabled

to allow users to do whatever they want. When the CASE environment is used with a method, all the tools must be disabled when not explicitly enabled by a toolbox. To disable a tool, all the above events have to be trapped. For menu items and buttons, both in toolbars and in dialogue boxes, it is even better to show to the user that they are disabled. This can be done by showing them in grey or by not showing them at all. In the DB-MAIN CASE environment, they will be shown in grey. To enable a tool is to show its menu entry and its buttons in their original colours, and to make all the events responsive again.

At some times, several manual primitive processes of various types, that is to say using different toolboxes, can be performed at the same time on different products. One toolbox does not have to interfere with the others. Since each product can only be modified by one process at a time, and since the Windows environment or most other graphical environments only have one active window at a time, it can be decided that the active toolbox is the one of the type of the process that modifies the product in the current window. Switching from one window to another implies switching toolboxes as well. This can be done by attaching the suitable toolbox to each window. A default toolbox containing all the tools of the CASE environment will always be available for this purpose during the performance of methodology neutral processes or projects.

ii. Database engineer's work

To work in a toolbox constrained environment, database engineers have to be aware of what they have to do because they are left to themselves; only the description and the help file section associated to the engineering process type using the toolbox can contain descriptions of what to do.

When the toolbox is used, at least one product type is passed to it in input or in update. All the products of the types passed in update can be modified. So a user can start modifying them with the enabled functions of the CASE environment only. This is the default behaviour of the CASE environment which corresponds to the most common hope of modifying all the concerned products. But, sometimes, some engineers may only want to modify a few products, or, for history presentation reasons, they may want to separate the modifications of several groups of products. So the same primitive process type can have several instances. A way to do this will be presented in the following section about recording the history.

When the user finishes the job, he or she has to indicate it to the CASE environment using the *End use of primitives* item in the *Engineering* menu for each process of the same manual primitive type.

iii. Job progress

As a consequence of the way of doing described above, a very simple means for the CASE environment to judge of the user's job progress is to look at all the processes of the same type. If they are all finished, then the CASE environment can present the *terminate* item in the process type contextual menu; if at least one instance is not declared finished by the engineer, the item cannot be shown.

The *execute again* item being aimed at allowing the engineer to start one more process of the process type, it has to be present in the contextual menu while the process type is in the *running* state.

c. Engineering process types

The execution of an engineering process type is a bit similar to the execution of a manual primitive process type: the database engineer has the responsibility to perform it and to decide on its termination. When the engineer decides to start the new engineering sub-pro-

cess, the current one is suspended, the content of the method window is automatically replaced by the strategy of the new engineering process type and all its sub-processes by which it can begin are put in the *allowed* state. The user can now perform the new process in the same way he or she was performing the suspended process.

When the engineering process comes to an end (when the end of the algorithm in the method window is reached, or can be reached, if a first branch of a some structure just ended for instance), the methodological engine will propose to the engineer to select output products and to definitely end it or to continue it. If the engineer confirms the termination, the method engine automatically validates the output products. If one of the product is not valid, the engineering process does not stop, a message pops up on screen to signal the problem (Figure 9.10 for instance), specifying what rule of what product model is violated by what product, and the method window does not change. The engineer will have to continue the process, and maybe some of its sub-processes, to correct the products and to try to terminate the engineering process again later.

During the whole execution of an engineering process, the contextual menu of the process type title only shows the *properties* and the *back* items. When the process is over, the *terminate* item appears. To draw the user's attention to this, the title is simply drawn in black during the whole execution, then in the same colour as the *running* state when the *terminate* item appears in the contextual menu. When the engineer selects this menu item, the methodological engine goes back to the last suspended engineering process and puts the terminated process type in the *done* state.

Engineering process executions, like toolbox uses, can be cancelled to allow database engineers to change their mind.

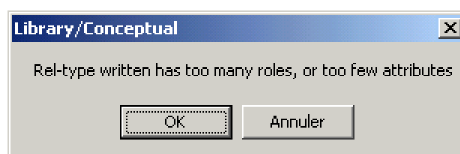


Figure 9.10A pop-up message signals that a component of the schema violates a rule of the schema model.

d. Synthesis about process types and state transition diagram

To summarise, a process type state transition diagram can be drawn, as shown in Figure 9.13. Each process type of the method is associated with such a diagram. When the project starts, each process type is in the *unused* state. When the turn of a given process type comes according to the project advancement, it is put in the *allowed* state. Then it can be put back in the *unused* state if a process of another type is performed (for instance if both process types are in two branches of a *one* structure), or it can be executed and be put in the *running* state. If the execution is cancelled (the arrow is drawn with a discontinued line to show that it is a correct but abnormal behaviour), the process type goes back to the *allowed* state, but, if the execution goes correctly to its end, the process type passes to the *done* state. If its turn comes again, the process type can be put in the *allowed* state again and the same scenario is followed.

This state chart has been constructed on the basis of the semantics of the process type strategies defined in Chapter 4. Indeed, a process type can be placed in any control structure defined in that Chapter:

- In a sequence, an *unused* process type is put in the *allowed* state when its turn comes, then in the *running* state when the analyst executes it, and finally in the *done* state when the analyst terminates it. If the sequence is followed a second time, whatever the reason, the process type can go from the *done* state to the *allowed* state again. See Figure 9.11.

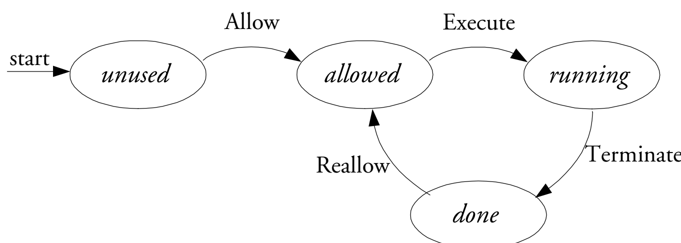


Figure 9.11 Process type state transition diagram in a sequence

- In a standard alternative, the process type can be put in the *allowed* state according to the result of the evaluation of the expression. Then the process state follows the same evolution as in a sequence. The state transition chart of a process in a standard alternative is the same as for a sequence in Figure 9.11.
- In a *some* or in a *each* structure, the process type is put in the *allowed* state at the same time as all the branches made of a single process type (for branches made of a control structure, see later). Then the process state follows the same evolution as in a sequence. It's state chart is thus the same as in Figure 9.11.
- In a *one* structure, the process type is put in the *allowed* state at the same time as all the branches made of a single process type (for branches made of a control structure, see later). It the process type is selected by the analyst for execution, its state passes to *running*, then later to *done*. It another branch of the *one* structure is selected, the process type goes back to the *allowed* state. If the *one* structure is executed a second time, the process type has to be put back to the *allowed* state again, and back to its previous state if it is not selected. The resulting state chart is shown in Figure 9.12.

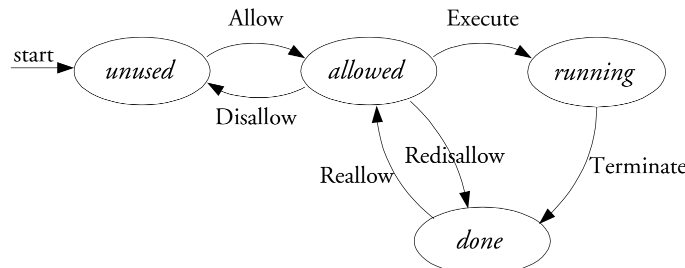


Figure 9.12 Process type state transition diagram in a one *structure.*

- In a loop, the process state follows the same evolution as if it was in a sequence followed several times. No matter whether the loop is a standard or non-standard one. So its state chart is the one shown in Figure 9.11.

Finally, the *execute again* function that allows a database engineer to perform several process of the same type is added to the state transition chart too. As well as the *cancel* function which is added to the CASE tool to allow analysts to undo a mistaken choice.

The complete state transition chart of a process type can be build by assembling the state charts from Figure 9.11 and from Figure 9.12 together, and by adding the *execute again* and *cancel* transitions. The result is shown in Figure 9.13.

e. Control structures

Control structures are driven by a state transition diagram too. Unlike process types, some control structures are made up of several parts: a condition and at least one body. Since these two kinds of parts are never performed at the same time, it is necessary to introduce two new states to follow them:

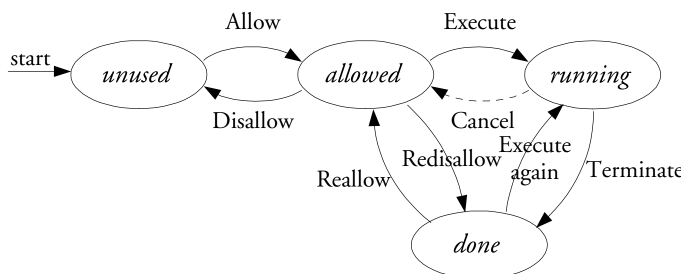


Figure 9.13 Process type state transition diagram

- The *expr-eval* state shows that the control structure expression is being evaluated; it will be shown on screen like the *running* state for process types (same colour).
- The *body-running* state shows that one body is pending. That is to say either a process type or a control structure of one body is in the *allowed*, *running*, or *body-running* state, or neither the engineer nor the methodological engine has decided to end the control structure. Since the body of a control structure is only materialised by its components, the *body-running* state is not visible on screen, it is not associated with a colour.

Each control structure has its own state transition diagram which is built according to the same principles as those used to build the process type state chart.

- Sequences do not have an expression. The database engineers do not have to explicitly start or stop them. When a sequence has to be started, the first component of its body has to be started, so the sequence itself is put in the *body-running* state, and the first component of its body has quit its *unused* state (to reach the *allowed* state for an engineering process, the *body-running* state for another sequence, or something else for the control structures examined hereafter). When the sequence ends, that is to say when what follows the sequence is performed or when the current engineering process strategy ends, the sequence state can be put in the *done* state. A state chart summarising this is shown in Figure 9.14.

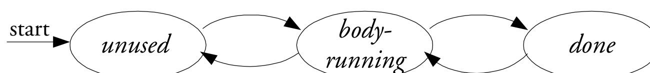


Figure 9.14 Sequence, one, some, each *state transition diagram*

- A *repeat* structure starts in the same way as a sequence: it is put in the *body-running state* and its body is put either in the *allowed* or in the *body-running* state. When the body ends, the *repeat* structure remains in the *body-running* state, and its body is put back in the *allowed* or in the *body-running* state again. At the same time, the upper structure (either the structure or the engineering process that contains this *repeat* structure) has to advance one step as if the *repeat* structure was really terminated and what follows the it has to be put in the *allowed* or *body-running* state too. It is only when something outside of the *repeat* structure is executed that the *repeat* structure must be put in the *done* state. The *repeat* structure follows the same state chart as the sequence in Figure 9.14.
- *One*, *some* and *each* structures are like sequences: they have no expression. The database engineers will not have to explicitly start or stop them. When a *one/some/each* structure has to be started, all the components of its body have to be started, so its state is set to *body-running* and all the branches of its body have to advance from their *unused* state. In a *one* structure, when a branch is executed, all other branches must be disabled and put back in their previous state. When this branch ends, the *one* structure must be put in the *done* state. In a *some* and a *each* structure, the execution of a branch does not modify

the state of the other branches. When a first branch of a *some* structure ends, the same principle as for the *repeat* structure must be applied, and it is only when a further process is executed that the *some* structure must be put in the *done* state. An *each* structure is put in the *done* state when all its branches are terminated. The *one/some/each* structures follow the same state chart as sequences in Figure 9.14.

- *If* structures have an expression that must be evaluated before the body can be run. When the method execution permits the evaluation of this expression, the control structures are put in the *allowed* state. When the engineer decides to evaluate the expression, the control structure is put in the *expr-eval* state, then to the *body-running* state when the expression is evaluated and one of the two bodies (*then* part or *else* part) can start (the first component of the body advances from its *unused* state). When the body ends, the *if* control structure is put in the *done* state. This is shown in Figure 9.15.

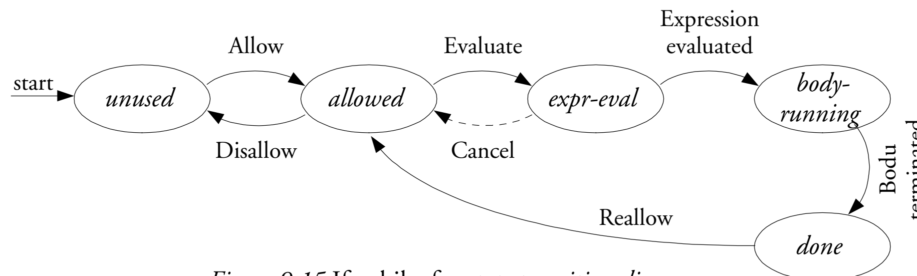


Figure 9.15 If, while, for *state transition diagram*

- The *which* and the *for* control structures follow the same path as the *if* structure up to the execution of the body. But, when the body is over, the state of the control structure is set back to *allowed* so that the engineer can evaluate the expression again. This is shown in Figure 9.16.

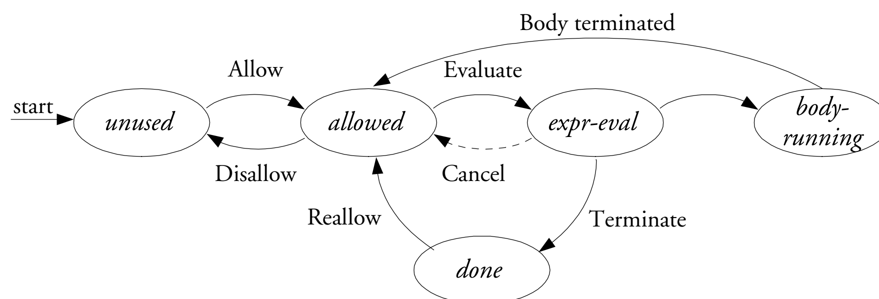


Figure 9.16 If, while, for *state transition diagram*

- The *until* control structure is different from the previous ones because the condition is evaluated after the performance of the body. The performance of the body is similar to the performance of a sequence. However, when the body ends, the state of the control structure is set to *allowed* in order to allow the engineer to evaluate the expression. According to the result, the state then passes either to *body-running* to perform the body again or to *done* to go on with the strategy. The state chart is shown in Figure 9.17.

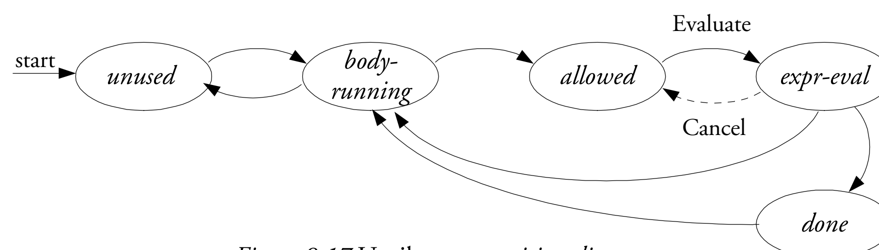


Figure 9.17 Until *state transition diagram*

B. Automated method execution

The step by step use of the method presented above is useful with small methods and with complex ones using a lot of non-deterministic control structures involving a lot of expertise from the database engineer. But some methods can be rather long and simple to follow, made mainly of traditional sequences and automatic primitive process types. In these cases, a lot of mechanical actions are required: selecting the first process type of the sequence, which is the only one in the *allowed* state, executing it, than the second one, than the third one,... always selecting the only process type in the *allowed* state without having to think about it. This is the case of the *Relational design* engineering process in Figure 6.8. This kind of tedious tasks can be automated.

An *auto-execute* item is added to the contextual menu of engineering process types at the same time as the *execute* item. The new engineering process will try to make the maximum by itself, requiring the intervention of the database engineer only when necessary, when decisions have to be taken: when more then one process type is in the *allowed* state, resulting of the presence of a non-deterministic control structure, and when a non-deterministic condition has to be evaluated.

9.2.4. Recording a history

Basically, the history is independent of the method because the same history can result from project supported or not by an MDL method. The CASE tool will receive a series of functions for managing histories. They are accessible both to the methodological engine that will use them automatically to make the history a reflect of the method, and to the users through the *engineering* menu. These functions will be analysed in detail, firstly in a method free project, secondly in a method supported project.

A. Recording a history in a method-free project

In a method-free project, database engineers can record histories in various ways:

- no recording at all
- recording a single log file containing everything
- recording a series of log files in sequence, one for each main phase of their project
- building manually a complete structured history
- building manually a complete structured history but only at the strategical level, without recording any log file.

The CASE environment has to offer all the needed functions to cover all these possibilities.

a. Recording the beginning of a method-free project

When a new project is created, a root engineering process is automatically created. The database engineers will have to decide whether they will use it as the root of a complex history, or just use it as a single workplace, without even paying attention to it.

b. Recording the execution of primitive processes during a method-free project

Automatic basic primitive processes will automatically leave their own trace in the current engineering process.

Manual primitive processes have to be created voluntarily. For this purpose, the *engineering* menu contains a *use of primitives* item. The database engineer has to select the products he or she wants in input or in update, then use this menu entry to start the primitive process and add it to the history. At that time, a dialogue box like the one in Figure 9.18 allows the

engineer to specify whether the selected products are to be used in input or in update. It forbids the use in update of a read-only product. The *description* button allows the user to add comments or to specify a few hypotheses that will influence the process. When the primitive process creation is confirmed, the engineer can open the products and modify them with a default toolbox containing all the tools of the CASE environment. All the actions performed by the engineer are automatically recorded in the log file of the current primitive process history.

When the job is finished, the engineer has to select the primitive process in the history to end it with the *end use of primitives* item of the *engineering* menu.

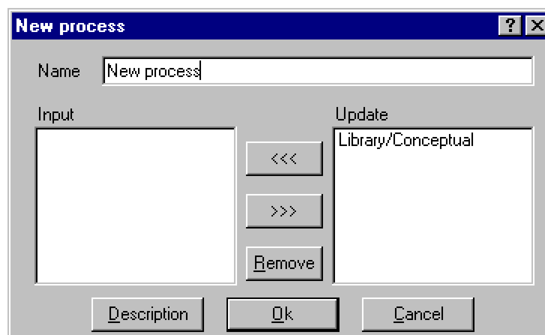


Figure 9.18 New process creation box with product use specification

c. Recording the execution of engineering processes in a method-free project

Engineering processes can be created in a similar way as primitive processes by selecting the products to take in input or in update, selecting the *new engineering process* item in the *engineering* menu and answering the same dialogue box (Figure 9.18). The new engineering process is created as a sub-process of the current engineering process shown in the method window. Then the new process becomes the new current engineering process. The database engineer can then do his or her job, and build the process graph by performing, recursively, new primitive processes and new engineering processes.

When the engineer wants to end the current engineering process, he or she uses the *end current process* item of the *engineering* menu. The dialogue box shown in Figure 9.19 appears to select the output products. In order to improve to usability of the CASE environment, the lists of the dialogue box are initialised with products selected in the graph before using the *end current process* function. When the selection is confirmed with the *OK* button, the graph is terminated and will not evolve anymore. The parent engineering process (which contains the one that is just finished) becomes the current one again.

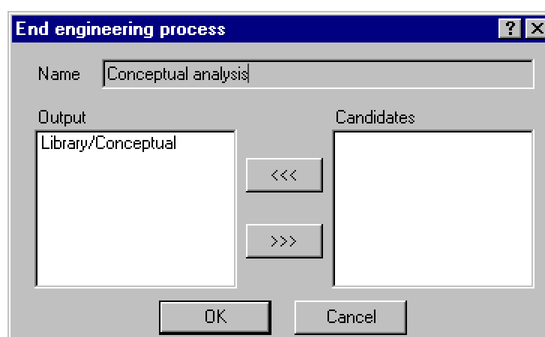


Figure 9.19 End of current process with output product selection

d. Recording a decision in a method-free project

When an engineer has terminated one (or many) process, he or she can decide to do it a

second time with the same products, with new hypotheses in mind, as in Figure 6.14. This results in the storage of several versions of a same product in the current engineering process among which the engineer can choose the best before going on. To record the decision, the engineer has to select all the products to take into account and to select the *take decision* item in the *engineering* menu. The dialogue box shown in Figure 9.20 appears with all the selected products in the left list. He or she will have to transfer the chosen product version(s) to the right list and to add a comment such as the rationales of the decision to complete the process. Upon confirmation, the new decision is stored in the graph of the current engineering process.

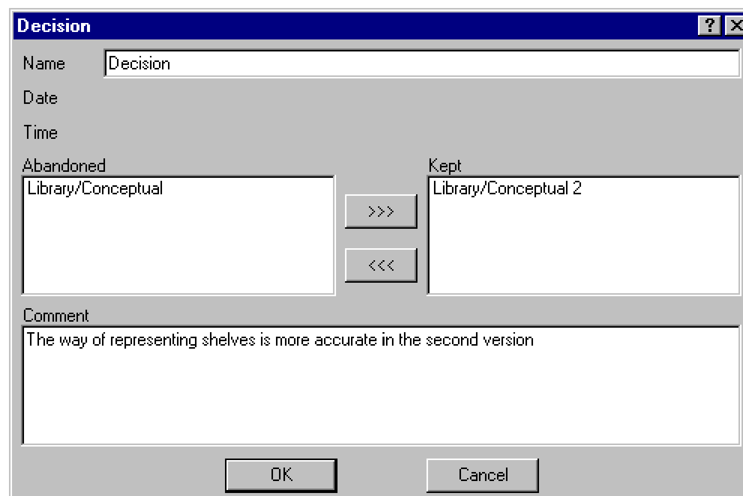


Figure 9.20 A decision taking dialogue box

B. Recording a history in a method supported project

When the project is supported by a method, the principles of history recording are fairly the same as with a method-free project, the few differences being the following ones:

- The same functions are used, but they are executed by the method rather than by the users with the menu.
- The use of the functions is accompanied by automatic change of current window to draw users attention to what they have to do and to reduce the number of manipulation they have to do.
- Each manual primitive process uses a specific toolbox rather than the default one.
- Decisions can still be taken by the engineers after having made several hypotheses, but also by the methodological engine itself when they are deterministic and imposed by the method.

When a database engineer selects the *execute* item in the contextual menu of a process type in the *allowed* state, the methodological engine acts on the state of several process types as described previously (the selected process type is put in the *running* state and all other process types in the *allowed* state are put back in their previous state), and automatically creates a process of the selected type and starts it. The action on the history depends on the kind of process type that is executed.

a. Recording the execution of primitive processes

An automatic primitive process type leaves its trace in the history by itself, so the methodological engine has nothing to do.

When a manual process type is encountered, the methodological engine executes the *use of*

primitives function. It does that more subtly than a database engineer would do by clicking on the item in the *engineering* menu because it also selects the suitable toolbox, according to the method, and it raises the history window to the front. This last action will generally improve the database engineer's ease of working because it shows the new process added to the history, and because the engineer must then open the products to use or modify. All the actions of the engineer are automatically stored in the log file of the process.

When the database engineer specifies the primitive process is finished by selecting the *end use of primitives* item in the *engineering* menu – this is one of the two only functions that cannot be performed by the methodological engine – the log file is closed and the method window is made current again to allow the engineer to *execute* a new process of the same type or to *terminate* the primitive process type with the respective entry in its contextual menu.

b. Recording the execution of engineering processes

When an engineering process type is encountered, the methodological engine simply executes the *new engineering process* function which acts the same way as if it was pressed by the engineer in a method-free project. The new engineering process becomes the current one so that forthcoming actions are stored in its history. The current window remains the method window (updated with the new process) because the operation most often performed next is the selection of the first sub-process type to execute.

When the strategy of the current engineering process type reaches its end, the control is automatically passed to the history window in order to allow the database engineer to select the output products and to put an end to the process, using the *end current process* item in the *engineering* menu – the second of the two functions that cannot be performed by the methodological engine. The history of the engineering process is closed and the parent engineering process becomes the current one again. The method window is automatically brought to the front for the engineer to terminate the strategy of the current process type with the *terminate* item in the process type title contextual menu.

c. Recording decisions

Some control structures of a strategy – *if...then...else*, *while*, *do...until* – impose the methodological engine to take decisions. These are not the same decisions as the one taken by the user on the choice of a product version. The decisions imposed by the method are *yes* or *no* decisions. They will have their own dialogue box with the *yes* and the *no* possibilities clearly shown, possibly modifiable by the user for weak decisions, always with the possibility of a comment as it can be seen in Figure 9.21. The decision and its comments are automatically stored in the history upon confirmation.

The first case study in Chapter 11 illustrates all this perfectly by showing the evolution step by step of a small project.

9.2.5. Complementary tools

A few tools provided by the methodological engine can be useful even without a method.

A. Schema analysis assistant

In a method driven environment, products are analysed when a process ends. Without a method, these analyses cannot be performed automatically since no product model is defined. But this kind of analysis can be interesting anyway. The CASE environment has to provide a mean to perform them. It is the aim of the *analysis assistant* shown in Figure 9.22. It allows engineers to conceive an analysis script with the structural predicates.

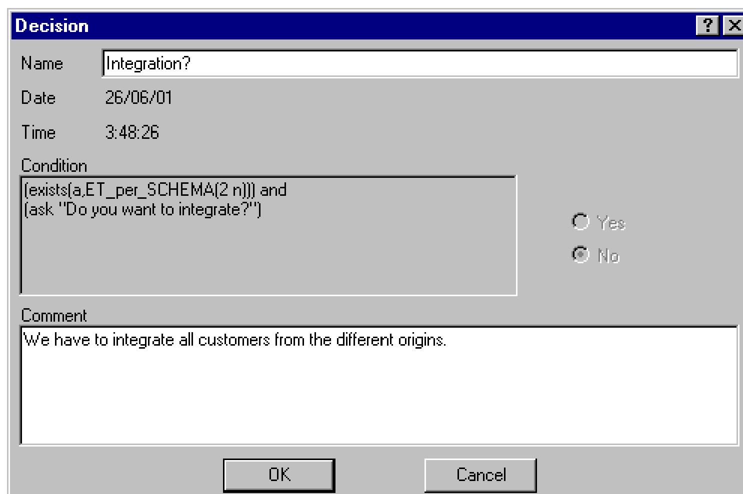


Figure 9.21 A forced decision

In this example, the exists *part of the condition returned* no, *so the result of the expression must be* no *to, and the* yes/no *choice is greyed.*

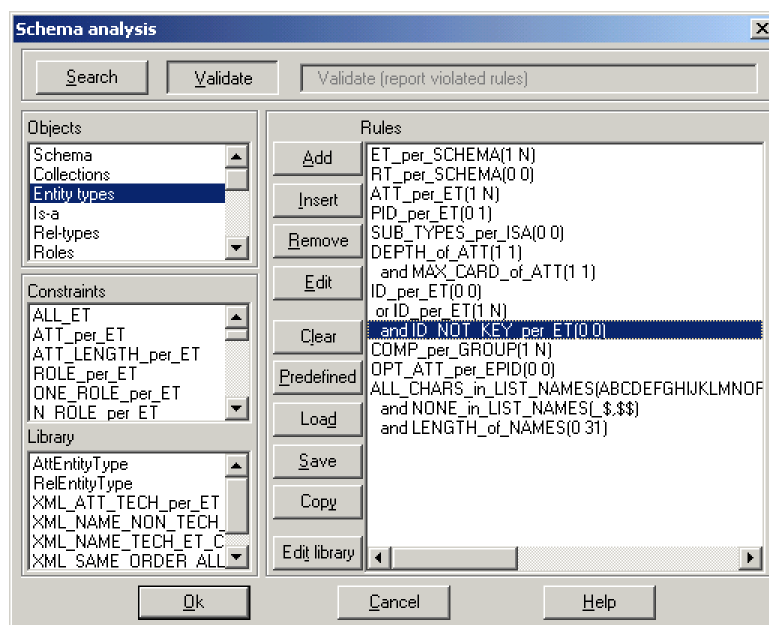


Figure 9.22 The schema analysis assistant

The large list at the right of the assistant window contains the script to use. Each line of this script can be selected, one at a time, using the mouse. The buttons next to this list allow the analyst to edit this script: to *Add* line at the end of the script, to *Insert* a line before the selected one, to *Remove* the selected line, to *Edit* the parameters of the selected line, to *Clear* the script, to *Save* it to disk and to *Load* it again, to use a built-in *Predefined* script[20], and to *Copy* the script in the clipboard for reuse in other programs (in a word processor for reporting, for instance).

The left column contains available components for building the script. From top to bottom, the first list contains categories of constraints (the objects concerned by the constraints: on entity types, on rel-types,...). On selection of a category, the second list is filled with constraints of this category. The analyst can select one of these constraints in order to *Add* or to *Insert* a new line in the script with this constraint. When the *Add* or the *Insert* button is pressed, a new dialogue box (Figure 9.23) appears on screen to edit the new line. It allows the analyst to edit the parameters of the constraint (some help about the syntax

20 Predefined scripts are general purpose scripts of common use directly built inside the CASE environment.

and the semantics of the constraint is available with the *Help* button), as well as to prefix the constraint with boolean operators: “and”, “or”, “and not”, “or not”.

On selection of a category of constraint in the assistant window, the third list in the left column is filled too, with predefined rules stored in a library. This library is a mean for the engineer to store some usual script chunks under a meaningful name. The library can be edited with the *Edit library* button which opens the dialogue box shown in Figure 9.24. The top right list of this window is the library. New entries can be added and existing entries can be deleted or renamed. When a new entry is created (see Figure 9.25), a category of constraint must be chosen for it. On selection of one library entry in the list of the library edition window, the bottom right list is filled with the definition of that library entry, and the left list is filled with all the constraints of the category associated with that library entry. This library entry can then be edited in the same way as the script in the schema analysis assistant. The whole library can be saved to disk and reloaded. A default library, which can be edited as well, is automatically loaded when the CASE tool is started.

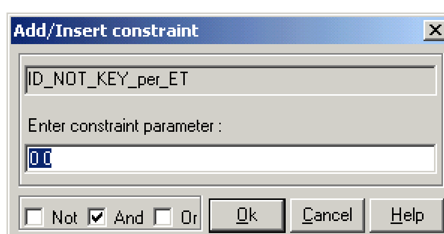


Figure 9.23 Line edition dialogue box for setting the parameters of the constraint and prefixing the line with and, or, not. *The* Help *button opens a help window with the syntax and semantics of parameters for that particular constraint.*

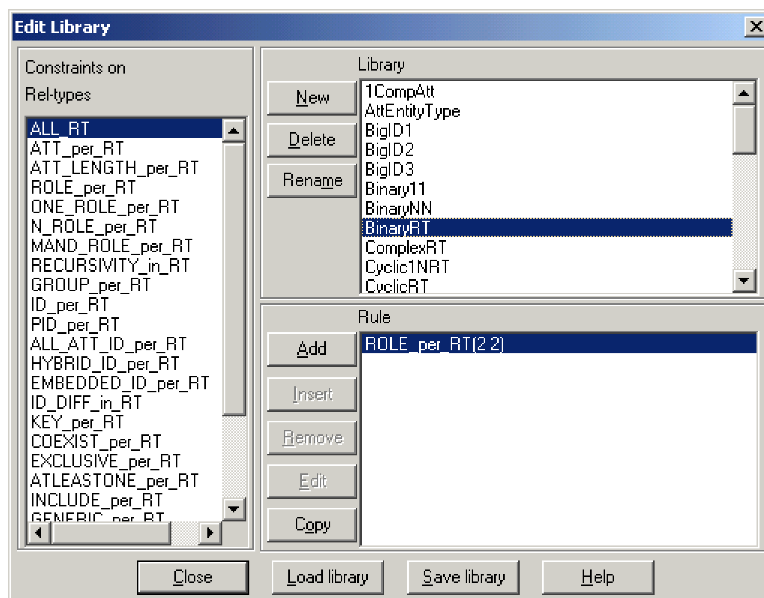


Figure 9.24 The schema analysis library edition dialogue box

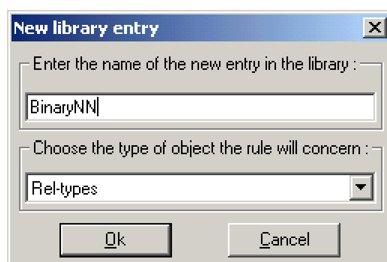


Figure 9.25 Creation of a new library entry

A script of the schema analysis assistant can be used within two modes (selectable in the topmost field of the window):

- used in *validation* (as shown in Figure 9.22), the script is used in the same way as with the method, the script states what a good product should be, and all the product components that violate these rules will be reported
- used in *search* mode, the rules describe what the engineer wants to find, and all the product components that match these rules will be reported.

When the assistant is executed (button *OK* pressed), a report is shown to the analyst if some rules are violated or found (according to the use mode), as shown in Figure 9.26. This window shows the first rule which is violated or found with the components of the schema which violates or satisfy the rule. The buttons *Previous* and *Next* allow the analyst to browse through all the rules of the script. When a schema component is selected, the *Goto* button is made available and allows the CASE environment to select the component in the schema and to show the schema with that selected component in the middle of the window. The *Select all* button allows the analyst to select all the components of the bottom list in the schema, while the *Mark all* button allows to mark[21] the same components in the schema. The *Report* buttons allows the analyst to save the whole report in a textual file.

Let us note that this assistant can be used by method analysts to define product models. Indeed, the analysts can use the assistant to write a script and to copy it to their MDL texts using the *Copy* button to defined schema models more easily.

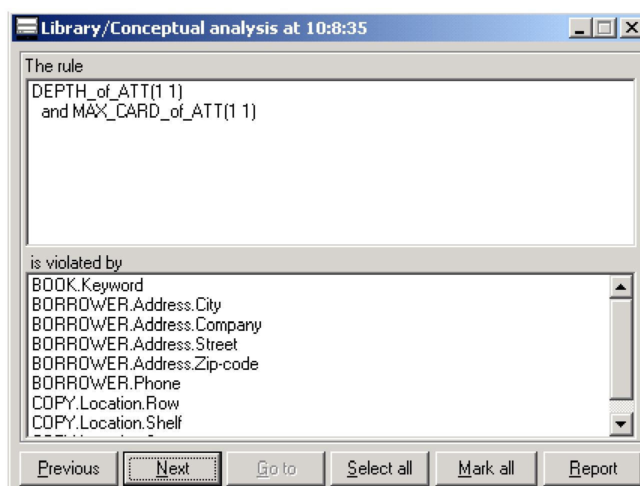


Figure 9.26 Schema analysis report

B. Global transformation assistant

Another facility provided by the MDL language that should be available anytime is the transformation scripts used for the configurable automatic process types. The *global transformation* assistant shown in Figure 9.27 responds to this need. Its structure is similar to the one of the analysis assistant. The large right panel shows a script of global transformations, which can be edited in the same way as the schema analysis scripts. The left column are the available components: from top to bottom, all the global transformations[22], all the control structures[22] and a library of predefined script chunks.

When a new entry is added to the scripts, parameters, which are structural rules, need to be specified. For that purpose, the schema analysis assistant is used in search mode, its top

21 Marking is a function of the DB-MAIN CASE environment that allows components to be marked in a persistent way (saved with the project) until voluntary unmarking, the traditional selection being volatile.

22 See Chapter 4 and Appendix C for a detailed description.

most panel being replaced to remind the global transformation, as shown in Figure 9.28, and its list of categories of constraints being limited to the object category concerned by the global transformation. In the example of Figure 9.28, the global transformation "GROUP_into_KEY" is only concerned by analysis constraints on groups.

The library edition window is shown in Figure 9.29. It works in the same way as the schema analysis library edition window.

The global transformation assistant can be used by method engineers to write more easily global transformation scripts which can be copied to the MDL file using the *Copy* button.

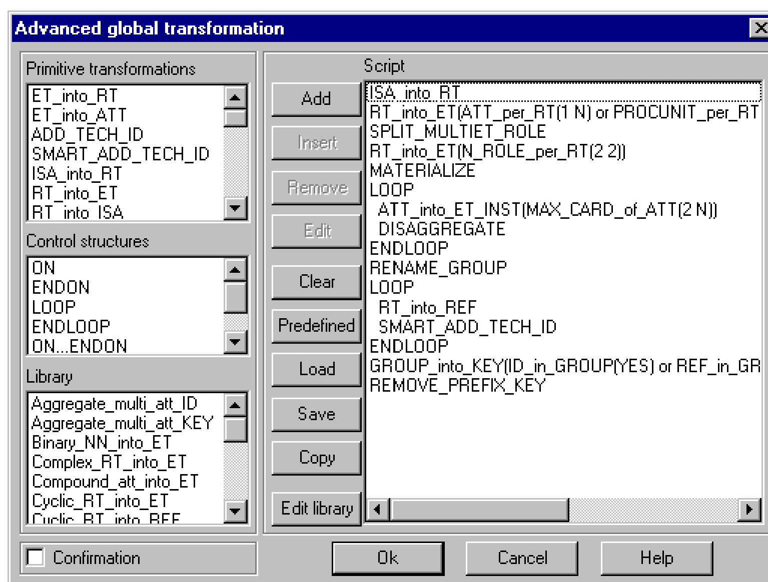


Figure 9.27 The global transformation assistant

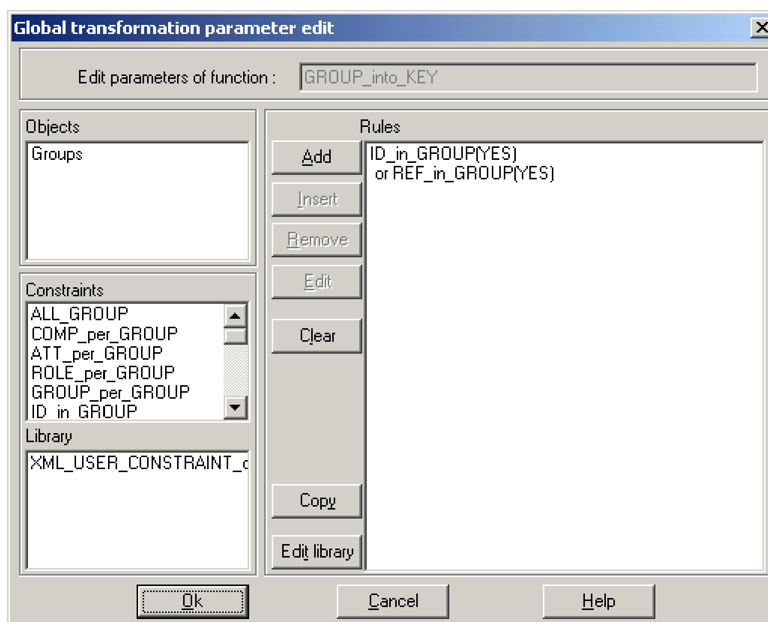


Figure 9.28 The analysis assistant used for global transformation parameter edition

C. Error correction

The CASE tool will have three functions to compensate for errors in method execution:

- An incorrectly started process can be deleted with the *delete* key when it is selected.

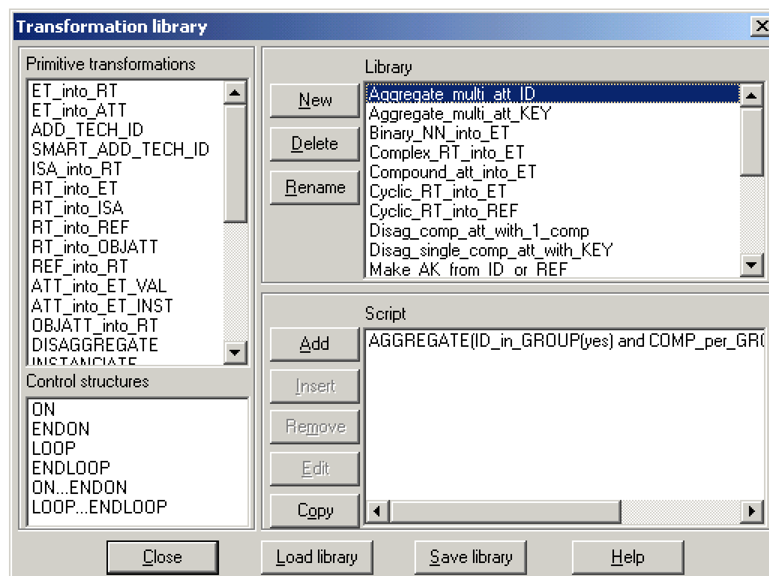


Figure 9.29 Global transformation library edition window

- A process that was incorrectly stopped, can be continued by selecting it and using the *continue process* item in the *engineering* menu.

These two functions have to be used directly after the mistake, preferably. Indeed, deleting a process in which a lot of work has already been performed is something people do not like. And continuing a process whose output products have already been reused by subsequent processes, and so modifying these products, is senseless. The third function is for parsimonious use in those "too late" cases.

- The *edit input/output/update products* item of the *engineering* menu allows the engineer to add input or update products to an already running process, or to add output products to an already terminated process.

9.2.6. Configuring the CASE environment

One of the requirements still not taken into account is the necessity of several levels of constriction of the CASE environment to the method. It is handled by a *control* item in the *engineering* menu that opens the dialogue box shown in Figure 9.30 to select a constriction mode. Four modes are proposed. The first mode, *strict use of the methodology*, and the second one, *Permissive use of the methodology*, are the two first levels of constriction cited in the requirements. The third and last required level of constriction was the possibility not to use the method. The DB-MAIN CASE environment offers a bit more with its third and fourth entries. The *No use of a methodology, but history control* mode allows the engineer to actually do not use the method, but to accept the help of the methodological engine to check the coherence of his or her actions when organising the history. The *No use of a methodology, no history control* mode really leaves the engineer alone, without any control. This is the mode of the CASE environment before the integration of the methodological engine.

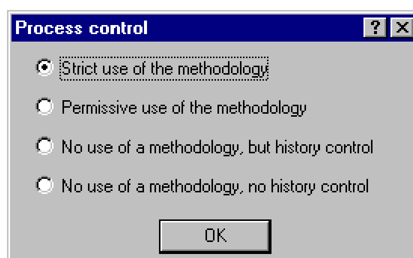


Figure 9.30 CASE environment control configuration

9.2.7. Browsing through a history

A. Browsing through a project history and its engineering process histories

Browsing through a history is the simplest way of using the history and certainly one of the most common, so it has to be as natural as possible to the CASE environment user. Hyperlinks seem to be the right tool. The history window always shows the current engineering process, with all its sub-processes symbolised by rectangles. When a project is started or loaded, the current engineering process is the root one. To see one of its engineering sub-processes, the user can simply double-click on it with the left mouse button. The content of the history window is then replaced with the selected engineering process. Doing the same again and again, the user can go deeper in the history hierarchy. To go back one upper level, the user can use the *close* function of the history window, by whatever mean provided by the operating system (*close* item in the window menu, window close icon, or keyboard shortcut). Closing the root engineering process window closes the project.

For the ease of browsing through the engineering process graphs, the history tree, as shown in Chapter 6, with an example in Figure 6.9, can be presented in another window, the *process hierarchy* window. Each entry of the tree, if it corresponds to an engineering process, which is shown with bold characters, will be an hyperlink to the process; selecting a bold tree entry will show the corresponding engineering process graph in the history window.

B. Browsing through primitive process histories

Primitive process log files can be very long and are, even if readable, rather difficult to understand, so people generally want to look at them much less often than engineering processes. Consequently, a fast access through a mouse click or a shortcut key combination is needless and a simple menu entry suffices.

A primitive process can modify several products and a product can be modified by several primitive processes. But every transformation, or log file entry, only modifies one product. Indeed, a product can be modified according to the content of one or several other products (in an integration process for instance), but each basic transformation is always applied to a single product. So a log file can be cut into several *slices*, each slice being made up of all the log file entries concerning the same product [WEISER,84]. Putting together all the slices concerning a given product gives its complete evolution. It is thus more interesting to record all these slices separately and to reassemble them according to the needs. A menu entry in the *Log* menu – this menu is supposed to be part of the supporting CASE environment – opens a dialogue box proposing to view log file slices by process or by product and finally shows the requested information in a simple text browser.

9.2.8. History replay and transformation

The replay of a primitive process is a simple task as shown in Chapter 7. Two menu entries allow a log file slice, or several ones, to be replayed either step by step or automatically.

History transformation is a personal task, as explained in Chapter 7. Indeed, each use of an history requires some particular tools which are specifically dedicated. These tools need to be programmed in Voyager 2 within the DB-MAIN CASE environment. The Voyager 2 language offers a series of facilities for these tasks:

- text file parsing that permits to read the log files easily
- string and list handling functions for managing the information buried in the histories
- access to the CASE environment repository for the management and transformation of the products

- GUI basic capabilities for human-machine dialogues
- traditional procedural language structure for easy learning and use.

Chapter 10

Architectural issues

After the presentation of the integration of the methodological engine in the CASE environment with the user's point of view in Chapter 9, it will be presented with the CASE environment developer's point of view. In a first time, the general architecture of the CASE environment and, grossly, the position of the methodological engine will be sketched. Secondly the DB-MAIN repository will be described with the enhancements to store the methods and the histories. The chapter will end with the update of the kernel and of the graphical user interface (GUI) for supporting the new requirements.

10.1. General architecture

The general architecture of the DB-MAIN CASE environment is sketched in Figure 10.1.

The kernel contains all the basic functions of the CASE environment. It uses a repository, which will be described in this chapter, to store permanent data such as all the database schemas and references to texts. The texts are stored independently of the repository but they have to be accessible by the CASE environment too. So the kernel includes a series of repository access and management functions, as well as functions for transforming and analysing the products and their components. Some analysis functions, mainly text analysis functions, use some patterns. These patterns are stored in libraries that can be used by several projects. The kernel also contains a series of all-purposes basic functions and is open to future developments.

At the top of the schema, the GUI is the link between the users and the kernel of the CASE tool. It shows all the products textually or graphically, in different ways, in a multi-windowed environment. It allows the users to handle these windows, to select and to act on some parts of the content of the window and to use the tools of the kernel. Its role is both to present all the data to the user and to control his or her actions.

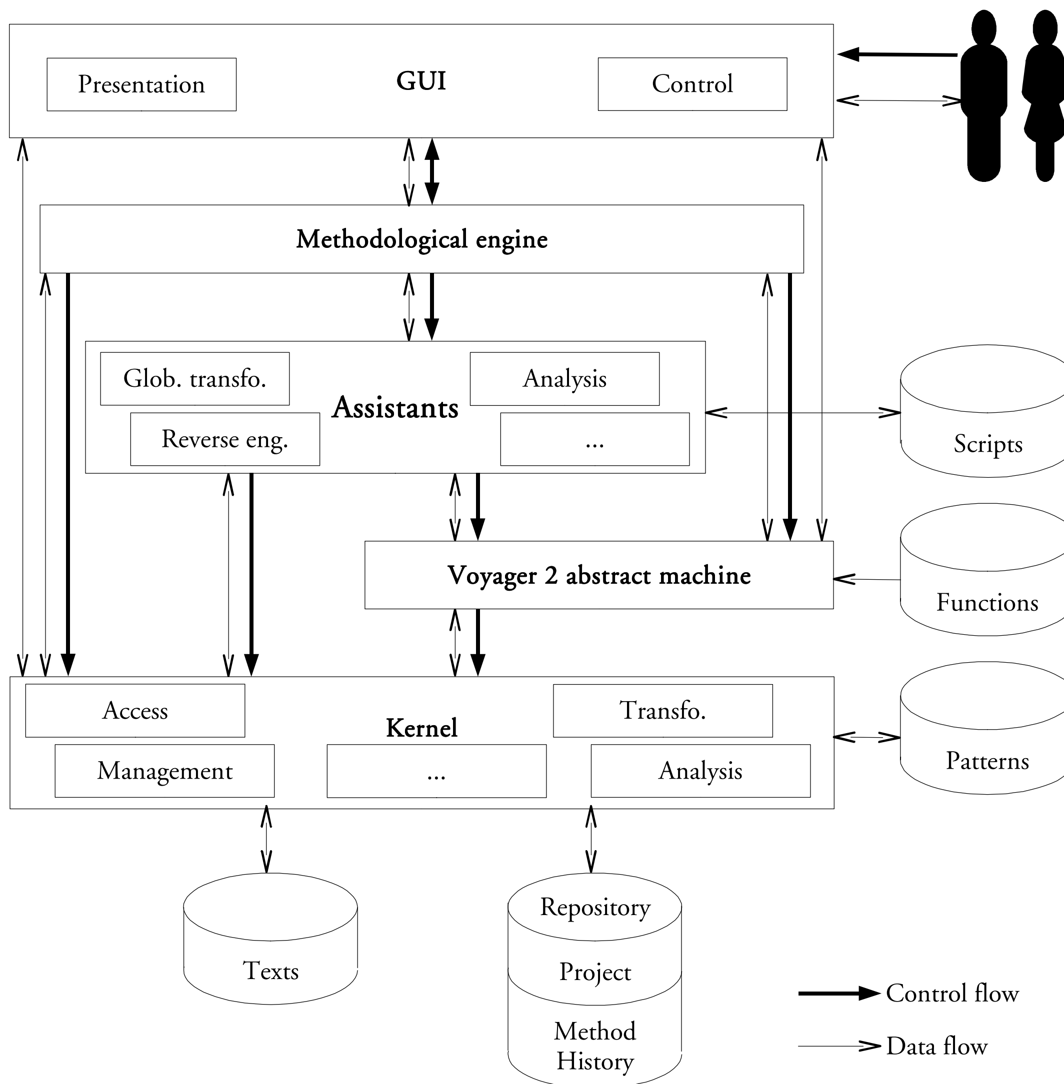


Figure 10.1 The general DB-MAIN architecture

Between those two levels, the assistants are complex components with a GUI interface that is aimed at automating some tedious and repetitive tasks that can be performed with the kernel. Some of these assistants (global transformations, schema analysis) can be driven by a script which can be stored independently.

Still between the GUI and the kernel, the Voyager 2 abstract machine is the component than can run Voyager 2 programs and functions which have already been compiled by the independent compiler. Voyager 2 functions can access and modify the repository through the kernel. Voyager 2 programs can be run directly by users through the GUI and Voyager 2 functions can be called by the assistants, in global transformation or product analysis scripts, as well as in response to the discovery by the reverse engineering assistant of some structures compliant with given patterns.

In this thesis the methodological engine component is added. It is incorporated underneath the GUI so that:

- users can dialogue with it through the GUI
- it can influence the control performed by the GUI by restricting the access to a series of functions that would normally be reachable in a method-free environment but which are forbidden in a particular context imposed by a method
- it can use the GUI to show the method it follows and the state of each process type
- it can launch some operations in the kernel, in an assistant, or a Voyager 2 function
- it can access the repository through the kernel in order to consult the method to follow and organise the history of the project.

The method and the history are stored in the repository. About the history, this seems natural since it is intimately linked to the project. Concerning the method, the reasons to store it in the repository may seem less straightforward, but a project can be very long, it may last several months. If the method evolves during that time to comply with the needs of newer projects, it cannot evolve for the current project, so it is necessary to make a copy of the method when the project starts. Indeed, if the part of the method which has already been used is modified, the history obtained with the original version of the method may not match the new version anymore and become obsolete. Moreover, it is easier to maintain strong links between the project history and the method when they are stored in the same place.

In conclusion, the implementation of the methodological engine must result in a brand new module, an extension of the repository, an extension of the kernel with new functions to access and manage the repository extension, and a thorough modification of the GUI. No modification of the existing functions of the kernel, of the Voyager 2 abstract machine and of the assistants are needed. The thorough modifications of the GUI are due to the fact that every component of the GUI has to be aware of the presence of the methodological engine when it shows itself: buttons and menu entries have to show differently if they are enabled or disabled by the method, and dialogue boxes and messages have to use the correct terminology according to the current product models.

10.2. The repository

The DB-MAIN repository is a C++ object base. It is stored into main memory, except texts, log files, descriptions and annotations which are stored in files. The kernel contains procedures for unloading/loading the repository in a file. This choice is relevant for the performance factor and because of the relatively small size of a project – the largest database schemas being made up of no more than a few tens of thousand components.

The original repository is designed to store a single project with all its schemas, references to its texts, and very basic semantic-free links between all the products. In this work, only the extension made to this original repository are presented: the new parts of the repository for storing the method to follow during the project and the history of the project.

10.2.1. Notations

The repository will be shown graphically, cut in several views, as a series of ERA schemas[23]. The following product model can be used to read these schemas:

```
schema-model CplusplusObjectBase
    title "C++ object base"
    description
        This C++ object base model is designed specifically for the representation
        of the DB-MAIN repository
    end-description
    concepts
        schema                      "object base"
        entity_type                 "class"
        is_a_relation               "inheritance"
        is_a                        "inherits"
        sub_type                    "sub-class"
        super-type                  "super class"
        rel-type                    "rel-type"
        attribute                   "property"
        atomic_attribute            "property"
        compound_attribute          "struct"
        object                      "class"
        processing_unit             "method"
        group                       "group"
        role                        "role"
        identifier                  "identifier"
        primary_identifier          "identifier"
        secondary_identifier        "identifier"
        coexistence_constraint      "coexistence constraint"
        exclusive_constraint        "exclusive constraint"
        at_least_one_constraint     "at-least-one constraint"
        exactly_one_constraint      "exactly-one constraint"
    constraints
        % This is a static model for schema reading, not requiring constraints
end-model
```

10.2.2. The original repository of the DB-MAIN CASE environment

The original repository of the DB-MAIN CASE environment is shown in Figure 10.2. A project is an object of the *System* class. A project is made of several *products* which are either *texts*, *schemas* or *prod_sets*. A *text* is a reference to an external file. A *schema* is a GER schema according to the model presented in Chapter 3. Most classes of this repository *(entity_type*, *rel_type*, *si_attribute*, *co_attribute, role*, *group*, *constraint*, *collection*, *proc_unit*,...*)* represent the concepts of this model. The *generic_object* class is a special class which is inherited by almost all other classes. It contains a global technical identifier, a class identifier, graphical positions, other presentation attributes, and a series of flags for runtime use. The following pages will present extensions of this repository.

23 As described in Chapter 3.

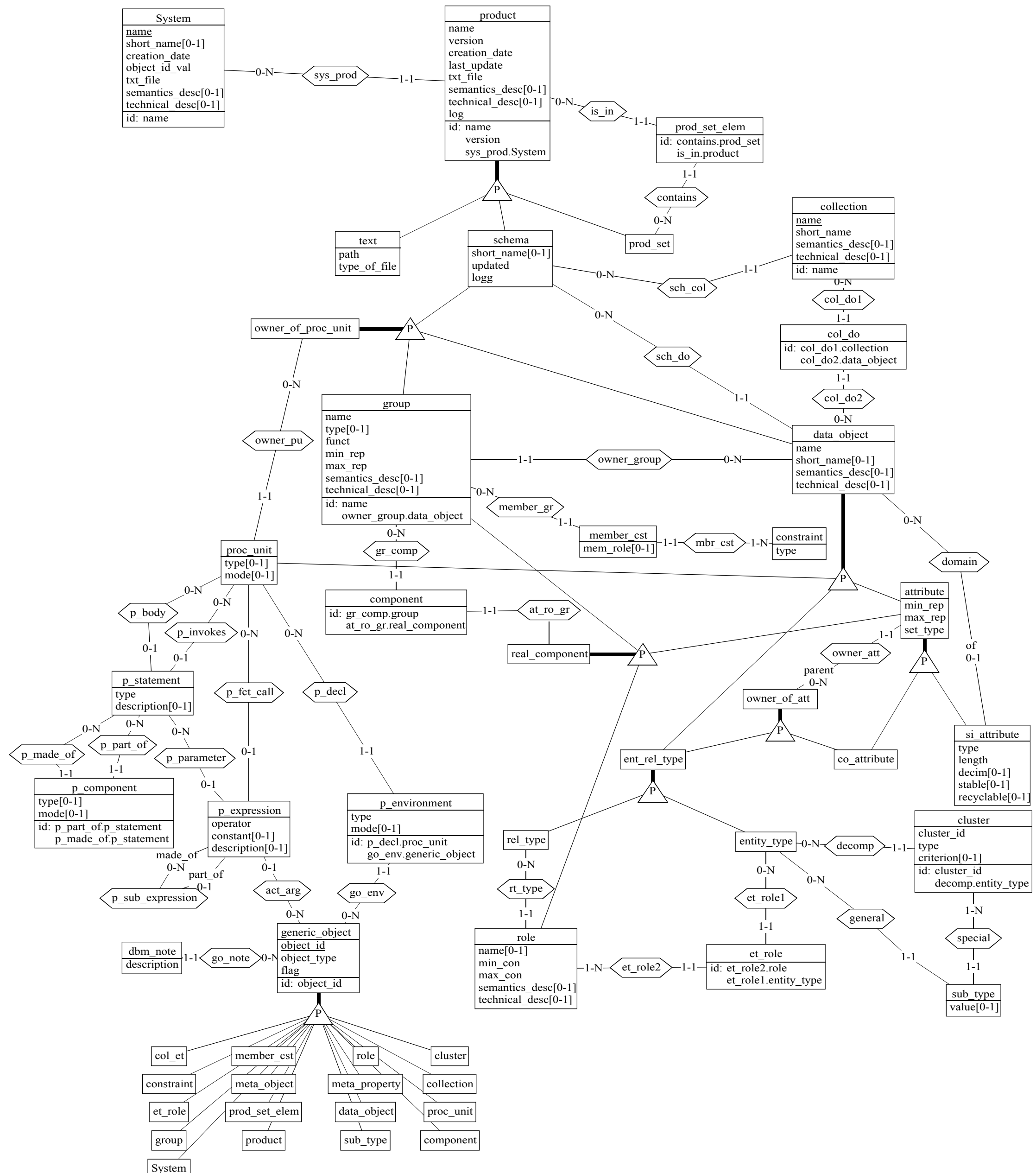


Figure 10.2 The original repository of the DB-MAIN CASE environment

10.2.3. The repository extension

The repository extension will be presented in two parts: the first one is aimed at the method and the second one at the history. The first part is sufficient in itself for some tasks like compiling a MDL source file. The second part can also be used independently of the first one like during the performance of a method-free project. The link between these two parts is straightforward and will be sketched in a third time.

Classes which are part of the original repository are drawn with a shadow.

A. The repository section concerning methods

This section of the repository is shown in Figure 10.3. In the bottom of the drawing, the *System* class is the *System* class of the original part of the DB-MAIN repository. Only the title of the class is shown because its properties and methods are not relevant here.

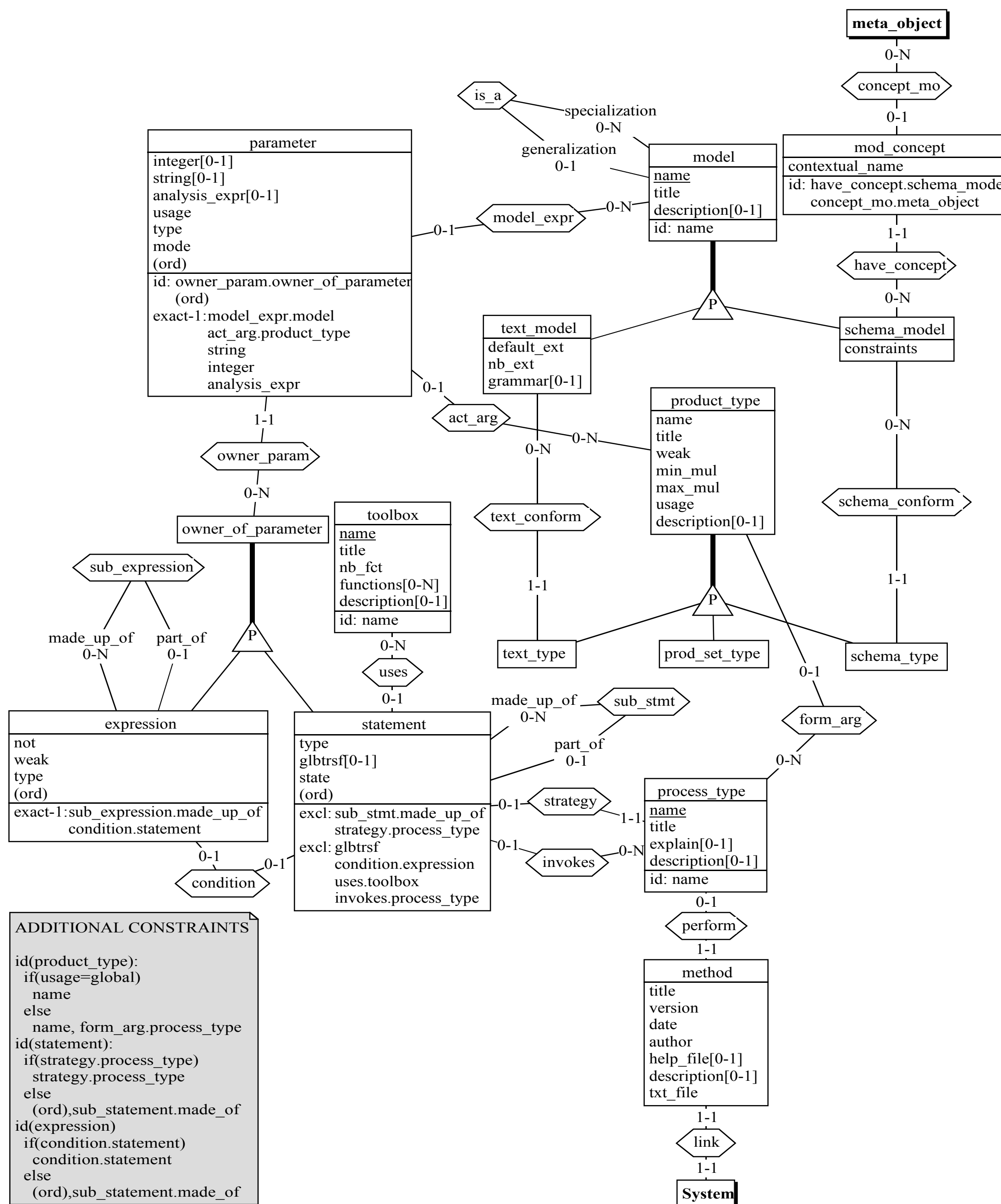


Figure 10.3 The repository part for storing the method

a. The method

A *method* class is attached to the system. The roles of the *link* show that every project has to follow exactly one method. In fact, even method-free project will have a default permissive method. This will permit several constraints to be defined and checked more easily. The *method* class is defined by the data declared in the *method* section of an MDL listing.

b. Process types and product types

Every method is made up of several process types which are stored as instances of the *process_type* class. Among all these instances, exactly one is the root process type of the method, as indicated by the *perform* rel-type. All others are process types used by this root process type or other sub-process types as declared in an MDL source. Every process type has some local *product_types* and a *strategy*. Both the process types and the product types are identified by their formal name and are characterised by their readable title and an optional

description. The *process type* class can also contain the declaration of a section (*explain*) in the method *help_file*. Each *product_type* is also characterised by a *weak* property specifying the degree of respect of products of this type to its model, by the *min_mul* and *max_mul* properties specifying the number of instances this class should have, as described in Chapters 4 and 5, and by its *usage*. This last property gives an interpretation to the *form_arg* rel-type: if a product type is declared as global, the *usage* property identifies this fact and no process type is linked through the *form_arg* relationship; if a product type is declared *input*, *output*, *update*, *intern* or *set* locally to a process type, the *usage* property reminds the declaration and a *form_arg* relation links the product type to the process type. Every *product_type* instance is either a *schema_type*, a *text_type* or a *prod_set_type* instance.

c. Product models

Every *text_type* must be of a *text_model* and every *schema_type* must be of a *schema_model*. *Schema_model* and *text_model* are both product *models*. As defined in Chapter 3, product models can inherit their characteristics from other product models through the *is-a* rel-type. Texts models are identified by their *name*, and they have a more readable *title*, an optional description as well as a list of characteristic file extensions (*default_ext*, *nb_ext*) and an optional *grammar* description file. Schema models have the same identifier, title and description, plus a list of constraints and a series of concepts. The instances of the *meta_object* class, which is part of the original repository, are a representation of all the possible product elements. The *mod_concept* class links some of these elements to schema models while giving them their *contextual_name*.

d. Process type strategy

The strategy of a process type is made up of *statements* which are themselves control structures, transformation scripts, toolbox uses and sub-process uses. The *type* property specifies this and indicates which of the optional *glbtrsf* property (transformation script), or the optional *condition.expression*, *uses.toolbox* or *invokes.process_type* roles have an instance. The *state* property is used during the performance of a project as explained in Chapter 9. Some control structures (*if*, *while*, *until*) need some expressions as a condition. These expressions can be very complex and made of several sub-expressions. Both statements and expressions sometimes need parameters. A parameter is either a product model, a product type, a string, an integer or a product analysis expression. The *ord* property in the *statement*, *expression* and *parameter* classes is put between parentheses to show that it is not a real property. It has been added to represent the fact that all the sub-statements of a statement, all sub-expressions of an expression, and all parameters of a statement or an expression are ordered. But this order is already kept by the fact that many-to-one rel-types are stored as lists in our C++ object base, as explained later in this chapter.

In fact, the strategy is itself a single statement which can be a control structure with sub-statements through the *sub_stmt* rel-type. All the strategy elements presented in Chapter 4 can be stored in this repository as follows.

- **sequence** *A*;*B* **end-sequence**: A statement with a *type* property set to *sequence* is linked through *sub_stmt* to two other statements, the first, *A*, with *ord* equal to 1, and the second, *B*, with *ord* equal to 2. But the *ord* property is simply kept by the fact that the relation between the *sequence* and the *A* statement is the first in the *made_up_of* list, and the relation between the *sequence* and the *B* statement is the second. The instances of the repository classes representing this simple sequence are sketched in Figure 10.4.
- **some** *A*;*B* **end-some**; **one** *A*;*B* **end-one**; **each** *A*;*B* **end-each**: These structures are stored in the same way as sequences, only the main statement *type* property differs, being *some*, *one* or *each* respectively.

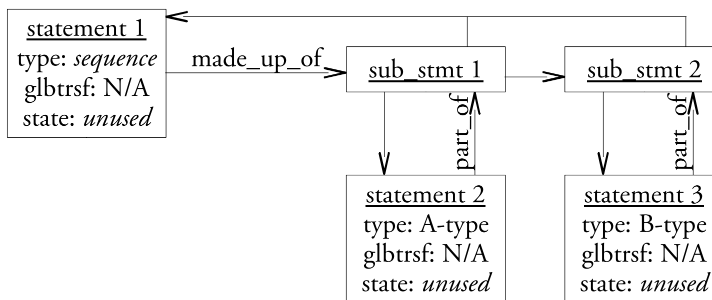


Figure 10.4 A sequence stored in the repository

- **if** *cond* **then** *A*;*B* **else** *C* **endif**: The storage of this structure is a bit more complex because the sub-statements have to be classified in two lists, when *A* and *B* are executed, *C* cannot be executed and conversely. To solve this problem, this statement is stored as the following one:

  if *cond* **then sequence** *A*;*B* **end-sequence else** *C* **end-if**

  More precisely, each of both lists of sub-statements containing more than one statement will be replaced by a sequence. Eventually, the *if-then-else* statements always have two sub-statements, or only one when there is no *else* part.

  Moreover, *if* statements have a condition. It will be stored as an expression. The storage of an expression has the form of its evaluation tree: the root node (the one linked to a statement, others are not) is the operator with the greatest priority (the *type* property specifies the operator, except the *not* and *weak* declarations represented by the *not* and the *weak* boolean properties), the operands being sub-expressions ordered as they have to be evaluated.

  Figure 10.5 shows the storage of the above statement.

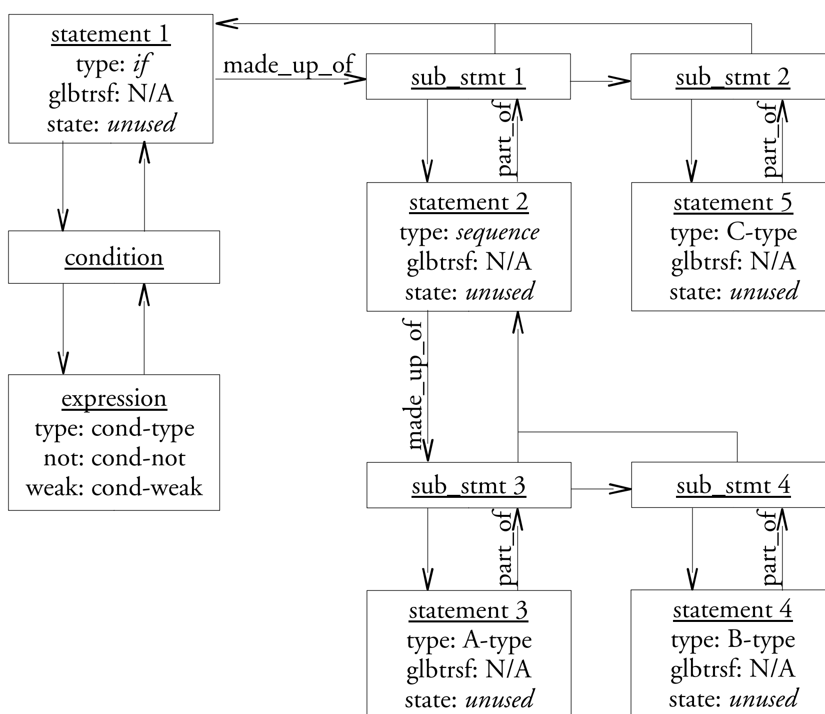


Figure 10.5 An if-then-else structure stored in the repository

- **repeat** *A*;*B* **end-repeat**; **while** *cond* **repeat** *A*;*B* **end-repeat**; **repeat** *A*;*B* **end-repeat until** *cond*: These structures can be stored in a way similar to the *if-then-else* structure. When the body of the loop is made up of several statements, *repeat* parts of these structures can be replaced by:

  repeat sequence *A*;*B* **end-sequence end-repeat**

  That way, *repeat*, *while* and *until* structure always have exactly one sub-statement. The difference between the three structures lays in the value of the *type* property and in the presence of a condition for the *while* and the *until* structures.

  Let us note that confusion between the sub-statements cannot arise with the *until* and the *while* structures as with the *if-then-else* structure because the body holds in a single group, so the storage space can be optimised by not adding the *sequence* structure and storing the *repeat* structure as the *sequence* structure.

- **for** (**one**|**some**|**each**) *S* **in** *P* **do** *A*;*B* **end-for**: The body of the loop (the sequence) is similar to the body of a *repeat* loop and can be stored in the same way. The main particularity of a *for* loop is the presence of two parameters: *S* and *P*. They are stored as instances of the *parameter* class and linked to the statement through the inheritance mechanism (*statement* is-a *owner_of_parameter*) and the *owner_param* rel-type. Both instances of *parameter* have the *type* property set to *product-type*. Parameter *S* is the first in the *owner_param* list (*ord*=1), its *usage* property is set to *output* and it is linked through the *act_arg* (actual argument) rel-type to the declared product set type. Parameter *P* comes next (*ord*=2), its *usage* is set to *input* and it is linked through the *act_arg* rel-type to the declared product type.

- **do** *A*(*P1*,*P2*): This statement, as well as all the following ones, is no more a control structure, there are no more sub-statements. The *statement* instance has a *type* value of *do* and a relation to the *process_type* instance *A*. According to the definition of *A*, the statement may need to have some parameters. In fact, the statement must be in relation with the same amount of parameters through the rel-type *owner_param* as *A* is in relation with product types via the *form_arg* rel-type, and they have to correspond two by two: if the first *product_type* instance in relation with *A* as a *usage* set to *input* (respectively *outout* and *update*), than the first *parameter* instance in relation with the statement must be I-compatible (respectively O-compatible and U-compatible) with it.

- **toolbox** *T*(*P*): A relation to the *toolbox T* through the *uses* rel-type is what characterises this statement in addition to the relation via *owner_param* with a parameter, itself in relation with the *product_type P* through the *act_arg* rel-type.

- **glbtrsf** (*P*,*T1*,*T2*,...): This is the only construct for which the *statement* instance has a value for the *glbtrsf* property, set to (*T1*,*T2*,...). The statement just has a relation with a parameter (possibly several) itself in relation with the product type *P*.

- **extern** *F* "*file*".*function* (*typeP1*,...) ... **external** *F*(*P1*,...): this two parts statement is actually stored as if it was declared in a single part, as a pseudo-statement of the form:

  external "*file*".*function* (*P1*,...)

  The first part being present only for the compile-time type checking, it does not need to be stored. Hence the *statement* instance has its *type* property set to *external* and it has a lot of relations with parameters. The first *parameter* instance has its *type* property set to *string* and its *string* property contains *file*. The second *parameter* instance has a similar form with its *string* property containing *function*. All other *parameter* instances are the parameters of the function in the order they are declared. They can be of any type: *integer*, *string*, *analysis_expr*, *product_type* or *product_model*.

- **extract** *Ext*(*source*,*dest*); **generate** *Gen*(*source*,*dest*): Both These two structures are simple *statements* with three *parameters*: a *string* with the name of the extractor or the generator (*Ext* or *Gen*), the source *product_type*, and the destination *product_type*.
- **define**(*P1*,*fct*(*P2*,...)): This simple *statement* is in relation with a single *parameter* – the set type (*prod_set_type* is-a *product_type*) – and with a single *expression*, the *type* of which is *fct*, having a series of relations with *parameters* to store *P2*,...

B. The repository section concerning histories

The new part of the repository for storing the complex histories is shown in Figure 10.6. Most non-shaded classes reduced to their title (*method*, *statement*, *expression*, *text_type*, *schema_type*, *prod_set_type*) are those of the method part described above.

a. Processes

The *process* class is the kernel of all the process definitions. It is identified by its *name*, its *start_date* and *start_time* altogether. The *end-date* and *end_time* are optional fields because they cannot have a value while the process is not finished. The boolean property *in_progress* is initialised to *true* when a process is created and is changed to *false* when the process finishes. It can be set back to *true* if an analyst decides to continue the process. Hence, the fact that *in_progress* is *true*, and that *end_date* and *end_time* have a value shows the process was stopped and is now continuing.

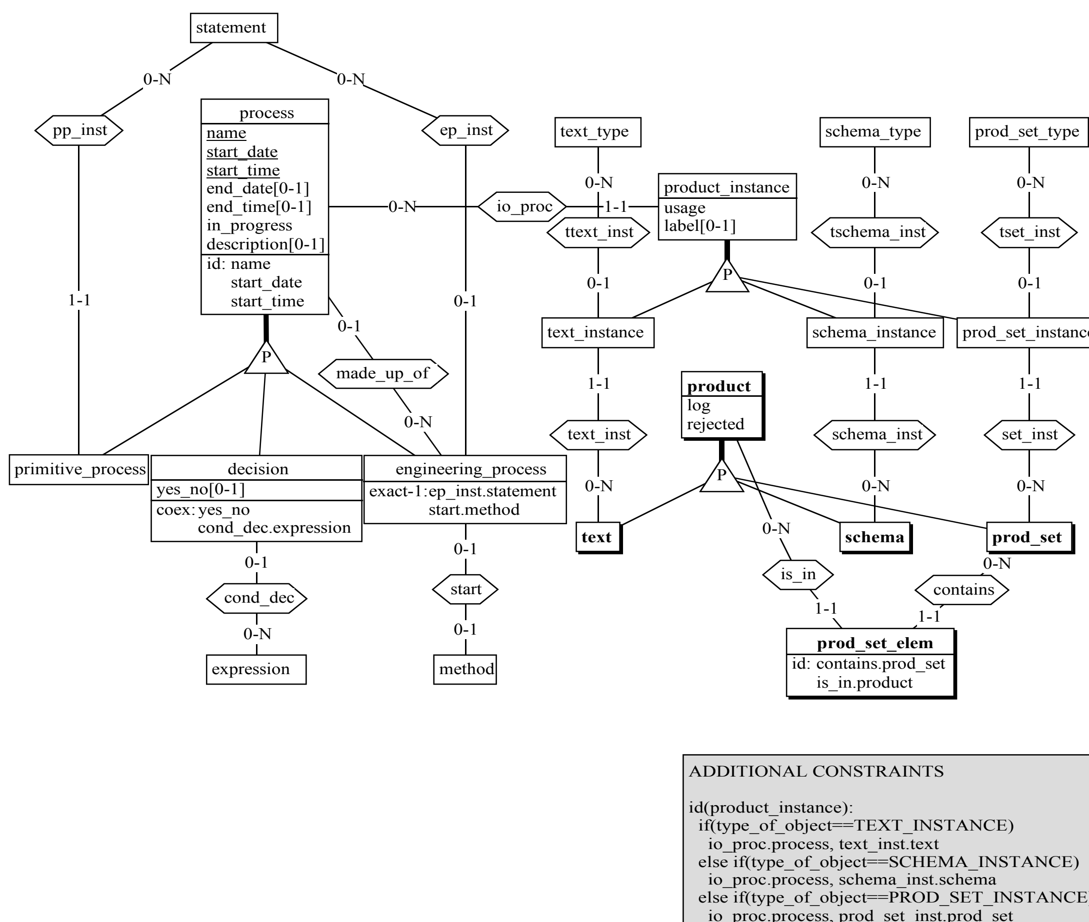


Figure 10.6 The repository section for storing histories

b. Primitive processes

An instance of the *primitive_process* class is always in relation with a statement in order to determine its characteristics. For instance, a schema copy primitive process must be in relation with a *statement* whose *type* is *copy*. During a project supported by a method, each execution of a primitive process type gives birth to a *primitive_process* instance. For the execution of a method-free project, in which there is no declared statement, two solutions can be implemented.

- A few classical *statement* instances (with type *new*, *copy*, *generate*,...) can be created by default at project start and all primitive processes can be connected to them. That way, all the copy primitive processes are connected to a single *copy* statement, and so on for all other primitive processes.
- A new *statement* is created automatically when a new *primitive_process* is created.

The first implementation has the advantage that the method part of the repository does not evolve during the project life. But it has the disadvantage that the loading of the project in a newer version of the CASE environment may make its functions unavailable since they have no corresponding statement. The second solution has the advantages of the ease of programming and of the independence of the project from the CASE environment release, but the inconvenience of the evolution of the method part of the repository. It can also be noted that the first solution has a fixed number of *statement* instances while this number will grow in the second solution, but the number of *primitive_instance* will always keep rather small, so this fact does not need to be taken into account.

c. Engineering processes

An *engineering_process* is an *ep_inst* instance of a *do* statement, except the root process of the history which is directly linked to the *method* itself by the *start* rel-type. Indeed, every engineering process type is used by a (possibly several) *do* statement, which is part of the strategy of another engineering process type, except the root process type which is never used by another engineering process type and which is declared as such in the *method* block of the MDL method description. An *engineering_process* is also linked to all its sub-processes through the *made_up_of* rel-type.

d. Decisions

A decision can be of two different kinds:

- The condition of an *if...then...else*, a *while*, or an *until* statement, when evaluated, gives a result (true/false, yes/no) which is the first kind of decision. In this case, the decision is linked through the *cond_dec* rel-type to the *expression* of the condition and the *yes_no* field stores the result of the expression evaluation.
- The decision to keep the best version(s) of a schema after several hypotheses had been made are totally independent of the method. They will never play a role in the *cond_dec* rel-type, and the *yes_no* attribute is useless. The decision is indicated by the *products* linked to the *decision*. The boolean property *rejected* of each *product* instance, which is initialised to *false*, is set to *true* for all the rejected product versions. Furthermore, the rejected product versions are stored as input products of the decision process, as explained below, and selected products are stored as update products of the decision process.

e. Products

Products are the main elements of the CASE tool. They are part of the original DB-MAIN repository. A *product* can be specialised as a *schema*, a *text*, or a *product_set*. The *product_sets*

can contain several *products* and a *product* can be in several *product_sets*. So, the *prod_set_elem* class is simply an implementation of a many-to-many rel-type.

In a method driven project, a product is created with a certain type by a process, then it can be passed to another process and be of another local type for this second process, and so on with several processes. This leads to a many-to-many-to-many ternary rel-type between a *process*, a *product_type* and a *product*. In a method-free project, no product type is defined and the rel-types are only between a *process* and a *product*. The two kinds of rel-types are implemented with the *product_instance* class, which must be either a *schema_instance*, a *text_instance* or a *prod_set_instance* class. The *usage* property specifies how the product is used by the process: in *input*, in *update*, in *output*, in *intern*, or as a *set-element* in a set passed in input or in update. The last usage is explained by the fact that if a process P passes a set S in input to another process P', then S itself is the input product, and all its elements I-compatible with the input product type must be accessible too without being real input products. If S is passed in update, the reasoning is the same.

As explained in Chapter 9, Section 9.2.7, the log file of a primitive process can be sliced so that all log entries modifying the same product are put together in their own slice. In the repository, all the slices concerning a same product are stored in a single log file, each slice being prefixed by an identifying label. This log file is a property of the product itself. The labels are used to initialise the *product_instance* property *label*. Indeed, a process can modify several products at the same time, but a product cannot be modified by several processes at the same time. So, if the log file is attached to the process, the extraction of the slice concerning a given product is rather complicated since it is necessary to browse all the log file, to examine every entry and to extract only entries concerning the right product. When the log file is attached to the product, extracting a slice is as simple as extracting a chunk enclosed between two labels.

For example, the "LIBRARY/Conceptual" product in Figure 6.8[24] will be stored in the repository as shown in Figure 10.7: it has an instance in each process in which it is generated or used, and each instance has its own type with a particular usage. Each *schema_inst* object has a *label* value. The "LIBRARY/Conceptual" product log file must look like this:

```
*POT "L1"
*POT "L2"
... (Schema analysis and design actions)
*POT "L3"
... (Schema normalisation transformations)
*POT "L4"
*POT "L5"
```

where *POT* is the reserved keyword to indicate a label (a POinT in the log file). The first one, "L1", is appended when the product is created by the "New schema" primitive process. The label "L2" begins the section dedicated to the primitive process "Analysis". In fact, the full "Analysis" log file is made of this section, plus all the sections attached to other products modified by the same process, but "library.txt" is the only other product used by the process, in input, so read-only. Then comes the "L3" label indicating both the end of the "Analysis" section and the beginning of the new one, for the "Normalisation" process. When the "Conceptual Analysis" engineering process ends, a *schema_inst* object is created to use "LIBRARY/Conceptual" as the output product of "Conceptual Analysis", and another *schema_inst* object is created to make the product an internal product of the "LIBRARY" engineering process. These *schema_inst* objects have no label because they are created only at the beginning of a process. When the "Logical Process" begins, the label "L4" is added in the same way. "L4" is immediately followed by "L5" marking the "Schema

24 see Chapter 6.

copy" process. Note that the schema copy being an automatic process type, no entry is appended to the log file during its execution.

C. The whole repository extension

The assembling of the two parts of the repository presented above is straightforward since the classes that have the same name in both schemas are the same classes: *statement*, *text_type*, *schema_type*, *prod_set_type*, *expression* and *method* classes.

When a project supported by a method starts, the method is stored in the repository as explained above and the history which follows this method is attached to it. For a method-free project, a simple default method built in the CASE environment is automatically used. This allows all the constraints (for instance, the 1-1 role played by *primitive_process* in *pp_inst*, and the exact-1 constraint in *engineering_process*) to be satisfied. This default method is shown in Figure 10.8. Its storage in the repository is shown in Figure 10.9.

When the extension is appended to the full repository, all the classes actually inherit of the *generic_object* class. This common class, which is part of the original repository, is not shown on the drawings to avoid to overload them without adding useful information.

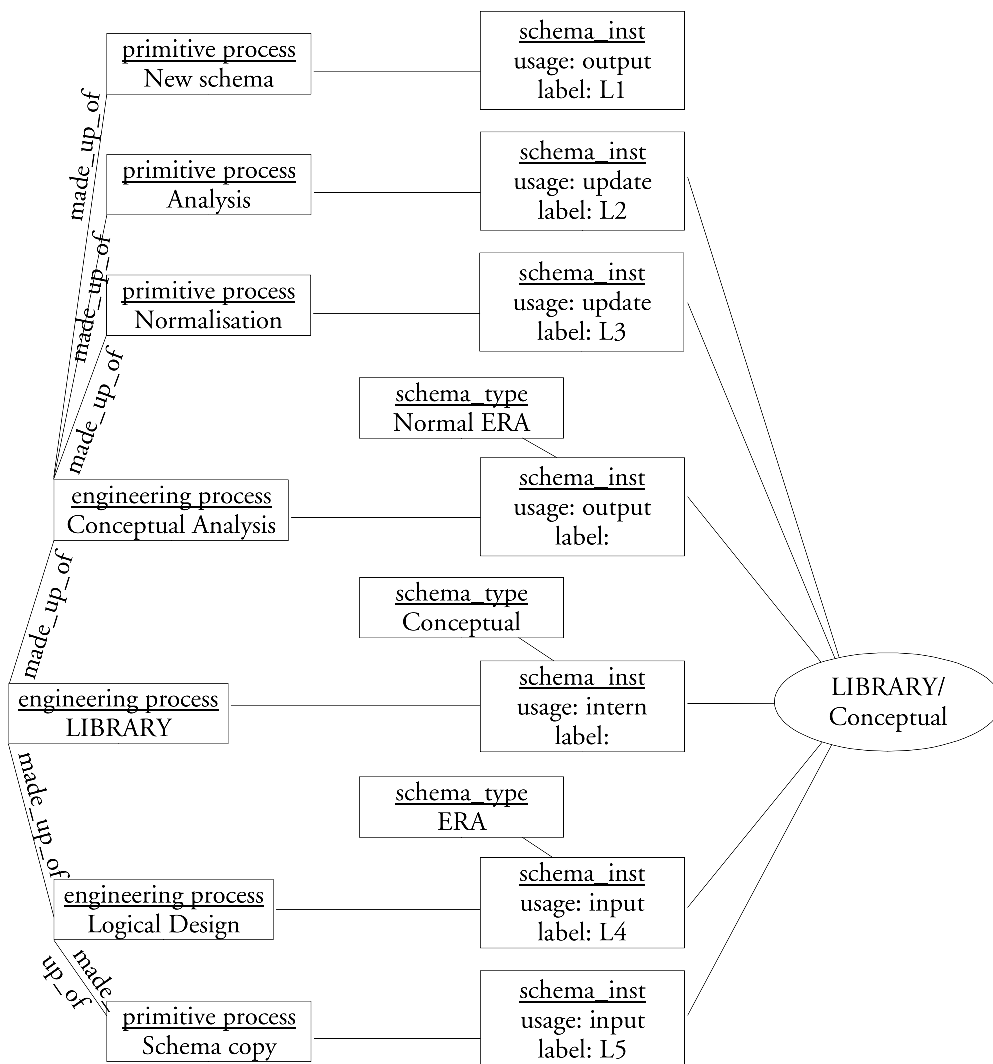


Figure 10.7 A schema in the repository

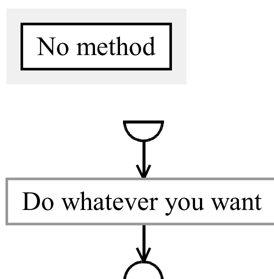


Figure 10.8 Default method for method-free projects

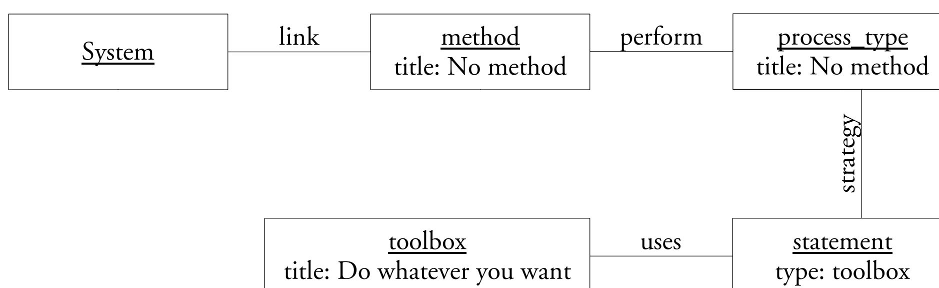


Figure 10.9 Default method storage in the repository

10.3. Parsing an MDL source file

An MDL source file is parsed with an LL(1) parser which uses a simple lexical analyser.

The lexical analyser reads one token in the source file each time it is called. A token is the largest sequence of characters which form a basic unit of the language. They can be:

- an *identifier*, beginning by a letter or symbol "_", and made of letters, figures and symbols "-" and "_"
- a *number*, made of figures only
- a *string*, made of any character enclosed between double quotes; to avoid confusion, a double quote inside a string must be prefixed by symbol "\", and this symbol itself must be doubled
- any other *character*.

Spaces, end-of-lines and comments (starting with symbol "%" and running to the end of the line) are simply ignored.

The parser works in a top-down fashion. The source file is read sequentially, token by token, one time. At every moment, the file has a *treated* first part and a *non-treated* second part. The first token of the second part is always read in advance and can be used to decide of the path the analysis has to follow. For instance, if the treated part of the file is the following:

```
... if (ask "Do you want an Oracle DDL?") then do GenerateOracle
```

the parser hopes to find either the end of the *if...then* structure or the second part of the *if...then...else* structure. Since the first token of the non-treated part is known, the parser can work in the following way:

```
if first-token="end-if" then
    do generate-if-then-structure-in-repository
else if first-token="else" then
    begin
        do parse-else-part
        go generate-if-then-else-structure-in-repository
    end
else
    error-message "syntax error"
```

The result of the parsing is stored in the repository.

For a more complete explanation about parsing texts written with LL(1) languages, [AHO, 89] is recommended.

10.4. The GUI

The DB-MAIN CASE environment is a C++ program designed for operating systems with a graphical user interface. In other words, it is build with an object oriented architecture and its GUI interface is event based. An **event** is anything that can happen outside an application that forces the application to react. For instance: a click on a mouse button, a mouse move, a press on a keyboard key, a move of a window which forces other windows to be redrawn,... Every window (product, history or dialogue box) is an object inheriting from objects of the programming environment libraries. All these objects have methods which are associated with events in order to be automatically executed when one event occurs. For instance, most dialogue boxes have an "OK" button. Such a dialogue box is managed by an object that contains an OK method. This method is associated with two events: a click with the mouse on the button or a press on the ENTER key.

This GUI will be modified in several steps:

1. The CASE tool must be able to load a method and to display it.
2. The history window must be extended in order to handle the complex histories.
3. The methodological engine must be implemented.
4. The GUI has to reflect the current state of the project.

10.4.1. Loading a method

The CASE environment must be able to load a method and to display it. The dialogue box associated to the *File/New project* menu entry must have a new field that allows the user to select a file generated by the MDL translator. When the method is loaded in the repository extension, a new window, the method window, must be created to show the method as presented in Chapter 4 and to allow users to browse it as presented in Chapter 9. The new method window can be created in the same way as other history and product windows. Its associated object can inherit of the same objects from the programming environment libraries as those windows. It must have its own methods for:

- drawing the method algorithms
- browsing through the method
- reporting functions (print and copy to clipboard).

10.4.2. History window extension

The history window must be extended in order to be able to show and to manage the com-

plex histories with all their views as presented in Chapter 6. The DB-MAIN CASE environment history window originally contains only products, no processes. These products already have a log file. The extension of this window consists in:

- expanding the history window to draw the processes stored in the repository extension and all the links between processes and products
- improving all existing methods of the history window associated object in order to take into account the processes
- expanding the history window to show the various views of the history
- adding all the methods needed to create, stop and continue processes
- adding all the methods that allow the analyst to take decisions
- adding all the methods needed to browse through the history

Furthermore, the engineering menu presented in Chapter 9 (Figure 9.9), which is aimed at starting the methods needed to create the complex history, can be implemented too.

10.4.3. The methodological engine

When both the method window and the history window are ready, the basic elements of the methodological engine can be implemented. A *methodological_engine* class has to be designed. When the CASE environment is started, one object of this class is created; it will be destroyed when the CASE environment is stopped, and there will never be a second instance. This class is responsible of the following activities:

- following of the method by managing the *state* property of all the statements in the method and calling the new methods of the history window for the creation and the termination of processes
- evaluating the conditions encountered in the control structures of the method
- checking the product validity when engineering process starts and when a primitive or an engineering process stops

So the methodological engine does not really have a direct impact on the GUI: it is hidden, only its actions can be seen. The methodological engine will be studied more precisely in Section 10.5.

10.4.4. The GUI look and feel

Finally, the GUI has to be updated in order to reflect the current state of the project. Two aspects have to be covered: the availability of the tools in the CASE environment, and the naming of every element of the GUI.

A. Availability of the tools

The availability of the tools in the CASE environment has to depend on the project progress. During a method-free project, every tool must be available at any moment[25]. During a method supported project, the availability of the tools has to depend on the use of the toolboxes: every product modification tool has to be disabled when no manual primitive process type instance is active, and only tools in the toolbox associated to such a process type must be available when one instance is pending. To do so, a complete list of the tools available in the CASE environment must be established, each tool must be to uniquely named, and this list must be made available to the MDL translator.

25 except when their own preconditions are not satisfied, but this is due to their intrinsic properties, not to the method, and this was already true in the original version of the CASE tool.

a. Menu entries

Each GUI element can be associated with one or several events and each event can be associated with at least one GUI element. Each event is associated with one method and a method can be associated with several events. In particular, each menu entry is associated with a *draw* event among others. This event occurs when the menu is shown. The class of the object associated with the windows containing that menu can contain a method which is executed in response to the event to specify how the menu entry has to be shown; this method can enable or disable the menu entry. So such a method has to be defined for every menu entry. This method must be aware of the current state of the project.

b. Tools in toolbars

The icons in toolbars can be linked to the same events as the menu items. So the methods designed for the management of the menus are automatically reused for the management of the toolbars too. The toolbars simply need to receive a *draw* event to be displayed correctly.

c. Keyboard shortcuts

Some keyboard shortcuts, as well as some clicks on a mouse button when the mouse points at some specific places can also be linked to the same events. To ease the use of the mouse, the CASE environment can be put in some special modes; for instance the *new entity-type mode* can be activated/disactivated in a schema graphical view when the *new entity* menu entry is selected, a new entity type being created each time the mouse button is pressed while this mode is active. These modes have to be taken into account too.

d. Dialogue boxes

Some dialogue boxes have to behave according to the project too: property boxes for product elements (for instance, an entity type property box) can have to allow analysts to modify the properties when the toolbox of an active manual process permits it, and not the remaining of the time. To do so, every property box associated class have to handle a *read-only* parameter which must be passed to them at opening time and which is used to initialise and to draw each dialogue element (edition zone, button,...).

e. Other events

Without responding to an event, a Voyager 2 program, some MDL primitive processes and the history replay function can also execute directly some functions of the CASE environment.

f. Tools inventory

To enable or disable the use of every function of the CASE environment by all the *activation means* presented above (menu, tool bar, keyboard shortcut, mouse click, dialogue box, Voyager 2 program, MDL method, replay function), a complete inventory of these functions must be prepared. This inventory has been made for the version 6.0 of DB-MAIN. It is shown below as a table, each line representing a function, each column representing an activation mean, each cell containing more information about the use of the activation mean of the column by the function of the line. A cell is left blank when the function cannot be used with the activation mean. Let us analyse a few functions and draw the inventory in the table below.

When DB-MAIN is started for the first time, the engineer selects the *file/new project* menu entry or the corresponding button in the *standard* tool bar. Both these events are connected to the *New project dialogue* function which opens the *Project properties* dialogue box. When

the engineer has entered some data, he or she clicks on the *OK* button, this starts the *Create project* function; *Create project* can also be started from the *Open project* dialogue box or by the *create(SYSTEM,...)* Voyager 2 function. When the engineer edits a schema, the standard *Copy* and *Paste* edition functions can be called with the *Edit* menu or with keyboard key combinations. In textual view, the engineer can draw a new entity type using the *New/New entity type* menu entry, using its corresponding button in the *standard* tool bar, or using the *New* button in the property box of another entity type. In a graphical view, the same three events simply put the CASE environment in the *New Entity Type Mode* (NETM), until one of the three is used again. While the NETM mode is active, each click of the mouse button in the middle of the schema window creates an entity type at the position pointed by the mouse by calling the *create entity type* function. It can also be called, in any mode, by selecting the *Edit/paste* menu entry, by its keyboard shortcut (CTRL+V), by the *integration* function, by a Voyager 2 program, or by the history *replay* function. The DB-MAIN CASE environment is transformation based; an example of transformation is the entity type into rel-type transformation which can be executed by several means too.

The *New project dialogue* functions just opens a dialogue box. The *create project* function cannot be performed from a toolbox, since it must be done only once before the toolboxes are available. The *Copy* function does not modify the product and so does not need to be disabled. But the *Paste* function can modify products by creating new elements, so it is worth to consider it as a tool that can be used or not in a toolbox, it will be named: *create*. This name is written in the right-most column of the table. Since the *create* tool works for any kind of product element, more restrictive tools can be defined as well: *create-entity-type*, *create-rel-type*, *create-attribute*,... In the same way, the creation of entity types in textual view or the entry in NETM mode in graphical view can be linked to the same *create-entity-type* tool or the more general *create* tool. Finally, the global transformation is itself a tool.

The same analysis has to be performed for all the functions available in the CASE environment. The result of the complete analysis is presented in Appendix E, while the reduced analysis above is summarised in the table in Figure 10.10.

Functions	Menu	Tool bar	KS	Mouse	Dialogue boxes	V2	Meth	Repl	Name
New project dialogue	File	Standard							
Create project					Project properties Open project	√			
Copy to clipboard	Edit		Ctrl+C						
Paste from clipboard	Edit		Ctrl+V						create create-entity-type create-rel-type create-attribute create-processing-unit create-role create-group create-collection
New entity type dialogue	New (TV)	Standard (TV)			ET properties (TV)				create-entity-type create
New entity type mode (=NETM)	New (GV)	Standard (GV)			ET properties (GV)				create-entity-type create
End new entity type mode	New Text standard Text compact Text extended Text sorted (NETM)	Standard (NETM)			ET properties (GV) (NETM)				
Create entity type	Edit/Paste		Ctrl+V	Left (NETM)	Schema integrate	√		√	
Entity type -> rel-type	Transform	Transfo			Global transfo. Adv. global transfo.			√	tf-ET-into-RT

Figure 10.10 An excerpt of the functions inventory

The naming of every element of the GUI has to be function of the project advancement. In a method-free project, as in the original methodology neutral version of the DB-MAIN CASE environment, every schema component has its GER-compliant name and every text component has its traditional text edition name. When a method supported project is active, all the concept names used in the GUI have to be the names defined in the model associated with the currently edited product. So, when a schema compliant with a relational model is being edited, the words “entity type” should appear nowhere and the word “table” should be used instead. The concerned GUI elements are: the menus, the status bar, the graphical tool bars, the dialogue boxes and every other window.

10.5. The methodological engine

The methodological engine is the new part of the CASE tool that guides database engineers during the whole performance of a project. It must:

- follow the method, that is to say:
  - manage the *state* of the statements in the method
  - execute automatic primitive process types
  - help the engineer in the use of manual primitive process types
  - automatically start new engineering processes and terminate them to help the engineer by doing tedious actions.
- evaluate expressions in conditional control structures (alternatives and loops)
- evaluate the conformity of products with their models.

This is why the engine naturally takes its place in the architecture where shown in Figure 10.1, just below the GUI to be able to control it entirely, with a direct link to the kernel to manage the *state* of the statements and to manage the history, and with direct links to the assistants and to the Voyager 2 abstract machine to execute automatic primitive processes and to check product validity.

10.5.1. Following a method

A. Management of the state of process types and control structures

Chapter 9 showed that every process type and control structure in a method can have several states according to the state charts in Figure 9.13, Figure 9.14, Figure 9.15, Figure 9.16, and Figure 9.17. The methodological engine has the responsibility of managing all the state charts individually and altogether. Indeed, a global coherence must be preserved. For example, a process type cannot be in the *done* state if some of its sub-process types are in the *running* state. The methodological engine will have to check the following rules whenever it has to perform an action:

- a process type or a control structure can only be put in the *allowed*, *running* or *body-running* state if its father (the process type or control structure that encompasses it) is in the *running* state or in the *body-running* state respectively
- the interrelations between a control structure and the components of its bodies should follow the requirements described in Chapter 9, Section 9.2.3.

Hence, the methodological engine needs:

- a function that checks the current state of the project in order to validate a transition in the state chart of a process type or a control structure according to the state of its father, according to its nature, and according to the state of some of its siblings

- a procedure that manages the transitions in various state charts according to the control structures in use.

B. Performing automatic process type

When a database engineer decides of the performance of an automatic primitive process type in the *allowed* state, not only the methodological engine has to change the state of the process type to *running* according to its state chart, but it can also order the correct tool to perform the job by simulating the actions (objects selection in project window and menu entries selection) the engineer should normally do to start the same job.

For example, when a "generate STD_SQL(PhysLibrary,LibraryScript)" primitive process type is ordered through the method window, the methodological engine will:

- put this *generate* process type in the *running* state
- select one product of type *PhysLibrary* in the project window
- select the *File/Generate/Standard SQL* menu entry that will perform the generation process on the selected product
- perform the two last operations again with each other product of type *PhysLibrary*.
- put the *generate* process type in the *done* state.

Each automatic primitive process type (see Chapter 5 for the complete list) has its own way of working, as depicted above for the *generate* process type. Doing the same analysis in detail for each of them is rather long and not of great interest here because it is very technical and dependent on the implementation of the CASE tool. It is a simple interfacing problem between the methodological engine and the CASE tool functions.

C. Performing manual process types

When a manual primitive process type state is set to *running*, a first process must always be created. Other instances are only created when a database engineer decides to perform several versions of the process. So the creation of the first process of a given type is a mandatory action, while the creation of the others is subject to the decision of the database engineers. To minimise the handling, the methodological engine can start the first new process by itself: it will emulate the selection of the *Engineering/ Use primitives* menu entry.

When the manual primitive process is performed, the database engineer is the only one who can decide of its end. The methodological engine has to leave him or her the responsibility of terminating the use of the primitive process type, so it can do nothing.

The most important effect of starting a new manual primitive process is the availability of the tools to the engineer according to the method. But, as explained in Chapter 9, section 9.2.3, the GUI can manage this without the need of the methodological engine since all the necessary information are directly accessible in the repository.

D. Starting and ending engineering processes

Similarly to the treatment of manual primitive process types, a first engineering process can be started automatically by the methodological engine when an engineering process type state chart transit from *allowed* to *running*. Other processes of the same type can be started manually when needed only.

When the strategy of the the engineering process type reaches its end, the process should be ended too. The methodological engine can do it by emulating the selection of the *End engineering process* entry in the *Engineering* menu.

10.5.2. Product and expression evaluation

A strategy can contain some control structures that require the evaluation of an expression: *if...then*, *while*, *until*, *for*. These expressions can be of several types:

- A simple question asked to the database engineer: **ask** "*question*". A dialogue box showing the question and two buttons labelled "yes" and "no" suffices to wait for the engineer's answer.
- An external function evaluation: **external** *function*(*parameters*,...). A simple call to the module executing external functions will do the job.
- A product set evaluation functions: **count-greater** (*product*-set, *nb*) for example. To evaluate such functions, the methodological engine requires a function that counts the numbers of elements in a set. The use of a traditional mathematical comparison operator (> in the example above) will do the remaining.
- A schema analysis function: **exists** (*schema-type-or-set*, *schema-analysis-rule*). This functions looks a schema for constructs that satisfy a structural rule. The methodological engine requires a complete schema analysis expression evaluation engine that must be able to evaluate all the predicates listed in Appendix A, possibly more, as well as more complex expressions (using *and*, *or*, *not* operators) made of these predicates. Such an evaluation engine is rather long to write (a lot of predicates) but rather simple: it consists in browsing through the CASE tool repository.
- A model evaluation function: **model** (*peroduct-set*, *product-model*). A model being made of several structural rules, this function has to look a schema for constructs that violate one or more of the structural rules. The methodological engine can use the same schema analysis expression evaluation engine as above to do so. The only two differences in the use of this engine are the following: (1) only one rule to evaluate with the *exists* function, several rules to evaluate with the *model* function, a schema being compliant with the model if and only if all the rules are satisfied; (2) the *exists* function looks for constructs that satisfy a rule, while the *model* function looks for constructs that violate a rule, one result being the negation of the other.

The product compliance evaluation that is required when a process ends (see Chapter 4, Section 4.2.2) is similar to the model evaluation function.

Chapter 11

Case studies

This chapter presents two case studies:

- The first one concerns a simple forward engineering process applied to a small library information system. This elementary case study shows how to define a simple method and to use the DB-MAIN CASE environment with the method step by step. This is a straightforward and imperative method without subtleties and difficult decisions to be taken by engineers.
- The second one is an excerpt of a complex reverse engineering process. Its aim is to show how a method can be used only to help and guide an engineer while giving him maximum freedom. The engineer has to make hypotheses and to take decisions.

11.1. First case study: a simple forward engineering project

The small case study concerns a library. It contains books that can be borrowed. The database is aimed at registering the books of the library, the borrowers and their borrowings. Its complete definition was given during an interview. The interview report will be used to start the project. During the design, this schema will be transformed into a relational schema and an SQL DDL script will be generated. In a first time, Madam method engineer defines a method to help engineers to conduct this kind of project. In a second time, Mister database engineer uses this method to perform the project.

11.1.1. Defining the method

The MDL development environment is started. The method engineer starts a new method using the *New* item in the *File* menu. A new blank text editor window appears on the screen and the engineer can start designing the new method. The whole method is listed in Appendix F.

A. Defining the product models

In a first step the method engineer makes an inventory of the products the database engineers will have at their disposal and what they will have to produce:

- The requirements of the projects database engineers will have to use as the starting point of their work are texts stored in *.txt* files.
- Database engineers will have to produce a conceptual schema which is a formal image of the requirements they will receive, a relational schema which is a semantically equivalent translation of the conceptual schema, a physical schema which expresses the relational structures according to a particular RDBMS, and an SQL DDL script for creating the database within the DBMS.

She can now define product models to represent all these products. She defines the "TEXT_FILE" model with a ".txt" extension for the requirements and the SQL_FILE model with a ".ddl" extension for the SQL DDL scripts. Then she defines the "CONCEPT_SCHEMA" model. She first gives the model the more readable name "Conceptual schema model". She adds a small description telling the purpose of the model that will appear on the users' screen when they need some help. Then she decides what concepts of the GER model have to be present in the conceptual model and what name they will have; she decides to keep every concepts but those that have a physical or navigational aspect – collection, referential constraint, inverse constraint, access key – as well as those that are process oriented – object, processing units, call, decomposition or in-out relations – and to keep their names. Finally, she declares a series (25) of constraints on the model: for instance, a conceptual schema must have at least one entity-type, each of them must have at least one attribute.

In the same way, she defines the "LOG_SQL_SCHEMA" and the "PHYS_SQL_SCHEMA" models, intended to specify relational logical schemas and physical SQL schemas respectively. The complete product models are in the listing in Appendix F.

B. Declaring process types

When the product models are defined, the method engineer can describe how to transform them through the process types. Since she knows what products will be available to database engineers and what products they will have to produce, she will work in a top-down fashion: she will begin by describing the root process, whose strategy decomposes the whole work in several main phases; then she will describe these phases by decomposing them into smaller tasks and so on.

a. The main process type

The main process type, named "FORWARD_ENGINEERING", is a simple sequence of main phases showing how to produce all the required products: after collecting all the interview reports at his disposal, the database engineer will have to analyse and translate them in a conceptual schema, then he will have to transform the conceptual schema into a logical one, to update it into a physical schema, and finally code the SQL DDL script. Its listing is in Appendix F and its graphical representation is shown in Figure 11.1.

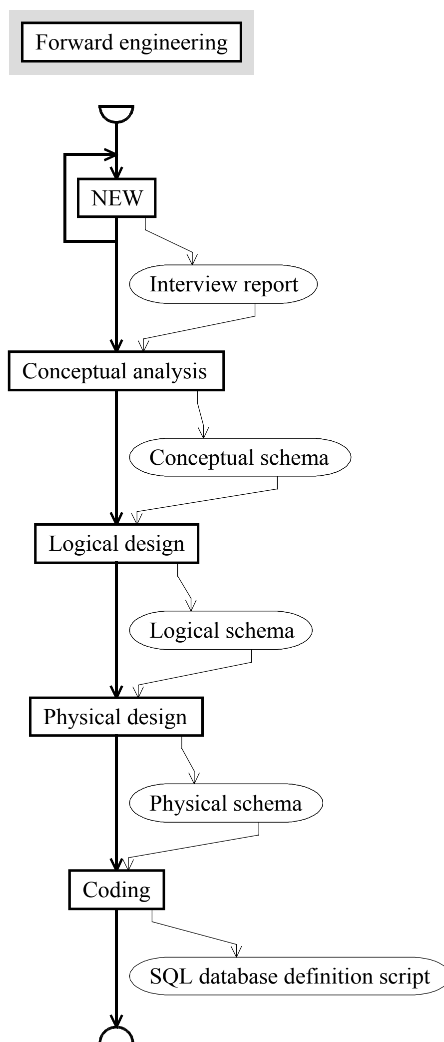


Figure 11.1 The main process of the method

b. The conceptual analysis phase

The conceptual analysis phase uses the interview reports as an input product and has to provide an output conceptual schema. It can be realised in three steps: preparing a blank working sheet, drawing a raw conceptual schema, and refining the conceptual schema. The first step is a standard one build in every CASE environment. The second step is a human task: the interview reports must be analysed and translated into a schema. To let human engineers work, the CASE environment has to provide them with tools in a toolbox. This toolbox, named "TB_ANALYSIS" contains tools for creating, modifying and deleting entity types, rel-types, attributes, roles and groups. Finally, the third step of the conceptual analysis phase is manual too. It uses the toolbox "TB_CONCEPTUAL_NORMALISATION" which provides tools for modifying and transforming the components of the schema drawn in the second step. It can be noted that a log file does not need to be

recorded for the analysis process, but it is needed during the conceptual normalisation. Indeed, since the drawing step starts with a blank sheet, the final schema suffices by itself to know what is done; the fact that a given entity type was drawn before or after another one is not important. But the method engineer wants the database engineers to be able to remember what normalisation transformations they performed during the third step. The "CONCEPTUAL_ANALYSIS" process type is shown in Figure 11.2. It must be noted that the "CONCEPTUAL_ANALYSIS" process type has to be placed before the "FOR-WARD_ ENGINEERING" description to avoid forward referencing that is forbidden in the MDL language.

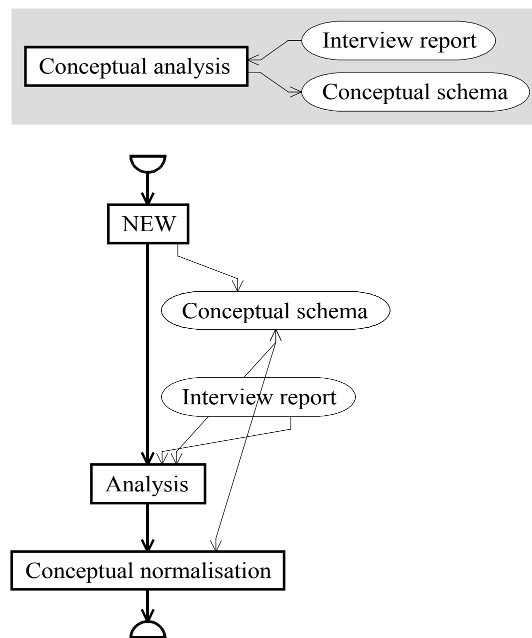


Figure 11.2 Conceptual analysis strategy

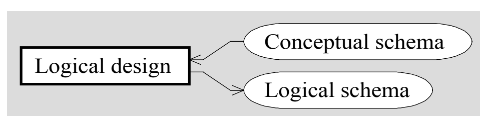
c. The logical design phase

The logical design phase uses the conceptual schema and produces a logical schema. This will be performed in two steps. In the first one the conceptual schema will be roughly converted automatically by global transformations of the CASE environment. They are grouped in a sub-process type named "RELATIONAL_TRANSLATION", itself divided in five simpler steps:

1. a global transformation for transforming all is-a relations into rel-types
2. three transformations for transforming complex rel-types as well as rel-types with multiple entity types roles and non-binary rel-types into many-to-one binary rel-types
3. transformations for flattening entity types that will be performed several times until a fix point is reached, that is to say while there remains compound or multivalued attributes
4. the fourth step prepares the job of the fifth one by adding some technical identifiers to entity types that need one
5. transformation of all the rel-types into referential attributes and constraints.

Since the conceptual schema is one of the final products of the project, it cannot be modified, so the logical design has to start by doing a copy of the schema, and it has to work on that copy. The method engineer decides that the raw logical schema will also be copied before the second step of the logical design in order to keep a trace of the intermediate state in the history. This second step of the logical design is a human activity aimed at cleaning

the logical schema that can be realised with the “TB_NAME_CONVERSION” toolbox. The graphical representation of the “LOGICAL_DESIGN” phase is shown in Figure 11.3 and its “RELATIONAL_TRANSLATION” in Figure 11.4.



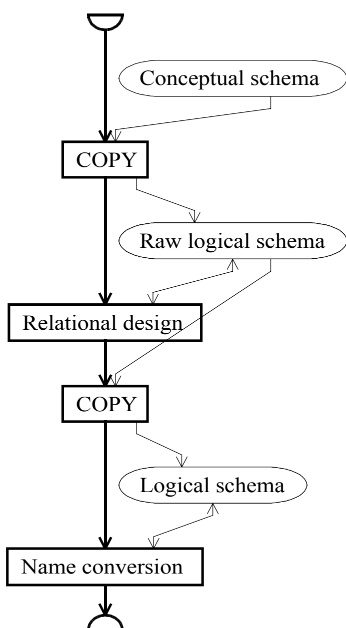
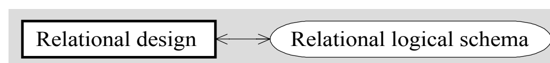


Figure 11.3 The logical design strategy



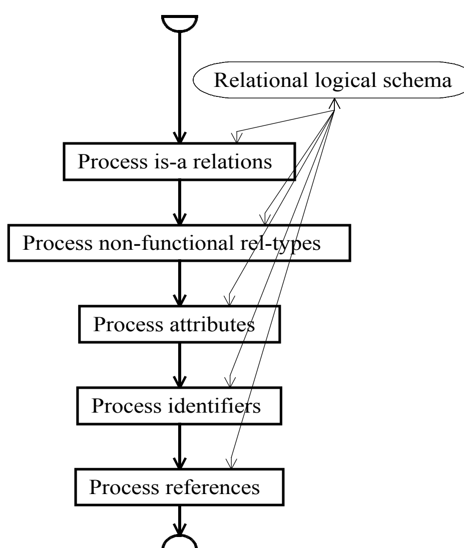


Figure 11.4 The relational design strategy

d. The physical design phase

The physical design phase (Figure 11.5), named “PHYSICAL_DESIGN”, begins by copying the logical schema to preserve it. The first task consists in setting access keys on primary keys and foreign keys that deserve it. Since simple rules exist to perform this task, it is done automatically by global transformations. The second task is aimed at distributing the tables (entity types are renamed tables in a relational schema) among files. It could be possible to automatically put all the tables in a single file or to put each table in its own file, but the method engineer prefers to permit the database engineers to decide on a better distribution, for instance, by grouping two relevant tables in one file and three other relevant tables in a second file. So she declares the “TB_STORAGE_ ALLOCATION” toolbox.

e. The coding phase

Finally, the phase named “CODING” (Figure 11.6) allows the CASE environment to automatically generate an SQL DDL script. This coding can be prepared by database engineers by copying the schema and adding some properties to the table description that can be understood by the DDL generator. So, the method engineer declares the “TB_SETTING_PARAMETERS” toolbox and plan its use just before the generation.

C. Declaring the method

The only remaining task for the method engineer is to specify some properties for the method itself in the mandatory *method* paragraph. The most important characteristic of this paragraph is certainly the *perform* clause that specifies that the “FORWARD_ ENGINEERING” process type is the root process type.

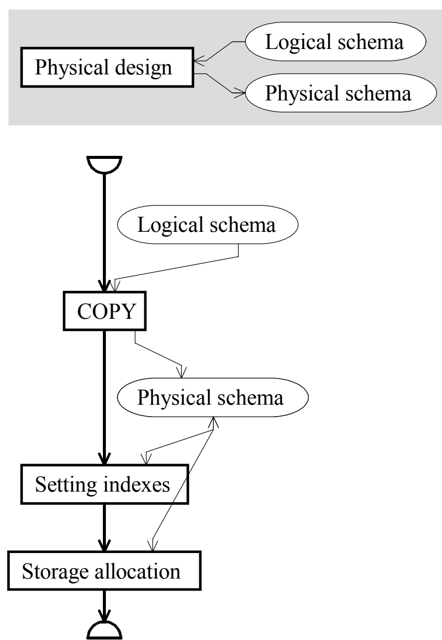


Figure 11.5 Physical design strategy

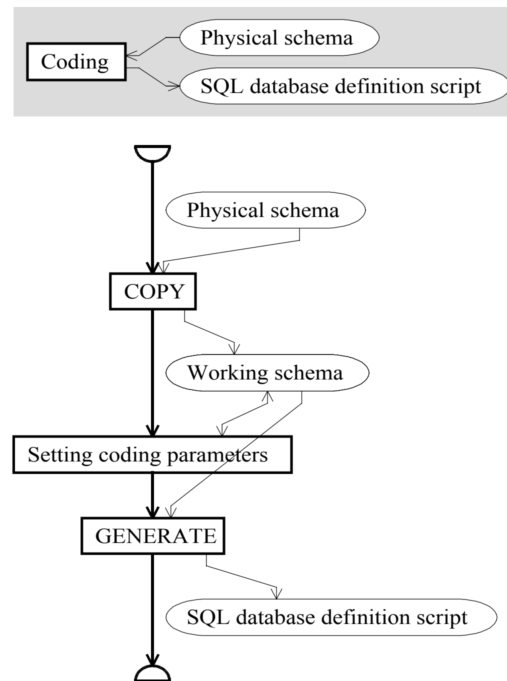


Figure 11.6 The coding process strategy

D. Compiling the method

When the method definition is finished (see listing in Appendix F), the engineer saves it and compiles it using the menu item **MDL/Compile**. A new window appears on the screen with the graphical representation of the root process shown in Figure 11.1. The engineer can then browse through the method to check her job.

Finally, she generates a *.lum* file with the compiled version using the menu item **File/Generate LUM**. This file can be distributed to database engineers who can use it to perform new projects.

11.1.2. Performing the project

A. Starting the new project

The DB-MAIN CASE environment is started, its workspace is blank. The analyst creates a new project that he will fulfil using the “forward.lum” method defined above.

When the project is created, the project window is opened, and, on top of it, another window containing the root process of the method (Figure 11.1) displayed in a graphical way, with the “New” process type shown in the *allowed* state (Figure 11.7).

The analyst executes a process of type “New”, using the **Execute** item of the contextual menu of this process type (Figure 11.8), to add an interview report (“library.txt”) to the project. In Figure 11.8, the “New text” process has been created, and the “library.txt” text has been added to the history. An arrow shows that the text is the output of the process.

In the method window, the “New” process type is in the *allowed* state, and a second one, “Conceptual analysis”, too. It means that the engineer can choose either to collect as many interview reports as he wants, or to proceed with the conceptual analysis of these reports. It is to be noticed that, during the execution of the “New text” process, the “New” process type was in the *running* state, shown with the associated colour.

In this example, the engineer will work on a single text.

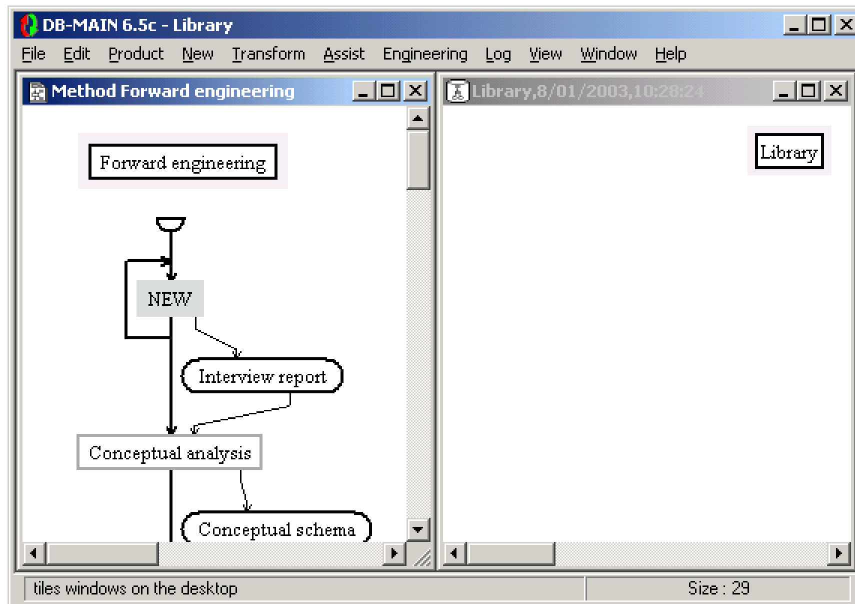


Figure 11.7 The project has just been created and the windows tiled. The method window shows what process can be executed, and the history window is empty.

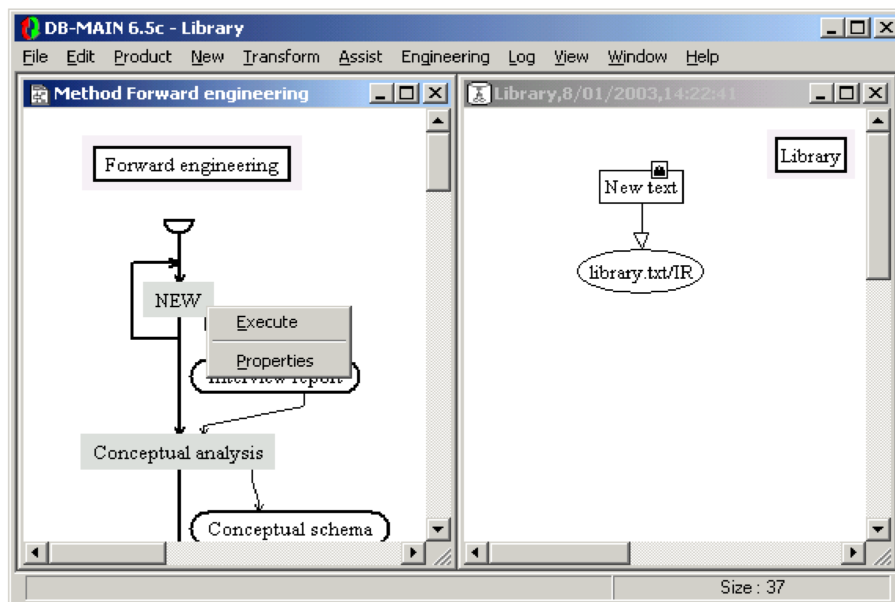


Figure 11.8 The first process is executed: a new text is added to the history. The analyst has now the possibility to perform a process of two different types: either add another new text or go on with the conceptual analysis. The contextual menu of the "New" process type is shown using the right mouse button.

B. Performing the conceptual analysis

The interview report must be analysed in order to draw the conceptual schema of the library management system. The analyst starts a new engineering process of type "Conceptual analysis". The content of the method window changes. It now shows the strategy of the "Conceptual analysis" process type (Figure 11.2 and Figure 11.9). The project window has changed in the same way, a "Conceptual analysis" engineering process has been created, and the window shows it. By opening the process hierarchy window (menu **Window/ Process hierarchy**), the analyst can see that Conceptual analysis is a sub-process of Library. The hierarchy window can be used to browse through the history.

The engineer can now start the conceptual analysis by creating a new schema that will be used as the drawing board. He calls this schema "Library/Conceptual". On this drawing board, he will introduce the conceptual schema of the library management system during the analysis process

The primitive process "Analysis" must be performed using a toolbox. By double clicking on the "Analysis" process type in the method window, the engineer can see what tools are available in this toolbox (Figure 11.10). They allow him to create and edit entity types, relationship types, attributes, roles and groups in the schema. The analyst has to open the interview report and the blank schema, and to fill it by creating the conceptual schema of the database by its own on the basis of the interview report, which is shown in Appendix F. Figure 11.11 shows the two products in use, as well as the **New** menu whose only available items are **Entity type...**, **Rel-type...**, and **Role/Rel-type...** according to the toolbox in use.

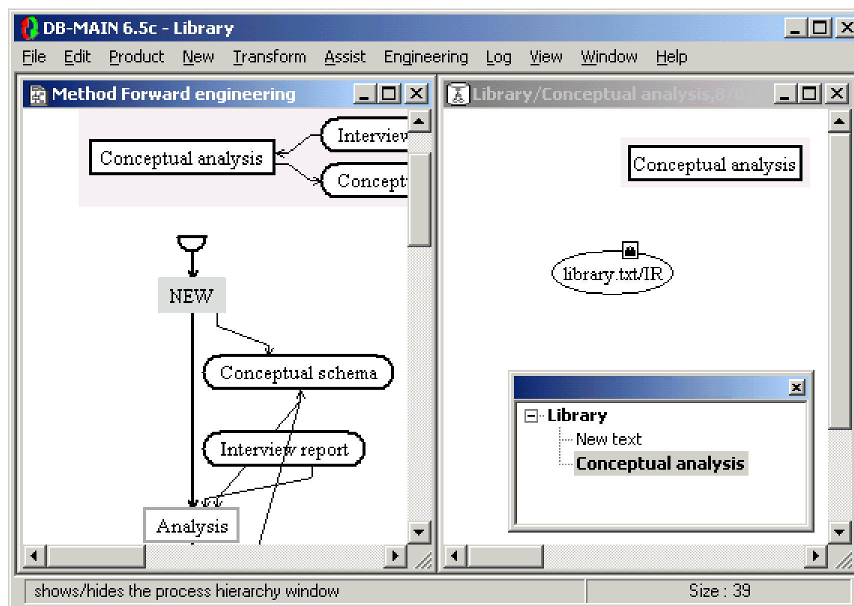


Figure 11.9 Beginning of the conceptual analysis process. The method window now shows the strategy to follow for the new process and the history window shows the new engineering process which contains one input product. The hierarchy window shows that the new engineering process is a child of the root process.

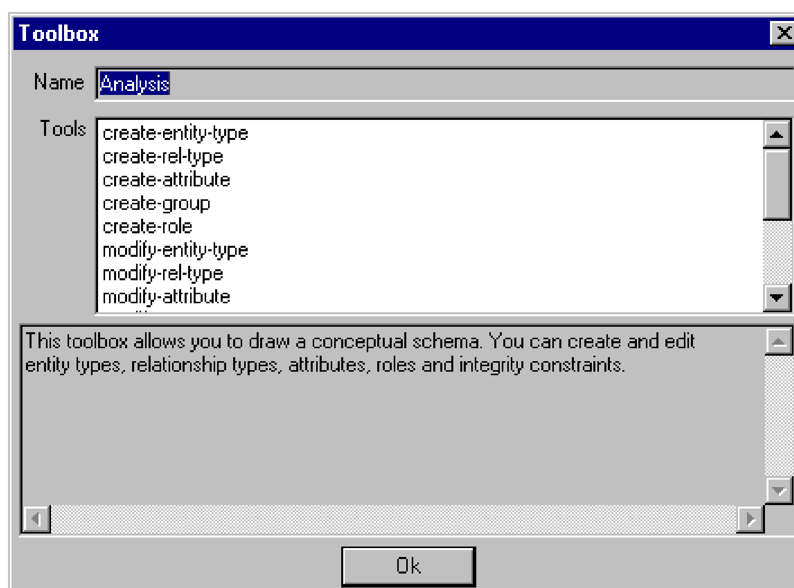


Figure 11.10 A toolbox for the conceptual analysis process

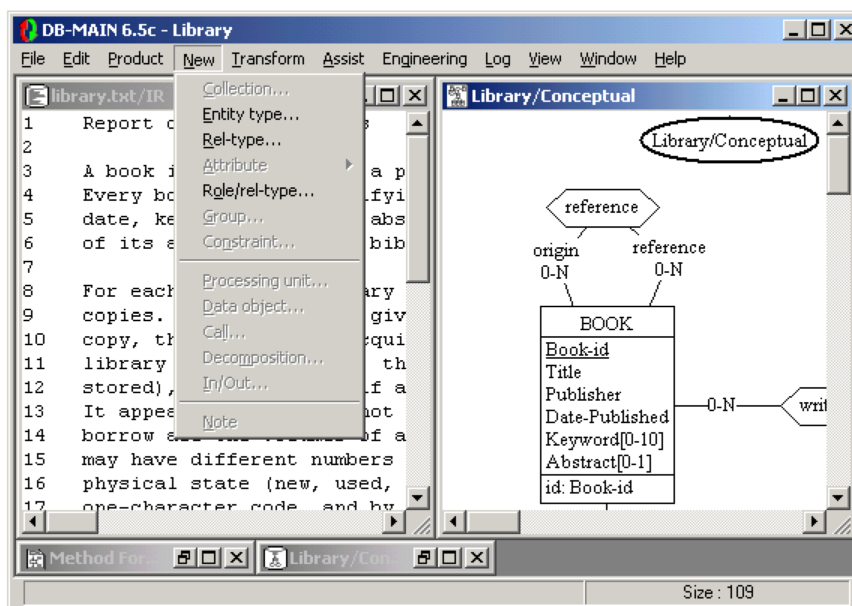


Figure 11.11The Analyst performs the Analysis primitive process, he draws the schema on the basis of the interview report. In the menu, only tools allowed by the toolbox are available.

When the engineer finishes the job, he has to signal it to the methodological engine using the **Engineering/End use of primitives** menu. Then the analyst signals he has finished with that process type using the **Terminate** item of its contextual menu.

The conceptual schema being introduced, it can be normalised. To know what this process means, the engineer double clicks on the "Conceptual normalisation" process type in the method window and reads its description. Then he starts the process, opens the schema and normalises it, and finally terminates the process.

The conceptual analysis is finished (see Figure 11.12). The CASE tool automatically terminates the process type: the CASE tool automatically performs the same action as the user could perform by selecting the menu entry **Engineering/End current process** with nothing selected in the project window. A dialogue box appears to allow the engineer to select output products, as shown in Figure 11.13. Since the process type specifies there should be conceptual schema(s) in output, and since there is only one conceptual schema in the project, this schema is automatically proposed in output. The engineer accepts this choice and terminates the use of the conceptual analysis process type.

Both the project window and the method window are back to their first view, the one of the root process, as shown in Figure 11.14.

C. Performing the logical design

In the method window, only the "Logical design" process type is now in the *allowed* state. The engineer executes it. The strategy of the logical process (Figure 11.3) appears in the method window. The engineer, according to the strategy of the process, copies the conceptual schema, naming the copy "Library/First logical", and starts an engineering process of the "Relational design" type.

The "Relational design" process type is a sequence of five primitive process types, as shown in Figure 11.4. The "New" and "Copy" process types are automatic basic primitive process types (see Chapter 2): the CASE environment knows by itself what to do. The "Analysis" process type met during the conceptual analysis was a manual primitive process type. The following ones are of a third kind: they are automatic configurable primitive process types. By double-clicking on them in the method window, one can see a script of transformations that were specified by the method engineer and that will be executed automatically by the

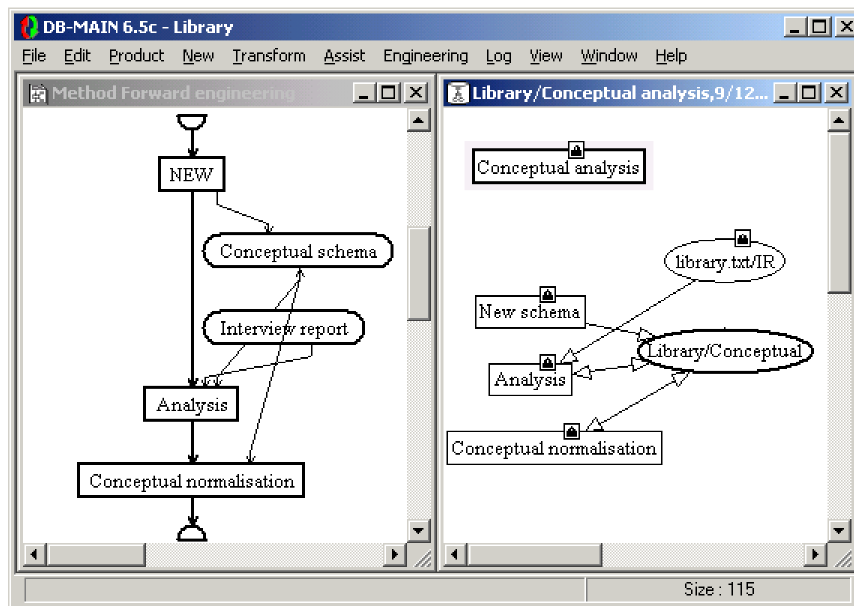


Figure 11.12 The conceptual analysis process is over

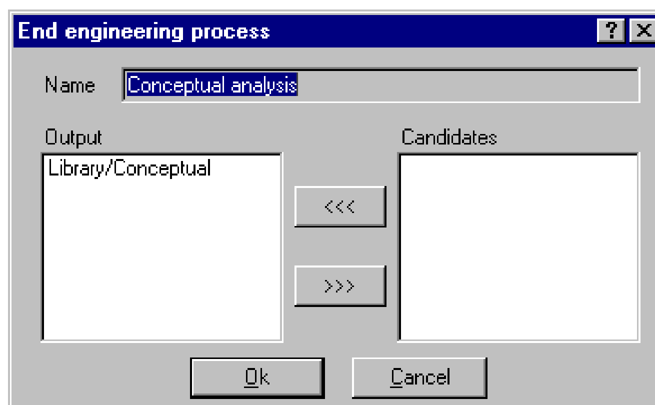


Figure 11.13The output product selection dialogue box

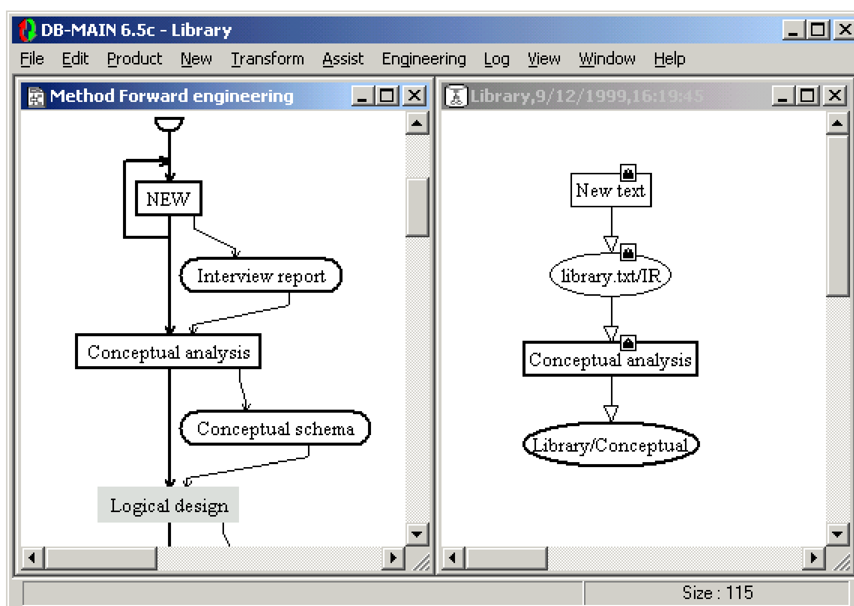


Figure 11.14 The "Conceptual analysis" engineering process is over. The "Library/Conceptual" schema is terminated and the method window proposes to the analyst to start the "Logical design" of the database.

CASE environment. For example, Figure 11.15 shows the definition of the "Non-functional rel-types" process type.

The engineer performs a process of each type in the order specified by the sequence, and terminates the relational design.

He goes on with the logical design by keeping a copy of the current state of the schema and transforming all the names in order for them to be compliant with the SQL standard.

The logical design is over and the CASE tool automatically terminates the process type: the schema "Library/Logical" is proposed in output, and the schema "Library/First logical " is put in the "candidates" list (see Figure 11.16), that is to say it is not proposed in output, but the user can decide to use it in output anyway. The engineer simply accepts the proposed solution.

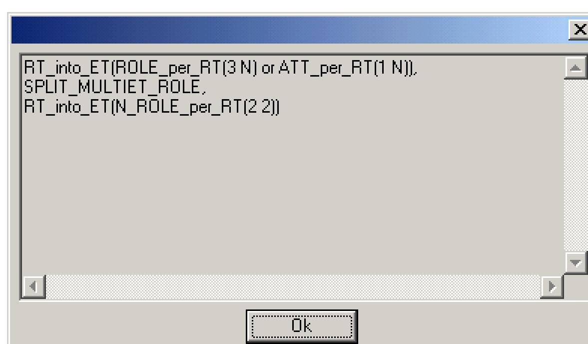


Figure 11.15 The definition of the "Non-functional rel-types" primitive process type.

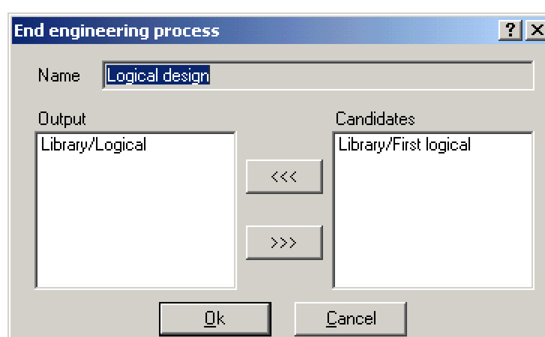


Figure 11.16 The logical design is terminated. One product is proposed in output. Another candidate is available, but it is left aside.

D. Performing the physical design

In the same way, the engineer can perform the physical design (see Figure 11.5) of the database.

After copying the input schema to "Library/Physical", a primitive process of an automatic configurable type creates indexes automatically where they are probably the most useful, that is to say on every primary and secondary keys and on every foreign keys, except the keys which are a prefix of another key.

A manual primitive process allows the database engineer to manually specify the database files to create and to distribute the tables among those files. He opens the physical schema, creates the two following collections and fills them:

- LIBRARY(AUTHOR,BOOK,COPY,KEYWORD,REFERENCE,WRITTEN)
- BORROWING(BORROWER,BORROWING,CLOSED_BORROWING,PHONE,

PROJECT)

Then the engineer closes the schema and the physical design is over. It is terminated automatically by the CASE tool with "Library/Physical" as proposed output product.

E. Performing the coding

Finally the coding phase (Figure 11.6) will generate the SQL-DDL script.

The engineers starts a new process of the "Coding" type and copies the input product to a working one. This last schema is an internal temporary schema aimed at preparing the coding. The technical descriptions of the components of this schema can be modified by introducing some coding parameters. They will be interpreted by the SQL generator. For instance, the technical description could specify, for each access key, if it must be implemented with a b-tree, or with hashing. Bothering with these optimisations will bring nothing interesting to this small case study, so this step can be skipped, assuming the default configuration will be all right for the SQL generator.

Finally, the SQL generator can be invoked.

Then the CASE tool automatically terminates the "Coding" process with "library.dll/1" as the proposed output product. Both the coding and the project are terminated.

11.1.3. The resulting history

When the project is over, the whole history of the job is recorded. It is possible to browse through it. Figure 11.17 shows the main tree of the history. Bold lines are engineering processes. Double clicking on one of them shows its graph in the history window. For instance, Figure 11.18 shows the *conceptual analysis* process. The three primitive processes beneath the *conceptual analysis* process in the tree (three non-bold lines) are shown as three rectangles in the graph of the process. The tree shows the order of their performance more clearly, while the graph shows the products involved.

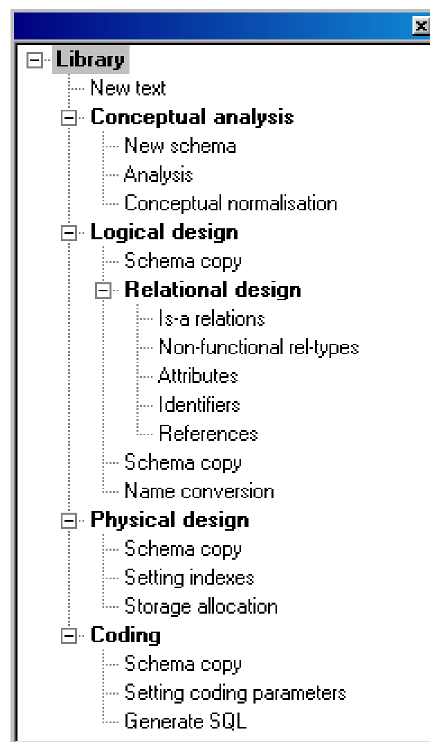


Figure 11.17 The history tree

The lines written with regular characters are primitive processes. Automatic processes, as well as manual processes whose type was declared with “[log off]” do not have a log file. Other primitive process log files can be opened with a text editor or treated with Voyager 2 user-written processors.

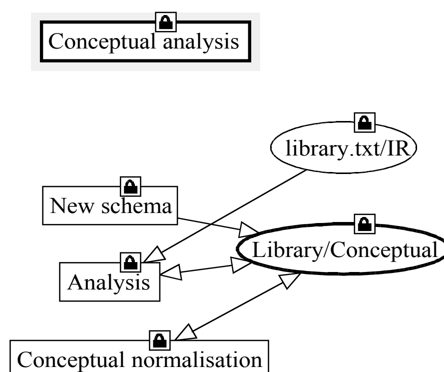


Figure 11.18 The conceptual analysis process

11.2. Second case study: a complex reverse engineering project

The second CASE study concerns a more complex job: the reverse engineering of a legacy database. This case study was formerly published in [HAINAUT,96d] which focuses on the reverse engineering aspects without using the methodological engine. This chapter will complete this case study. This project needs more intelligence than the first case study. Indeed, the latter was mainly a straightforward sequence of actions and contained a few transformations scripts which are automatic processes. In the second case study, the strategies are more complex, use more non-deterministic control structures, have less automatic processes, and often require much database engineers’ expertise. The use of the method sometimes needs so much intuition that engineers will have to try various options with different hypotheses, and to take decisions afterwards.

In the following, the method designed by Mister method engineer will be quickly described. Then, Madam database engineer will perform one part of a project using this method. Finally, the resulting history will be transformed.

11.2.1. Method description

A. The reverse engineering method

The aim of the reverse engineering process is to analyse a legacy database, made up of a collection of COBOL files, which is used in production for several years, which has evolved along the years, and for which the documentation is poor, erroneous, or even non-existent. This analysis should produce the following results:

- a detailed conceptual schema of the database
- a possible history of the original design
- a mapping between the components of the conceptual and the physical schemas.

From this list of goals, it can be deduced that the method needs to recognise a series of product models: COBOL programs, COBOL physical schemas, COBOL logical schemas and conceptual schemas. The mapping and the possible design history can be obtained by analysis and transformation of the reverse engineering project history.

In this case study, some schema models are defined in two layers using the inheritance mechanism: a logical schema model is defined to declare a few general properties which are

usual for logical schemas, then a COBOL logical schema model is defined by inheriting properties of the previous model and refining it. A physical schema model and a COBOL physical schema model are defined in the same way, as shown in Figure 11.19. If the method had to evolve in order to treat other kinds of databases (for instance, SQL databases), the new logical and physical model can be designed by inheriting the same LOG_SCHEMA and PHYS_SCHEMA models.

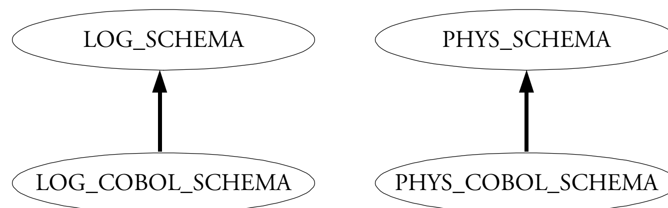


Figure 11.19 The model hierarchy

At the highest strategic level, the reverse engineering activity to model is made up of five phases:

1. COBOL program files collection.
2. Data structure extraction for retrieving the complete physical schema.
3. Schema cleaning, for retrieving a logical schema by removing all physical constructs.
4. Data structures de-optimisation and untranslation to recover the conceptual schema.
5. Conceptual normalisation to make the conceptual schema more compliant to some presentation rules, to make it more readable.

The second step is the most complex one, it requires a lot of human expertise. This case study mainly focuses on it. The complete method listing can be found in Appendix F.

B. The data structure extraction phase

The **Data Structure Extraction** process consists in recovering the logical schema of the database, including all the implicit and explicit structures and constraints. It mainly consists of three distinct sub-processes:

- *DDL text analysis*. A first-cut schema is produced through parsing the DDL texts or through extraction from data dictionaries.
- *Schema refinement*. This schema is then refined through specific analysis techniques [HAINAUT,96b] that search non-declarative sources of information for evidences of implicit constructs and constraints, that is many important constructs and constraints that are not explicitly declared, but rather are managed through procedural section, or even are left unmanaged. The analysts will recover structures such as field and record hierarchical structures, identifiers, foreign keys, concatened fields, multivalued fields, cardinalities and functional dependencies.
- *Schema integration*. If several schemas have been recovered, they have to be integrated. The output of this process is, for instance, a complete description of COBOL files and record types, with their fields and record keys (explicit structures), but also with all the foreign keys that have been recovered through program and data analysis (implicit structures).

a. The process type

The extraction of all the data structures can be performed according to the strategy shown

in Figure 11.20. A COBOL application generally comprises a collection of COBOL source code files. Some of them contain an *input-output section* in an *environment division* that specifies the files and their characteristics, and a *data division* which is a rough description of the data records. A record description is a list of fields, possibly decomposed in several levels, each field being characterised by a level, a name, a data type that can be undefined, and possibly an array sizing. There is no constraints on field values and no relations between records. The record identifier, or *record key*, is expressed in the *input-output section*. All other additional constraints are managed by the procedural part of the files, in the *procedure division*.

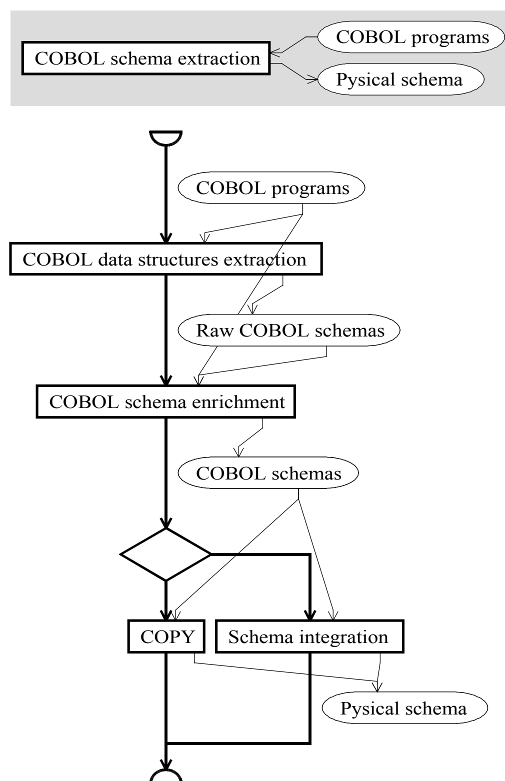


Figure 11.20 The data structure extraction strategy

The analysis of the *environment division* and the *data division* is a task that can be performed automatically. It will be handled by the "COBOL data structures extraction" process type which will extract a schema from each COBOL source file containing an *environment division* or a *data division*.

The analysis of the *procedure division* for enriching the extracted schemas is the most complex task which is managed by the "COBOL schema enrichment" process type.

When all the schemas are completed, they can be integrated by a process of the "Schema integration" type in order to provide a single physical schema of the whole system. The condition in the diamond is:

count-greater(COBOL_schema,1)

which indicates that an integration process is necessary only if more than one schema has been extracted, else, a simple copy of the schema suffices.

b. The COBOL extraction step

The "COBOL data division extraction" process type strategy is shown in Figure 11.21. The analyst has to select the files deserving such an extraction and to treat them one by

one, either with the automatic extractor or manually. Typically, analysts should prefer the automatic way, except if a COBOL file contains peculiarities unrecognised by the extractor. The automatic tool creates a new schema representing the extracted structures. The manual way has to do the same, a new schema has to be created and, once put in the "cobsch" set to distinguish it from the previously created schemas, it has to be edited manually. Let us note that the set "cobsch" always contains a single schema of type "COBOL schemas", which is in fact the product type used in update by "Manual extract".

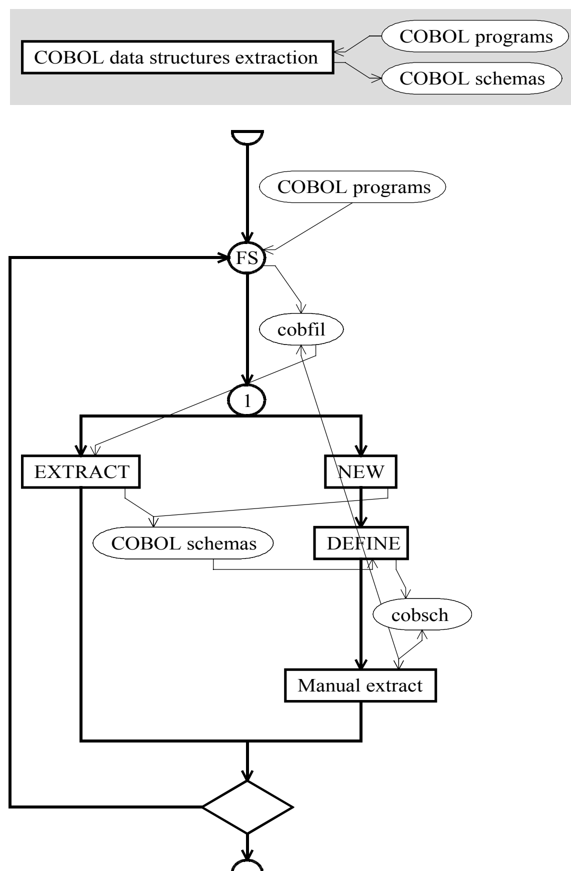


Figure 11.21 The data division extraction strategy

c. The schema enrichment step

The "COBOL schema enrichment" process type has to be performed for each schema extracted previously as shown in Figure 11.22. The enrichment must be done on the basis of information found in all COBOL source files, possibly from the file from which the schema was extracted, possibly from any other source file. In practice, most information will be found in files with a *procedure division*.

The strategy of the enrichment process type to be performed once for each schema is shown in Figure 11.23. First of all, the extracted schema is copied. The fact of taking the extracted schema in input and copying it to work on the copy only has two purposes. Firstly, it allows the history to be complete by keeping a copy of the result of the extraction and to modify its copy only. Secondly, it allows the engineers to make several copies to try various hypotheses and to choose the copy in output.

The engineer can choose either to use an expert process type to be guided during the job, or to do it entirely manually. Both methods can be combined: the expert can do a maximum, then the engineer can terminate with a few refinements, or the engineer can do the job in the manual way then use the expert process type to validate the result.

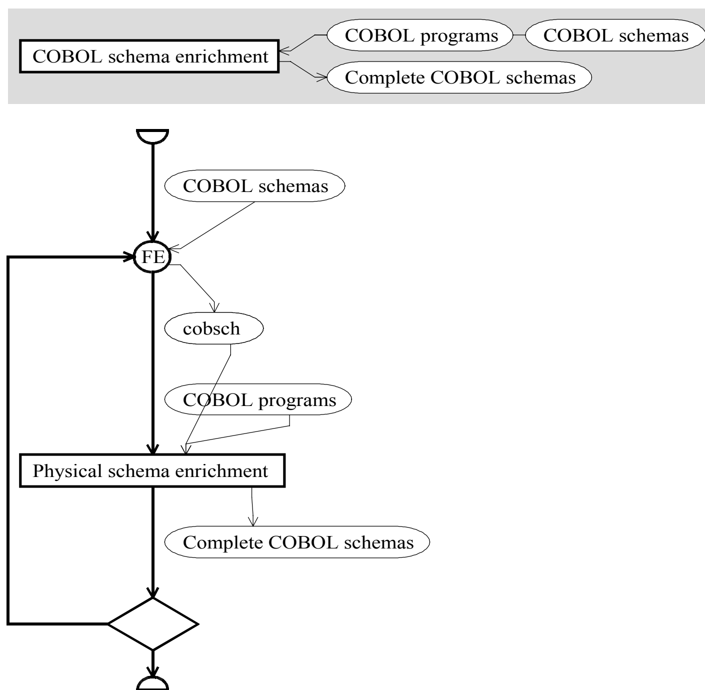


Figure 11.22 The global schema enrichment strategy

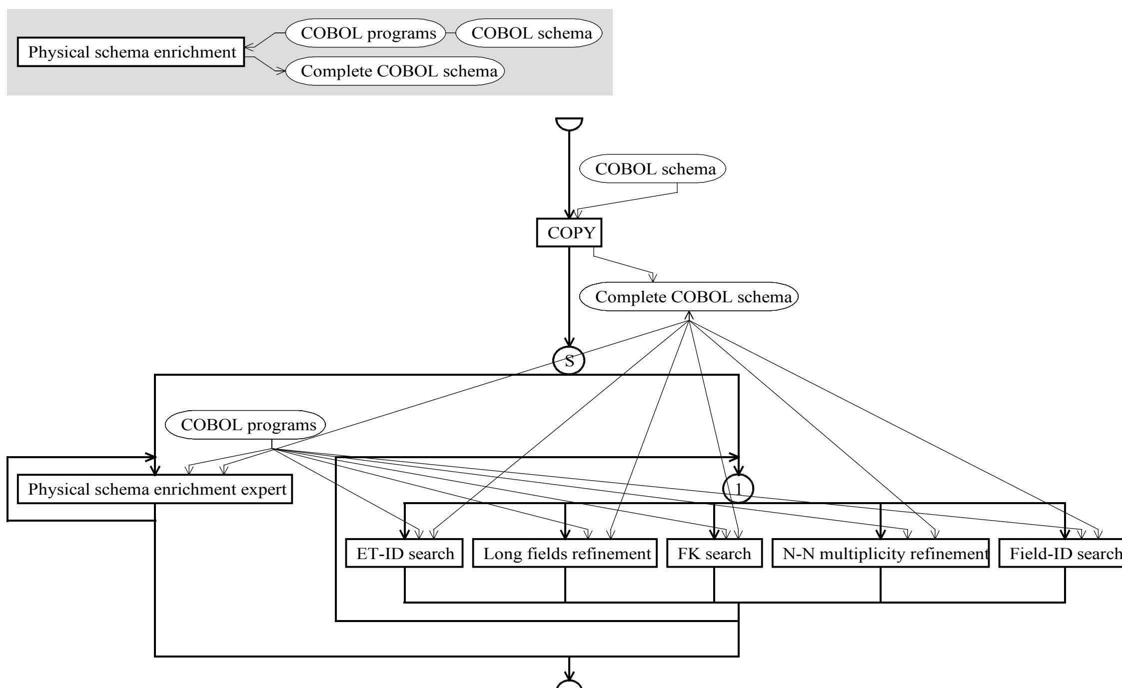


Figure 11.23 The true enrichment strategy for one schema

The manual way of working simply provides the engineer with a series of toolboxes aimed at several precise tasks:

- The “ET-ID search” toolbox provides tools for analysing texts and creating groups, in order to search for entity type identifiers. One of the tools, the program slicing[26] facility,

26 Program slicing [WEISER,84] consists in analysing the procedural code of the programs in order to detect evidence of additional data structures and integrity constraints by extracting instructions having a direct or

allows the analyst to find some programming patterns in the source files that check the uniqueness of one (possibly some) field value among all the records in a same datafile.

- The “Long fields refinement” toolbox provides tools for analysing texts and creating attributes. It is a common practice in COBOL to declare two data structures, a single rather large string and a compound field made of several sub-levels, possibly with repetitive components, and to copy (*move* in COBOL) one in the other and conversely. In that case, the second data structure can be seen as a refinement of the first one. Pattern matching is a text analysis function that is well suited to finding such *move* instructions.
- The “FK search” toolbox provides tools for analysing texts, looking for foreign keys by field names analysis and creating groups. Relations between tables are not declared in COBOL, they are hidden in the source code. Both the already cited program slicing and the pattern matching facilities can help, but DB-MAIN also provides a foreign key assistant which allows the comparison of attribute names, length and types and which can look for perfect or approaching matches.
- The “N-N multiplicity refinement” toolbox provides tools for analysing texts and modifying attribute properties. The COBOL syntax allows the declaration of repetitive fields (“OCCURS n TIMES”) which clearly show the maximum multiplicity of the field. But it gives no information about the minimum multiplicity. This last information must be found in the source code with the pattern matching and the program slicing facilities.
- The “Field-ID search” toolbox provides tools for analysing texts and creating groups in order to discover and create instance identifiers for multi-valued compound fields.

The expert process type, shown in Figure 11.24, uses the same toolboxes in sequence and embedded in *if* control structures whose conditions are examples of simple heuristics that give hints rather than absolute rules to follow blindly.

- The first conditional structure demands that the “ET_ID_search” toolbox is used when the condition “exists(COBOL_schema,ID_per_ET(0 0))” is satisfied, i.e. when there exists at least one entity type without primary identifier.
- The second conditional structure concerns the same toolbox, but with a different heuristic in the condition: “exists(COBOL_schema,NONE_in_LIST_CI_NAMES(ID*, *ID))” which looks for field names beginning or ending by “ID”.
- The third condition concerns the presence of long fields. But the notion of long field is subjective, it can be more than 10 characters, more than 20, 50, 100,... So the conditions is an informal one: “ask "Are there long fields?"”.
- The fourth condition simply looks for entity types without foreign keys: “exists (COBOL_schema,REF_per_ET(0 0))”. This condition is only a tip: the fact that each record has a foreign key does not mean that all foreign keys have been found because some entity types can have several foreign keys, and each entity type does not need a foreign key. But every entity type having at least one foreign key can be adopted as a completion condition.
- When the multiplicity of multi-valued fields as been refined, the minimum cardinality ends up being 0 or 1 most of the time, so it is a good trick to try to refine N-N multiplicities when the minimum one is greater or equal to 2: “exists(COBOL_schema, MIN_CARD_of_ATT(2 N))”.
- The fifth condition looks for compound multi-valued fields with no identifier: “exists(COBOL_schema,MAX_CARD_of_ATT(2 N) and SUB_ATT_per_ATT(1 N) and ID_per_ATT(0 0))”.

---

indirect incidence on the value of a given variable at a given point.

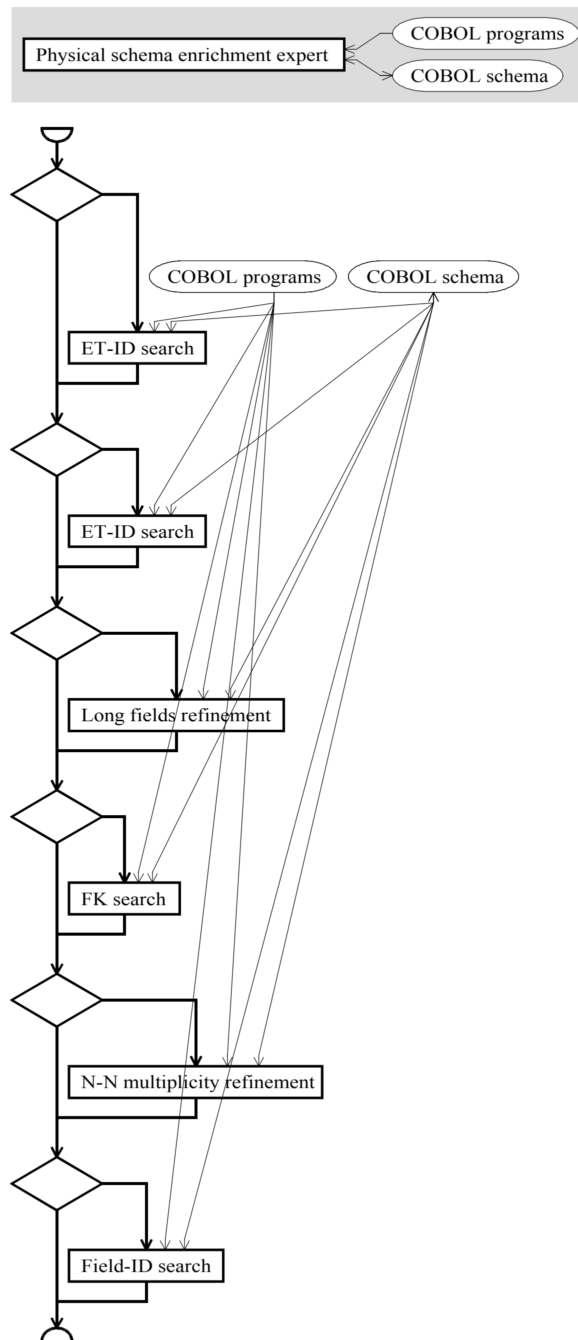


Figure 11.24 The enrichment expert strategy

11.2.2. Project performance

The method described above will be used to reverse engineer the database of a very small order management system. This application is written in COBOL. It is a very small program that holds in a single source file, shown in Appendix F.

A. Beginning of the project

The beginning of the project is straightforward and simply follows the method:

- The engineer has to add a new COBOL source file to the project. She selects a single source file: “Order.cob”.
- The engineer must then perform a process of type “COBOL schema extraction” which itself requires the performance of a process of type “COBOL data division extraction”.

- The “COBOL data division extraction” strategy requires the engineer to choose some products from which to extract data structures. Since “Order.cob” is the only available product and since it contains a *data division*, the engineer performs an “EXTRACT” primitive process with it. This last process creates a new schema named “ORDER/extracted”.
- The engineer terminates the process of type “COBOL data division extraction” with the new schema in output.
- Back to the “COBOL schema extraction” strategy in the method window, the engineer has to perform a process of type “COBOL schema enrichment”.
- The “COBOL schema enrichment” strategy simply demands that a process of type “Physical schema enrichment” be performed with each schema in input, that is to say once with schema “ORDER/extracted” and all source files, i.e. “Cobol.cob”, in input. In fact, the *for each* control structure makes a set, called “cobsch”, with each extracted schema at its turn, and passes this set as well as all source files to the new process.

B. The physical enrichment process

The performance of the “physical enrichment process” (same name as the process type because no confusion is possible between them, and their relationship appears more easily) is the most interesting part of the project because it is the one that requires the more human expertise. It will be described in detail. It follows the strategy shown in Figure 11.23.

a. Starting the process

When the process is started, its history is as shown in Figure 11.25. It contains two inputs: the set, “cobsch”, with its content, schema “ORDER/extracted”, and the “Order.cob” COBOL source file. “ORDER/extracted” is shown in Figure 11.26, and “Order.cob” is listed in Appendix F.

The first step of the process is imposed by the single sub-process type in the *allowed* state in the strategy: to make a copy of the original schema. The engineer knows the process is complex and thinks she will possibly make several copies of the schema during the process, at various intermediate states, so she decides to give the copy an extension that shows the temporary state of the product and calls it: “ORDER/draft-1”.

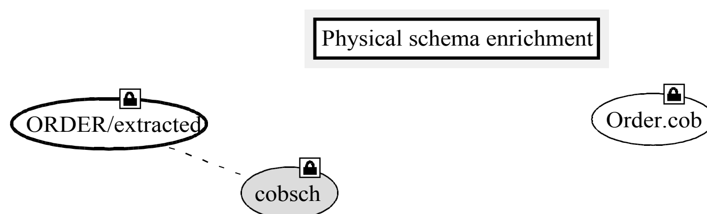


Figure 11.25 The beginning of the enrichment process

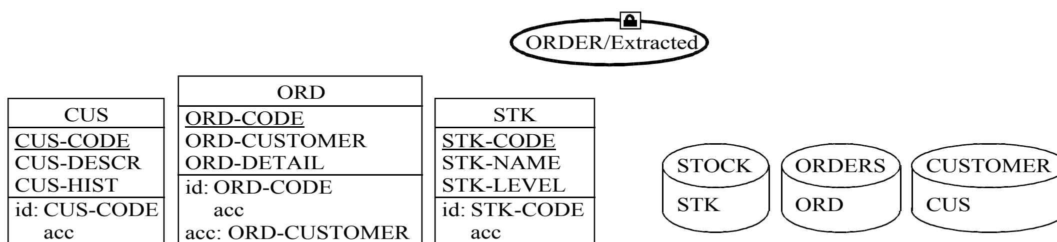


Figure 11.26 The “C-ORD/extracted” schema to enrich

When the copy is performed, the control flow follows the sequence in the strategy and arrives to the *some* structure. Its two branches are allowed. In the left one, the “Physical schema enrichment expert” process type is allowed. In the right branch, the *one* structure is allowed, and so all its branches, which are all primitive processes. So, the engineer has the choice either to use the expert or to start the job with a toolbox of her choice, as it can be seen in Figure 11.27.

The engineer prefers to proceed manually. When she looks at the schema “ORDER/draft-1”, in Figure 11.26, she thinks it would be better to concentrate her attention on one table only, in order to apprehend the reverse engineering activity thoroughly and to better understand the studied application (source file organisation, programming style, programming subtleties,...). She decides to examine the table “CUS”.

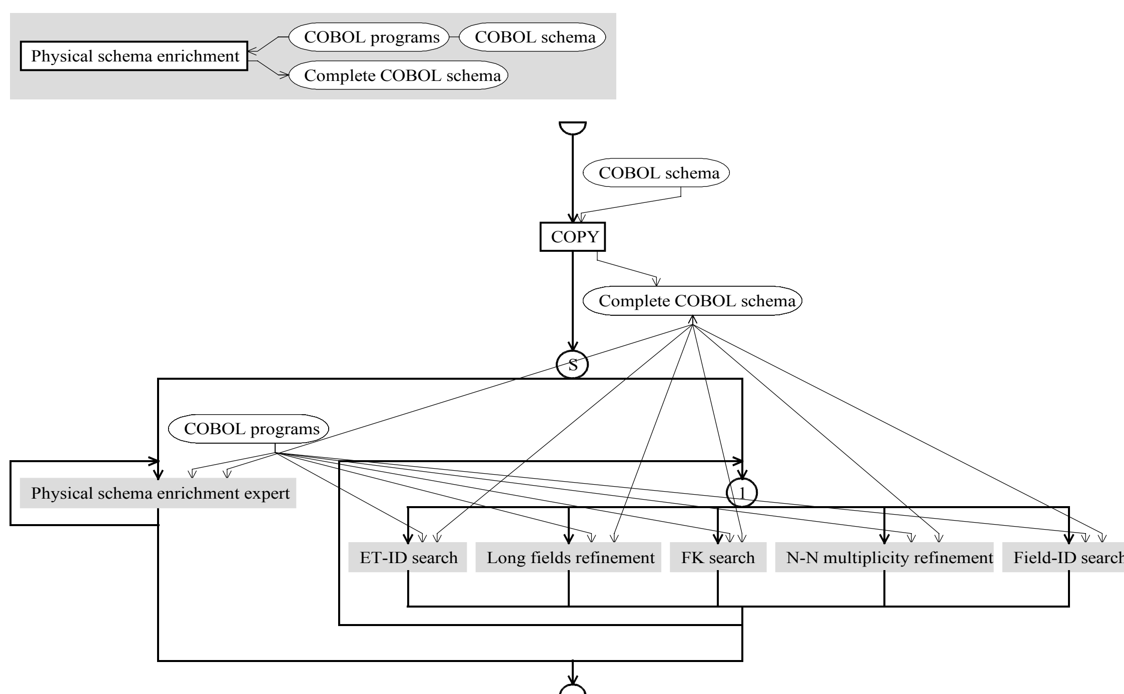


Figure 11.27 The engineer must decide what to do by herself.
Allowed state is shown with a box filled in light grey, done state with black borders.

b. Long field refinement

The engineer notices that attribute “CUS_DESCR” is made of 80 characters, and attribute “CUS_HIST” is made of 1000 characters. These are rather long fields. By double-clicking on the “Long fields refinement” process type, the window shown in Figure 11.28 appears. The description suits her needs, so she closes this window and starts a new primitive process of this type. The method window then appears as in Figure 11.29. All other processes of the *one* structure are disabled.

She uses the pattern matching tool, which is a text analysis tool, to look for the “MOVE” instructions involving the “CUS” record, which gave birth to the “CUS” entity type. She finds the instructions:

- “MOVE DESCRIPTION TO CUS-DESCR” in procedure “NEW-CUS”
- “MOVE CUS-DESCR TO DESCRIPTION” in procedure “NEW-ORD”.

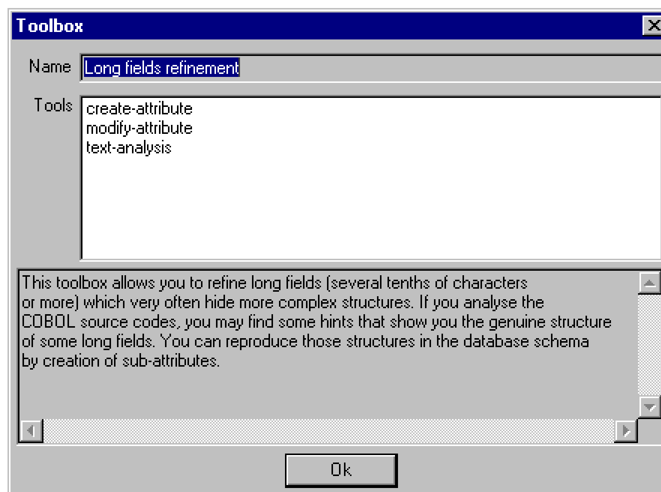


Figure 11.28 The "Long fields" refinement *process type properties*

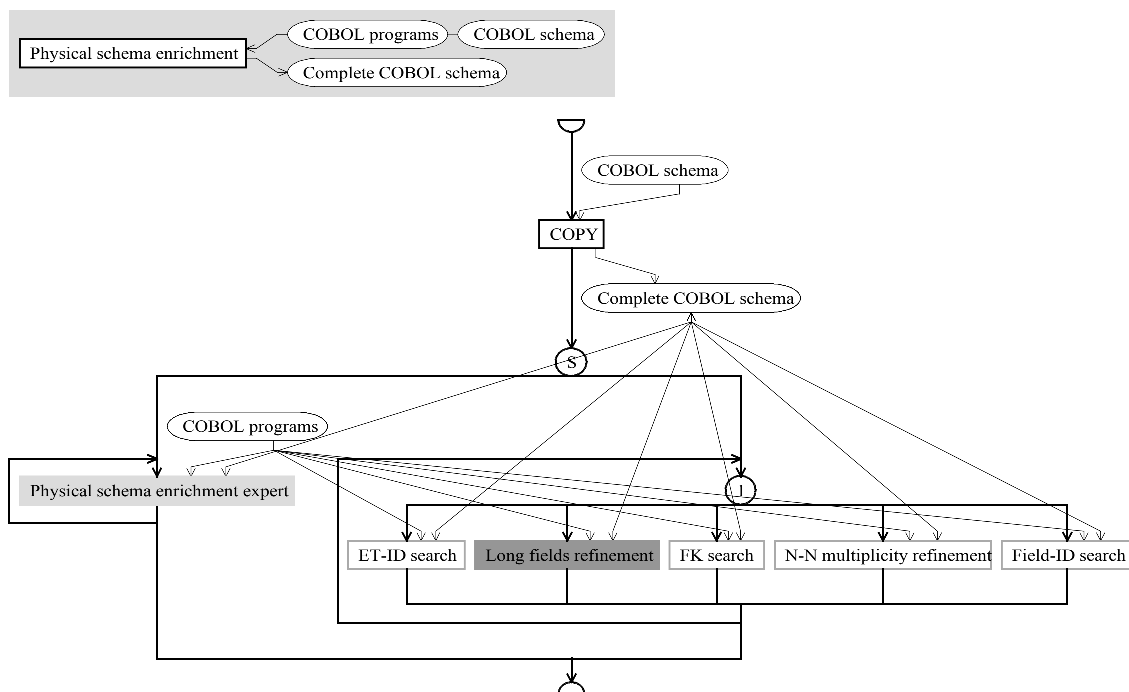


Figure 11.29 The engineer performs a process of the "Long fields refinement" type.
Running state is shown with a dark grey rectangle, unused state with white rectangle with grey borders.

In the declaration of the "DESCRIPTION" record, she also notices that its size is the same as "CUS-DESCR". So she assumes that "DESCRIPTION" is a refinement of "CUS-DESCR" and she creates new sub-fields to "CUST_DESCR" in the schema which correspond to the fields of the record "DESCRIPTION" in the source file.

She refines the field "CUS-HIST" in the same way[27]. The result is shown in Figure 11.30. Then she ends the primitive process, and she declares she has finished with the "Long fields refinement" type with the "Terminate" entry in the conceptual menu of the process type.

The *one* structure ends, and the *repeat* control structures allows the *one* structure body to be executed again, making the strategy look like in Figure 11.27 again.

c. N-N multiplicity refinement: 2 versions

The first refinement introduced an array of size [100-100] in the schema. The engineer

27 More details in [HAINAUT,96d]

knows that the array is the only available structure in COBOL to implement lists or sets. Moreover, if the array hides another structure, the actual minimum cardinality frequently is not really 100. She will use the "N-N multiplicity refinement" toolbox to try to solve the problem. But she does not know if the array is used to code a list or a set. A rapid look at the source files does not help her. So she decides to try both hypotheses. She manually makes two copies of the schema (using the *copy* function of the CASE tool, not the "copy" process type in the strategy, and choosing the same schema type as the original schema) that she calls "ORDER/draft-2" and "ORDER/draft-3".

i. First hypothesis

In the method window, the engineer selects the "Long fields refinement" process type and executes it. A dialogue box appears at the creation of the new primitive process, as shown in Figure 11.31. The engineer changes the name of the process to "N-N refinement - CUS-list" in order to show clearly and in short that the process concerns the refinement of the N-N records of the record "CUS" using lists. By clicking on the "Description button", she can write a longer and more complete text with the hypothesis. The "Input" and the "Update" lists contain the products of the "COBOL programs" and "Complete COBOL schema" types. The read only products, "Order.cob" which is an input of the current engineering process, and "ORDER/draft-1" which can no longer be modified after being copied, are proposed in input. Others, "ORDER/draft-2" and "ORDER/draft-3" are proposed in update. But the engineer only needs "Order.cob" and "ORDER/draft-2" so she removes the others from the lists, and confirms the creation of the primitive process.

The engineer opens both products and examines the "INIT-HIST" procedure. She notices that the data structure is initialised with "0" everywhere, meaning that it can be "empty". So she edits the properties of the "CUS-HIST.PURCH" field to change the minimum cardinality to "0" and to change the type from "array" to "list". Then she closes the two products and terminates the primitive process.

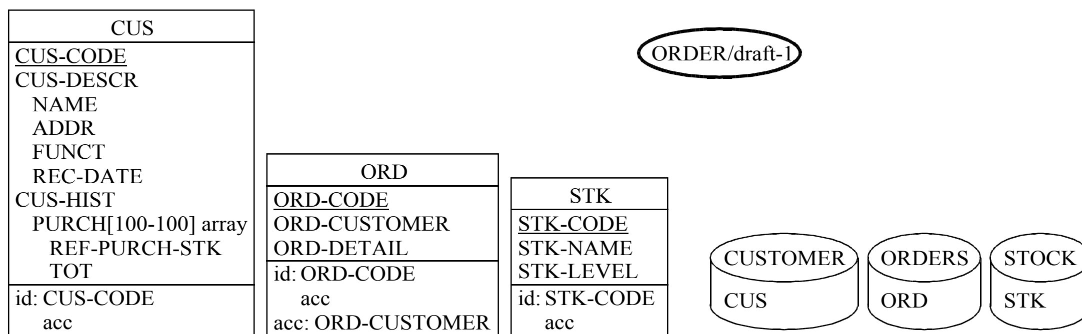


Figure 11.30 The long fields of record CUS are refined

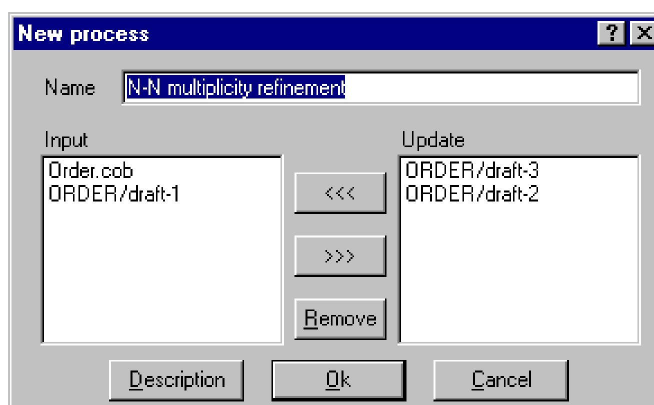


Figure 11.31 The new primitive process dialogue

ii. Second hypothesis

The engineer can then perform the same process type again with the second hypothesis by selecting the “Execute again” entry in the “N-N multiplicity refinement” process type contextual menu, and creating the new primitive process with the name “N-N refinement - CUS-set”, with the products “Order.cob” in input and “ORDER/draft-3” in update, and with the second hypothesis detailed in the description.

She then opens “ORDER/draft-3”, edits the “CUS-HIST.PURCH” properties to change its type from “array” to “set”, and to change the minimum cardinality to “0”.

When every process has been carried out with all the hypotheses, the use of the process type can be stopped by selecting the “Terminate” entry in its contextual menu.

d. Searching for field identifiers in both product versions

i. In the first product version

After examination of all the versions of the resulting schema, the engineer still cannot choose the best solution, so she continues the job. She looks again at the “CUS-HIST” data structure, as a set and as a list, and wonders if there could not be an identifier in that structure. So she decides to perform processes of type “Field-ID search”, she selects the “Execute” entry in the contextual menu and creates a new primitive process using products “Order.cob” and “ORDER/draft-2”.

With the text analysis tools[28], she discovers the “UPDATE-CUS-HIST” procedure that looks for an element of the list with “REF-PURCH-STK” being equal to a given “PROD-CODE”. If one is found, it is updated, else, a new element with that “PROD-CODE” is added to the list. She concludes that the field “REF-PURCH-STK” identifies list elements and she adds a new identifying group to the list with “REF-PURCH-STK” as the only component. She terminates the process.

ii. In the second product version

By selecting the “execute again” entry in the process type contextual menu, she starts the same process with “Order.cob” and “ORDER/draft-3” in input. But the job to perform is exactly the same. To avoid to do the same things several times, she opens the schema “ORDER/draft-2” again and stores its log in an independent file, then she comes back in “ORDER/draft-3" and replays automatically the part of the log file concerning the last process. This is easy to do since the starting of each new primitive process adds a tag to the log file containing the name of the process.

e. Taking a decision

When the engineer has ended the primitive process and the use of the primitive process type, she looks again at both versions of the schema and thinks it is time to take a decision, to choose one of them to continue the project. In the history window, she selects both versions of the product. Then she selects the “make decision” entry in the “engineering” menu[29]. She transfers “ORDER/draft-3” in the “Kept” list and writes the rationales of her decision, as show in Figure 11.32. The history then looks like in Figure 11.33 into which the reader can notice the difference between the arrows connected to the decision.

After having transformed the table “CUS”, the engineer wants to keep an image of the current state of the product transformation, so she makes a copy of “ORDER/draft-3”, that she names “ORDER/draft-4”.

28 More details in [HAINAUT,96d]
29 See Chapter 9.

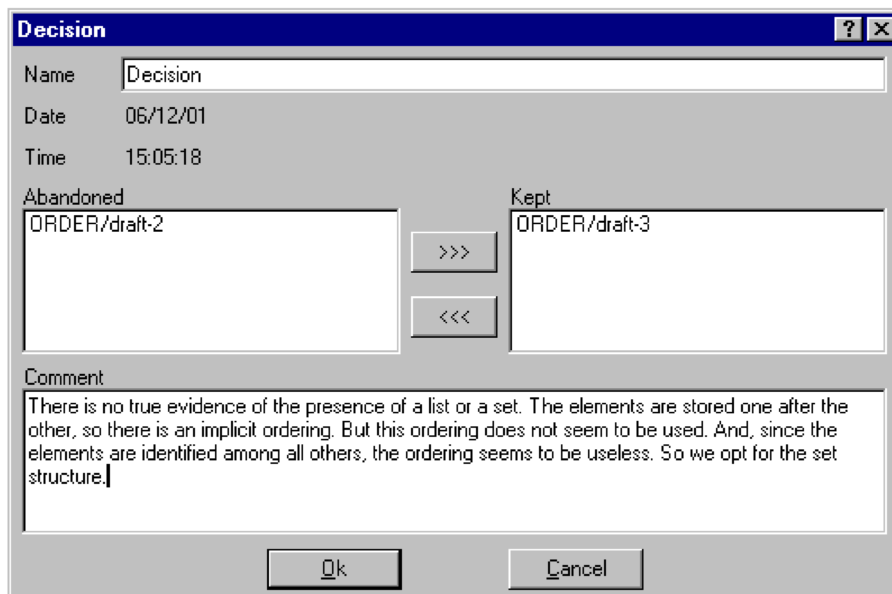


Figure 11.32 A decision taking dialogue box.

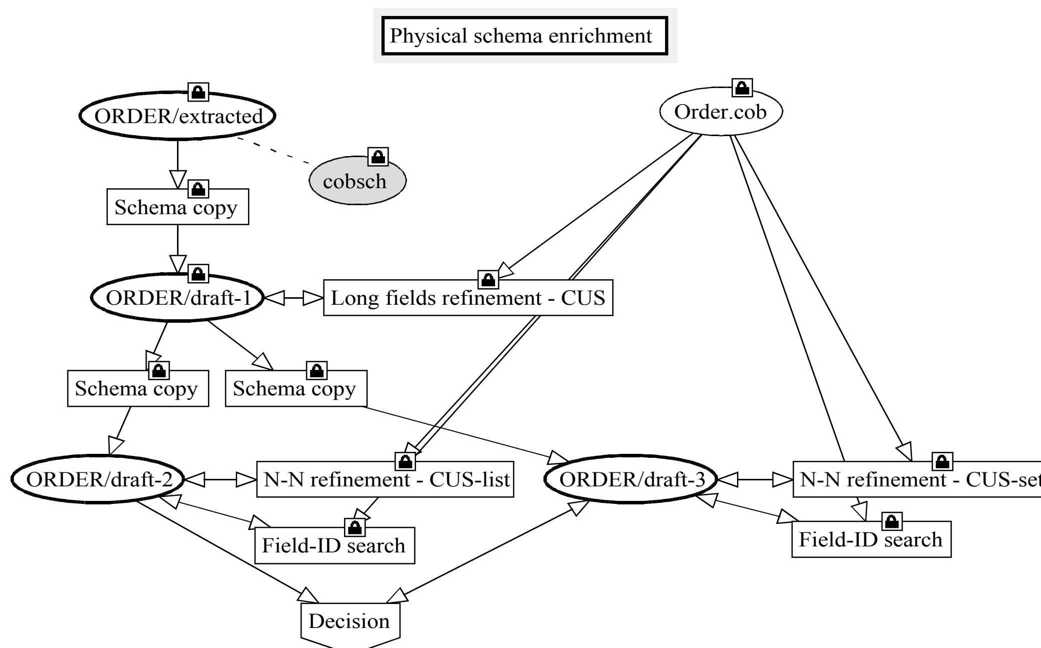


Figure 11.33 The history after a decision.

f. Enriching the remaining of the schema

She goes on with the project and does to the remaining of the schema what she did with the "CUS" table: she performs again processes of types "Long field refinement" with fields "CUS-DESCR" and "ORD-DETAIL", "N-N multiplicity refinement" and "Field-ID search" with the field "ORD-DETAIL". Since she already did the job once with the record "CUS", she knows how to work, so she does not need to make several hypotheses again. The resulting schema, with all the fields refined, is shown in Figure 11.34.

g. Looking for foreign keys

All the records being refined, the engineer decides to look for the links between the record types. So she starts a process of type "FK search" and she opens both the COBOL source file and the schema.

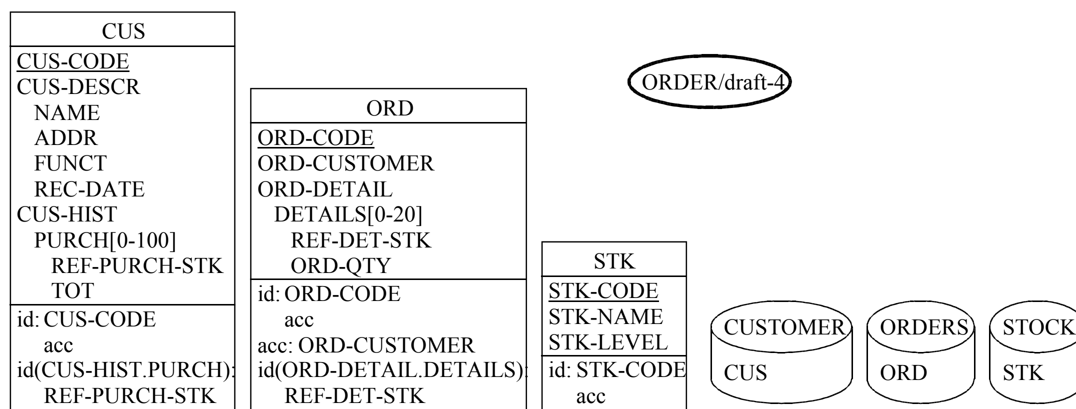


Figure 11.34 The schema after the refinement of all fields

i. Using intuition and the search tool

She notices the fields “REF-PURCH-STK” in “CUS”, and “REF-DET-STK” in “ORD” which are prefixed by “ref” (standing for “reference”) and which are ended by “STK” which is the name of the third record type. The properties of these two fields, as well as the properties of “STK-CODE”, which is the primary identifier of “STK”, show they are of the same type and have the same length. The use of the search tool in the source file to find all instances of “STK-CODE” allows the engineer to find the following instruction in procedure “READ-PROD-CODE”:

MOVE PROD-CODE TO STK-CODE.

This instruction comes just before a “READ” instruction to check the existence of a “STK” record in file “STOCK” with “STK-CODE” equals to “PROD-CODE”. Then the procedure “READ-PROD-CODE” calls the procedure “UPDATE-ORD-DETAIL” which stores the value of “PROD-CODE” in “REF-DET-STK” and which itself calls the procedure “UPDATE-CUST-HIST” which stores the value in “REF-PURCH-STK”.

ii. Using program slicing

The program slicing assistant can be used in conjunction with the search assistant to confirm all this. It shows that the “MOVE” instruction above is at the beginning of the program slices made of all the instructions of the application which have an influence on the value of “PROD-CODE” at the two program points: the “MOVE” instructions that copy that value in “REF-DET-STK”, and the “MOVE” instruction that copies the same value in “REF-PURCH-STK”.

iii. Adding foreign keys to the schema

With these hints, the engineer is pretty sure there is a foreign key in “CUS” made of the field “REF-PURCH-STK” which references “STK”, and another one in “ORD”, made of the field “REF-DET-STK”, which references “STK” too. She creates them in the schema.

In the same way, she discovers a third foreign key in “ORD”, made of the field “ORD-CUSTOMER” which references “CUS”.

When all the tables seem to be connected and when the engineer sees no more sign of other reference keys, she terminates the process and the use of its type.

h. End of the enrichment process

The refinement process being finished, the schema changes its status from draft to final. So the engineer changes the schema version from “draft-4” to “completed”.

Finally, the product looks like in Figure 11.35. The complete physical schema enrichment process history is shown in Figure 11.36. The engineer terminates it with the schema "ORDER/completed" as its only output. In the method window, the title is now in the *running* state and can be terminated.

Back to the "COBOL schema enrichment" process (Figure 11.22), the *for each* loop is terminated since there is only one schema in the set. The current engineering process can be terminated too, with schema "ORDER/completed" as its only output product.

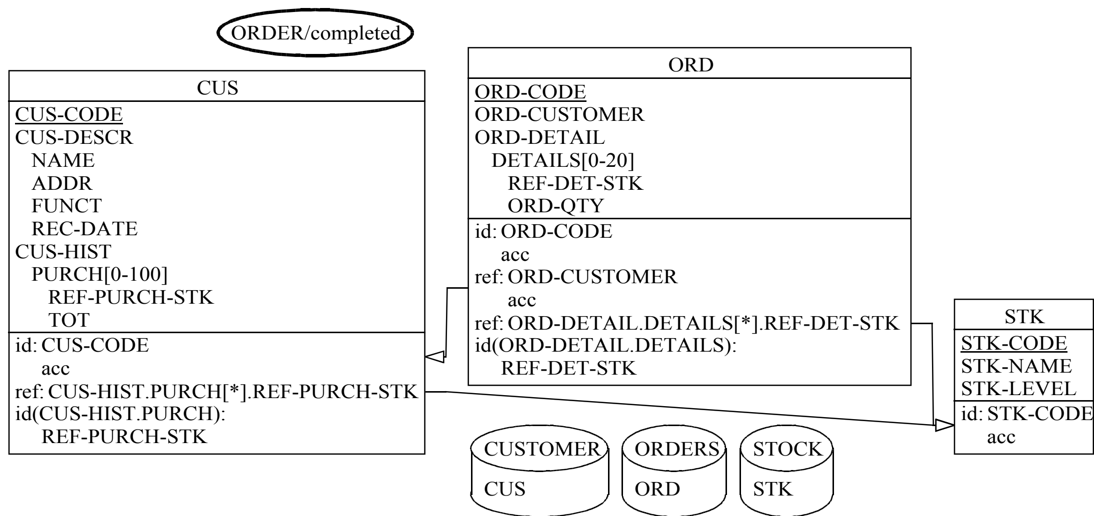


Figure 11.35 The refined physical schema

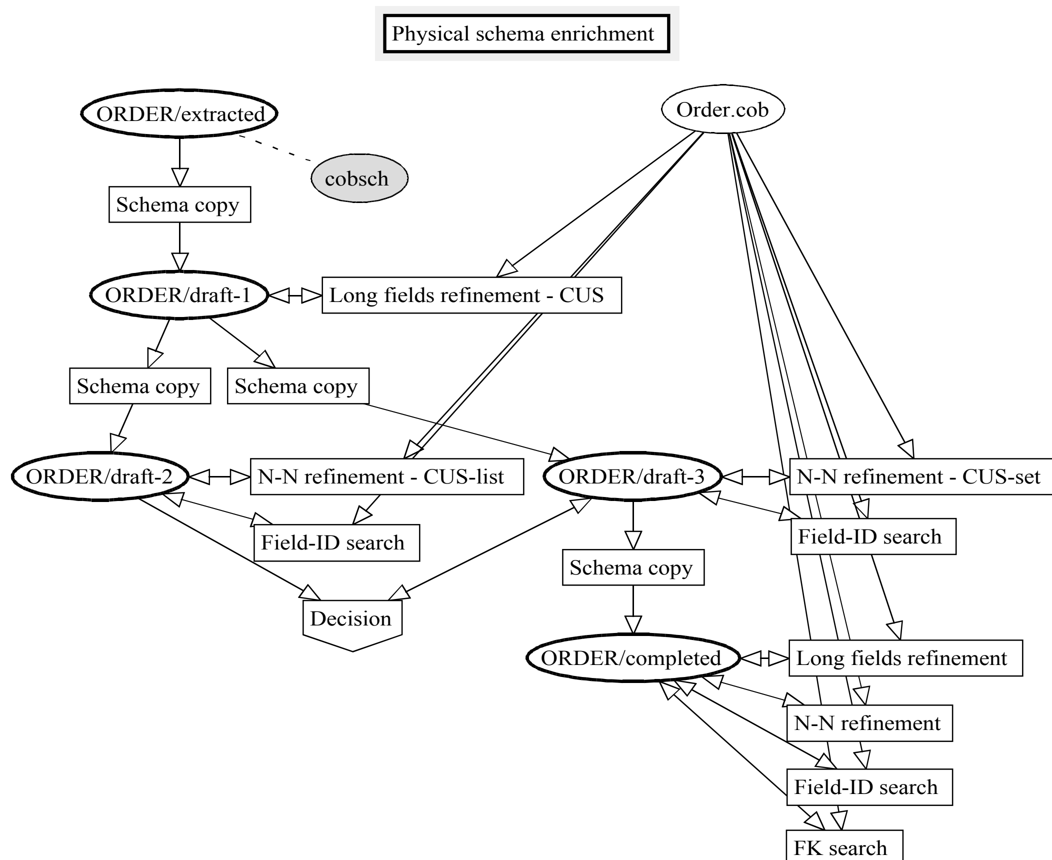


Figure 11.36 The complete physical schema enrichment process history

C. Remaining of the project

The remaining of the project is more simple, so it will be presented more rapidly.

a. Remaining of the COBOL schema extraction

The method window shows the "COBOL schema extraction" process type (Figure 11.20) again with the "COBOL schema enrichment" process type in the *done* state, and only the diamond of the *if* control structure in the *allowed* state. The engineer can double-click on it to see the condition:

cardinality(COBOL schemas) > 1

This is a condition that can be evaluated automatically by the CASE tool. Since "ORDER/completed" is the only product of type "COBOL schemas", the condition is not satisfied. So, when the engineer orders the evaluation of the condition, the methodological engine orients the control flow to the bottom of the diamonds, and the "COPY" process is put in the *allowed* state.

The engineer names the copy "ORDER/physical". This process ends the current engineering process ("COBOL schema extraction") whose output is "ORDER/physical". When the engineer explicitly terminates the use of the current engineering process type, the method window goes back to the strategy of the main engineering process type and the history window displays the root node of the history tree.

b. COBOL schema cleaning

The remaining of the project is more or less straightforward.

During the "COBOL schema cleaning" phase, the schema in Figure 11.35 is copied as "ORDER/logical", then the components of the copy are renamed to make the schema more readable:

- Each of the three record types is stored in its own file whose name seems more explicit, so the three record types can take the name of their file.
- All the fields of each record type have the same prefix. These prefixes are useless and can be removed.
- Some field names can be completed ("DESCR" → "DESCRIPTION", "ADDR" → "ADDRESS",...), and the field names can be capitalised (first letter upper case, others lower case).

Finally, the three files are removed, as well as the access keys.

c. Conceptualisation

During the "Schema conceptualisation" phase, the schema is copied again as "ORDER/raw conceptual". In the copy, attributes (what was called a field in the physical model is now called an attribute) "History" and "Detail" which are made of one component only are disaggregated. Attributes "Purchase" and "Details", which are very complex and look like optimisations, are separated from their entity types and transformed into entity types named "PURCHASE" and "DETAIL". Foreign keys are untranslated and transformed into rel-types.

d. Normalisation

Finally, during the "Conceptual normalisation" phase, a new "ORDER/conceptual" schema, shown in Figure 11.37, is created as a copy of the raw conceptual schema into which entity types "PURCHASE" and "DETAIL" are transformed into rel-types.

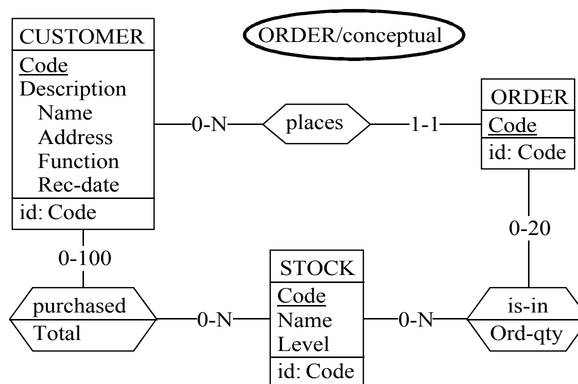


Figure 11.37 The normalised conceptual schema

11.2.3. The resulting history

The complete history tree is shown in Figure 11.38. Its global dependency view is shown in Figure 11.39. This simple view shows all the products that were generated during the schema extraction phase, derived from “Order.cob”. The abandoned schema version (“ORDER/draft-2”) appears at the end of a dead branch. The bottom of this view, from “ORDER/physical” to “ORDER/conceptual” is the straightforward conceptualisation part.

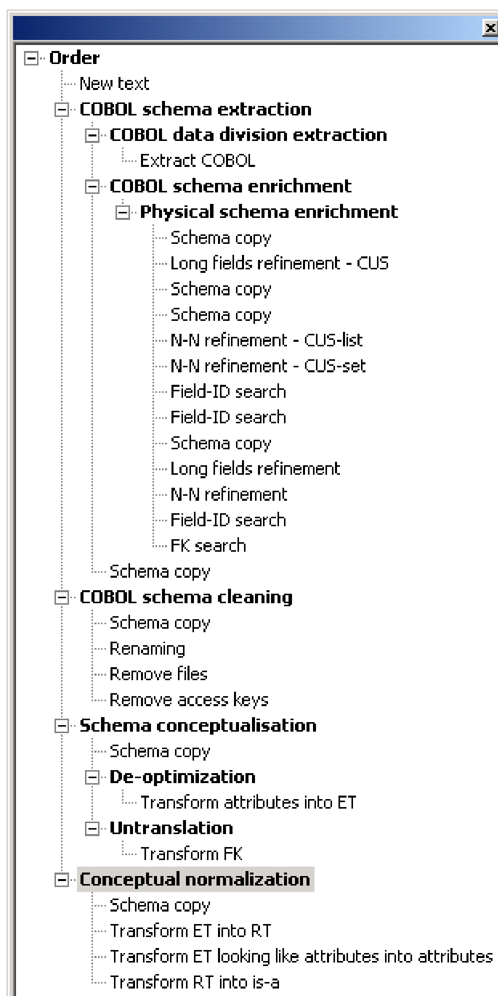


Figure 11.38 The history tree

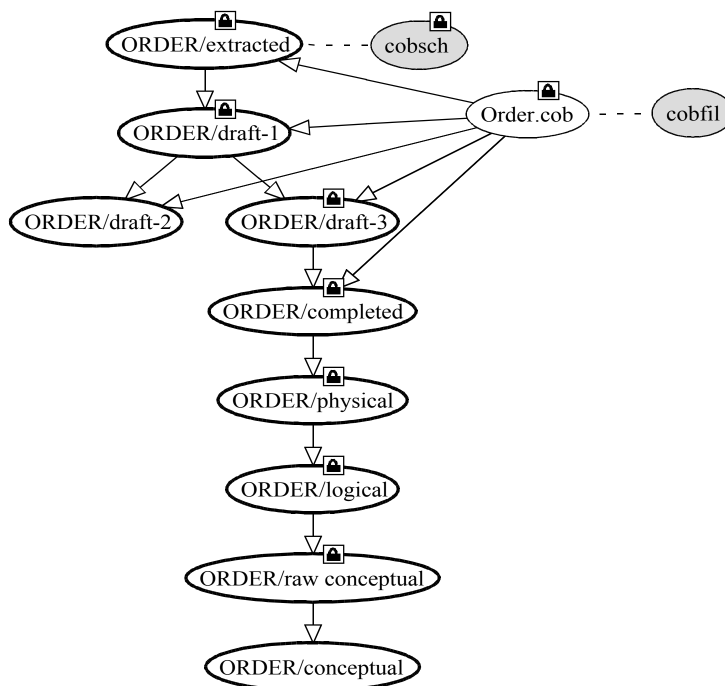


Figure 11.39 The global dependency view of the history

According to the implementation of the DB-MAIN CASE tool, the log files attached to schemas “ORDER/logical”, “ORDER/raw conceptual” and “ORDER/conceptual” are the concatenation of the log files of all the primitive processes that modify them. So, the engineer can flatten the conceptualisation phase of the project by concatenating the three log files associated to these schemas. The engineer can now use the new complete log file, “reverse.log”, to fulfil the three goals of the process engineering project:

- The detailed schema of the database, “ORDER/conceptual”, is the first goal.
- The second goal is a possible history of the original design. This goal can be reached by transformation of “reverse.log” using the history inversion transformation presented in Chapter 7, section 7.7 to produce the file “forward.log”. This history transformation is presented hereafter, in Section 11.2.4.
- The third goal is a mapping between all the components of the conceptual schema and the physical schema. For example, the attribute “Code” of entity type “CUSTOMER” in the conceptual schema (Figure 11.37) is the field “CUS-CODE” of record “CUS” in the physical schema (Figure 11.35), and the rel-type “places” in the conceptual schema is the field “ORD-CUSTOMER” of “CUS” in the physical schema. In fact, two different mappings can be computed:
  - A *direct mapping*, by drawing a two column table. In the left column, the engineer lists all the components of the conceptual schema. Then she follows “rev-forward.log” focussing her attention on each element of the left list and she writes in the right column the name of the corresponding element in the physical schema, possibly the result of several transformations. Since the log file is a simple text file listing transformations performed in sequence, a simple static analysis program can do the task.
  - An *inverse mapping*, by copying the name of each component of the physical schema in an attached technical note in the beginning of the project, and by taking care of all the notes during the reverse engineering process. At the end of the process, each component of the conceptual schema should have a note with the name of its physical counterpart. Once again, the copying process can easily be automated. A call to

this automated process could even be appended to the method, in the “Schema cleaning” engineering process type strategy, just before the “Renaming” primitive process type.

This mapping is simpler to compute than the first one, but it can be less precise because some particular transformations (for instance the merge of two components) can loose some notes, while other transformations can create new components without notes. Taking care of this kind of situations is difficult and only possible up to a certain point, possibly by using some heuristics, with the static analysis program of the first method.

11.2.4. Design recovery

The design recovery is the building of a possible history (“forward.log”) of the design of the database by inversion of the history (“reverse.log”) of the reverse engineering activity, as it was presented in Chapter 7, section 7.7.

An extended log file (required for design recovery) can be big, so interpreting it by hand could be tedious. Since it has a formal syntax, it can easily be interpreted by a program. Such a program can be written in almost any computer language that can handle strings, and read and write in text files. This program can either be written by the method engineer or be a function, built-in or add-on, of the CASE tool.

First of all, only the transformations are interesting, not the moves. Indeed, the automatic COBOL extractor created a new schema into which it placed the extracted tables and files. Then the analyst moved these constructs several times along the evolution of the schema in order to keep it readable (to make all the tables hold in the window or to avoid overlaps for instance). The trace of these moves is useless noise in the log history. The log files can be cleaned from these moves. A small C program that does the job is shown in Appendix F.

Figure 11.40 shows a cleaned version of the “reverse.log” file. In fact, this log file has been cleaned, only the first line of each entry is shown with a short description of its content. Only the last transformation is shown completely. This is an entity type into rel-type transformation (with the “TRF et_to_rt” header). The transformation is applied to the “DETAIL” entity type.

The log file transformation program[30] has to read the source file, each log entry at a time in the reverse order, the last one first and the first one at the end, and it has to write in the target file the inverse of each transformation. So this program has to understand every possible log entry and know their inverse. Applied to the “reverse.log” log file shown in Figure 11.40, the inversion transformation will produce the “forward.log” log file shown in Figure 11.41.

The new “forward.log” log file, the recovered design, is the new documentation of the legacy system, which can be reused to make the legacy database evolve, or to re-engineer the legacy system. For instance, it can be replayed on the final schema (see Figure 11.37). The resulting schema exactly corresponds to the logical schema resulting form the reverse engineering activity.

11.3. Conclusion

The first case study showed:

- how simple it is for an engineer to conduct a project with the guidance of the methodological engine

30 The complete program that does this job is much larger than the cleaning program because of the particular treatment for each possible log entry, so it cannot be printed in this thesis.

- how to build a complex structured history.

The second case study showed:

- the various degrees of freedom that can be given to the database engineer by the method engineer
- how the database engineer can record all his or her reasoning including trials and errors
- how the database engineer can bring his or her own touch to the project progress by making some simple processes which are not prescribed by the method, and which do not perturb it
- how the resulting history can be reused and transformed.

These two case studies simply show in a pragmatic way that the goals presented in Chapter 1, Section 1.4 are reached. The database engineers are well guided, their work is simplified, a correct and a useful documentation of their work is generated automatically.

```
*POT "begin-file"                          tag indicating the beginning of the log file
*POT "Renaming       20011206155142"       tag indicating the renaming of construxts in the schema
*MOD ENT                                   renaming ET "CUS" into "CUSTOMER"
*TRF prefix                                removing prefix "CUS-" in ET "CUSTOMER"
*MOD ENT                                   renaming ET "ORD" into "ORDER"
*TRF prefix                                removing prefix "ORD-" in ET "ORDER"
*MOD ENT                                   renaming ET "STK" into "STOCK"
*TRF prefix                                removing prefix "STK-" in ET "STOCK"
*MOD COA                                   renaming attribute "DESCR" into "DESCRIPTION"
*MOD SIA                                   renaming attribute "ADDR" into "ADDRESS"
*MOD SIA                                   renaming attribute "FUNCT" into "FUNCTION"
*MOD COA                                   renaming attribute "HIST" into "HISTORY"
*MOD COA                                   renaming attribute "PURCH" into "PURCHASE"
*MOD SIA                                   renaming attribute "TOT" into "TOTAL"
*MOD SIA                                   renaming attribute "REF-PURCH-STK" into "REF-PURCH-STOCK"
*MOD SIA                                   renaming attribute "REF-DET-STK" into "REF-DETAIL-STOCK"
*TRF name_proc                             name processing to capitalise attribute names
*POT "Remove files   20011206160632"       tag indicating the removing of files
*DEL COL                                   removing collection "STOCK"
*DEL COL                                   removing collection "ORDERS"
*DEL COL                                   removing collection "CUSTOMER"
*POT "Remove access  20011206160641"       tag indicating the removing of access keys
*MOD GRP                                   removing AK on identifier of "STOCK"
*MOD GRP                                   removing AK on identifier of "ORDER"
*MOD GRP                                   removing AK on foreign key in "ORDER"
*MOD GRP                                   removing AK on identifier of "CUSTOMER"
*POT "Schema concept 20011206160701"       tag indicating the beginning of the "Schema conceptualisation" process
*POT "De-optimizatio 20011206160951"       tag indicating the beginning of the "De-optimization" process
*POT "Transform attr 20011206161014"       tag indicating the beginning of the "Transform attributes into ET" process
*TRF desaggre_att                          disaggregation of attribute "CUSTOMER.History"
*TRF att_to_et_inst                        transformation of attribute "CUSTOMER.Purchase"
*TRF desaggre_att                          disaggregation of attribute "ORDER.Details"
*TRF att_to_et_inst                        transformation of attribute "ORDER.Details"
*MOD ENT                                   renaming ET "Purchase" into "PURCHASE"
*MOD REL                                   renaming ET "CUS_Pur" into "CUS_PUR"
*POT "Untranslation  20011206161212"       tag indicating the beginning of the "Untranslation" process
*POT "Transform FK   20011206161219"       tag indicating the beginning of the "De-optimization" process
*TRF att_to_rt                             transformation of FK from "PURCHASE" to "STOCK" into RT
*TRF att_to_rt                             transformation of FK from "DETAILS" to "STOCK" into RT
*TRF att_to_rt                             transformation of FK from "ORDER" to "CUSTOMER" into RT
*POT "Conceptual nor 20011206161627"       tag indicating the beginning of the "Conceptual normalization" process
*POT "Transform ET i 20011206161638"       tag indicating the beginning of the "Transform ET into RT" process
*TRF et_to_rt                              transformation of ET "PURCHASE" into RT "purchased"
*TRF et_to_rt                              transformation of ET "DETAIL" into RT "is-in".
%BEG                                       This transformation is listed in full, with all its details.
  %NAM "DETAIL
  %OWN "ORDER"/"conceptual"
  *CRE REL
  %BEG
    %NAM "is-in"
    %OWN "ORDER"/"conceptual"
  %END
```

Figure 11.40 The "revese.log" log file, first part...

```
  &MOD SIA
  %BEG
    *OLD SIA
    %BEG
      %NAM "Ord-qty"
      %OWN "ORDER"/"conceptual"."DETAIL"
    %END
    %NAM "Ord-qty"
    %OWN "ORDER"/"conceptual"."is-in"
  %END
  &MOD GRP
  %BEG
    *OLD GRP
    %BEG
      %NAM "IDDETAIL"
      %OWN "ORDER"/"conceptual"."DETAIL"
    %END
    %NAM "IDDETAIL"
    %OWN "ORDER"/"conceptual"."is-in"
  %END
  &DEL ROL
  %BEG
    %OWN "ORDER"/"conceptual"."is_in"
    %ETR "ORDER"/"conceptual"."DETAIL"
    %CAR 1-1
  %END
  &MOD ROL
  %BEG
    *OLD ROL
    %BEG
      %OWN "ORDER"/"conceptual"."is_in"
      %ETR "ORDER"/"conceptual"."STOCK"
    %END
    %NAM "is_in"
    %OWN "ORDER"/"conceptual"."is-in"
    %ETR "ORDER"/"conceptual"."STOCK"
  %END
  &DEL REL
  %BEG
    %NAM "is_in"
    %OWN "ORDER"/"conceptual"
  %END
  &DEL ROL
  %BEG
    %OWN "ORDER"/"conceptual"."ORD_DET"
    %ETR "ORDER"/"conceptual"."DETAIL"
    %CAR 1-1
  %END
  &MOD ROL
  %BEG
    *OLD ROL
    %BEG
      %OWN "ORDER"/"conceptual"."ORD_DET"
      %ETR "ORDER"/"conceptual"."ORDER"
    %END
    %NAM "ORD_DET"
    %OWN "ORDER"/"conceptual"."is-in"
    %ETR "ORDER"/"conceptual"."ORDER"
  %END
  &DEL REL
  %BEG
    %NAM "ORD_DET"
    %OWN "ORDER"/"conceptual"
  %END
  &DEL GRP
  %BEG
    %NAM "IDDETAIL"
    %OWN "ORDER"/"conceptual"."is-in"
    %COM "ORDER"/"conceptual"."is-in"."STOCK"
    %COM "ORDER"/"conceptual"."is-in"."ORDER"
    %TYP A
    %FLA "P"
  %END
  &DEL ENT
  %BEG
    %NAM "DETAIL"
    %OWN "ORDER"/"conceptual"
  %END
%END
*POT "Transform ET l 20011206161933"      tag indicating the beginning of "Transform ET looking like attributes"
*POT "Transform ET i 20011206161936"      tag indicating the beginning of the "Transform RT into is-a" process
*POT "end-file"                           tag indicating the end of the log file
```

Figure 11.40 The "revese.log" log file, last part.

Only the first line of each menu entry is shown, with comments, except the last entry which is shown completely

```
*POT "begin-file"                    tag indicating the beginning of the log file
*TRF rt_to_et                        transformation of RT "is-in" into ET "DETAIL"
%BEG
  %NAM "is-in"
  %OWN "ORDER"/"conceptual"
  *CRE ENT
  %BEG
    %NAM "DETAIL"
    %OWN "ORDER"/"conceptual"
  %END
  *CRE REL
  %BEG
    %NAM "ORD_DET"
    %OWN "ORDER"/"conceptual"
  %END
  *CRE REL
  %BEG
    %NAM "is_in"
    %OWN "ORDER"/"conceptual"
  %END
  &CRE ROL
  %BEG
    %OWN "ORDER"/"conceptual"."ORD_DET"
    %ETR "ORDER"/"conceptual"."DETAIL"
    %CAR 1-1
  %END
  &MOD ROL
  %BEG
    *OLD ROL
    %BEG
      %OWN "ORDER"/"conceptual"."is-in"
      %ETR "ORDER"/"conceptual"."ORDER"
    %END
    %OWN "ORDER"/"conceptual"."ORD_DET"
    %ETR "ORDER"/"conceptual"."ORDER"
  %END
  &CRE ROL
  %BEG
    %OWN "ORDER"/"conceptual"."is_in"
    %ETR "ORDER"/"conceptual"."DETAIL"
    %CAR 1-1
  %END
  &MOD ROL
  %BEG
    *OLD ROL
    %BEG
      %OWN "ORDER"/"conceptual"."is-in"
      %ETR "ORDER"/"conceptual"."STOCK"
    %END
    %OWN "ORDER"/"conceptual"."is_in"
    %ETR "ORDER"/"conceptual"."STOCK"
  %END
  &MOD SIA
  %BEG
    *OLD SIA
    %BEG
      %NAM "Ord-qty"
      %OWN "ORDER"/"conceptual"."is-in"
    %END
    %NAM "Ord-qty"
    %OWN "ORDER"/"conceptual"."DETAIL"
  %END
  &CRE GRP
  %BEG
    %NAM "IDDETAIL"
    %OWN "ORDER"/"conceptual"."DETAIL"
    %COM "ORDER"/"conceptual"."is-in"."STOCK"
    %COM "ORDER"/"conceptual"."ORD_DET"."ORDER"
    %TYP A
    %FLA "P"
  %END
  &DEL REL
  %BEG
    %NAM "is-in"
    %OWN "ORDER"/"conceptual"
  %END
%END
*TRF rt_to_et                        transformation of RT "purchased" into ET "PURCHASE"
*TRF rt_to_att                       transformation of RT "places" into a FK
*TRF rt_to_att                       transformation of RT "is_in" into a FK
*TRF rt_to_att                       transformation of RT "from" into a FK
*MOD REL                             renaming ET "CUS_PUR" into "CUS_Pur"
*MOD ENT                             renaming ET "PURCHASE" into "Purchase"
```

Figure 11.41 The "forward.log" log file, first part...

```
*TRF et_to_att        transformation of ET "Details" into attribute "ORDER.Details"
*TRF aggre_gr         aggregation of group made up of attribute "ORDER.Details"
*TRF et_to_att        transformation of ET "Purchase" into attribute "CUSTOMER.Purchase"
*TRF aggre_gr         aggregation of group made up of attribute "CUSTOMER.History"
*MOD GRP              adding AK to identifier of "CUSTOMER"
*MOD GRP              adding AK to foreign key in "ORDER"
*MOD GRP              adding AK to identifier of "ORDER"
*MOD GRP              adding AK to identifier of "STOCK"
*CRE COL              adding collection "CUSTOMER"
*CRE COL              adding collection "ORDERS"
*CRE COL              adding collection "STOCK"
*TRF name_proc        name processing to convert all attribute names to upper case
*MOD SIA              renaming attribute "REF-DETAIL-STOCK" into "REF-DET-STK"
*MOD SIA              renaming attribute "REF-PURCH-STOCK" into "REF-PURCH-STK"
*MOD SIA              renaming attribute "TOTAL" into "TOT"
*MOD COA              renaming attribute "PURCHASE" into "PURCH"
*MOD COA              renaming attribute "HISTORY" into "HIST"
*MOD SIA              renaming attribute "FUNCTION" into "FUNCT"
*MOD SIA              renaming attribute "ADDRESS" into "ADDR"
*MOD COA              renaming attribute "DESCRIPTION" into "DESCR"
*TRF prefix           adding prefix "STK-" in attributes of ET "STOCK"
*MOD ENT              renaming ET "STOCK" into "STK"
*TRF prefix           adding prefix "ORD-" in attributes of ET "ORDER"
*MOD ENT              renaming ET "ORDER" into "ORD"
*TRF prefix           adding prefix "CUS-" in attributes of ET "CUSTOMER"
*MOD ENT              renaming ET "CUSTOMER" into "CUS"
*POT "end-file"
```

Figure 11.41 The "forward.log" log file, last part.

This log file is the result of the inversion of the "reverse.log" file.

Chapter 12

Professional use

The DB-MAIN CASE tool enhanced with the methodological engine has been used to support the methodological aspect of several projects. This chapter presents the feed-back from users and their comments.

The DB-MAIN CASE tool enhanced with the methodological engine has been used by other researchers as a tool to accomplish their projects. This chapter presents two projects, as well as the the feed-back and the comments of researchers in charge of these projects. The first project addresses the database evolution problem already presented in Chapter 7. The second project concerns the design of XML database schemas.

12.1. List of questions

The following questions were asked to the users:

1. For what purpose was the process modelling facility used?
2. What advantages did you get from the process modelling facility?
3. What critics can you formulate about the process modelling facility?
4. What enhancements would you suggest?
5. If you had to do your project again, would you still use the process modelling facility?

12.2. Relational database applications evolution

The relational database applications evolution problem is presented in Chapter 7, section 7.3. It is also presented in [HICK,98] and in [HICK,99], and fully detailed in [HICK,01].

Here are Jean-Marc Hick's answers to the questions:

1. I used the process modelling facility to present the method to follow in a formal way and to automate the use of this method.
2. The syntax was sufficient to describe the method. The use of the process modelling facility to formalise the method allowed me to highlight difficulties and incoherence, as well has to make that method clear. So, process modelling helped at enhancing the methodology.
   In the DB-MAIN CASE tool, the process modelling facility allows me to guide the user who wishes to follow a complex evolution method.
3. Following the method in the CASE tool is pretty satisfying. But, while designing a method, the necessity of building a new project each time the method is updated is a bit heavy.
4. About the tools, a debugger would be useful.
5. To start to use a method in the field of database engineering, the process modelling facility seems to be well adapted. It is a precious step in setting up a new methodology. Hence the answer is yes.

12.3. XML Engineering

The second project is about engineering XML databases. It is presented in [DELCROIX, 01] and in [ESTIEVENART,02]. It follows a general path similar to the one of the first case study in Chapter 11, but with transformations which are particular to XML.

The XML model is a hierarchical model. While the ER model is not. So the transformation process has to derive a hierarchical schema from a unconstrained schema. This cannot be done automatically, and requires some decision taking. Indeed, after the transformation of is-a relations and non-functional rel-types, in the same way as in the first case study of Chapter 11, an ER schema is a graph. Some entity types appear as the root of a hierarchy. The rel-types between these root nodes and their children are marked as inheritance links.

Then those children themselves have children, so the links between them are marked too, and so on. When no more inheritance links can be discovered, some unmarked rel-types may remain in the graph. Some of them can be suppressed and replaced by a reference group, then the search of more inheritance relations can go on. The decision of which unmarked rel-type to suppress is not trivial. It is a human decision. When no more unmarked rel-type remains, the graph is a forest which can be transformed into a tree by adding a root node, which is the father of all previous root nodes. Then, the sub-types of an entity type must be ordered in a sequence, this is, once again, a non-trivial task. For instance, Figure 12.1 shows an intermediate version of a schema during the transformation process (from [DELCROIX,01]). All rel-types are binary one-to-many without attributes, and some *many* roles have the name "f" ("father"). The rel-types in which they are played must be kept. Rel-types without such a role ("dét_PRO" in the example) will be deleted. An entity type "root" has been added so that the hierarchy has a single root node. "root" contains a special group which indicates the sequence order of its sub-types (the entity-types linked to it through the many-to-one rel-types).

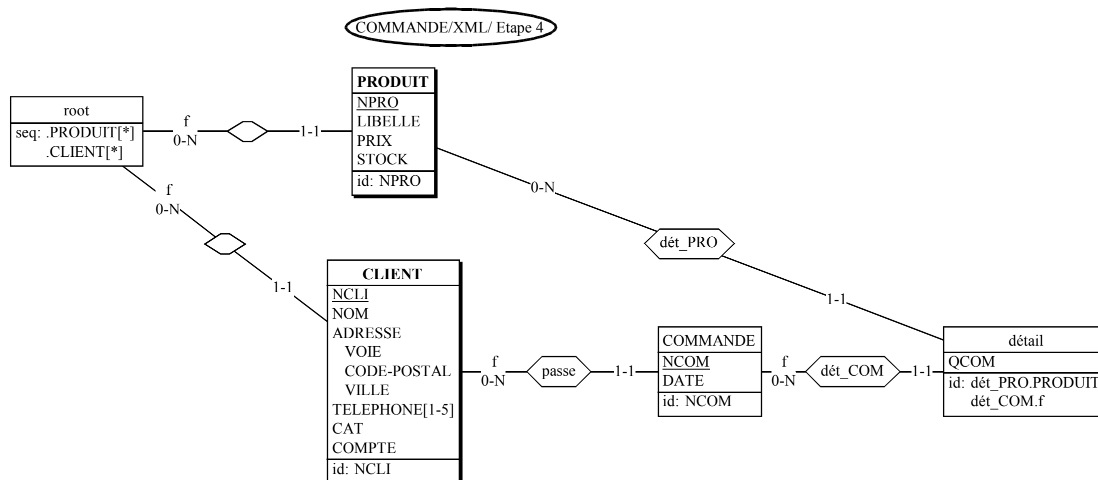


Figure 12.1 An XML schema transformation example

Many required functions (detection of the root of a hierarchy, detection of the children of one node,... for instance) are not part of the CASE tool kernel and have been implemented with the 4GL Voyager 2. Some transformation scripts do a part of the job too.

Here are the answers of F. Estiévenart to the questions:

1. The process modelling facility allows us to implement the method for transforming a conceptual schema into an XML (DTD or XML schema) model compliant schema.
2. The advantages are the following:
   - Graphical view of the transformation process (terminated processes and processes still waiting to be performed are easily distinguishable).
   - Possibility to use either global transformation scripts, validation scripts, or other Voyager 2 programs using a single interface.
   - Possibility to cope with schemas compliant with various models.
3. The critics are the following:
   - When a process finishes automatically, it is necessary to explicitly start the following one using the "execute" menu item. In the same way, at the end of a sub-process, the user must click on "terminate" to go back to the main process. When a method contains a lot of process types, too many clicks are necessary. A remedy to this "friendli-

ness weaknesses" is to merge several sub-processes into a single one, but this reduces the clarity and the precision of the global process.

- Interaction with DB-MAIN. Necessity to export the conceptual schema made in one project in an external format to be able to use it in the XML design project using the process modelling facility.

4. The CASE tool should be able to start and finish some automatic process types automatically, without the need for the user to click many times.
5. Yes, in a different way. The method is mainly a sequence of process types now. Most decisions to take are asked to the analyst through Voyager 2 dialogue boxes. If the same project had to be done again, a better use of MDL, using more of the control structures it offers, could reduce the need of programming in Voyager 2.

12.4. Conclusion

The first true users of the DB-MAIN CASE tool augmented with the process modelling facility seem to be happy with it, although some enhancements are still necessary to make it a fully industrial tool.

The DB-MAIN team having many contacts with private companies in many domains can notice that most of them still do not use methodological support and are, a priori, not very interested in them. This lack of methodology leads them to poorly documented and unmanageable systems which need being reverse engineered. So further effort of education should be necessary to make them aware of the usefulness of methodological support before their commitment to it and their feed-back can be analysed.

Part 4

Future work

Chapter 13

Method evolution

This thesis addresses the process of writing and using a method. But a method also has to evolve along the time. This chapter will begin with the reasons for making a method evolve and with a classification of these reasons. Then the problem will be stated more formally and a solution will be proposed. It is based on temporal databases that allow the repository to store all the versions of the method, as well as the complete history of the project with correct links between each part of the history and the method version which guided it.

13.1. Presentation

This thesis addresses the problem of writing and using a method. But a method also has to evolve along the time for various reasons:

- A new project has to start which looks pretty like a previous one, with just a few differences (in the physical product model, for instance).
- A project following a method showed that a few aspects of the method could be enhanced. The method engineer wants to update the method for future projects.
- During a project, the database engineer faces a situation in which the method does not allow him or her to perform a required task. The method should be updated before going further in the project.
- During a project, due to a method design problem, the methodological engine has reached a deadlock. For example, a toolbox does not provide the useful tools to make a product compliant with a given product model. In this case, an update of the method (or the disabling of the method, as shown in Chapter 9) is required.

These reasons can be classified in two categories: the static updates (first and second cases) and the dynamic updates (third and fourth cases). A *static update* is performed before the beginning of a project; it is a further step in the design of the method. A *dynamic update* is performed during a project, so that a part of the project is performed with the unmodified method and another part of the project with the modified method. The same classification is also made in [DOWSON,94].

Other project authors, such as [CUGOLA,99], [CONRADI,94a], [POHL,99], and many more, also think taking possible deviations into account is necessary.

In Chapter 4, various process modelling techniques were compared. Most of them allow dynamic updates. In fact, they take dynamic update as a requirement, according to the fact that the third and fourth reasons above are common because of the design of the methods by human beings. But this raises a question about giving a non-mature method to the user to help. Ensuring that a method is correct is surely a good practice. If a large computer program can reach several millions of lines of code, a large method will seldom go over a few thousands of lines of code, a typical method being about several hundreds of lines of code only as it could be observed in real projects. Verifying it and testing it are possible and realistic activities. Anyway, updating dynamically a method can always prove to be useful.

13.2. The problem

In this section, the problem will be stated more precisely. The possible modifications that can be done to a method and their consequences, both on the remaining of the method and on the history, will be examined within both declarative and procedural method definition paradigms.

The main kinds of method modifications are the following:

- insertion, modification or deletion of a product model
- insertion, modification or deletion of a product type
- insertion, modification or deletion of a process type.

13.2.1. Product models and product types

In the MDL framework, the definition of product models and product types follow a declarative paradigm. So the following analysis holds for either the MDL framework, most

other procedural models, or declarative models.

Inserting a new product model is a really easy task since it has no direct impact on the remaining of the method. It does not even has an impact on the history since no product of a type compliant with this model already exist. Indirect consequences will generally appear due to the fact that this insertion is often accompanied by the modification of several product types or process types.

Inserting a new product type or a new process type has similar impacts.

Modifying a product model may have various impacts according to the kind of modification. Either the list of concepts or the list of constraints can be modified. Adding a new concept or modifying an existing one (that is to say changing its renaming) has no impact on the remaining of the method nor on the history. Removing a concept can have a lot of consequences: constraints may be invalid and process types concerned by this concept may become senseless. Adding or modifying a constraint can also have a large impact on the process type and on the history. Removing a constraint can cause product model compliance problem with parameters during future use of sub-processes, but it has no impact on the history.

Modifying a product type can have various impacts too. Modifying the product model of the product type can have an impact on other product type definition since product model compliance must still be insured in sub-process use, and an impact on the history since the existing products may not be compliant with the product model anymore. Modifying the cardinality constraints can also have an impact on parameter definitions, but also on the strategy of the process type(s) using this product type. Modifying the cardinality constraints can also have consequences on the validity of the history.

13.2.2. Process types

Process types are very different within a procedural paradigm, including the MDL framework, or within a declarative paradigm.

Within a procedural paradigm, *modifying a process type* is rather simple since it suffices to replace the old one by the new one for the method to remain coherent. But the history may suffer a lot of such a modification. Indeed, the recorded processes may not follow the new strategy anymore. The links between the method and the history can only survive if the new process type version still allows to perform the job in the same way. For instance, if the modification simply adds a new branch to a *some* structure, the history is still valid with respect to the new process type because performing the added branch is not mandatory and it could have always been left apart in the past.

Deleting a product model, a product type or a process type has evident consequences on the remaining of the method, each reference to the deleted component becoming invalid, and on the history if it contains products or processes of this type or of this model.

Within a declarative paradigm, the insertion, the deletion or the modification of a process type seem to be equally easy because no other process is directly involved. But the integrity of the whole method still needs to be verified. Indeed, a situation into which a process type can no longer be enacted, or into which no more process can be enacted, because of a precondition or post-condition mismatch, can be reached.

13.2.3. The method evolution problem

In summary, the method evolution problem is twofold:

- The method itself has to keep its coherence.
- The history should keep its characteristic of following the method. In this respect, the

idea of automatically updating the history[31] in parallel with the method is not an option because it should not be altered.

For the method to keep its coherence, the problems impacted by a modification must be detected and corrected with other modifications. Several impact detection techniques exist:

- Dependency graphs: graphs into which the nodes are product models or product types, and the vertices are dependency relationships. For example, a dependency graph may show that product type *A* depends on ("is of" in reality) product model *M*, as well as product type *B*. It may also show that products of type *C* in one process type become products of type *D* in another process type used by the first one.
- Call graphs: graphs into which the nodes are process types and the vertices show that one process type uses another process type.
- Declaration graphs: graphs into which the nodes are product types and process types, and the vertices show in what process types each product type is declared (for local product types) or used (for global product types).
- Program slicing [WEISER,84]: it consists in analysing the MDL code of the methods in order to detect integrity problems by extracting MDL instructions having a direct or indirect incidence on the instances of a given product type at a given point.
- ...

But the use of these detection techniques is out of the scope of this chapter which is concerned about the correct storage of every needed modifications in a valid method. In the following section a solution to the evolution problem will be described in detail.

13.3. Solution proposal

A solution that keeps both the history unchanged and the links to the method it followed is to keep the old method unchanged in the repository together with the modified method. The solution proposed in this section to the problem of method evolution is to transform the repository depicted in Chapter 10 to make it temporal. In a first time, a few elements about temporal databases will be reminded. They will be applied to the method evolution problem in a second time.

13.3.1. Temporal databases

A description of temporal database techniques can be found in [DETIENNE,01]. In this section the aspects of this paper of interest for this work are summarised.

In a traditional database, only the current state of a given entity is kept. In a temporal database, all the successive states of this entity are kept together with the date and time of their modification. For example, a customer is known with his or her current address only in a traditional database, and all the successive addresses of the customer are kept with the date they moved from one to the other in a temporal database. In fact, either the true date and time of the move can be saved, this is called the *valid time*, or the date and time of the moment the information is stored, this is the *transaction time*. In the following, only the transaction time is of concern. It will be stored by tagging entity types with timestamps.

A **timestamp** is a data structure that allows to represent moments. It can be a date, a pair (date,time), just a time, including or not the century, the year, seconds, tenths of seconds, hundredths of seconds,... It all depends on the granularity the problem needs: an information system managing a stock in real time to which thousands of terminals are connected does not require the same timestamps as an information system on a standalone computer

31 Is this realistic? This surely needs complex actions and decision taking. It would be worth another thesis.

for managing lecturers and books in a small library; a timestamp as precise as a thousandth of a second may be useful in the first case, while a timestamp with the precision of a day may suffice in the second case. In order to avoid this granularity problem and to ease the representation, integers will be used in this thesis, 0 representing the moment the information system is activated, 999, or any large number, representing the future, any other integer value between 0 and 999 representing discrete moments between the activation and the present time, itself represented by a variable *now*. This can be done without loss of generality since a new table associating these integer values to real moments can always be introduced[32]. Figure 13.1 shows an example of time line using these notations.

Figure 13.1 A time line

A **temporal entity type** is an entity type to which two timestamps are attached: one, the *start time*, that represents the moment at which the entity type is created, and the other, the *end time,* representing the time at which the data are "modified" or "deleted". Indeed, the values stored in a temporal data structure are never modified nor suppressed, they are simply marked as such. When a new entity of a temporal type is stored in the database, the start time is set to the value of *now* and the end time is set to 999 in order to show that it will still be correct in the future. This entry is the first state of the entity. When the entity must be modified, the end time of its last state is set to the value of *now*, then a new state of the same entity is added to the database with the new values. As a result, an entity is represented by one entry in a non-temporal database, but an entity is made of several entries (several states) in a temporal database. To "delete" this entity, the end time of its last state is simply set to the value of *now*, only states having an end time of 999 being valid.

The example in Figure 13.2 shows that a new customer, Smith, was encoded at time 5 with the customer ID 123. Another new customer, Jones, was encoded at time 8 with ID 178. At time 12, Mr Smith told us he moved from Roses street to Lemons square: his first state was invalidated by setting the end time to 12, and a new state was created with the new address, new timestamps (12 for the start time, 999 for the end time) and the former values for all other attributes. Mr Jones moved twice and told it to us at times 27 and 49. Finally, at time 68, Mr Jones was removed. This example contains two entities and five states. At the present time, the correct address of Mr Smith is known by the line with an end time value of 999, as well as all his previous addresses with the period at which he lived at each one. The fact that Mr Jones was a customer in the past is known too.

CustId	*Name*	*Address*	*Start time*	*End time*	*Other data*
123	Smith	45, Roses street	5	12	...
178	Jones	13, Pines avenue	8	27	...
123	Smith	14, Lemons square	12	999	...
178	Jones	6, Grapefruit street	27	49	...
178	Jones	26, Pineapple avenue	49	68	...

Figure 13.2 A temporal data structure example

A temporal data structure has some properties that imply a few constraints:

- Since the database contains several states of the same entity (in the example above, several lines concerning the same customers, with the same CustId value), the *natural iden-*

32 Furthermore, this reference table can also insure the uniqueness of timestamps since two events appearing exactly at the same time will be assigned two consecutive integer values.

tifier of an entity type, that is to say what identifies in the real world the concept represented by the entity in the database, is not an identifier. A possible identifier is the natural identifier plus the timestamps. Since both the start time and the end time are unique for all the states of a same entity, only one of them suffice. And since the end time is initialised with a default value before being changed later on, it seems more natural to use the start time. The genuine identifier of an entity type is thus the natural identifier plus the start time. The entity type corresponding to the example above is shown in Figure 13.3, both in a non-temporal and in a temporal versions.

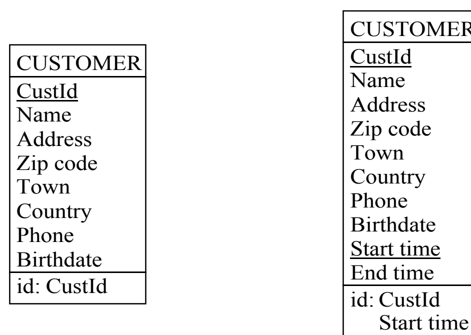


Figure 13.3 The Customer entity type, non-temporal version and temporal version

- The natural identifier of an entity type must be stable. In other words, the value of the natural identifier must remain unchanged among all its states. All other attributes can vary, except if otherwise expressed in the temporal entity type declaration.
- The two timestamp values of a state define an interval. To be precise and to avoid confusion, this interval is closed at left, open at right: [*start time*, *end time*[. The interval of the different states of an entity type E (with natural identifier I) do not overlap and leave no hole:

  $\forall$ e_1, $e_2 \in$ E such that $e_1 \neq e_2$ and $e_1.I = e_2.I$,
  [e_1.start time,e_1.end time[$\cap$ [e_2.start time,e_2.end time[= $\emptyset$

  If e_{first} is the first state of an entity e of type E, such that e.I = id, and if e_{last} is its last state, then

  $\bigcup_{i \text{ such that } e_i.I = id}$ [e_i.start time, e_i.end time[= [e_{first}.start time, e_{last}.end time[

[DETIENNE,01] implements such temporal databases within the relational SQL model, interprets all the traditional relational operators, and define new necessary ones in order to be able to handle temporal data structures. It also presents the two interpretations of the time – the transaction time and the valid time – and how to cope with both of them at the same time. Within the framework of this thesis, only the transaction time is of interest, and an object oriented model is used for the design of the repository.

Within an object oriented model, classes (entity types) are linked with rel-types. A role played by an object in a functional rel-type (one-to-many without attributes) is part of the object, like an attribute. So, creating a rel-type between two objects must force the creation of a new state for the temporal objects. Let us examine how the object oriented schemas have to evolve. Figure 13.4 shows a simple situation into which each object of class B has to be in relation with exactly one object of class A. If class A becomes a temporal class, as shown in Figure 13.5, each object of class B has to always be in relation with exactly one valid state of an object of class A (end time=999), and possibly one or several other states, of the same object or others, which are no longer valid (end time<999). So, the cardinality constraint of the role played by B in R has to be transformed from "1-1" into "1-N" and

the following constraint has to be added to the class B (the group labelled “Cst” in Figure 13.5):

$$\cup_{a \in b.R.A} [a.start\ time, a.end\ time[= [\min_{a \in b.R.A} a.start\ time, 999[$$
$$\wedge\ \exists!\ a \in b.R.A \text{ such that } a.end\ time=999$$

If the cardinality constraint of the role played by B in Figure 13.4 was “0-1” instead of “1-1”, it can become “0-N” in Figure 13.5 and the constraint is no more necessary.

If class B becomes temporal, see Figure 13.6, the roles do not need to be changed. Indeed, each state of an object of class B can be in relation with the same or with various objects of class A without any contradiction, and adding new relationships does not perturb the existence of objects of type A. Note, that instances of R will never be removed because of the “1-1” cardinality constraint and because of the fact that B is now temporal and none of its instance states will be removed. This implies that no constraint needs to be added. Having “1-1” instead of “0-1” in Figure 13.4 does not disturb this reasoning and the cardinality constraint would be kept too.

Figure 13.7 shows the same transformations in a schema containing a one-to-one rel-type. The constraint to be added when class A is made temporal is the same as in the case of the many-to-one rel-type above.

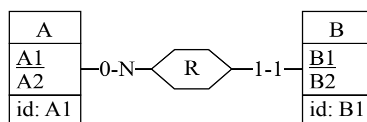


Figure 13.4 A non-temporal schema with a 1-N rel-type

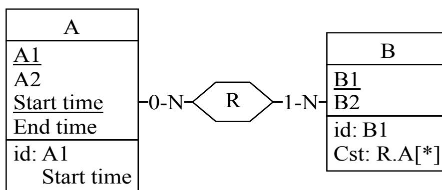


Figure 13.5 Class A has been made temporal

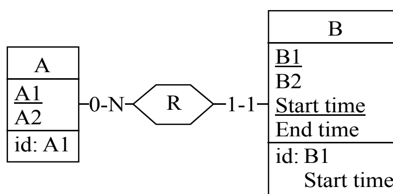


Figure 13.6 Class B has been made temporal

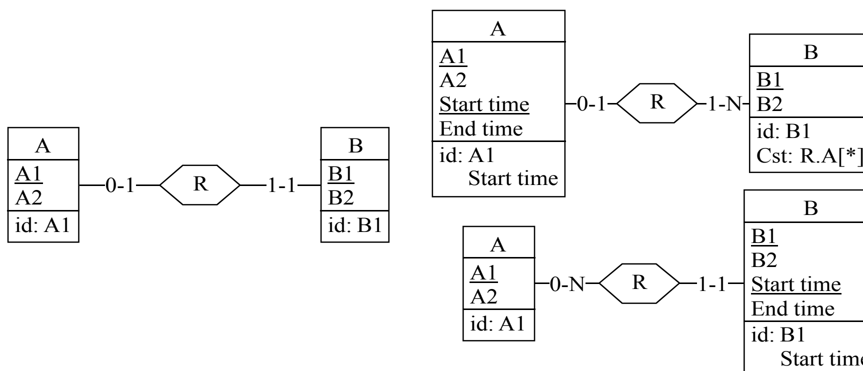


Figure 13.7 Transformations of a 1-1 rel-type

13.3.2. A solution proposal for the method evolution problem

The temporal database notions presented above can be used to enhance the repository presented in Chapter 10 in order for it to store all the successive states of a method during a project. When a product is used or a process is performed, its history is in relation with its type in Chapter 10. Now, its history will be put in relation with the current state of its type when it is executed. A product will always remain in relation with its type at the time of its creation. If a process type is performed several times, but in different states (it is modified between two uses), all the executions will be correctly stored and remain compliant with the state into which they were performed.

The part of the repository for the method shown in Figure 10.3 has to evolve.

All the paragraphs (schema-model, text-model, product type, toolbox, process type, method) in an MDL listing can evolve. So their counterpart class in the repository has to become temporal. But two of them, *schema_model* and *text_model* are specializations of a common super-class: model. So, the following classes must become temporal: *method*, *process_type*, *statement*, *toolbox*, *product_type*, and *model*. Two new attributes, *start_time* and *end_time*, must be added and the identifiers must be updated.

The specializations of a temporal class are automatically temporal too since they share the same attributes and identifiers. So *text_model*, *schema_model*, *text_type*, *schema_type* and *prod_set_type* can remain unchanged. An expression could be made temporal too because it can evolve. But since it is identified by the statement into which it is defined, a modification in an expression can be seen as a modification in the statement. So the full expression can be stored each time a new state of the statement that contains it is stored, in order to avoid complex time interval computation when accessing an expression. Usually, an expression is rather small in size, so this technique will not consume a lot of space. Since the *statement* class is temporal and the *expression* class is not, the *owner_of_parameter* class has to remain non temporal too. The same reasoning can be done for the *parameter* class and the *mod_concept* class as for the *expression* class, so they can remain unchanged too. To terminate the transformation, the rel-types still have to be updated too, and more specifically their roles, according to the rules presented in the previous section.

The format of timestamps has to be defined. When a method must be updated, several modifications may have to be taken into account. If no database engineer uses the method during the update – this can honestly be taken as an hypothesis – there is no need to distinguish the precise moment of the different modifications. The complete update, including all the modifications, can be a single transaction made at a single precise moment. So, simply numbering the method versions and using this integer number as a timestamp is sufficient. The method is created at moment 0, the first update at moment 1,...

Between two updates of the method, the CASE environment has to know which state is the last one. In other words, it has to know the value of *now*. This value has to be stored in a new attribute, named *now*, of the System class.

The new repository is shown in Figure 13.8. The following constraints must be added:

- mod_concept.cst:

  $\cup_{o \in have_concept.schema_model}[o.start_time, o.end_time[$
  $= [min_{o \in have_concept.schema_model}\ o.start_time, 999[$
  $\wedge\ \exists!\ o \in have_concept.schema_model$ such that $o.end_time=999$

- schema_type.cst:

  $\cup_{o \in schema_conform.schema_model}[o.start_time, o.end_time[$
  $= [min_{o \in schema_conform.schema_model} o.start_time, 999[$
  $\wedge\ \exists!\ o \in R.A$ such that $o.end_time=999$

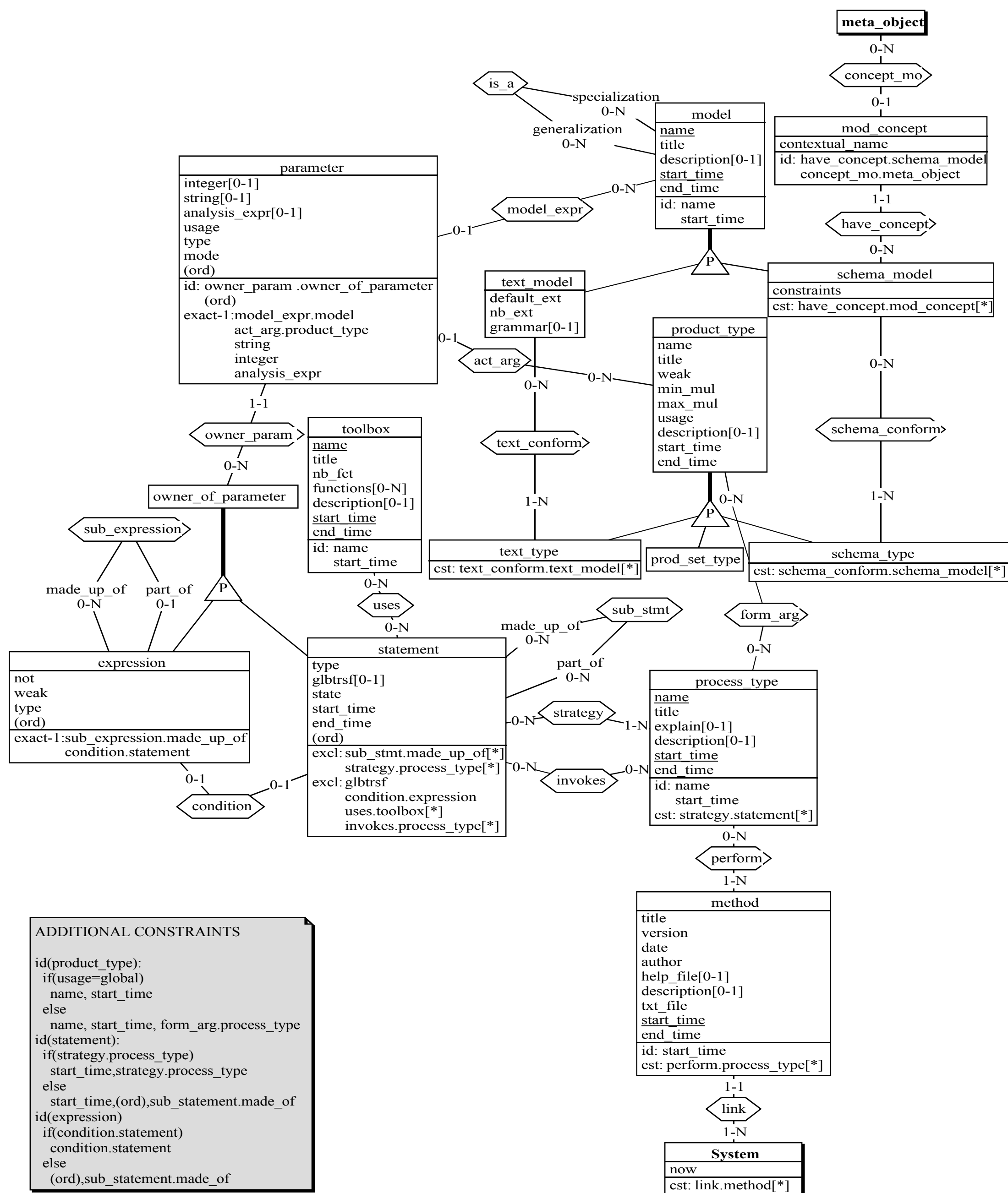


Figure 13.8 The temporal version of part of the repository for storing a method

- text_type.cst:

  $\bigcup_{o \in \text{text_conform.text_model}} [\text{o.start_time}, \text{o.end_time}[$
  $= [\min_{o \in \text{text_conform.text_model}} \text{o.start_time}, 999[$
  $\wedge\ \exists!\ o \in$ text_conform.text_model such that o.end_time=999

- process_type.cst:

  $\bigcup_{o \in \text{strategy.statement}} [\text{o.start_time}, \text{o.end_time}[$
  $= [\min_{o \in \text{strategy.statement}} \text{o.start_time}, 999[$
  $\wedge\ \exists!\ o \in$ strategy.statement such that o.end_time=999

- method.cst:

  $\bigcup_{o \in \text{perform.process_type}} [\text{o.start_time}, \text{o.end_time}[$
  $= [\min_{o \in \text{perform.process_type}} \text{o.start_time}, 999[$
  $\wedge\ \exists!\ o \in$ perform.process_type such that o.end_time=999

- System.cst:
  $\cup_{o\in \text{link.method}}[\text{o.start_time},\text{o.end_time}[$
  $= [\min_{o\in \text{link.method}} \text{o.start_time}, 999[$
  $\wedge\ \exists!\ o \in$ link.method such that o.end_time=999

In practice, the work to do to implement method evolution is the following:

- Implement the new repository.
- Update the MDL translator for it to compare the new MDL source text with the stored method state and to store only the new version of the modified parts of the method with an incremented value of *now*. The way to perform the comparison is still to be worked out.
- Update the CASE tool GUI to allow database engineers to see the various versions of the method.
- Update the methodological engine so that it always uses the last version (with end_time=999) of each component of the method.

This chapter just sketched the main lines of the method evolution problem. A more complete study is necessary in order to implement it correctly. This is left for future research.

Chapter 14

Conclusion and future works

14.1. Conclusion

In Chapter 1 it was claimed that database engineering processes can be modelled so that a CASE tool using the models can:

- force database engineers to perform projects in a specific way, the same as all their colleagues
- guide and help the database engineers perform their work
- keep a complete and reusable history of project developments

In the following chapters, database oriented schemas were formally defined, as well as other kinds of products that can be useful, and process types with their strategy. A semi-procedural Method Definition Language was developed. A complete history representation model has been defined too, onto which a set of selection and transformation operators have been designed. Then everything was put in practice by implementing new functions in an existing database engineering CASE tool to support process modelling and history recording and the use of these tool was experimented with case studies. The prototype has even been used for real projects which are not presented in this thesis.

The various uses of the current implementation of the CASE environment with a method has brought a series of comments from the database engineers, as well as several observations made by the author of this thesis:

- The help brought to the database engineers is real. The way the method is presented makes additional documentation unnecessary even if a little bit of training is required to understand the algorithms correctly.
- Designing a method really is different from writing a piece of program with a procedural language, as shown in Chapter 8, and a learning period is required.
- The use of a single click to start predefined processes (mainly primitive processes of an external type) allows engineers to work faster and to concentrate on the project in itself, rather than on the way to organise it.
- In some cases, some improvements would be useful. For instance, it is necessary to start manually all the automatic processes in a sequence (for instance, the five global transformations of the logical design process type in the first case study in Chapter 11). Users reports this to be a bit tedious. Indeed, even if the number of mouse clicks is reduced comparing to the same job without a method, database engineers still have to perform several times the same actions without the need of thinking and taking decision. Auto-

mation was presented in Chapter 9, but it is still not implemented and this lack of implementation proves the usefulness of the idea.

- When the method engineer and the database engineer are the same person (during the test phase of a method design process, for example), method evolution (Chapter 13) is lacking, so further study of the problem and implementation should come pretty soon.

Hence, overall, some work is still needed in order to get a professionally usable tool, but the first results are very promising. We are definitely on the right path. Some work still need to be done to make this work really industrially viable as expected in Chapter 1, section 1.4, but it is already no more a simple prototype.

This work concentrated on modelling processes for database engineering, but it can be extended to other domains of interest too, although a few problems have to be taken into account. Indeed, neither the model, nor the strategy part of the language is specific to a given paradigm. Among the problems to solve, the following ones can be cited:

- Database engineering with the DB-MAIN CASE tool does not need interfacing with third party tools for complementary process types. Software engineering, for instance, may require the use of high level text editors, compilers,... So an integration mechanism is required, especially concerning the definition and the management of the history. [POHL,99] addresses this problem.
- The classification of primitive processes in four types may have to be reviewed. For instance, a fifth type consisting of external tools that cannot be configured at all to guide the user may be necessary.
- New kinds of product models of interest may require new type of descriptions and new transformations. So, either the *schema-model* section of the MDL language can be updated by a new list of concepts and constraints (appendix A) and the *text-model* section (Appendix B) can be adapted to new requirements, or either these two sections maybe replaced by new sections. In the same way, the list of tools to put in toolboxes (Appendix E) and the list of global transformations (Appendix C) have to be replaced too.

In short, the main structure of the model and the language is valid whatever the paradigm, and only the details of the new product models and primitive process types need to be strongly revisited to cope with new paradigms, as well as the syntax and the generation of primitive process types histories.

14.2. Future works

The research work performed in this thesis has not reached an end. Future research lines are still open.

14.2.1. Method evolution implementation

Chapter 13 was entirely dedicated to the method evolution problem. It proposed an idea of solution and drew a path in the direction of this solution, but the big part of the job has still to be done.

14.2.2. Method engineering methodology

We proposed, in Chapter 8, a few elements of methodology for designing database engineering methods with the MDL language. But there is much more to write about the subject. Traditional methodological theories and techniques about procedural programming include structured programming, invariant theory, recursive design and recursion suppres-

sion techniques, program termination proving, and much more. These theories and techniques only hold for deterministic systems. Since an engineering method is oriented towards non-deterministic human beings, all these theories need to be revisited.

14.2.3. Method recovery

The history of a reverse engineering project can be used to recover a possible design history of the reverse engineered database, as presented in Chapter 7. The history of a method-free project can also be analysed in order to recover the method implicitly followed by the analyst. Indeed, the discovery of some patterns in the performance of actions and their ordering can help to discover typical design behaviours. From there, a possible method may be induced.

A few examples can enlighten us about this reasoning:

- The log files may show that the same sequence of transformations is carried out on similar structure patterns of the schema. This can be the tip for a loop in the method description. If it can be shown that the transformations tend to make the product satisfy some rules or a particular model, than these rules or this model can be used as a condition for the loop. Else, it can be a non-deterministic, condition-less loop.
- If a simple transformation is performed systematically to all the objects of a given type (for example to all rel-types), maybe a global transformation or an external function can do the job.
- If the log file is made up of a first block of transformations which are typical of a particular abstraction level (for example, the transformation of binary rel-types into referential groups of attribute is typical of a relational logical design), followed by a second block of transformations which are typical of another abstraction level (setting indexes on referential groups is a typical transformation of a physical level for example), then two engineering process types should describe these activities.
- Maybe more attention should be paid to a part of the log file where hesitancy appears than in a part that looks more fluent. For instance, if a few transformations are followed by their reverse in a short time, it seems that the engineer is looking for something. But if several transformations are performed, without being corrected or undone, and with a rather short and constant time interval between them, it seems that the engineer is working easily. So maybe the induced method should be of a greater help in the first case, than in the second one. Maybe the degrees of freedom left to the engineer should be different in the two situations.

To perform such analyses, large text files analysis and induction techniques are required. [HABRA,92] presents a solution using the RUSSEL rule-based language for analysing audit trails; it can be used with the primitive process log files too. A complete set of induction rules still have to be defined for the particular framework of the MDL language.

14.2.4. Graphical method development environment

In the beginning of Chapter 9 we enumerated various kinds of development environment, ranging from the simple command line to a complex RAD environment. For priority (the translator was the main objective) and usefulness reasons, an intermediate solution was chosen, made up of a simple development environment including an editor, the translator, an error report window and a simple graphical browser. For the ease of use of the method engineer and in order to make the tool look more professional, the development environment should evolve toward a complex RAD. It could include:

- A dialogue box to define schema models, including a list box with all the predefined concepts and the use of the schema analysis assistant presented in Chapter 9, and shown

in Figure 9.22, to specify the constraints.

- A dialogue box to define text models (to be completely designed from scratch).
- A dialogue box to define toolboxes, using a list into which the method engineer could simply select the right tools instead of typing their name.
- A graphical editor that would allow the method engineer to draw the algorithms directly.
- Refined text models
- ...

14.2.5. Extending to software engineering in general

From the start, the scope of this thesis was voluntarily reduced to database engineering only. This limitation has been justified in Chapter 1. But the software engineering community in general deserves to possess good tools to do their job, to be guided and to get a complete and well integrated history of the performed projects.

14.2.6. Supporting a Meta-CASE

Chapter 10 showed how the method support is implemented in a generic CASE tool. Since more and more meta-CASE tools are available on the market, it should be interesting to count with them as well. This is much more complex, it certainly deserves a new complete design, from the requirements analysis down to the implementation.

14.2.7. Supporting co-operative design

This thesis classify concerned people in two categories, namely method engineers and database engineers. But the latter could be further classified in sub-categories. Instead of allowing everybody to follow a method, the method engineer could assign some tasks to some specific categories of people. For instance, conceptual analysis could be assigned to analysts, while the physical tuning of a database could be assigned to system administrators.

Furthermore, the DB-MAIN CASE environment, including the MDL implementation, currently is a single-user application. Implementing the methodological engine in a multi-user environment would make new problems appear, such as access rights or concurrency.

Bibliography

AHO,89 Alfred Aho, Ravi Sethi, Jeffrey Ullman, *Compilateurs: Principes, techniques et outils*, InterÉditions, Paris, 1989, translated from *Compilers: Principles, Techniques and Tools*, Addison-Wesley, Mass., 1986.

BACHMAN,69 Charles W. Bachman, *Data Structure Diagrams*, DATA BASE 1(2) pp. 4-10, 1969.

BAETEN,95 J.C.M. Baeten, C. Verhoef, *Concrete process Algebra*, Einhoven University of Technology, Department of Mathematics and computing Science.

BANDINELLI,93 S. C. Bandinelli, A. Fuggetta, *Software Process Model Evolution in the SPADE Environment*, IEEE Transactions on Software Engineering, Vol. 19, N° 12, December 1993, pp. 1128-1144. Also in [GARG,96].

BARGHOUTI,90 N. S. Barghouti, G. E. Kaiser, *Consistency and Automation in Multi-User Rule-Based Development Environments*, Columbia University, Department of Computer Science, Technical Report CUCS-047-90, October 31, 1990.

BARROS,97 A. P. Barros, A. H. M. ter Hofstede, H. A. Proper, *Towards Real-Scale Business Transaction Workflow Modelling*, CAiSE'97, Advanced Information Systems Engineering, Barcelona, June 1997, LNCS 1250, pp. 319-332.

BELKHATIR,94 N. Belkhatir, J. Estublier, W. Melo, *The ADELE-TEMPO experience: an environment to support process modeling and enaction*, in [FINKELSTEIN,94], pp. 187-222.

BENGHEZALA,01 H. Hadjami Ben Ghezala, R. Beltaifa Hajri, *Towards a Systematic Reuse Based on both General Purpose and Domain-Specific Approaches*, Proc. of Japan-Tunisia Workshop on INformatics, JTWIN 2001, University of Tsukuba, October 25-26, 2001.

BOBILLIER,99 M.-E. Bobillier, *Les transferts d'apprentissage dans le cadre des transferts technologiques informatiques: le cas du maquettage en conception informatique*, PhD thesis, Université de Metz, Psychologie, 1999.

BODART,95 F. Bodart, A. Hennebert, J. Lheureux, I. Provot, B. Sacré, J. Vanderdonckt. *Towards a systematic building of software architecture: The TRIDENT methodological guide*. In Design, Specification and Verification of Interactive Systems, pp. 262-278, Vienna, 1995. Springer Verlag.

BOEHM,88 B. Boehm, *A Spiral Model of Software Development and Enhancement*, IEEE Computer, vol.21, N° 5, May 1988, pp 61-72.

BOGUSCH,99 R. Bogusch, B. Lohmann, W. Marquardt, *Computer-Aided Process Modeling with MODKIT*, Technical Report LPT-1999-23, Lehrstuhl für Prozeßtechnik, RWTH Aachen, 1999.

BRATAAS,97, G. Brataas, P. H. Hughes, A. Sølvberg, *Performance Engineering of Human and Computarized Workflows*, CAiSE'97, Advanced Information Systems Engineering, Barcelona, June 1997, LNCS 1250, pp. 187,202.

BRINKKEMPER,01 S. Brinkkemper, M. Saeki, F. Harmsen, *A Method Engineering Language for the Description of Systems Development Methods (Extended Abstract)*, CAiSE 2001, LNCS 2068, pages 473-476.

BROCKERS,93 A. Bröckers, V. Gruhn, *Computer-Aided Verification of Software Process Model Properties*, Colette Rolland, François Bodart, Corine Cauvet editors: Advanced Information Systems Engineering, CAiSE'93, Paris, France, June 8-11, 1993, LNCS 685, pp. 521-546.

BRUNO,95 G. Bruno, R. Agarwal, *M_{CASE}: Model-based CASE*, Proc. of the 7th International Workshop on Computer-Aided Software Engineering CASE'95, Toronto, Ontario, Canada, July 9-14, 1995, pp 152-161.

CAPGEMINI,95 Cap Gemini Sogeti, *Process Weaver, General Information Manual, Version PW2.1*, 1995.

CASTANO,99 S. Castano, V. De Antonellis, M. Melchiori, *A Methodology and Tool Environment for Process Analysis and Reengineering*, Data & Knowledge Engineering 31, 1991, Elsevier Science, pp. 253-278.

CATARCI,00 T. Catarci, *What happened when database researchers met usability*, Information Systems Vol. 25, N° 3, pp. 177-212, 2000.

CHANG,73 C.-L. Chang, R. C.-T. Lee, *Symbolic Logic and Mechanical Theorem Proving*, Academic Press, Computer Science Classics, 1973, ISBN 0-12-170350-9.

CHEN,71 Wai-Kai Chen, *Applied Graph Theory*, North-Holland Publishing Company - Amsterdam, London, 1971.

CHEN,76 P. P. Chen, *The Entity-Relationship model: Toward a Unified View of Data*, ACM TODS, vol. 1, n°1, 1976.

CHUNG,91 L. Chung, P. Katalagarinos, M Marakakis, M. Mertikas, J. Mylopoulos, Y. Vassiliou, *From Information System Requirements to Designs: a Mapping Framework*, Information Systems, Vol. 16, N° 4, PP. 429-461, 1991.

CLOCKSIN,84 W. F. Clocksin, C. S. Mellish, *Programming in Prolog*, second edition, Springer-Verlag, 1984, ISBN 3-540-15011-0, 0-387-15011-0.

COLE,95 A. J. Cole, S. J. Wolak, J. T. Boardman, *A Computer-based Process Handbook for a System Engineering Business*, Proc. of the 7th International Workshop on Computer-Aided Software Engineering CASE'95, Toronto, Ontario, Canada, July 9-14, 1995, pp. 172-181.

COLLINS,95 English *Dictionary*, Harper Collins Publishers, The Cobuild Series, Great Britain, 1995.

COLLONGUES,89 A. Collongues, J. Hugues, B. Laroche, R. Malgoire, *Merise: 1. méthode de conception*, Dunod Paris, 1989.

CONRADI,93 R. Conradi, *Customization and Evolution of Process Models in EPOS*, IFIP 8.1 ISDP'93, Como, Italy, September 1-3, 1993.

CONRADI,94a R. Conradi, C. Fernström, A. Fuggetta, *Concepts for Evolving Software Processes*, in [FINKELSTEIN,94], pp. 9-31.

CONRADI,94b R. Conradi, M. Hagaseth, J.-O. Larsen, M. N. Nguyên, B. P. Munch, P. H. Westby, W. Z. Letizia Jaccheri, C. Liu, *EPOS: Object-Oriented Cooperative Process Modelling*, in [FINKELSTEIN,94], pp. 33-70.

CUGOLA,95 G. Cugola, E. Di Nitto, C. Ghezzi, M. Mantione, *How To Deal With Deviations During Process Model Enactment*, Proc. of 17th Intl. Conf. on Software Engineering, Seattle 1995.

CUGOLA,99 G. Cugola, C. Ghezzi, *Design and Implementation of PROSYT: a Distributed Process Support System*, IEEE 8th Inth Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises, Palo Alto, California, June 16-18, 1999.

CURTIS,92 B. Curtis, M. I. Kelner, J. Over, *Process Modeling*, Communications of the ACM, September 1992, Vol.35 No.9.

DAHL,67 O.-J. Dahl, K. Nygaard, *SIMULA: a language for programming and description of discrete event systems*, Norwegian Computing Center, Oslo, 1967.

DAMI,97 S. Dami, J. Estublier, M. Amiour, *APEL: a Graphical Yet Executable Formalism for Process Modeling*, in Automated Software Engineering: an International Journal, Vol. 5, N° 1, January 1998, pp. 61-96.

DBMAIN,02a DB-MAIN team, *DB-MAIN 6.5 Reference Manual*, FUNDP Institut d'Informatique, www.db-main.be, 2002.

DBMAIN,02b DB-MAIN team, *DB-MAIN.HLP file*, included in the DB-MAIN package, FUNDP Institut d'Informatique, 2002.

DEITERS,90 W. Deiters, V. Gruhn, *Managing Software Process in the Environment MELMAC*, ACM SIGSOFT Software Engineering Processes in the Environment MELMAC, Vol. 15, N° 6, December 1990, pp. 193-205.

DELCROIX,01 Ch. Delcroix, *Plan de transformation d'un schéma conceptuel en un schéma XML*, FUNDP, Institut d'Informatique, Technical report, 2001.

DETIENNE,01 V. Detienne, J.-L. Hainaut, *CASE Tool Support for Temporal Database Design*, H.S. Kunii, S. Jajodia, and A. Sølvberg Editors, Proc. of the 20th International Conference on Conceptual Modeling (ER 2001), Yokohama, Japan, November 2001, Springer LNCS 2224, pp. 208-224.

DEWITTE,97 P. S. de Witte, C. Pourteau, *IDEF enterprise engineering methodologies support simulation*, Manufacturing Systems: Information Technology for Manufacturing Managers, March 1997, pp. 70-75.

DIJKSTRA,62 E. W. Dijkstra, *Primer of Algol 60 Programming*, Academic Press, June 1962.

DIJKSTRA,68 E. W. Dijkstra, *Go To Statement Considered Harmful*, Communications of the ACM, Vol. 11, N° 3, March 1968, pp. 147-148.

DITTRICH,00 K. Dittrich, D. Tombros, A. Geppert, The Future of Software Engineering, A. Finkelstein editor, ACM Press, May 2000.

DOMGES,98 R. Dömges, K. Pohl, *Adapting Traceability Environments to Project-Specific Needs*, Communications of the ACM, Vol. 41, N° 12, December 1998.

DOMINGUEZ,97 E. Domínguez, M. A. Zapata, J. Rubio, *A Conceptual Approach to Meta-Modelling*, CAiSE'97, Advanced Information Systems Engineering, Barcelona, June 1997, LNCS 1250, pp. 319-332.

DOWSON,94 M. Dowson, C. Fernström, *Towards Requirements for Enactment Mechanisms*, Proc. of the 3rd European Workshop on Software Process Technology (EWSPT94), Villard de Lens, France, Feb. 7-9, 1994, LNCS 772, pages 90-106.

DSOUZA,98 D. F. D'Souza, A. C. Wills, *Objects, components and Framework with UML; the Catalysis Approach*, Addisson-Wesley, Object Technology Series, 1998, ISBN 0-201-31012-0.

DUBOIS,94 E. Dubois, P. Du Bois, M. Petit, *ALBERT II: An Agent-oriented Language for Building and Eliciting Requirements for real-Time systems*, Proc. of the 27th Hawaii Intl. Conf. on System Science (HICSS-27), 1994.

ENGELS,94 Gregor Engels and Luuk P.J. Groenewegen, *SOCCA: Specifications of Coordinated and Cooperative Activities*, in [FINKELSTEIN,94], pp. 71-102.

ENGLEBERT,95 V. Englebert, J. Henrard, J.-M. Hick, D. Roland, J.-L. Hainaut, *DB-MAIN: un atelier d'ingénierie de bases de données*, in Proc. of the "11èmes journées Base de Données Avancées", Nancy (France), September 1995. Also in Ingénierie des systèmes d'information, Vol. 4, N° 1/1996, pp. 87-116.

ENGLEBERT,99 V. Englebert, *Voyager 2 Reference Manual*, technical DB-MAIN documentation.

EPOS,95 http://www.idi.ntnu.no/~epos/OVERVIEW/epos/epos.html

ESTIEVENART,02 F. Estiévenart, *Méthode et outils pour la conception de bases de données XML natives*, FUNDP, Institut d'Informatique, mémoire, 2002.

ESTUBLIER,94 J. Estublier, R. Casallas, *The Adele Configuration Manager*, in Tichy editor, Configuration Management. John Wiley & Sons, 1994.

ESTUBLIER,96 J. Estublier, S. Dami, *Process Engine Interoperability: An Experiment*, in Proc. European Workshop on Software Process Technology (EWSPT5), Nancy, France, October 9-11, 1996. LNCS 1149.

FAUSTMANN,99 G. Faustmann, *Enforcement vs. Freedom of Action – An Integrated Approach to Flexible Workflow Enactment*, ACM SIGGROUP Bulletin, Vol. 20, Issue 3, December 1999.

FEILER,93 P. H. Feiler, W. S. Humphrey, *Software Process Development and Enactment: Concepts and Definitions*, Proc. of Second International Conference on the Software Process, (ICSP-2), IEEE Press, February 1993.

FERNSTROM,93 C. Fernström, *PROCESS WEAVER: Adding Process Support to UNIX*, Proc of the 2nd International Conference on the Software Process, Berlin, Germany, February 25-26, 1993. Also in [GARG,96].

FICKAS,85 S. F. Fickas, *Automating the Transformational Development of Software*, IEEE Transactions on Software Engineering, Vol. 11, N°11, November 1985.

FINKELSTEIN,92 A. Finkelstein, J. Kramer, M. Hales, *Process Modelling: a critical analysis*, Integrated Software Reuse: Management and Techniques, P. Walton, N. Maiden editors, Chapman and Hall and UNICOM, 1992, pages 137-148.

FINKELSTEIN,94 A. Finkelstein, J. Kramer, B. Nuseibeh editors, *Software Process Modelling and Technology*, Research Studies Press Ltd., England, John Wiley & Sons Inc., ISBN 0-86380-169-2

FINKELSTEIN,00 A. Finkelstein, J. Kramer, *Software Engineering: A Roadmap*, The Future of Software Engineering, A. Finkelstein editor, ACM Press, May 2000.

FOUCAUT,78 O. Foucaut, Colette Rolland, *Concepts for Design of an Information System Conceptual Schema and its Utilization in the REMORA Project*, Proc of the 4th VLDB conf., September 13-15, 1978, West Berlin, Germany, pp. 342-350.

FROEHLICH,95 G. Froehlich, J.-P. Tremblay, P. Sorenson, *Providing Support for Process Model Enaction in the Metaview Metasystem*, Proc. of the 7th International Workshop on Computer-Aided Software Engineering CASE'95, Toronto, Ontario, Canada, July 9-14, 1995, PP 141-149.

GARG,96 P. K. Garg, M. Jazayeri editors, *Process-Centered Software Engineering Environments*, IEEE Computer Society Press, Los Alamitos, California, 1996.

GHEZZI,91 C. Ghezzi, M. Jazayari, D. Mandrioli, *Fundamentals of Software Engineering*, Prentice-Hall, 1991, ISBN 0-13-818204-3.

GREEN,00 P. Green, M. Rosemann, *Integrated Process Modeling: an Ontological Evaluation*, Information Systems Vol. 25, N° 2, pp. 73-87, 2000.

HABRA,92 N. Habra, B. Le Charlier, A. Mounji, I. Mathieu, *ASAX: Software Architecture and Rule-Based Language for Universal Audit Trail Analysis*, Proc. of ESORICS'92, European Symposium on Research in Computer Security, November 23-25, Toulouse, Springer-Verlag, 1992.

HAINAUT,89 J-L. Hainaut, *A Generic Entity-Relationship Model*, in Proc. of the IFIP WG 8.1 Conf. on Information System Concepts: an in-depth analysis, North-Holland, 1989.

HAINAUT,94 J-L. Hainaut, V. Englebert, J. Henrard, J-M. Hick, D. Roland, *Evolution of database Applications: the DB-MAIN Approach*, in Proc. of the 13th Int. Conf. on ER Approach, Manchester, Springer-Verlag, LNCS 881, 1994

HAINAUT,95 J-L Hainaut, V. Englebert, J. Henrard, J-M. Hick, D. Roland, *Requirements for Information System Reverse Engineering Support*, in Proc. of the IEEE Working Conference on Reverse Engineering, Toronto, IEEE Computer Society Press, July 1995

HAINAUT,96a Hainaut J.-L., Englebert V., Henrard J., Hick J.-M., Roland D., *Database Reverse Engineering: from Requirement to CARE tools*, Journal of Automated Software Engineering, 3(2), 1996, Kluwer Academic Press.

HAINAUT,96b Hainaut J.-L., Henrard J., Hick J.-M., Roland D., Englebert V., *Database Design Recovery*, in Proc of the 8th Conf. on Advanced Information Systems Engineering (CAISE'96), Heraklion (Crete, Greece), Springer-Verlag, 1996.

HAINAUT,96c Hainaut J.-L., *Specification preservation in schema transformations - Application to semantics and statistics*, Data & Knowledge Engineering, 16(1), 1996, Elsevier Science Publish.

HAINAUT,96d Hainaut J.-L., Roland D., Englebert V., Hick J.-M., Henrard J., *Database Reverse Engineering - A Case Study*, in Actes du 2ème Forum International d'Informatique Appliquée (FIIA96), Tunis, March 12-14, 1996.

HARANI,98 Y. Harani, *Modèle de produit et modèle de processus pour la représentation de l'acrtivité de conception*, Revue Internationale d'Ingénierie des Systèmes de Production Mécanique, N° 1, novembre 1998, pp. V-11 – V-20.

HAUMER,99 P. Haumer, M. Jarke, K. Pohl, K. Weidenhaupt, *Improving Reviews of Conceptual Models by Extended Traceability to Captured System Usage*, Crews Report 99-16, Information Systems, RWTH Aachen, Germany, 1999.

HENDERSON,90 B. Henderson-Sellers, J. M. Edwards, *The Object-Oriented Systems Life Cycle*, Communications of the ACM, Vol. 33, N° 9, September 1990.

HENRARD,96 Henrard J., Hick J.-M., Roland D., Englebert V., Hainaut J.-L., *Techniques d'analyse de programmes pour la rétro-ingénierie de base de données*, submitted to INFORSID'96, 1996.

HENRARD,98 J. Henrard, D. Roland, V. Englebert, J.-M. Hick, J.-L. Hainaut, *Outils d'analyse de programmes pour la rétro-conception de bases de données*, Proc. of INFORSID'98, May 12-15, 1998, Montpellier, France.

HICK,98 Hick, J-M., Hainaut J-L., *Maintenance et évolution d'applications de bases de données*, Research Paper RP-98-005, FUNDP, Journées sur la Ré-ingenierie des Systèmes d'Information - RSI'98, Lyon (France), 1-2 avril 1998.

HICK,99 J.-M. Hick, J.-L. Hainaut, V. Englebert, D. Roland, J. Henrard, *Stratégies pour l'évolution des applications de bases de données relationnelles : l'approche DB-MAIN*, in XVIIe congrès INFORSID, Toulon, 1999.

HICK,01 J.-M. Hick, *Évolution d'applications de bases de données relationnelles: méthodes et outils*, PhD thesis, FUNDP, Institut d'Informatique, Namur, Belgium, September 26, 2001.

HUMPHREY,95 W. S. Humphrey, *A Discipline for Software Engineering*, Addisson-Wesley, the SEI Series in Software Engineering, 1995, ISBN 0-201-54610-8.

JACCHERI,98 M. L. Jaccheri, P. Lago, G. P. Picco, *Eliciting Software Process Models with E3 Language*, ACM Transcations on Software Engineering and Mathodology, Vol. 7, N° 4, 1998, pp 368-410.

JAMART,94 P. Jamart, A. van Lamsweerde, *A Reflective Approch to Process Model Customization, Enactment and Evolution*, Proc. ICSP3, 3rd Intl. Conf. on Software Process, IEEE Computer Society Press, 1994, pp. 21-32.

JARKE,92 M. Jarke, J. Mylopoulos, J. W. Schmidt, Y. Vasciliou, *DAIDA: An Environment for Evolving Information Systems*, ACM Transactions on Information Systems, Vol. 10, N° 1, January 1992, pp. 1-50.

JARKE,93 M. Jarke, editor. *Database Application Engineering with DAIDA*, Springer - Verlag, 1993.

JENSEN,78 K. Jensen, N. Wirth, *Pascal User Manual and Report*, second edition, Springer-Verlag, ISBN 0-387-90144-2, 3-540-90144-2.

JORGENSEN,99 H. D. Jorgensen, S. Carlsen, *Emergent Workflow: Planning and Performance of Process Instances*, Proc. of Workflow Management '98, Münster, Germany, November 9, 1999.

JORGENSEN,00a H. D. Jorgensen, *Software Process Model Reuse and Learning*, Proc. of Process Support for Distributed Team-based Software Development (PDTSD'00), SCI2000, Orlando, July 2000.

JORGENSEN,00b H. D. Jorgensen, *Supporting Knowledge Work with Emergent Process Models*, CSCW 2000, Workshop: Beyond Workflow Management, Supporting Dynamic Organisational Processes, Philadelphia, December 2000.

JUNKERMANN,94 G. Junkermann, B. Peuschel, W. Schäfer, S. Wolf, *Merlin: Supporting Cooperation in Software Development through a Knowledge-based Environment*, in [FINKELSTEIN, 94], pp. 103-130.

KATAYAMA,89 T. Katayama, *A Hierarchical and Functional Software Process Description and its Enaction*, Proc. of the 11th International Conference on Software Engineering (ICSE '89), Pittsburgh, USA, 1989, pp. 343-352.

KELLY, 96, S. Kelly, K. Lyytinen, *MetaEdit+, A Fully Configurable Multi-User and Multi-Tool CASE and CAME Environment*, CAiSE'96, Advanced Information System Engineering, Heraklion, Crete, Greece, May 1996, LNCS 1080, pp. 1-21.

KIM,95 Y.-G. Kim, S. T. March, *Comparing Data Modeling Formalisms*, Communications of the ACM, Vol. 38, N° 6, June 1995, pp. 103-115.

KORTH,91 H. F. Korth, A. Silberschatz, *Database System Concepts*, second edition, McGraw Hill Intl Editions, Computer Science Series, 1991, ISBN 0-07-100804-7.

KRASNER,92 H. Krasner, J. Terrel, A. Linehan, P. Arnold, W. H. Ett, *Lessons Learned from a Software Process Modeling System*, Communications of the ACM, September 1992, Vol. 35, N° 9, pp. 91-100.

KRUCHTEN,01 Ph. Kruchten, *A Rational Development Process*, white paper, Rational Software Corp. Vancouver, Canada, 2001.

LACAZE,02 X. Lacaze, Ph. Palanque, D. Navarre, *Analyse de performance et modèles de tâches comme support à la conception rationelle des systèmes interactifs*, IHM 2002, November 26-29, 2002, Poitiers, France, ACM Press, pages 17-24.

MARCHAL,01 B. Marchal, *XML by example*, QUE publishing, ISBN 0-7897-2504-5.

MARTTIIN,88 P. Marttiin, M. Koskinen, *Similarities and Differences of Method Engineering and Process Engineering Approaches*, M. Khosnowpour editor, Effective Utilization and Management of Emerging Information Technologies, Harrisburg, Idea Group Publishing, pages 420-427.

MAYER,98 R. J. Mayer, P. C. Benjamin, B. E. Caraway and M. K. Painter, *A Framework and a Suite of Methods for Business Process Reengineering*, www.idef.comarticles, October 30, 1998.

MCCARTHY J. McCarthy, *Recursive Functions of Symbolic Expressions and their computation by machine*, Communications of the ACM, April 1960.

MCLEOD K. McLeod, *Jackson Structured Programming, & Introduction to JSD,* http://www.cee.hw.ac.uk/ism/ug1/jsd/jspjsd.htm.

MUETZELFELDT,01 R. Muetzelfeldt, J. Taylor, *Getting to know SIMILE the visual modelling environment for ecological, biological and environmental research*, University of Edinburgh, Institute of Ecology and Resource Management, www.ierm.ed.ac.uk/simile, 2001.

MYLOPOULOS,92 J. Mylopoulos, L. Chung, B. Nixon, *Representing and Using Nonfunctional Requirements: A Process-Oriented Approach*, IEEE TSE, Vol. 18, No. 6, June 1992.

NATURE,96 Nature Team, *Defining Visions In Context: Models, Processes And Tools For Requirements Engineering*, Information Systems, Vol. 21, No 6, 1996.

NUSEIBEH,93 B. Nuseibeh, A. Finkelstein, J. Kramer, *Fine-Grain Process Modelling*, Proc. of the 7th Intl Workshop on Software Specification and Design, Redondo Beach, California, USA, December 6-7, 1993, IEEE CS Press, Pages 42-46.

OIVO,92 M. Oivo, V. R. Basili, *Representing Software Engineering Models: The TAME Goal Oriented Approach*, IEEE transactions on Software Engineering, Vol. 18, N° 10, October 1992, pp. 886-898.

OMG,01 OMG, *Unified Modeling Language, V. 1.4*, www.omg.org, September 2001.

OSTERWEIL,97 L. J. Osterweil, *Software Process are software too, revisited: An Invited Talk on the Most Influential Paper of ICSE9**, Proc. of the 19th Intl Conf. on Software Engineering, Boston, USA, May 1997.

POHL,96 K. Pohl, *Process-Centered Requirements Engineering*, Research Studies Press Ltd,1996

POHL,97 K. Pohl, R. Dömges, M. Jarke, *Towards Method-Driven Trace Capture*, CAiSE'97, Advanced Information Systems Engineering, Barcelona, June 1997, LNCS 1250, pp. 103-116.

POHL,99 K. Pohl, K. Weidenhaupt, R. Dömges, P. Haumer, M. Jarke, R. Klamma, *PRIME: Towards process-integrated environments*, ACM Transactions on Software Engineering and Methodology, Vol. 8, N° 4, October 1999.

POTTS,88 C. Potts, G. Bruns, *Recording the Reasons for Design Decisions*, in ICSE 88, 1988.

PROFORMA,99 Proforma Corporation, *Enterprise Application Modeling*, technical paper, 1999.

RALYTE,01a J. Ralyté, *Ingénierie des méthodes à base de composants*, PhD thesis, Université Paris 1 – Sorbonne, France, 2001

RALYTE,01b J. Ralyté, C. Rolland, *An approach for Method Reengineering*, Proc. of the 20th Intl Conf. on Conceptual modeling, Yokohama, Japan, November 2001, Springer LNCS 2224, pp 471-484.

ROLAND,97 D. Roland, J.-L. Hainaut, *Database Engineering Process Modelling*, Proceedings Of The First International Workshop On The Many Facets Of Process Engineering, Gammarth, Tunisia, September 22-23, 1997.

ROLAND,99 D. Roland, , J-L. Hainaut, J. Henrard, J-M. Hick, V. Englebert, *Database engineering process history*, Proceedings of the second International Workshop on the Many Facets of Process Engineering, Gammarth, Tunisia, May 1999.

ROLAND,00 D. Roland, J-L. Hainaut, J.-M. Hick, J. Henrard, V. Englebert, *Database Engineering Processes with DB-MAIN*, Proc. of the 8th European Conference on Information Systems, ECIS 2000, Vienna, July 3-5, 2000, pp. 244-251.

ROLLAND,93 C. Rolland, *Modeling the Requirements Engineering Process*, in 3rd European-Japanese Seminar on Information Modeling and Knowledge Bases, Budapest, May 1993.

ROLLAND,95 C. Rolland, C. Souveyet, M. Moreno, *An Approach For Defining Ways-Of-Working*, Information Systems, Vol. 20, No. 4, pp. 337-359, 1995.

ROLLAND,96 C.Rolland, *L'ingénierie des processus de développement de système : un cadre de référence,* Ingénierie des systèmes d'information, Vol. 4, No 6, 1996.

ROLLAND,97 C. Rolland, *A primer for method engineering*, CREWS Report Series 97-06, Proceedings of the conference INFORSID, Toulouse, France, June 10-13, 1997.

ROSENTHAL,94 A. Rosenthal, D. Reiner, *Tools and Transformations - Rigourous and Otherwise - for Practical Database Design*, ACM TODS, Vol. 19, No. 2, June 1994

ROYCE,70 W. W. Royce, *Managing the Development of Large Software Systems*, Proceedings of IEEE WESTCON, San Francisco, August 1970.

SADIQ,00 W. Sadiq, M. E. Orlowska, *Analysing Process Models Using Graph Reduction Techniques*, Information Systems Vol. 25 N° 2 pp. 117-134, 2000.

SAEKI,94 M. Saeki, K. Wenyin, *Specifying Software Specification & Design Methods*, Proc. of Advanced Information Systems Engineering, CAiSE'94, Utrecht, The Netherlands, June 6-10, 1994,

SCHLENOFF,96 C. Schelnoff, A. Knutilla, A. Ray, *Unified Process Specification Language: Requirements for Modeling Process*, US Department of Commerce, Technology Administration, National Institue of Standards and Technology, NISTR 5910, September 1996.

SCHLENOFF,00 C. Schlenoff, M. Gruninger, F. Tissot, J. Valois, J. Lubell, J. Lee, *The Process Specification Language (PSL), Overview and Version 1.0 Specification*, National Institute of Technology, Gaithersburg, NISTIR6459, 2000.

SISAID,96 S. Si-Said, C. Rolland, G. Grosz, *MENTOR: A Computer Aided Requirements Engineering Environment*, CAiSE'96, Advanced Information System Engineering, Heraklion, Crete, Greece, May 1996, LNCS 1080, pp. 22-43. Same in French: G. Grosz, S. Si-Said, C. Rolland, *Mentor: un environnement pour l'ingénierie des méthodes et des besoins*, INFORSID'96, Bordeaux, 4-7 juin 1996.

SMITH,01 J. Smith, *A comparison of RUP and XP*, Rational Software White Paper, 2001.

SORENSON,88 P. G. Sorenson, J. P. Tremblay, A. J. McAllister, *The Metaview System for Many Specification Environments*, IEEE Software, Vol.5, N° 2, March 1988, pages 30-38.

SOUQUIERES,93 J. Souquières, N. Lévy, *Description of Specification Developments*, in Proceedings of RE'93, San Diego (CA), 1993.

STEELE,90 Guy L. Steele, *Common Lisp: The Language, 2nd Edition*, Digital Press, 1990. 1029 pages, ISBN 1-55558-041-6.

SUTCLIFFE,00 A. G. Sutcliffe, *Requirements Analysis for Socio-Technical System Design*, Information System Vol. 25, N° 3, PP. 213-233, 2000.

SUTTON,90 S. M. Sutton Jr., D. Heimbigner, L. J. Osterweil, *Language Constructs for Managing Change in Process-Centered Environments*, in Proc. of the 4th International Symposium on Practical Software Development Environment, ACM SIGSOFT notes, Vol. 15, N° 6, pp. 206-217, 1990.

TAWBI,99 M. Tawbi, C. Souveyet, *Guiding Requirement Engineering with a Process map*, Proceedings of the second International Workshop on the Many Facets of Process Engineering, Gammarth, Tunisia, May 1999.

TAYLOR,88 R. N. Taylor, F. C. Belz, L. A. Clarke, L. Osterweil, R. W. Selby, J. C. Wileden, A. L. Wolft, M. Young, *Foundations for the Arcadia Environment Architecture*, Proc. of the 3rd ACM SIGSOFT/SIGPLAN symposium Practical Software Development Environments, ACM Press, New-York, 1988, pp. 1-13. Also in [GARG,96].

TOLVANEN,98 J.-P. Tolvanen, *Incremental Method Engineering with Modeling Tools: Theoretical Principles and Empirical Evidence*, PhD Thesis, University of Jyväskylä, 1998, ISBN 951-39-0303-6.

VANDERAALST,02 W.M.P. van der Aalst, B.F. van Dongen, *Discovering Workflow Performance Models from Timed Logs,* In Y. Han, S. Tai, and D. Wikarski, editors, International Conference on Engineering and Deployment of Cooperative Information Systems (EDCIS 2002), LNCS 2480, pages 45-63. Springer-Verlag, Berlin, 2002

VANDERDONCKT,97 J. Vanderdonckt. *Conception assistée de la présentation d'une interface homme-machine ergonomique pour une application de gestion hautement interactive,* PhD thesis, Facultés Universitaires Notre-Dame de la Paix, Institut d'Informatique, Namur, 1997.

VONDRAK,01 Ivo Vondrák, *Business Process Modeling and Workflow Automation*, Proc. of the 4th International Conference on Information Systems Modelling, ISM'01, Hradec nad Moravici, Czech Republic, May 9-11, 2001.

WANG,95 X. Wang, P. Loucopoulos, *The Development of Phedias: a CASE Shell*, Proceedings of the Seventh International Wokshop on Computer-Aided Software Engineering, Toronto, July 10-14, 1995.

WEISER,84 M. Weiser, *Program Slicing*, IEEE Transactions on Software Engineering, Vol. 10, N° 4, July 1984, pp. 352-357.

WELZEL,92 D. Welzel, *Embedding and Evaluating of Software Assessment within a Process Model*, Proc. ERCIM Workshop on Software Quality Principles & Techniques, Pisa, May 21-22, 1992.

XP, www.xprogramming.com

YONESAKI,93 N. Yonesaki, M. Saeki, J. Ljungberg, T. Kinnula. *Software Process Modeling with the TAP Approach - Tasks-Agents-Products*, in 3rd European-Japanese Seminar on Information Modeling and Knowledge Bases, Budapest, May 1993.

ZAMFIROIU,98 M. Zamfiroiu, *Contribution à la traçabilité du processus de conception en génie logiciel*, PhD thesis, Université de Paris IX-Dauphine, UFR Sciences des Organisations, Paris, December 1998.

ZAMFIROIU,01 M. Zamfiroiu, N. Prat, *Traçabilité du processus de conception des systèmes d'information*, in Ingénierie des systèmes d'information, Corine Cauvet, Camille Rosenthal-Sabroux editors, Hermès Sciences Publication, Paris, 2001, pp. 245-276.

ZEROUAL,92 K. Zeroual, P.-N. Robillard, *KBMS: A Knowledge-Based System for Modeling Software System Specifications*, IEEE Transactions on Knowledge and Data Engineering, Vol. 4, N° 3, June 1992.

Appendix A

Schema analysis predicates

This appendix is a complete listing of the structural predicates on schemas included in version 6,* of the DB-MAIN CASE environment. These predicates are used for constraints in schema models, for formal expressions in conditional instructions of the strategies and by the schema analysis assistant of the CASE environment.

A.1. Constraints on schema

ET_per_SCHEMA (***min max***)
The number of entity types per schema must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or N.

RT_per_SCHEMA (***min max***)
The number of rel-types per schema must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or N.

COLL_per_SCHEMA (***min max***)
The number of collections per schema must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or N.

DYN_PROP_of_SCHEMA (***dynamic-property parameters***)
Check some properties of the dynamic properties.
☞ See Section A.16.

SELECTED_SCHEMA
Search for all selected objects. This constraint should not be used for validation.
☞ No parameters.

MARKED_SCHEMA
Search for all marked objects. This constraint should not be used for validation.
☞ No parameters.

V2_CONSTRAINT_on_SCHEMA (***V2-file V2-predicate parameters***)
A call to a Voyager 2 boolean function. This constraint returns the result of the function. It provides an easy way to add any new constraint.
☞ See Section A.17.

A.2. Constraints on collections

ALL_COLL
Used for a search, this constraint finds all collections. It should not be used for a valida-

tion.
☞ No parameters.

ET_per_COLL (*min max*)
The number of entity types per collection must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

DYN_PROP_of_COLL (*dynamic-property parameters*)
Check some properties of the dynamic properties.
☞ See Section A.16.

SELECTED_COLL
Search for all selected objects. This constraint should not be used for validation.
☞ No parameters.

MARKED_COLL
Search for all marked objects. This constraint should not be used for validation.
☞ No parameters.

V2_CONSTRAINT_on_COLL (*V2-file V2-predicate parameters*)
A call to a Voyager 2 boolean function. This constraint returns the result of the function. It provides an easy way to add any new constraint.
☞ See Section A.17.

A.3. Constraints on entity types

ALL_ET
Used for a search, this constraint finds all entity types. It should not be used for a validation.
☞ No parameters.

ATT_per_ET (*min max*)
The number of attributes per entity type must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

ATT_LENGTH_per_ET (*min max*)
The sum of the size of all the attributes of an entity type must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

ROLE_per_ET (*min max*)
The number of roles an entity type can play must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

ONE_ROLE_per_ET (*min max*)
Entity types play between *min* and *max* roles with maximum cardinality = 1.
☞ *min* and *max* are integer constants or **N**.

N_ROLE_per_ET (*min max*)
Entity types play between *min* and *max* roles with maximum cardinality > 1.
☞ *min* and *max* are integer constants or **N**.

MAND_ROLE_per_ET (*min max*)
The number of mandatory roles played by entity types must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

OPT_ROLE_per_ET (*min max*)
The number of optional roles played by entity types must be at least *min* and at most

max.
☞ *min* and *max* are integer constants or **N**.

GROUP_per_ET (***min max***)
The number of groups per entity type must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

ID_per_ET (***min max***)
The number of identifiers per entity type must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

PID_per_ET (***min max***)
The number of primary identifiers per entity type must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

ALL_ATT_in_ID_ET (***yn***)
If parameter is **yes**, all the identifiers of an entity type contain all attributes (possibly with or without some roles) or the entity type has no explicit identifier. If parameter is **no**, an entity type must have at least one identifier which does not contain all the attributes of the entity type.
☞ *yn* is either **yes** or **no**.

ALL_ATT_ID_per_ET (***min max***)
The number of primary identifiers made of attributes only must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

HYBRID_ID_per_ET (***min max***)
The number of hybrid identifiers (made of attributes, roles or other groups) must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

KEY_ID_per_ET (***min max***)
The number of identifiers that are access keys must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

ID_NOT_KEY_per_ET (***min max***)
The number of identifiers that are not access keys must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

KEY_ALL_ATT_ID_per_ET (***min max***)
The number of identifiers made of attributes only which are access keys must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

EMBEDDED_ID_per_ET (***min max***)
The number of overlapping identifiers must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

ID_DIFF_in_ET (***type***)
All the identifiers of an entity type are different. Similarity criteria are function of the specified *type*: **components** indicates that all the elements of both identifiers are the same, possibly in a different order, **components_and_order** forces the components in both identifiers to be in the same order for the identifiers to be identical. For instance, let an entity type have two identifiers, one made of attributes A and B, the other made of attributes B and A. They will be said to be identical when *type* is *components* and different in the other case.
☞ *type* is either **components** or **components_and_order**.

KEY_per_ET (*min max*)
The number of access key groups per entity type must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

ALL_ATT_KEY_per_ET (*min max*)
The number of access keys made of attributes only must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

HYBRID_KEY_per_ET (*min max*)
The number of hybrid access keys must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

ID_KEY_per_ET (*min max*)
The number of access keys that are identifiers too must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

KEY_PREFIX_in_ET (*type*)
An access key is a prefix of another identifier or access key. *type* specifies whether the order of the attributes must be the same in the access key and in the prefix or not. This constraint is particularly well suited for using the assistant for search. To use it in order to validate a schema, it should be used with a negation (not KEY_PREFIX_in_ET).
☞ *type* is either **same_order** or **any_order**.

REF_per_ET (*min max*)
The number of reference groups in an entity type must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

REF_in_ET (*type*)
Referential constraints reference groups of type *type*.
☞ *type* is either **pid** to find ET with primary identifiers or **sid** to find ET with secondary identifiers.

COEXIST_per_ET (*min max*)
The number of coexistence groups per entity type must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

EXCLUSIVE_per_ET (*min max*)
The number of exclusive groups per entity type must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

ATLEASTONE_per_ET (*min max*)
The number of at-least-one groups per entity type must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

INCLUDE_per_ET (*min max*)
The number of inclusion constraints per entity type must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

INVERSE_per_ET (*min max*)
The number of inverse constraints per entity type must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

GENERIC_per_ET (*min max*)
The number of generic constraints per entity type must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

PROCUNIT_per_ET (*min max*)
The number of processing units per entity type must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

COLL_per_ET (*min max*)
The number of collections an entity type belongs to must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

DYN_PROP_of_ET (*dynamic-property parameters*)
Check some properties of the dynamic properties.
☞ See Section A.16.

SELECTED_ET
Search for all selected objects. This constraint should not be used for validation.
☞ No parameters.

MARKED_ET
Search for all marked objects. This constraint should not be used for validation.
☞ No parameters.

V2_CONSTRAINT_on_ET (*V2-file V2-predicate parameters*)
A call to a Voyager 2 boolean function. This constraint returns the result of the function. It provides an easy way to add any new constraint.
☞ See Section A.17.

A.4. Constraints on is-a relations

ALL_ISA
Used for a search, this constraint finds all is-a relations. It should not be used for a validation.
☞ No parameters.

SUB_TYPES_per_ISA (*min max*)
An entity type can not have less than *min* sub-types or more than *max* sub-types.
☞ *min* and *max* are integer constants or **N**.

SUPER_TYPES_per_ISA (*min max*)
An entity type can not have less than *min* super-types or more than *max* super-types.
☞ *min* and *max* are integer constants or **N**.

TOTAL_in_ISA (*yn*)
Is-a relations have (yes) or do not have (no) the *total* attribute.
☞ *yn* is either **yes** or **no**.

DISJOINT_in_ISA (*yn*)
Is-a relations have (yes) or do not have (no) the *disjoint* attribute.
☞ *yn* is either **yes** or **no**.

DYN_PROP_of_ISA (*dynamic-property parameters*)
Check some properties of the dynamic properties.
☞ See Section A.16.

SELECTED_ISA
Search for all selected objects. This constraint should not be used for validation.
☞ No parameters.

MARKED_ISA
Search for all marked objects. This constraint should not be used for validation.

☞ No parameters.

V2_CONSTRAINT_on_ISA (*V2-file V2-predicate parameters*)
A call to a Voyager 2 boolean function. This constraint returns the result of the function. It provides an easy way to add any new constraint.
☞ See Section A.17.

A.5. Constraints on rel-types

ALL_RT
Used for a search, this constraint finds all rel-types. It should not be used for a validation.
☞ No parameters.

ATT_per_RT (*min max*)
The number of attributes per rel-type must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

ATT_LENGTH_per_RT (*min max*)
The sum of the size of all the attributes of a rel-type must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

ROLE_per_RT (*min max*)
The number of roles played in a rel-type must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

ONE_ROLE_per_RT (*min max*)
Rel-types have between *min* and *max* roles with maximum cardinality = 1.
☞ *min* and *max* are integer constants or **N**.

N_ROLE_per_RT (*min max*)
Rel-types have between *min* and *max* roles with maximum cardinality > 1.
☞ *min* and *max* are integer constants or **N**.

MAND_ROLE_per_RT (*min max*)
The number of mandatory roles in a rel-types must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

RECURSIVITY_in_RT (*min max*)
The number of times an entity type plays a role in a rel-type should be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

GROUP_per_RT (*min max*)
The number of groups per rel-type must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

ID_per_RT (*min max*)
The number of identifiers per rel-type must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

PID_per_RT (*min max*)
The number of primary identifiers per rel-type must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

ALL_ATT_ID_per_RT (*min max*)
The number of identifiers made of attributes only must be at least *min* and at most *max*.

☞ *min* and *max* are integer constants or **N**.

HYBRID_ID_per_RT (***min max***)
The number of hybrid identifiers (made of attributes, roles or other groups) must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

EMBEDDED_ID_per_RT (***min max***)
The number of overlapping identifiers must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

ID_DIFF_in_RT (***type***)
All the identifiers of a rel-type are different. Similarity criteria are function of the specified *type*: **components** indicates that all the elements of both identifiers are the same, possibly in a different order, **components_and_order** forces the components in both identifiers to be in the same order for the identifiers to be identical. For instance, let an entity type have two identifiers, one made of attributes A and B, the other made of attributes B and A. They will be said to be identical when *type* is **components** and different in the other case.
☞ *type* is either **components** or **components_and_order**.

KEY_per_RT (***min max***)
The number of access keys per rel-type must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

COEXIST_per_RT (***min max***)
The number of coexistence groups per rel-type must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

EXCLUSIVE_per_RT (***min max***)
The number of exclusive groups per rel-type must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

ATLEASTONE_per_RT (***min max***)
The number of at-least-one groups per rel-type must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

INCLUDE_per_RT (***min max***)
The number of inclusion constraints per rel-type must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

GENERIC_per_RT (***min max***)
The number of generic constraints per rel-type must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

PROCUNIT_per_RT (***min max***)
The number of processing units per rel-type must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

DYN_PROP_of_RT (***dynamic-property parameters***)
Check some properties of the dynamic properties.
☞ See Section A.16.

SELECTED_RT
Search for all selected objects. This constraint should not be used for validation.
☞ No parameters.

MARKED_RT
Search for all marked objects. This constraint should not be used for validation.

☞ No parameters.

V2_CONSTRAINT_on_RT (*V2-file V2-predicate parameters*)
A call to a Voyager 2 boolean function. This constraint returns the result of the function. It provides an easy way to add any new constraint.
☞ See Section A.17.

A.6. Constraints on roles

ALL_ROLE
Used for a search, this constraint finds all roles. It should not be used for a validation.
☞ No parameters.

MIN_CARD_of_ROLE (*min max*)
The minimum cardinality of role must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

MAX_CARD_of_ROLE (*min max*)
The minimum cardinality of role must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

ET_per_ROLE (*min max*)
The number of entity types per role must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

DYN_PROP_of_ROLE (*dynamic-property parameters*)
Check some properties of the dynamic properties.
☞ See Section A.16.

SELECTED_ROLE
Search for all selected objects. This constraint should not be used for validation.
☞ No parameters.

MARKED_ROLE
Search for all marked objects. This constraint should not be used for validation.
☞ No parameters.

V2_CONSTRAINT_on_ROLE (*V2-file V2-predicate parameters*)
A call to a Voyager 2 boolean function. This constraint returns the result of the function. It provides an easy way to add any new constraint.
☞ See Section A.17.

A.7. Constraints on attributes

ALL_ATT
Used for a search, this constraint finds all attributes. It should not be used for a validation.
☞ No parameters.

MIN_CARD_of_ATT (*min max*)
The minimum cardinality of an attribute must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

MAX_CARD_of_ATT (*min max*)
The maximum cardinality of an attribute must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

DEPTH_of_ATT (***min max***)

The depth of a compound attribute, that is the amount of encompassing compound attributes plus one, must be at least <min> and at most <max>. For instance, in order to select all sub-attributes, use this constraint with <min>=2 and <max>=N.

☞ *min* and *max* are integer constants or **N**.

SUB_ATT_per_ATT (***min max***)

The number of sub-attributes of a compound attribute is at least *min* and at most *max*.

☞ *min* and *max* are integer constants or **N**.

TYPES_ALLOWED_for_ATT (***list***)

List of allowed types of attribute.

☞ *list* is the list of all allowed types (**BOOLEAN**, **CHAR**, **DATE**, **FLOAT**, **NUMERIC**, **VARCHAR**), separated with a space.

TYPES_NOTALLOWED_for_ATT (***list***)

List of all forbidden types of attribute.

☞ *list* is the list of all forbidden types, separated with a space: **BOOLEAN CHAR DATE FLOAT NUMERIC VARCHAR**.

TYPE_DEF_for_ATT (***type parameters***)

Specification of the parameters for a type of attributes. For instance, to specify that all numbers should be coded with 1 to 5 digits and 0 to 2 decimals:

TYPE_DEF_for_ATT NUMERIC 1 5 0 2

☞ *type* is the type of attribute for which the parameters must be specified, *parameters* is the list of parameters for the type; the content of that list depends on the type:

CHAR *min-length max-length*
FLOAT *min-size max-size*
NUMERIC *min-length max-length min-decimals max-decimals*
VARCHAR *min-length max-length*
BOOLEAN *min-size max-size*
DATE *min-size max-size*

min-... and *max-...* are integer constants or **N**.

PART_of_GROUP_ATT (***min max***)

The number of groups the attribute is a component of is at least *min* and at most *max*.

☞ *min* and *max* are integer constants or **N**.

ID_per_ATT (***min max***)

The number of identifiers per attribute is at least *min* and at most *max*.

☞ *min* and *max* are integer constants or **N**.

PID_per_ATT (***min max***)

The number of primary identifiers per attribute is at least *min* and at most *max*.

☞ *min* and *max* are integer constants or **N**.

PART_of_ID_ATT (***min max***)

The number of foreign keys the attribute is a component of is at least *min* and at most *max*.

☞ *min* and *max* are integer constants or **N**.

KEY_per_ATT (***min max***)

The number of access keys per attribute is at least *min* and at most *max*.

☞ *min* and *max* are integer constants or **N**.

REF_per_ATT (***min max***)

The number of referential group per attribute is at least *min* and at most *max*.

☞ *min* and *max* are integer constants or **N**.

PART_of_REF_ATT (***min max***)
The number of referential groups the attribute is a component of is at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

DYN_PROP_of_ATT (***dynamic-property parameters***)
Check some properties of the dynamic properties.
☞ See Section A.16.

SELECTED_ATT
Search for all selected objects. This constraint should not be used for validation.
☞ No parameters.

MARKED_ATT
Search for all marked objects. This constraint should not be used for validation.
☞ No parameters.

V2_CONSTRAINT_on_ATT (***V2-file V2-predicate parameters***)
A call to a Voyager 2 boolean function. This constraint returns the result of the function. It provides an easy way to add any new constraint.
☞ See Section A.17.

A.8. Constraints on groups

ALL_GROUP
Used for a search, this constraint finds all groups. It should not be used for a validation.
☞ No parameters.

COMP_per_GROUP (***min max***)
The number of terminal components in a group must be at least *min* and at most *max*. A component is terminal if it is not a group. For instance, let *A* be a group made of an attribute *a* and another group *B*. *B* is made of two attributes *b1* and *b2*. Then *A* has got three terminal components: *a*, *b* and *c*.
☞ *min* and *max* are integer constants or **N**.

ATT_per_GROUP (***min max***)
The number of attributes per group must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

ROLE_per_GROUP (***min max***)
The number of roles per group must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

GROUP_per_GROUP (***min max***)
The number of groups per group must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

ID_in_GROUP (***yn***)
Identifiers are (yes), are not (no) allowed.
☞ *yn* is either **yes** or **no**.

PID_in_GROUP (***yn***)
Primary identifiers are (yes), are not (no) allowed.
☞ *yn* is either **yes** or **no**.

KEY_in_GROUP (***yn***)
Access keys are (yes), are not (no) allowed.

☞ *yn* is either **yes** or **no**.

REF_in_GROUP (*yn*)
Reference groups are (yes), are not (no) allowed.
☞ *yn* is either **yes** or **no**.

COEXIST_in_GROUP (*yn*)
Coexistence groups are (yes), are not (no) allowed.
☞ *yn* is either **yes** or **no**.

EXCLUSIVE_in_GROUP (*yn*)
Exclusive groups are (yes), are not (no) allowed.
☞ *yn* is either **yes** or **no**.

ATLEASTONE_in_GROUP (*yn*)
At_least_one groups are (yes), are not (no) allowed.
☞ *yn* is either **yes** or **no**.

INCLUDE_in_GROUP (*yn*)
Include constraints are (yes), are not (no) allowed.
☞ *yn* is either **yes** or **no**.

INVERSE_in_GROUP (*yn*)
Inverse constraints are (yes), are not (no) allowed.
☞ *yn* is either **yes** or **no**.

GENERIC_in_GROUP (*yn*)
Generic constraints are (yes), are not (no) allowed.
☞ *yn* is either **yes** or **no**.

LENGTH_of_ATT_GROUP (*min max*)
The sum of the length of all components of a group must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

DYN_PROP_of_GROUP (*dynamic-property parameters*)
Check some properties of the dynamic properties.
☞ See Section A.16.

SELECTED_GROUP
Search for all selected objects. This constraint should not be used for validation.
☞ No parameters.

MARKED_GROUP
Search for all marked objects. This constraint should not be used for validation.
☞ No parameters.

V2_CONSTRAINT_on_GROUP (*V2-file V2-predicate parameters*)
A call to a Voyager 2 boolean function. This constraint returns the result of the function. It provides an easy way to add any new constraint.
☞ See Section A.17.

A.9. Constraints on entity type identifiers

ALL_EID
Used for a search, this constraint finds all entity type identifiers. It should not be used for a validation.
☞ No parameters.

COMP_per_EID (***min max***)
The number of components of an entity type identifier must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or N.

ATT_per_EID (***min max***)
The number of attributes per entity type identifier must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or N.

OPT_ATT_per_EID (***min max***)
An entity type identifier must have between *min* and *max* optional attributes.
☞ *min* and *max* are integer constants or N.

MAND_ATT_per_EID (***min max***)
An entity type identifier must have between *min* and *max* mandatory attributes.
☞ *min* and *max* are integer constants or N.

SINGLE_ATT_per_EID (***min max***)
An entity type identifier must have between *min* and *max* single-valued attributes.
☞ *min* and *max* are integer constants or N.

MULT_ATT_per_EID (***min max***)
An entity type identifier must have between *min* and *max* multi-valued attributes.
☞ *min* and *max* are integer constants or N.

MULT_ATT_per_MULT_COMP_EID (***min max***)
An entity type identifier made of several components must have between *min* and *max* multi-valued attributes.
☞ *min* and *max* are integer constants or N.

SUB_ATT_per_EID (***min max***)
An entity type identifier must have between *min* and *max* sub-attributes.
☞ *min* and *max* are integer constants or N.

COMP_ATT_per_EID (***min max***)
An entity type identifier must have between *min* and *max* compound attributes.
☞ *min* and *max* are integer constants or N.

ROLE_per_EID (***min max***)
The number of roles per entity type identifier must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or N.

OPT_ROLE_per_EID (***min max***)
An entity type identifier must have between *min* and *max* optional roles.
☞ *min* and *max* are integer constants or N.

MAND_ROLE_per_EID (***min max***)
An entity type identifier must have between *min* and *max* mandatory roles.
☞ *min* and *max* are integer constants or N.

ONE_ROLE_per_EID (***min max***)
An entity type identifier must have between *min* and *max* single-valued roles.
☞ *min* and *max* are integer constants or N.

N_ROLE_per_EID (***min max***)
An entity type identifier must have between *min* and *max* multi-valued roles.
☞ *min* and *max* are integer constants or N.

GROUP_per_EID (***min max***)
The number of groups per entity type identifier must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

ALL_EPID
Used for a search, this constraint finds all entity type primary identifiers. It should not be used for a validation.
☞ No parameters.

COMP_per_EPID (***min max***)
The number of components of a entity type primary identifier must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

ATT_per_EPID (***min max***)
The number of attributes per entity type primary identifier must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

OPT_ATT_per_EPID (***min max***)
An entity type primary identifier must have between *min* and *max* optional attributes.
☞ *min* and *max* are integer constants or **N**.

MAND_ATT_per_EPID (***min max***)
An entity type primary identifier must have between *min* and *max* mandatory attributes.
☞ *min* and *max* are integer constants or **N**.

SINGLE_ATT_per_EPID (***min max***)
An entity type primary identifier must have between *min* and *max* single-valued attributes.
☞ *min* and *max* are integer constants or **N**.

MULT_ATT_per_EPID (***min max***)
An entity type primary identifier must have between *min* and *max* multi-valued attributes.
☞ *min* and *max* are integer constants or **N**.

MULT_ATT_per_MULT_COMP_EPID (***min max***)
An entity type primary identifier made of several components must have between *min* and *max* multi-valued attributes.
☞ *min* and *max* are integer constants or **N**.

SUB_ATT_per_EPID (***min max***)
An entity type primary identifier must have between *min* and *max* sub-attributes.
☞ *min* and *max* are integer constants or **N**.

COMP_ATT_per_EPID (***min max***)
An entity type primary identifier must have between *min* and *max* compound attributes.
☞ *min* and *max* are integer constants or **N**.

ROLE_per_EPID (***min max***)
The number of roles per entity type primary identifier must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

OPT_ROLE_per_EPID (***min max***)
An entity type primary identifier must have between *min* and *max* optional roles.
☞ *min* and *max* are integer constants or **N**.

MAND_ROLE_per_EPID (***min max***)
An entity type primary identifier must have between *min* and *max* mandatory roles.
☞ *min* and *max* are integer constants or **N**.

ONE_ROLE_per_EPID (***min max***)
An entity type primary identifier must have between *min* and *max* single-valued roles.
☞ *min* and *max* are integer constants or **N**.

N_ROLE_per_EPID (***min max***)
An entity type primary identifier must have between *min* and *max* multi-valued roles.
☞ *min* and *max* are integer constants or **N**.

GROUP_per_EPID (***min max***)
The number of groups per entity type primary identifier must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

DYN_PROP_of_EID (***dynamic-property parameters***)
Check some properties of the dynamic properties.
☞ See Section A.16.

SELECTED_EID
Search for all selected objects. This constraint should not be used for validation.
☞ No parameters.

MARKED_EID
Search for all marked objects. This constraint should not be used for validation.
☞ No parameters.

V2_CONSTRAINT_on_EID (***V2-file V2-predicate parameters***)
A call to a Voyager 2 boolean function. This constraint returns the result of the function. It provides an easy way to add any new constraint.
☞ See Section A.17.

A.10. Constraints on rel-type identifiers

ALL_RID
Used for a search, this constraint finds all rel-type identifiers. It should not be used for a validation.
☞ No parameters.

COMP_per_RID (***min max***)
The number of components of a rel-type identifier must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

ATT_per_RID (***min max***)
The number of attributes per rel-type identifier must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

OPT_ATT_per_RID (***min max***)
A rel-type identifier must have between *min* and *max* optional attributes.
☞ *min* and *max* are integer constants or **N**.

MAND_ATT_per_RID (***min max***)
A rel-type identifier must have between *min* and *max* mandatory attributes.
☞ *min* and *max* are integer constants or **N**.

SINGLE_ATT_per_RID (***min max***)
A rel-type identifier must have between *min* and *max* multi-valued attributes.
☞ *min* and *max* are integer constants or **N**.

MULT_ATT_per_RID (***min max***)
A rel-type identifier must have between *min* and *max* single-valued attributes.
☞ *min* and *max* are integer constants or **N**.

MULT_ATT_per_MULT_COMP_RID (***min max***)
A rel-type identifier made of several components must have between *min* and *max* multi-valued attributes.
☞ *min* and *max* are integer constants or **N**.

SUB_ATT_per_RID (***min max***)
A rel-type identifier must have between *min* and *max* sub-attributes.
☞ *min* and *max* are integer constants or **N**.

COMP_ATT_per_RID (***min max***)
A rel-type identifier must have between *min* and *max* compound attributes.
☞ *min* and *max* are integer constants or **N**.

ROLE_per_RID (***min max***)
The number of roles per rel-type identifier must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

OPT_ROLE_per_RID (***min max***)
A rel-type identifier must have between *min* and *max* optional roles.
☞ *min* and *max* are integer constants or **N**.

MAND_ROLE_per_RID (***min max***)
A rel-type identifier must have between *min* and *max* mandatory roles.
☞ *min* and *max* are integer constants or **N**.

ONE_ROLE_per_RID (***min max***)
A rel-type identifier must have between *min* and *max* single-valued roles.
☞ *min* and *max* are integer constants or **N**.

N_ROLE_per_RID (***min max***)
A rel-type identifier must have between *min* and *max* multi-valued roles.
☞ *min* and *max* are integer constants or **N**.

GROUP_per_RID (***min max***)
The number of groups per rel-type identifier must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

ALL_RPID
Used for a search, this constraint finds all rel-type primary identifiers. It should not be used for a validation.
☞ No parameters.

COMP_per_RPID (***min max***)
The number of components of a rel-type primary identifier must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

ATT_per_RPID (***min max***)
The number of attributes per rel-type primary identifier must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

OPT_ATT_per_RPID (*min max*)
A rel-type primary identifier must have between *min* and *max* optional attributes.
☞ *min* and *max* are integer constants or N.

MAND_ATT_per_RPID (*min max*)
A rel-type primary identifier must have between *min* and *max* mandatory attributes.
☞ *min* and *max* are integer constants or N.

SINGLE_ATT_per_RPID (*min max*)
A rel-type primary identifier must have between *min* and *max* single-valued attributes.
☞ *min* and *max* are integer constants or N.

MULT_ATT_per_RPID (*min max*)
A rel-type primary identifier must have between *min* and *max* multi-valued attributes.
☞ *min* and *max* are integer constants or N.

MULT_ATT_per_MULT_COMP_RPID (*min max*)
A rel-type primary identifier made of several components must have between *min* and *max* multi-valued attributes.
☞ *min* and *max* are integer constants or N.

SUB_ATT_per_RPID (*min max*)
A rel-type primary identifier must have between *min* and *max* sub-attributes.
☞ *min* and *max* are integer constants or N.

COMP_ATT_per_RPID (*min max*)
A rel-type primary identifier must have between *min* and *max* compound attributes.
☞ *min* and *max* are integer constants or N.

ROLE_per_RPID (*min max*)
The number of roles per rel-type primary identifier must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or N.

OPT_ROLE_per_RPID (*min max*)
A rel-type primary identifier must have between *min* and *max* optional roles.
☞ *min* and *max* are integer constants or N.

MAND_ROLE_per_RPID (*min max*)
A rel-type primary identifier must have between *min* and *max* mandatory roles.
☞ *min* and *max* are integer constants or N.

ONE_ROLE_per_RPID (*min max*)
A rel-type primary identifier must have between *min* and *max* single-valued roles.
☞ *min* and *max* are integer constants or N.

N_ROLE_per_RPID (*min max*)
A rel-type primary identifier must have between *min* and *max* multi-valued roles.
☞ *min* and *max* are integer constants or N.

GROUP_per_RPID (*min max*)
The number of groups per rel-type primary identifier must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or N.

DYN_PROP_of_RID (*dynamic-property parameters*)
Check some properties of the dynamic properties.
☞ See Section A.16.

SELECTED_RID
Search for all selected objects. This constraint should not be used for validation.
☞ No parameters.

MARKED_RID
Search for all marked objects. This constraint should not be used for validation.
☞ No parameters.

V2_CONSTRAINT_on_RID (*V2-file V2-predicate parameters*)
A call to a Voyager 2 boolean function. This constraint returns the result of the function. It provides an easy way to add any new constraint.
☞ See Section A.17.

A.11. Constraints on attribute identifiers

ALL_AID
Used for a search, this constraint finds all attribute identifiers. It should not be used for a validation.
☞ No parameters.

COMP_per_AID (*min max*)
The number of components of an attribute identifier must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

ATT_per_AID (*min max*)
The number of attributes per attribute identifier must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

OPT_ATT_per_AID (*min max*)
An attribute identifier must have between *min* and *max* optional attributes.
☞ *min* and *max* are integer constants or **N**.

MAND_ATT_per_AID (*min max*)
An attribute identifier must have between *min* and *max* mandatory attributes.
☞ *min* and *max* are integer constants or **N**.

SINGLE_ATT_per_AID (*min max*)
An attribute identifier must have between *min* and *max* single-valued attributes.
☞ *min* and *max* are integer constants or **N**.

MULT_ATT_per_AID (*min max*)
An attribute identifier must have between *min* and *max* multi-valued attributes.
☞ *min* and *max* are integer constants or **N**.

MULT_ATT_per_MULT_COMP_AID (*min max*)
An attribute identifier made of several components must have between *min* and *max* multi-valued attributes.
☞ *min* and *max* are integer constants or **N**.

SUB_ATT_per_AID (*min max*)
An attribute identifier must have between *min* and *max* sub-attributes.
☞ *min* and *max* are integer constants or **N**.

COMP_ATT_per_AID (*min max*)
An attribute identifier must have between *min* and *max* compound attributes.
☞ *min* and *max* are integer constants or **N**.

GROUP_per_AID (***min max***)
The number of groups per attribute identifier must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

ALL_APID
Used for a search, this constraint finds all attribute primary identifiers. It should not be used for a validation.
☞ No parameters.

COMP_per_APID (***min max***)
The number of components of an attribute primary identifier must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

ATT_per_APID (***min max***)
The number of attributes per attribute primary identifier must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

OPT_ATT_per_APID (***min max***)
An attribute primary identifier must have between *min* and *max* optional attributes.
☞ *min* and *max* are integer constants or **N**.

MAND_ATT_per_APID (***min max***)
An attribute primary identifier must have between *min* and *max* mandatory attributes.
☞ *min* and *max* are integer constants or **N**.

SINGLE_ATT_per_APID (***min max***)
An attribute primary identifier must have between *min* and *max* single-valued attributes.
☞ *min* and *max* are integer constants or **N**.

MULT_ATT_per_APID (***min max***)
An attribute primary identifier must have between *min* and *max* multi-valued attributes.
☞ *min* and *max* are integer constants or **N**.

MULT_ATT_per_MULT_COMP_APID (***min max***)
An attribute primary identifier made of several components must have between *min* and *max* multi-valued attributes.
☞ *min* and *max* are integer constants or **N**.

SUB_ATT_per_APID (***min max***)
An attribute primary identifier must have between *min* and *max* sub-attributes.
☞ *min* and *max* are integer constants or **N**.

COMP_ATT_per_APID (***min max***)
An attribute primary identifier must have between *min* and *max* compound attributes.
☞ *min* and *max* are integer constants or **N**.

GROUP_per_APID (***min max***)
The number of groups per attribute primary identifier must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

DYN_PROP_of_AID (***dynamic-property parameters***)
Check some properties of the dynamic properties.
☞ See Section A.16.

SELECTED_AID
Search for all selected objects. This constraint should not be used for validation.

☞ No parameters.

MARKED_AID
Search for all marked objects. This constraint should not be used for validation.
☞ No parameters.

V2_CONSTRAINT_on_AID (*V2-file V2-predicate parameters*)
A call to a Voyager 2 boolean function. This constraint returns the result of the function. It provides an easy way to add any new constraint.
☞ See Section A.17.

A.12. Constraints on access keys

ALL_KEY
Used for a search, this constraint finds all access keys. It should not be used for a validation.
☞ No parameters.

COMP_per_KEY (***min max***)
The number of components of an access key must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

ATT_per_KEY (***min max***)
The number of attributes per access key must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

OPT_ATT_per_KEY (***min max***)
An access key must have between *min* and *max* optional attributes.
☞ *min* and *max* are integer constants or **N**.

MAND_ATT_per_KEY (***min max***)
An access key must have between *min* and *max* mandatory attributes.
☞ *min* and *max* are integer constants or **N**.

SINGLE_ATT_per_KEY (***min max***)
An access key must have between *min* and *max* single-valued attributes.
☞ *min* and *max* are integer constants or **N**.

MULT_ATT_per_KEY (***min max***)
An access key must have between *min* and *max* multi-valued attributes.
☞ *min* and *max* are integer constants or **N**.

MULT_ATT_per_MULT_COMP_KEY (***min max***)
An access key made of several components must have between *min* and *max* multi-valued attribute.
☞ *min* and *max* are integer constants or **N**.

SUB_ATT_per_KEY (***min max***)
An access key must have between *min* and *max* sub-attributes.
☞ *min* and *max* are integer constants or **N**.

COMP_ATT_per_KEY (***min max***)
An access key must have between *min* and *max* compound attributes.
☞ *min* and *max* are integer constants or **N**.

ROLE_per_KEY (***min max***)
The number of roles per access key must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

OPT_ROLE_per_KEY (*min max*)
An access key must have between *min* and *max* optional roles.
☞ *min* and *max* are integer constants or N.

MAND_ROLE_per_KEY (*min max*)
An access key must have between *min* and *max* mandatory roles.
☞ *min* and *max* are integer constants or N.

ONE_ROLE_per_KEY (*min max*)
An access key must have between *min* and *max* single-valued roles.
☞ *min* and *max* are integer constants or N.

N_ROLE_per_KEY (*min max*)
An access key must have between *min* and *max* multi-valued roles.
☞ *min* and *max* are integer constants or N.

GROUP_per_KEY (*min max*)
The number of groups per access key must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or N.

DYN_PROP_of_KEY (*dynamic-property parameters*)
Check some properties of the dynamic properties.
☞ See Section A.16.

SELECTED_KEY
Search for all selected objects. This constraint should not be used for validation.
☞ No parameters.

MARKED_KEY
Search for all marked objects. This constraint should not be used for validation.
☞ No parameters.

V2_CONSTRAINT_on_KEY (*V2-file V2-predicate parameters*)
A call to a Voyager 2 boolean function. This constraint returns the result of the function. It provides an easy way to add any new constraint.
☞ See Section A.17.

A.13. Constraints on referential groups

ALL_REF
Used for a search, this constraint finds all referential constraints. It should not be used for a validation.
☞ No parameters.

COMP_per_REF (*min max*)
The number of components of a reference group must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or N.

ATT_per_REF (*min max*)
The number of attributes per reference group must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or N.

OPT_ATT_per_REF (*min max*)
A reference group must have between *min* and *max* optional attributes.
☞ *min* and *max* are integer constants or N.

MAND_ATT_per_REF (*min max*)
A reference group must have between *min* and *max* mandatory attributes.
☞ *min* and *max* are integer constants or N.

SINGLE_ATT_per_REF (***min max***)
A reference group must have between *min* and *max* single-valued attributes.
☞ *min* and *max* are integer constants or **N**.

MULT_ATT_per_REF (***min max***)
A reference group must have between *min* and *max* multi-valued attributes.
☞ *min* and *max* are integer constants or **N**.

MULT_ATT_per_MULT_COMP_REF (***min max***)
A reference group made of several components must have between *min* and *max* multi-valued attribute.
☞ *min* and *max* are integer constants or **N**.

SUB_ATT_per_REF (***min max***)
A reference group must have between *min* and *max* sub-attributes.
☞ *min* and *max* are integer constants or **N**.

COMP_ATT_per_REF (***min max***)
A reference group must have between *min* and *max* compound attributes.
☞ *min* and *max* are integer constants or **N**.

ROLE_per_REF (***min max***)
The number of roles per reference group must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

OPT_ROLE_per_REF (***min max***)
A reference group must have between *min* and *max* optional roles.
☞ *min* and *max* are integer constants or **N**.

MAND_ROLE_per_REF (***min max***)
A reference group must have between *min* and *max* mandatory roles.
☞ *min* and *max* are integer constants or **N**.

ONE_ROLE_per_REF (***min max***)
A reference group must have between *min* and *max* single-valued roles.
☞ *min* and *max* are integer constants or **N**.

N_ROLE_per_REF (***min max***)
A reference group must have between *min* and *max* multi-valued roles.
☞ *min* and *max* are integer constants or **N**.

GROUP_per_REF (***min max***)
The number of groups per reference group must be at least *min* and at most *max*.
☞ *min* and *max* are integer constants or **N**.

LENGTH_of_REF (***operator***)
The length of a reference group (the sum of the length of its components) must be equal, different, smaller than or greater than the length of the referenced group.
☞ *operator* is either **equal**, **different**, **smaller** or **greater**.

TRANSITIVE_REF (***yn***)
The group is a transitive referential constraints. For instance, A(a,b), B(<u>a,b</u>) and C(<u>b</u>) are 3 entity types. (A.a,A.b) is a reference attribute of (B.a,B.b), A.b is a reference attribute of C.b and B.b is a reference attribute of C.b. In that case, the referential constraint from A.b to C.b is redundant and should be suppressed.
☞ *yn* is either **yes** or **no**.

DYN_PROP_of_REF (***dynamic-property parameters***)
Check some properties of the dynamic properties.

☞ See Section A.16.

SELECTED_REF
Search for all selected objects. This constraint should not be used for validation.
☞ No parameters.

MARKED_REF
Search for all marked objects. This constraint should not be used for validation.
☞ No parameters.

V2_CONSTRAINT_on_REF (*V2-file V2-predicate parameters*)
A call to a Voyager 2 boolean function. This constraint returns the result of the function. It provides an easy way to add any new constraint.
☞ See Section A.17.

A.14. Constraints on processing units

ALL_PROCUNIT
Used for a search, this constraint finds all processing units. It should not be used for a validation.
☞ No parameters.

DYN_PROP_of_PROCUNIT (*dynamic-property parameters*)
Check some properties of the dynamic properties.
☞ See Section A.16.

SELECTED_PROCUNIT
Search for all selected processing units. This constraint should not be used for validation.
☞ No parameters.

MARKED_PROCUNIT
Search for all marked processing units. This constraint should not be used for validation.
☞ No parameters.

V2_CONSTRAINT_on_PROCUNIT (*V2-file V2-predicate parameters*)
A call to a Voyager 2 boolean function. This constraint returns the result of the function. It provides an easy way to add any new constraint.
☞ See Section A.17.

A.15. Constraints on names

CONCERNED_NAMES (*list*)
This predicate retains all the objects of specified types. This is a very special predicate in the sense that it does not really treats about object names, but that it should only be used in conjunction with other predicates on names. Indeed, it has no real sense by itself, but it allows other predicates to restrict their scope. For instance, to restrict entity type and rel-type names to 8 characters, the following validation rule can be used:

```
CONCERNED_NAMES ET RT
    and LENGTH_of_NAMES 1 8
  or not CONCERNED_NAMES ET RT
```

☞ *list* is a list of object types separated by spaces. The valid object type names are those used as the suffixes of all the prodecates: SCHEMA, COLL, ET, RT, ATT, ROLE, ATT, GROUP, EID, EPID, RID, RPID, AID, APID, KEY, REF, PROCUNIT.

NONE_in_LIST_NAMES (*list*)

The names of the schema, entity types, rel-types, attributes, roles, groups, processing units and collections are not in the list *list*.

☞ *list* is a list of words separated by a comma. All the characters between two commas belong to a word, spaces included. The syntax of the words is the same as for the name processor. Hence, it is possible to use the following special characters: ^ to represent the beginning of a line, $ to represent its end, ? to represent any single character and * to represent any suite of characters. For instance: ^_*, *_$. This list forbids any name that begins by _ or end by _.

NONE_in_LIST_CI_NAMES (*list*)

The names of the schema, entity types, rel-types, attributes, roles, groups, processing units and collections are not in the list *list*. The comparison between names and words in the list is case insensitive.

☞ *list* is a list of words separated by a comma. All the characters between two commas belong to a word, spaces included. The syntax is similar to the one described in the NONE_in_LIST_NAMES constraint.

ALL_in_LIST_NAMES (*list*)

The names of the schema, entity types, rel-types, attributes, roles, groups, processing units and collections are in the list *list*.

☞ *list* is a list of words separated by a comma. All the characters between two commas belong to a word, spaces included. The syntax is similar to the one described in the NONE_in_LIST_NAMES constraint.

ALL_in_LIST_CI_NAMES (*list*)

The names of the schema, entity types, rel-types, attributes, roles, groups, processing units and collections are in the list *list*. The comparison between names and words in the list is case insensitive.

☞ *list* is a list of words separated by a comma. All the characters between two commas belong to a word, spaces included. The syntax is similar to the one described in the NONE_in_LIST_NAMES constraint.

NONE_in_FILE_NAMES (*file_name*)

The names of the schema, entity types, rel-types, attributes, roles, groups, processing units and collections can not be in the file with the name *file_name*.

☞ *file_name* is the name of an ASCII file that contains a list of all the forbidden names. Each line of the file contains a name. All the characters of a line are part of the name, except the end of line characters. The syntax is similar to the one described in the NONE_in_LIST_NAMES constraint.

NONE_in_FILE_CI_NAMES (*file_name*)

The names of the schema, entity types, rel-types, attributes, roles, groups, processing units and collections can not be in the file with the name *file_name*. The comparison between names and words in the file is case insensitive.

☞ *file_name* is the name of an ASCII file that contains a list of all the forbidden names. Each line of the file contains a name. All the characters of a line are part of the name, except the end of line characters. The syntax is similar to the one described in the NONE_in_LIST_NAMES constraint.

ALL_in_FILE_NAMES (*file_name*)

The names of the schema, entity types, rel-types, attributes, roles, groups, processing units and collections are in the file with the name *file_name*.

☞ *file_name* is the name of an ASCII file that contains a list of all the forbidden names. Each line of the file contains a name. All the characters of a line are part of the name,

except the end of line characters. The syntax is similar to the one described in the NONE_in_LIST_NAMES constraint.

ALL_in_FILE_CI_NAMES (*file_name*)

The names of the schema, entity types, rel-types, attributes, roles, groups, processing units and collections are in the file with the name *file_name*. The comparison between names and words in the file is case insensitive.

☞ *file_name* is the name of an ASCII file that contains a list of all the forbidden names. Each line of the file contains a name. All the characters of a line are part of the name, except the end of line characters. The syntax is similar to the one described in the NONE_in_LIST_NAMES constraint.

NO_CHARS_in_LIST_NAMES (*list*)

The names of the schema, entity types, rel-types, attributes, roles, groups, processing units and collections can not contain any character of the list *list*.

☞ *list* is a list of characters with no separator. For example: &é"'()§è!çà{}@#[]

ALL_CHARS_in_LIST_NAMES (*list*)

The names of the schema, entity types, rel-types, attributes, roles, groups, processing units and collections must be made of the characters of the list *list* only.

☞ *list* is a list of characters with no separator.
For example: ABCDEFGHIJKLMNOPQRSTUVWXYZ

LENGTH_of_NAMES (*min max*)

The length of names of the schema, entity types, rel-types, attributes, roles, groups, processing units and collections must be at least *min* and at most *max*.

☞ *min* and *max* are integer constants or **N**.

DYN_PROP_of_NAMES (*dynamic-property parameters*)

Check some properties of the dynamic properties.

☞ See Section A.16.

SELECTED_NAMES

Search for all selected objects. This constraint should not be used for validation.

☞ No parameters.

MARKED_NAMES

Search for all marked objects. This constraint should not be used for validation.

☞ No parameters.

V2_CONSTRAINT_on_NAMES (*V2-file V2-predicate parameters*)

A call to a Voyager 2 boolean function. This constraint returns the result of the function. It provides an easy way to add any new constraint.

☞ See Section A.17.

A.16. Using DYN_PROP_OF_... constraints

All dynamic property constraints are of the form:

DYN_PROP_of_XXX (*dynamic-property parameters*)

where:

- XXX is either SCHEMA, COLL, ET, ISA, RT, ROLE, ATT, GROUP, EID, RID, AID, KEY, REF, PROCUNIT or NAMES.
- *dynamic-property* is the name of a dynamic property defined on constructs of type XXX. If the name contains a space character, it must be surrounded by double quotes. The

name cannot itself contain double quotes. E.g.: owner, "account number" are valid names.

- *parameters* is a series of parameters, the number and the type of which depend on the dynamic-property, as shown hereafter.

The dynamic property can be declared either mono-valued or multi-valued.

1. If the dynamic property is mono-valued, the parameters string format depends on the type of the dynamic property:

- If the dynamic property is of type Integer, parameters are: *min max*
  The dynamic property value must be comprised between *min* and *max*, integer constants or **N**.
- If the dynamic property is of type Char, parameters are: *min max*
  The dynamic property value must be comprised, in the ASCII order, between *min* and *max*, two character constants.
- If the dynamic property is of type Real, parameters are: *min max*
  The dynamic property value must be comprised between *min* and *max*, two real constants.
- If the dynamic property is Boolean, the single parameter is either **true** or **false**.
  The dynamic property value must be either true or false.
- If the dynamic property is of type String, parameters are comparison_operator string
  The comparison operator must be one of: **=**, **<**, **>**, **=ci**, **<ci**, **>ci**, and **contains**. **=** is the strict equality of both the string value and the dynamic property value, **<** means string comes before the dynamic property value in alphabetical order, and **>** is the inverse; **=ci**, **<ci** and **>ci** are the case insensitive equivalents of **=**, **<**, **>**; **contains** is the sub-string operator that checks whether string is a sub-string of the dynamic property value.

2. If the dynamic property is multi-valued, the parameters string is one of the following:

- **count** *min max*
  The number of values (whatever they are) is comprised between *min*, an integer number, and *max*, an integer number or **N**.
- **one** *mono-valued-dynamic-property-parameters*
  Exactly one of the values must satisfy the *mono-valued-dynamic-property-parameters*. In fact, each values treated as if the dynamic property was mono-valued; all the values that satisfy the property are counted and the multi-valued property is said to be satisfied if the count equals one.
- **some** *mono-valued-dynamic-property-parameters*
  At least one of the values must satisfy the *mono-valued-dynamic-property-parameters*. In fact, each value is treated as if the dynamic property was mono-valued; all the values that satisfy the property are counted and the multi-valued property is said to be satisfied if the count is greater or equal to one.
- **each** *mono-valued-dynamic-property-parameters*
  Every value must satisfy the *mono-valued-dynamic-property-parameters*. In fact, each value is treated as if the dynamic property was mono-valued and the multi-valued property is said to be satisfied if every value satisfy the “mono-valued property”.

Examples:

- DYN_PROP_of_ATT (view count 2 N)

Searches for all attributes used in at least two views (view is the DB-MAIN built-in dynamic property for the definition of views)

- DYN_PROP_of_ET(owner = "T. Smith")
  Assuming owner is a mono-valued string dynamic property defined on entity types, this constraints looks for all entity types owned by T. Smith.
- DYN_PROP_of_ET("modified by" some contains Smith)
  Assuming modified by is a multi-valued string dynamic property defined on entity types which contains the list of all the persons who modified the entity type, this constraint looks for all entity types modified by Smith.
- DYN_PROP_of_ATT(line 50 60)
  line is a mono-valued integer dynamic property defined on all constructs generated by the COBOL extractor. This constraint looks for all constructs obtained from the extraction of a specific part (lines 50-60) of the COBOL source file.

A.17. Using Voyager 2 constraints

Voyager 2 can be used to implement user-defined constraints with all object types. They are used with the V2_CONSTRAINT_on_XXX constraints, where XXX stands for SCHEMA, ET, RT, ROLE, ATT, EID, EPID, AID, EPID, RID, RPID, REF, KEY, ISA, or NAMES. This may be very useful to look for complex patterns that cannot be expressed with the predefined constraints.

All the V2-CONSTRAINT_on_... are used the same way, they all need three parameters:

V2_CONSTRAINT_on_... (*V2-file V2-function parameters*)

where

- *V2-file* is the name of the Voyager 2 program that contains the function to execute
- *V2-function* is the name of the Voyager 2 function
- *parameters* all the parameters to pass to the function. The number of such parameters may vary according to the function definition.

The Voyager 2 function must be declared as an integer function with two parameters: the object of the repository that must be analysed (an entity type for instance) and a string containing all the parameters. The value returned by this function must be 0 if the constraint is not satisfied and any value different of 0 otherwise. The function must be declared as exportable.

Example:

Let Num_tech_id_per_et be the name of a Voyager 2 function that verifies if an entity type has a valid number of technical identifiers. It is placed in the program ANALYSE.V2, compiled as ANALYSE.OXO in directory C:\PROJECT. This function needs two parameters, one that is a minimum valid number of technical identifiers and the other that is a maximum valid number. The declaration of the Voyager 2 function in the file ANALYSE.V2 should look like:

export function integer Num_tech_id_per_et(entity_type: ent, string: arguments)

and the constraint in the analyser script should look like:

V2_CONSTRAINT_on_ET (ANALYSE.OXO Num_tech_id_per_et 0 1)

Appendix B

The PDL syntax

This appendix lists the abstract syntax of the Pattern Definition Language (PDL) used in the DB-MAIN CASE environment by the reverse engineering assistant and in the MDL language to define the grammar of text models.

B.1. BNF notation

::= is the definition symbol. The left member is the defined term, the right member its definition. For instance,
<a> ::= t means that *<a>* is defined as *t*.

<...> angle brackets encompass the terms that have a definition. When placed at the left side of ::=, it shows the term that is defined. At the right side of that symbol, it must be replaced by its definition. For instance, *<b> ::= t*, defines *<b>* as *t*, and in *<a> ::= r<b>s*, *<b>* is replaced by its definition and thus *<a>* is defined as *rts*.

| stands for an alternative. Either the left member or the right member may be used. They are two possible definitions. For instance, *<a> ::= <b>|<c>* means that *<a>* may be defined either as *<b>* or *<c>*.

[...] encompasses a facultative part of a definition. For instance, *<a> ::= <b>[<c>]* means that *<a>* may be defined either as *<b>* or as *<b><c>*

{...} encompasses a repeatable part of a definition. That part may be used zero, one or many times. For instance, *<a> ::= <b>{<c>}* means that *<a>* may be defined either as *<b>*, *<b><c>*, *<b><c><c>*,...

{...}$_{m-n}$ encompasses a repeatable part of a definition with a limit on the number of repetitions. That part may be used at least m times and at most n times. For instance, *<a> ::= <b>{<c>}*$_{0-3}$ means that *<a>* may be defined either as *<b>*, *<b><c>*, *<b><c><c>* or *<b><c><c><c>*.

B.2. The PDL language

<pattern> ::= <pattern_name> **::=** {<segment>};
<pattern_name> ::= <letter>{<valid-character>}$_{0-29}$
This is the name of the pattern
<valid-character> ::= <letter>|<figure>
<letter> ::= **a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z|**
A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
<figure> ::= **1|2|3|4|5|6|7|8|9|0**

<segment>	::= <terminal_seg>**|**<pattern_name>**|**<variable>**|**<range>**|**<optional_seg>**|** <repeat_seg>**|**<group_seg>**|**<choice_seg>**|**<regular_expr>
<terminal_seg>	::= **"**string**"** *Match the string, case sensitive, /t = tabulation; /n = new line*
<variable>	::= **@**<pattern_name> *The "@" symbol indicates that the segment is a variable. If a variable appears two times in a pattern, then both occurences have the same value. When a pattern is found, the value of the variables can be known. A variable can not appear into a repetitive structure.*
<range>	::= **range(**<any-character>**-**<any-character>**)** *Is any character between the two specified*
<any-character>	::= *any ASCII character*
<optional_seg>	::= **[**<segment>**]** *Optional segment*
<repeat_seg>	::= <segment>***** *Repetitive segment: match one or more time <segment>*
<group_seg>	::= **(**{<segment>}**)**
<choice_seg>	::= <segment>{**|**<segment>} *Match any of the <segment>.*
<regular_exp>	::= **/g"**<regular expression>**"** *<regular expression> is a regular expression à la grep*

Appendix C

Global transformations

This appendix lists all the global transformations of the DB-MAIN 6.* CASE environment. These transformations are used by the **glbtrsf** instruction of the MDL language and by the "advanced global transformation assistant" of the CASE environment.

C.1. Transformations

A transformation is designed to perform a given action on a set of objects. A default set is defined for each transformation. This set may be refined to a subset defined by a predicative rule (see Appendix A).

Here follows a list of all transformations with their default scope:

ET_into_RT, default scope: all entity types.

Transform all entity types satisfying the preconditions of the elementary transformation into rel-types.

ET_into_ATT, default scope: all entity types.

Transform all entity types satisfying the preconditions of the elementary transformation into attributes.

ADD_TECH_ID, default scope: all entity types.

Add a technical identifier to all entity types. This transformation should never be used without refinement of the scope.

SMART_ADD_TECH_ID, default scope: all entity types.

Add a technical identifier to all entity types that do not have one but should for all rel-types to be transformable into referential constraints.

ISA_into_RT, default scope: all is-a relations.

Transform all is-a relations into binary one-to-one rel-types.

RT_into_ET, default scope: all rel-types.

Transform all rel-types into entity types. This transformation should never be used without refinement of the scope.

RT_into_ISA, default scope: all rel-types.

Transform all binary one-to-one rel-types that satisfy the preconditions of the elementary transformation into is-a relations if it can be done without dilemma (the remaining

is-a relations can possibly be transformed with the elementary transformation).

RT_into_REF, default scope: all rel-types.

Transform all rel-types into referential attribute(s).

RT_into_OBJATT, default scope: all rel-types.

Transform all rel-types into object-attribute(s).

REF_into_RT, default scope: all referential attribute.

Transform all referential attributes into rel-types.

ATT_into_ET_VAL, default scope: all attributes.

Transform all attributes into entity types using the value representation of the attributes. This transformation should never be used without refinement of the scope.

ATT_into_ET_INST, default scope: all attributes.

Transform all attributes into entity types using the instance representation of the attributes. This transformation should never be used without refinement of the scope.

OBJATT_into_RT, default scope: all object attributes.

Transform all object attributes into a rel-type.

DISAGGREGATE, default scope: all attributes.

Disaggregate all compound attributes.

INSTANCIATE, default scope: all attributes.

Transforms all multivalued atomic attributes into a list of single-valued attributes.

MATERIALIZE, default scope: all attributes.

Materializes all user-defined attributes, replaces them with their definition.

SPLIT_MULTIET_ROLE, default scope: all roles.

Split all the rel-types that contain one or more multi-ET roles.

AGGREGATE, default scope: all groups

Aggregate all groups. This transformation should never be used without refinement of the scope.

GROUP_into_KEY, default scope: all groups

Add the access key property to all groups.

RENAME_GROUP, default scope: all groups

Give a new meaningful name to each group. This name is unique in the schema. Note that the old name is lost forever.

REMOVE_KEY, default scope: all access keys

Remove all access keys.

REMOVE_PREFIX_KEY, default scope: all access keys

Remove all access keys that are a prefix of another one.

REMOVE_TECH_DESC, default scope: all objects of the schema

Remove the technical description of all the objects of the schema.

REMOVE, default scope: NONE; scope definition is mandatory

Remove all the objects that are in the specified scope. The deleted objects are lost forever.
Note that this transformation is very special, it does not exactly conform to the definition of a transformation since there is no default scope.

NAME_PROCESSING, default scope: NONE; scope definition mandatory

Process the name and short name of the selected objects. The parameters (in the script) must be interpreted in two parts. The second one is the rule defining the set of objects to process. The first parameter is the patterns; it has the following syntax:
"L" stands for the conversion of uppercase letters to lowercase letters;
"U" stands for the conversion of lowercase letters to uppercase letters;
"C" stands for "capitalization";
"A" stands for accents removal;
"S" stands for shortening and is followed by the maximum size of new names;
"P" stands for patterns and is followed by the list of patterns with the following syntax (in the patterns, semi-colons and backslashes are prefixed by a backslash):

search_pattern_1;replace_pattern_1;...;search_pattern_n;replace_pattern_n;

MARK, default scope: NONE; scope definition mandatory

Mark all the objects that are in the specified scope.

Note that this transformation is very special, it does not exactly conform to the definition of a transformation since there is no default scope and no real transformation.

UNMARK, default scope: NONE; scope definition mandatory

Remove the mark of all the marked objects that are in the specified scope.

Note that this transformation is very special, it does not exactly conform to the definition of a transformation since there is no default scope and no real transformation.

EXTERN, default scope: NONE; scope definition mandatory

Call an external Voyager 2 function, that is a user defined function. This function may work on any type of objects.

C.2. Control structures

ON (<rule>)... **ENDON**

This structure allows us to reduce the scope of a set of transformations. The *rule* is evaluated and the set of objects it finds will be the scope of all the subsequent transformations until the ENDON keyword.

During execution, it is possible that a transformation destroys an object of the scope. In that case, this object is no more available for the following transformations. It is also possible that a transformation creates an object that validates the rule of the ON clause. In that case, this object will not be added to the scope. To address this question, the ON...ENDON structure can be inserted in a LOOP...ENDLOOP structure.

Note that ON...ENDON structure can not overlap, there can not be an ON...ENDON structure inside another ON...ENDON structure.

LOOP...ENDLOOP

This structure allows us to perform the same actions several times until a fix point is reached. The LOOP keyword is just a label: when encountered, it does nothing. All the

transformations that follow it are performed until the ENDLOOP keyword is reached. Then, if one or more transformations have effectively modified the schema, all these transformations are performed once more. This will continue until the schema has reached a fix point for these transformations, i.e. none of them modifies the schema.

Note that LOOP...ENDLOOP structures can be included one into another.

Appendix D

The MDL syntax

D.1. BNF notation

::= is the definition symbol. The left member is the defined term, the right member its definition. For instance,
<a> ::= t means that *<a>* is defined as *t*.

<...> angle brackets encompass the terms that have a definition. When placed at the left side of ::=, it shows the term that is defined. At the right side of that symbol, it must be replaced by its definition. For instance, *<b> ::= t*, defines *<b>* as *t*, and in *<a> ::= r<b>s*, *<b>* is replaced by its definition and thus *<a>* is defined as *rts*.

| stands for an alternative. Either the left member or the right member may be used. They are two possible definitions. For instance, *<a> ::= <b>|<c>* means that *<a>* may be defined either as *<b>* or *<c>*.

[...] encompasses a facultative part of a definition. For instance, *<a> ::= <b>[<c>]* means that *<a>* may be defined either as *<b>* or as *<b><c>*

{...} encompasses a repeatable part of a definition. That part may be used zero, one or many times. For instance, *<a> ::= <b>{<c>}* means that *<a>* may be defined either as *<b>*, *<b><c>*, *<b><c><c>*,...

{...}$_{m-n}$ encompasses a repeatable part of a definition with a limit on the number of repetitions. That part may be used at least m times and at most n times. For instance, *<a> ::= <b>{<c>}*$_{0-3}$ means that *<a>* may be defined either as *<b>*, *<b><c>*, *<b><c><c>* or *<b><c><c><c>*.

D.2. Miscellaneous rules

D.2.1. Spaces and comments

For the readability of the grammar, spaces between grammar elements are not specified. In fact, they should be appended "intelligently":

- no spaces between letters of a word or between figures forming a number
- mandatory spaces between separated words both made of letters and/or figures
- optional spaces between special symbols or words and symbols.

For example:

do normalise(SQL-schema)

Spaces are mandatory between do and normalise and optional everywhere else; the following is equivalent:

do   normalise   (SQL-schema   )

A space is any series of blank (ASCII code 32), tab (ASCII code 8) or new line (ASCII codes 13 and/or 10) characters.

Comments are also considered as spaces: they can be put anywhere a space is allowed. A comment begins with the **%** character and is terminated with the end of the line. For instance:

do normalise(SQL-schema) **%** this is a comment
do optimise(SQL-schema) **%** this is another comment

D.2.2. Forward references

Forward references are not allowed.

D.3. Multi-purpose definitions

These definitions make a useful set for the following. They include the definition of special characters such as an end-of-line, an end-of-file,... and the definition of special strings such as valid-names that will serve as identifiers, human readable texts, comments,...

The characters used are the following:

<EOL>	::= *End-Of-Line character*
<EOF>	::= *End-Of-File character*
<letter>	::= **a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z|** **A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z**
<figure>	::= **1|2|3|4|5|6|7|8|9|0**
<valid-character>	::= <letter>|<figure>|**-**|**_** *characters recognised by the language for identifiers*
<readable-character>	::= *any readable ASCII character but* <EOL> *and* <EOF>. *These characters are used for messages that appear on the screen* *A double quote must be doubled ("").*
<any-character>	::= *any character but* <EOL> *and* <EOF>
<really-any-character>	::= *any character but* <EOF>

Those characters can be combined. A valid-name is a string that is recognised by the language as an identifier. And some readable text is any text that will be displayed on screen such as messages, contextual names,... Strings are used for any suite of parameters of any type. Numbers are positive integers.

<free-text>	::= {<any-character>}
<totally-free-text>	::= {<really-any-character>}
<valid-name>	::= {<valid-character>}$_{1-100}$ *a name used for identifiers*
<readable-name>	::= **"**{<readable-character>}$_{0-100}$**"** *a human readable and meaningful name*
<string>	::= **"**{<readable-character>}$_{0-255}$**"** *a human readable and meaningful string of characters*
<textual-description>	::= **description** <totally-free-text> **end-description** *A description is a free text of any length, the "|" character may be used as the left margin indicator. Almost every block can have a description.*
<number>	::= <figure>{<figure>}

D.4. Expressions

Some expressions are required by several control structures in a strategy.

<expression> ::= <and-expression> [**or** <expression>]

<and-expression> ::= <not-expression> [**and** <and-expression>]

<not-expression> ::= [**not**] <weak-expression>

<weak-expression> ::= [**weak**] <elem-expression>

<elem-expression> ::= <exists-expr>|<model-expr>|<external-expr>|<ask>|<built-in-expr>| <parenth-expr>

<exists-expr> ::= **exists (** <product-name> **,** <sch-anal-expr> {**,** <sch-anal-expr>} **)**
the comma acts as a and*, all expressions must be* true *for the result to be* true

<model-expr> ::= **model (** <product-name> **,** <model-name> **)**

<external-expr> ::= **external** <external-fct-name> **(** [<ext-parameters>] **)**

<ext-parameters> ::= <ext-parameter> {**,** <ext-parameter>}

<ext-parameter> ::= [**content:**]<product-name>|<string>|<number>

<ask> ::= **ask** <string>

<built-in-expr> ::= <built-in-fct-name> <misc-parameters>

<built-in-fct-name> ::= <valid-name>

<misc-parameters> ::= **(** <parameter> {**,** <parameter>} **)**

`<parameter> ::= <product-name>|<string>|<number>`

<parenth-expr> ::= **(** <expression> **)**

<sch-anal-expr> ::= <and-sch-anal-expr> [**or** <sch-anal-expr>]

<and-sch-anal-expr> ::= <not-sch-anal-expr> [**and** <and-sch-anal-expr>]

<not-sch-anal-expr> ::= [**not**] <elem-sch-anal-expr>

<elem-sch-anal-expr> ::= <constraint-name> <cstr-param>
a first-order logic predicate

<constraint-name> ::= <valid-name>
the name of a validation function of the supporting CASE environment

<cstr-param> ::= **(**{<any-character>}$_{0\text{-}255}$**)**
strings for the parameters of predicates; their syntax may vary depending on the context in which they are used. Characters "(", ")" and "\" must be prefixed by "\"

D.5. Method description

A single special paragraph describes the method itself with a title, a version, an author, a date, possibly a description or a help file and the main process type.

<Method> ::= <block> {<block>} <method-description>

<block> ::= <extern-decl>|<schema-model>|<text-model>|<product-type>|<toolbox>| <task-model>

<method-description> ::= **method** <method-title> <method-version> [<textual-description>] <method-author> <method-date> [<method-help>] <method-perform> **end-method**

<method-title> ::= **title** <readable-name>

<method-version> ::= **version** <version-name>

<version-name> ::= **"**{<readable-character>}$_{16}$**"**

<method-author> ::= **author** <readable-name>

<method-date> ::= **date** <date>

<date> ::= **"**<day>**-**<month>**-**<year>**"**

<day> ::= {<figure>}$_{2\text{-}2}$

<month> ::= {<figure>}$_{2\text{-}2}$

<year> ::= {<figure>}$_{4\text{-}4}$

<method-help> ::= **help-file** <felp-file-name>

<help-file-name> ::= <string>

<method-perform> ::= **perform** <task-name>

D.6. External declaration

The language allows the methodological engine to use external functions, that is user-defined functions written in the Voyager 2 language. These functions must be declared before they can be used.

<extern-decl> ::= **extern** <external-fct-name> <real-ext-fct-name> ([<ext-param-decl>])

<external-fct-name> ::= <valid-name>

<real-ext-fct-name> ::= <voyager-file>**.**<voyager-fct>

<voyager-file> ::= <readable-name>
<voyager-fct> ::= <valid-name>
<ext-param-decl> ::= <ext-param> {**,** <ext-param>}
<ext-param> ::= <ext-param-type> [<ext-param-name>]
<ext-param-type> ::= **list | type | integer | string**
<ext-param-name> ::= <valid-name>

D.7. Schema model description

A schema model is a specialisation of the GER model. It is identified by a name and a more readable title. A small description can be added to ease its understanding by database engineers. The specialisation is made up of (1) a concept selection and renaming list and (2) a series of structural constraints.

<schema-model> ::= **schema-model** <model-header> <model-title> [<textual-description>] [<schema-concepts>] [<model-constraints>] **end-model**
<model-header> ::= <model-name>[**is** <model-name>]
<model-name> ::= <valid-name>
<model-title> ::= **title** <readable-name>
title to be written to the screen
<schema-concepts> ::= **concepts** {<concept-line>}
<concept-line> ::= <concept-name> <readable-name>
one concept with its conceptual name
<concept-name> ::= <valid-name>
<model-constraints> ::= **constraints** {<constraint-block>}
<constraint-block> ::= <rule> <diagnosis-line>
one single constraint line
<rule> ::= <sch-anal-expr>
<diagnosis-line> ::= **diagnosis** <diagnosis-string>
the message to be displayed when the constraint is violated
<diagnosis-string> ::= <string>
a readable and meaningful string

D.8. Text model description

A text is any product that is not a schema in the sense above. An identifying name must be given to a text model as well as a readable name and a list of possible file extensions.

<text-model> ::= **text-model** <model-header> <model-title> [<textual-description>] <extensions> [<grammar>] **end-model**
<extensions> ::= **extensions** <extension-name> {**,** <extension-name>}
<grammar> ::= **grammar** <grammar-file>
<grammar-file> ::= <readable-name>
<extension-name> ::= <string>

D.9. Product type description

A product type has an identifying name, a readable name, a reference model and possibly a description.

<product-type> ::= **product** <product-name> <product-title> [<textual-description>] <product-model> [<multiplicity>] **end-product**
<product-name> ::= <valid-name>
<product-title> ::= **title** <readable-name>
title to be written to the screen
<product-model> ::= **model** [**weak**] <model-name>
<multiplicity> ::= **multiplicity** <min-max-mult>
<min-max-mult> ::= [<min-mult> - <max-mult>]
<min-mult> ::= <number>
<max-mult> ::= <number>**|n|N**

D.10. Toolbox description

A tool is a product transformation. For instance, a function for adding an entity-type is a tool. A toolbox is a set of such tools. It can be defined from an empty toolbox in which we add all the tools we need or from another one by adding or removing tools.

<toolbox>	::= **toolbox** <toolbox-header> <toolbox-title> [<textual-description>] <toolbox-body> **end-toolbox**
<toolbox-header>	::= <toolbox-name> [**is** <toolbox-name>]
<toolbox-name>	::= <valid-name>
	the toolbox identifier
<toolbox-title>	::= **title** <readable-name>
	name to be written to the screen
<toolbox-body>	::= <toolbox-line> {<toolbox-line>}
<toolbox-line>	::= <add-line>|<remove-line>
<add-line>	::= **add** <tool-name>
<remove-line>	::= **remove** <tool-name>
<tool-name>	::= <valid-name>
	the name of a function of the supporting CASE environment

D.11. Process type description

A process type is defined in three parts: a header with its name, its external definition and its internal definition.

The external definition contains some methodological aspects and a static definition of the process. Firstly, a title in clear text. It is that title that the user will see on screen. Secondly, the name of a section in the help file that should contain a description of the process. The user can read that file whenever he wants while performing a process of that type. Finally, the static description of the process type simply shows what product types are required in input and what product types are provided in output or updated, with the model they are conform to and possibly their cardinality constraint. The internal definition begins with the schema types used as the internal workplaces. Finally, the strategy shows how the process has to be performed.

<task-model>	::= **process** <task-name> <task-body> **end-process**
<task-name>	::= <valid-name>
	the task identifier
<task-body>	::= <task-title> [<textual-description>] <models-section> [<explain-line>] <strategy>
<task-title>	::= **title** <readable-name>
	title to be used by the user interface
<models-section>	::= [<input-line>] [<output-line>] [<update-line>] [<intern-line>] [<set-line>]
<input-line>	::= **input** <product-list>
	the product types expected in input that will not be modified
<product-list>	::= <product-element> {**,** <product-element>}
<product-element>	::= <product-name> [<multiplicity>] [<UI-name>] **:** [**weak**] <model-name>
<UI-name>	::= <readable-name>
<output-line>	::= **output** <product-list>
	the product types produced in output
<update-line>	::= **update** <product-list>
	the product types expected in update (input, transformation, output)
<intern-line>	::= **intern** <product-list>
	the product types to which the internal schemas must (or should) conform
<set-line>	::= **set** <product-set-list>
	the product types to which the internal schemas must (or should) conform
<product-set-list>	::= <product-set-element> {**,** <product-set-element>}
<product-set-element>	::= <product-set-name> [<multiplicity>] [<UI-name>]
<product-set-name>	::= <valid-name>
<explain-line>	::= **explain** <explain-section>
	the section in the help file where explanation and suggestions can be found
<explain-section>	::= <readable-name>

<strategy> ::= **strategy** <action>
body of a process

<action> ::= |<elem-action>|<compl-action>
action to be carried out, possibly no action

<elem-action> ::= <do-action>|<toolbox-action>|<external-action>|<glbtrsf-action>|
<extract-action>|<generate-action>|<message-action>|<built-in-action>

<do-action> ::= **do** <task-name> ([<do-prod-parameters>])

<do-prod-parameters> ::= [**content:**]<product-name> {**,** [**content:**]<product-name>}

<toolbox-action> ::= **toolbox** <toolbox-name> [[**log** <log-level>]] <tb-prod-parameters>

<log-level> ::= **off | replay | all**

<tb-prod-parameters> ::= **(** <product-name> {, <product-name>} **)**

<external-action> ::= **external** <external-fct-name> [[**log** <log-level>]] <ext-parameters>

<glbtrsf-action> ::= **glbtrsf** ["<transfo-name>"] [[**log** <log-level>]] **(** <product-name> **,** <global-trsf>
{, <global-trsf>} **)**

<global-trsf> ::= <transfo-name> **(**<free-text>**)**

<transfo-name> ::= <valid-name>

<extract-action> ::= **extract** <extractor-name> **(** <source-file> **,** <dest-schema> **)**

<extractor-name> ::= <valid-name>

<source-file> ::= <product-name>

<dest-schema> ::= <product-name>

<generate-action> ::= **generate** <generator-name> **(** <source-schema> **,** <dest-file> **)**

<generator-name> ::= <valid-name>

<source-schema> ::= <product-name>

<dest-file> ::= <product-name>

<message-action> ::= **message** <string>

<bulti-in-action> ::= <new-action>|<copy-action>|<import-action>|<cast-action>|<define-action>

<new-action> ::= **new (**<product-name>**)**

<copy-action> ::= **copy (**<product-name>,<product-name>**)**

<import-action> ::= **import (**<product-name>**)**

<cast-action> ::= **cast (**<product-name>,<product-name>**)**

<define-action> ::= **define (**<product-set-element>,<product-set-expr>**)**

<product-set-expr> ::= <product-set-op> <product-set-expr-list>

<product-set-op> ::= **union | inter | minus | subset | origin | target | choose-one | choose-many |**
first | remaining

<product-set-expr-list> ::= **(** <product-set-expr> {, <product-set-expr>} **)**

<compl-action> ::= <sequence>|<iterate>|<choose>|<alternate>
complex action

<sequence> ::= **sequence** <action-list> **end-sequence** | <action-list>
perform all actions of the body in the specified order

<action-list> ::= <action> {**;** <action>}
a list of actions separated by semi-colons

<iterate> ::= <repeat>|<while-repeat>|<repeat-until>|<for>

<repeat> ::= **repeat** <action> **end-repeat**

<while-repeat> ::= **while** <parenth-expr> <repeat>

<repeat-until> ::= <repeat> **until** <parenth-expr>

<for> ::= **for** <one-some-each> <product-name> **in** <product-name> **do** <action> **end-for**

<one-some-each> ::= **one | some | each**

<choose> ::= <one>|<some>|<each>

<one> ::= **one** <action-list> **end-one**
perform one action from the list

<some> ::= **some** <action-list> **end-some**
perform at least one action from the list in any order

<each> ::= **each** <action-list> **end-each**
perform each action from the list in any order

<alternate> ::= **if** <parenth-expr> **then** <action> [**else** <action>] **end-if**
carry out one action or the other according to the condition

Appendix E

DB-MAIN functions

The following pages show a table with all the functions of DB-MAIN version 6 with the different ways to use them. This can be by selecting a menu entry, by selecting a tool in a toolbar, by using a keyboard shortcut, by clicking on a mouse button when the mouse is pointing a place in a window, by a dialogue box, by a Voyager 2 program, by a method, or by the history replay function. Finally, when a function deserves to be known by the MDL language to be put in a toolbox, a name (possibly several when the context matters) is given to the function.

For instance, the *mark selected* function which marks the currently selected part of a product th the current marking plan can be called through the *Edit* menu, by clicking on the corresponding tool in the *standard* toolbar, by using a Voyager 2 function in a Voyager 2 program, or by a method (using a global transformation). We will name it *mark*. So, when defining a toolbox with the MDL language, the mark keyword can be used to allow the use of this *mark selected* function, as well as the *mark* function which allows to mark some specific product components.

Functions	**Menu**	**Tool bar**	**KS**	**Mouse**	**Dialogue boxes**	**V2**	**M**	**R**	**Name**
New project dialogue	File	Standard							
Create project					Project properties Open project	√			
Open project dialogue	File	Standard							
Save project dialogue	File	Standard							
Save project as dialogue	File	Standard							
Close project	File			⊠		√			
Project properties	File								
Modify project properties					Project properties	√			
Import dialogue	File								
Export dialogue	File								
Execute Voyager dialogue	File	Standard							
Execute Voyager					Execute Voyager	√	√	E	externals
Continue Voyager	File	Standard						E	externals
Rerun Voyager	File	Standard						E	externals
<Voyager programmes>	File						√	E	externals
Extract ... dialogue	File								
Extract ...					Extract		√		
Generate ... dialogue	File								
Generate ...					Generate Global transfo.		√		
Edit text file	File								
Print dictionary	File								

Functions	Menu	Tool bar	KS	Mouse	Dialogue boxes	V 2	M	R	Name
Printer Setup	File								
Configuration	File								
Exit	File								
Save point	Edit						√		
Rollback	Edit								
Copy to clipboard	Edit		Ctrl+C						
Paste from clipboard	Edit		Ctrl+V						create create-entity-type create-rel-type create-attribute create-processing-unit create-role create-group create-collection
Copy graphic to clipboard	Edit	Graphical							
Select All	Edit		Ctrl+A						
Select				Left		√			
Mark					Mark view Schema anal. result	√		√	mark
Mark selected	Edit	Standard				√	√		mark
Select marked	Edit					√	√	√	
Colour selected	Edit	Standard							colour
Colour						√		√	
Delete					Integrate two objects Remove view	√	√	√	
Delete selected	Edit		Del						delete delete-entity-type delete-rel-type delete-attribute delete-processing-unit delete-role delete-group delete-collection
Change font	Edit								
New schema dialogue	Product								
Create schema						√	√		
Add text dialogue	Product								
Create text						√	√		
New set dialogue	Product								
Create product set						√	√		
Open product	Product			Double left					
Product properties dialogue	Product								
Modify product properties					Product properties Graphical settings	√			
Copy product dialogue	Product						√		
Copy product					Copy product	√			
Define view dialogue	Product/View								create-view
Generate view dialogue	Product/View								create-view
Create view					Generate view				
Mark view dialogue	Product/View								mark
Remove view dialogue	Product/View								delete-view
Copy view dialogue	Product/View								create-view
Rename view dialogue	Product/View								modify-view
Meta-properties dialogue	Product/Meta								create-meta-prop delete-meta-prop modify-meta-prop
Create meta-property					Meta-property	√			create-meta-prop
Delete meta-property					Meta-property	√			delete-meta-prop
Modify meta-property					Meta-property	√			modify-meta-prop
User domains dialogue	Product								create-user-domain delete-user-domain modify-user-domain

Functions	Menu	Tool bar	KS	Mouse	Dialogue boxes	V2	M	R	Name
Create user domains						√			create-user-domain
Delete user domains						√			delete-user-domain
Modify user domains						√			modify-user-domain
Lock/unlock	Product								
New collection dialogue	New (TV)	Standard (TV)			Collection properties (TV)				create-collection create
New collection mode (=NCM)	New (GV)	Standard (GV)			Collection properties (GV)				create-collection create
End new collection mode	New Text standard Text compact Text extended Text sorted (NCM)	Standard (NCM)							
Create collection	Edit/Paste		Ctrl+V	Left (NCM)	Schema integrate	√		√	
New entity type dialogue	New (TV)	Standard (TV)			ET properties (TV)				create-entity-type create
New entity type mode (=NETM)	New (GV)	Standard (GV)			ET properties (GV)				create-entity-type create
End new entity type mode	New Text standard Text compact Text extended Text sorted (NETM)	Standard (NETM)							
Create entity type	Edit/Paste		Ctrl+V	Left (NETM)	Schema integrate	√		√	
New rel-type dialogue	New (TV)	Standard (TV)			RT properties (TV)				create-rel-type create
New rel-type mode (=NRTM)	New (GV)	Standard (GV)			RT properties (GV)				create-rel-type create
End new rel-type mode	New Text standard Text compact Text extended Text sorted (NRTM)	Standard (NRTM)							
Create rel-type	Edit/Paste		Ctrl+V	Left (NRTM)	Schema integrate	√		√	
New first attribute dialogue	New/ Attribute	Standard			Attribute properties ET properties RT properties Schema integrate				create-attribute create
New next attribute dialogue	New/ Attribute	Standard			Attribute properties Schema integrate Integrate two objects				create-attribute create
Create attribute	Edit/Paste		Ctrl+V		Attribute properties	√		√	
New processing unit dia- logue	New				Proc. unit properties Schema integrate				create-processing-unit create
Create processing unit	Edit/Paste		Ctrl+V		Proc. unit properties	√		√	
New role	New	Standard			Role properties (TV) RT properties (TV)				create-role create
New role mode (=NRM)	New (GV)	Standard (GV)			Role properties (GV) RT properties (GV)				create-role create
End new rel-type mode	New Text standard Text compact Text extended Text sorted (NRM)	Standard (NRM)							
Create role				Left (NRM)	Schema integrate	√		√	
New group dialogue	New								create-group create
Create group	Edit/Paste	Standard			Group properties Ref. key assistant Schema integrate	√		√	
Create identifier		Standard			Schema integrate	√		√	create-identifier create

Functions	Menu	Tool bar	KS	Mouse	Dialogue boxes	V2	M	R	Name
New constraint dialogue	New				Group properties				create-constraint create
Create constraint					Constraint properties Schema integrate	√		√	
Delete constraint					Constraint properties	√		√	delete-constraint
Constraint properties dialogue					Group properties	√		√	create-constraint delete-constraint modify-constraint create delete modify
Modify constraint					Constraint properties				
Entity type -> rel-type	Transform	Transfo			Global transfo. Adv. global transfo.			√	tf-ET-into-RT
Entity type -> attribute	Transform	Transfo			Global transfo. Adv. global transfo.			√	tf-ET-into-att
Is-a -> rel-types	Transform	Transfo			Global transfo. Adv. global transfo.			√	tf-isa-into-RT
Rel-types -> is-a	Transform	Transfo			Global transfo. Adv. global transfo.			√	tf-RT-into-isa
Split/merge	Transform	Transfo			Global transfo. Adv. global transfo.			√	tf-split-merge
Add technical identifier	Transform	Transfo			Global transfo. Adv. global transfo.			√	tf-add-tech-id
Rel-type -> entity type	Transform	Transfo			Global transfo. Adv. global transfo.			√	tf-RT-into-ET
Rel-type -> attribute	Transform	Transfo			Global transfo. Adv. global transfo.			√	tf-RT-into-att
Rel-type -> object attribute	Transform	Transfo			Global transfo. Adv. global transfo.			√	tf-RT-into-obj-att
Attribute -> entity type	Transform	Transfo			Global transfo. Adv. global transfo.			√	tf-att-into-ET
Disaggregation	Transform	Transfo			Global transfo. Adv. global transfo.			√	tf-disaggregate
Multi att. -> single	Transform	Transfo			Global transfo. Adv. global transfo.			√	tf-multi-att-into-single
Single att. -> multi	Transform	Transfo			Global transfo. Adv. global transfo.			√	tf-single-att-into-multi
Multi att -> list single	Transform	Transfo			Global transfo. Adv. global transfo.			√	tf-multi-att-into-list
Multi attribute conversion	Transform	Transfo			Global transfo. Adv. global transfo.			√	tf-multi-att-conversion
Materialize domain	Transform	Transfo			Global transfo. Adv. global transfo.			√	tf-materialize-domain
Object attribute -> rel-type	Transform	Transfo			Global transfo. Adv. global transfo.			√	tf-obj-att-into-RT
Multi-ET role -> rel-type	Transform	Transfo			Global transfo. Adv. global transfo.			√	tf-multi-ET-role-into-RT
Group -> rel-type	Transform	Transfo			Global transfo. Adv. global transfo.			√	tf-ref-group-into-RT
Aggregate	Transform	Transfo			Global transfo. Adv. global transfo.			√	tf-aggregate
Group -> multi attribute	Transform	Transfo			Global transfo. Adv. global transfo.			√	tf-list-into-multi-att
Change prefix	Transform							√	change-prefix
Name processing dialogue	Transform				Global transfo.				name-processing
Name processing					Global transfo.			√	
Relational model	Transform							√	
Quick SQL	Transform								
Global transfo. dialogue	Assist								global-transfo
Global transformations					Globval transfo.			√	
Adv. global transfo. dialogue	Assist								advanced-global-transfo
Advanced global transfo.					Adv. global transfo.		√	√	

Functions	Menu	Tool bar	KS	Mouse	Dialogue boxes	V 2	M	R	Name
Schema analysis dialogue	Assist						√		schema-analysis
Schema analysis					Schema analysis		√		
Schema integration dialogue	Assist								schema-integration
Schema integration					Schema integration				
Object integration dialogue	Assist								object-integration
Object integration					Object integration			√	
Load pattern	Assist/text anal.								text-analysis
Search	Assist/text anal.	Rev. Eng.							text-analysis
Search next	Assist/text anal.	Rev. Eng.							text-analysis
Execute	Assist/text anal.								text-analysis
Dependency	Assist/text anal.								text-analysis
Progarm slicing	Assist/text anal.	Rev. Eng.							text-analysis
Mark slice	Assist/text anal.								text-analysis
Slicing assistant	Assist/text anal.								text-analysis
Dependency/SDG	Assist/text anal.								text-analysis
Load SDG	Assist/text anal.								text-analysis
Save SDG	Assist/text anal.								text-analysis
Load parsing	Assist/text anal.								text-analysis
Free SDG	Assist/text anal.								text-analysis
Text analysis settings	Assist/text anal.								text-analysis
Text analysis color	Assist/text anal.								text-analysis
Goto	Assist/text anal.	Rev. Eng.							text-analysis
Referential key dialogue	Assist	Rev. Eng.							ref-key-search
Use primitives	Engineering	Process					√		
Copy schema & use prim.	Engineering								
End use of primitives	Engineering	Process							
New engineering process	Engineering	Process					√		
End current process	Engineering	Process							
Continue process	Engineering	Process							
Take decision	Engineering	Process					√		
Process properties	Engineering								
Edit in/out/update prod.	Engineering								
Controlled history	Engineering								
Trace	Log								
Add check point	Log						√		
Add schema	Log								
Add description	Log								
Clear log	Log								
Save log as	Log								
Replay automatic	Log								
Replay interactive	Log								
Text compact	View								
Text standard	View	Standard							
Text extended	View								
Text sorted	View								
Graph compact	View								
Graph standard	View	Standard							
Graph dependency	View								
Graphical settings dialogue	View								
Alignment ...	View	Graphical						√	
Autodraw	View								
View engineering method	View								
...	Window								
...	Help								

Functions	Menu	Tool bar	KS	Mouse	Dialogue boxes	V2	M	R	Name
Object properties dialogue			Enter	Double left					modify modify-entity-type modify-rel-type modify-attribute modify-processing-unit modify-role modify-group modify-collection
Modify object properties					... properties				
Edit semantic description		Standard			... properties Conflict resolution	√		√	modify-sem-desc
Edit technical description		Standard			... properties Conflict resolution	√		√	modify-tech-desc
Edit meta-properties dia-logue		Standard			... properties				modify-meta-prop-value
Modify meta-properties					Edit meta-properties Define view Remove view Copy view Rename view	√		√	
Change mark plan		Standard				√			
Independant graph. objects		Graphical			Graphical settings				
Zoom +10%		Graphical							
Zoom -10%		Graphical							
Zoom		Graphical			Graphical settings	√			
Connection dialogue					Text properties Schema properties	√			edit-connection
Center object				Right				√	

Abbreviations:

KS	keyboard shortcut
V2	Voyager 2 language
M	method
R	replay function
(TV)	textual view
(GV)	graphical view
E. (in the “R” column)	Effects: the Voyager 2 function executions are not stored in the repository, but all the transformations they perform are stored.
☒	closing button in the title bar of a window

Appendix F

Case study listings

This appendix contains the material required to perform the two case studies in Chapter 11: the MDL listings of the methods followed, the texts required to start the case studies, and the complete script of the first case study.

F.1. The first case study: a forward engineering method

```
% Product models definitions
%%%%%%%%%%%%%%%%%%%%%%%%%%%%

text-model TEXT_FILE
  title "Text file"
  description
    A text file contains some free text. In this method, we will use them
    to store reports written in natural language.
  end-description
  extensions "TXT"
end-model

text-model SQL_FILE
  title "SQL file"
  description
    An SQL script containing SQL instructions for the creation of a database
    including create database, create table, create index, alter table with
    checks, create trigger,...
  end-description
  extensions "SQL", "DDL"
end-model

schema-model CONCEPT_SCHEMA
  title "Conceptual schema model"
  description
    The conceptual schema model allows an analyst to draw a representation of the
    real world. A schema compliant with that model shows precisely, in a readable
    way, the semantics of the database. It cannot be directly implemented. Its
    main purpose is to be a basis for documenting the database, to be a support
    for dialogue.
  end-description
  concepts
    schema                  "schema"
    entity_type             "entity type"
    rel_type                "relationship type"
    attribute               "attribute"
    atomic_attribute        "attribute"
    compound_attribute      "compound attribute"
    role                    "role"
    is_a_relation           "is-a relation"
    sub_type                "sub-type"
    super_type              "super-type"
    note                    "note"
    group                   "group"
    identifier              "identifier"
```

```
    primary_identifier      "primary identifier"
    secondary_identifier    "secondary identifier"
    constraint              "constraint"
    at_least_one_constraint "at-least-one constraint"
    exactly_one_constraint  "exactly-one constraint"
    coexistence_constraint  "coexistence constraint"
    exclusive_constraint    "exclusive constraint"
    user_constraint         "constraint"
  constraints
    ET_per_SCHEMA (1 N)        % At list one ET required
      diagnosis "Schema &NAME should have an entity type"
    COLL_per_SCHEMA (0 0)      % No collection allowed
      diagnosis "The schema should have no collection"
    ATT_per_ET (1 N)           % At least one attribute per ET
      diagnosis "Entity type &NAME should have at least one attribute"
    KEY_per_ET (0 0)           % No access keys
      diagnosis "Entity type &NAME should not have an access key"
    REF_per_ET (0 0)           % No foreign key
      diagnosis "Entity type &NAME should not have a foreign key"
    ID_per_ET (1 N)            % If there are identifiers, one is primary
        and PID_per_ET (1 1)
      or ID_per_ET (0 0)
      diagnosis "One of the identifiers of entity type &NAME should be primary"
    EMBEDDED_ID_per_ET (0 0)   % Embedded identifiers are not allowed"
      diagnosis "Embedded identifiers should be removed in entity type &NAME"
    ID_DIFF_in_ET (components) % All identifiers must have different components
      diagnosis "Ids made up of the same components should be avoided in &NAME"
    TOTAL_in_ISA (no)          % Total is-a relations should concern at least
      or TOTAL_in_ISA (yes)    % two subtypes
        and SUB_TYPES_per_ISA (2 N)
      diagnosis "Total is-a relations are not allowed with only one sub-type"
    DISJOINT_in_ISA (no)       % Disjoint is-a relations should concern at
      or TOTAL_in_ISA (yes)    % least two subtypes
        and SUB_TYPES_per_ISA (2 N)
      diagnosis "Disjoint is-a relations must have at least two sub-types"
    ROLE_per_RT (2 2)          % 2 <=  degree of a rel-type  <= 4
      or ROLE_per_RT (3 4)     % if 3 or 4, the rel-type cannot have a one role
        and ATT_per_RT (1 N)   %  or it must also have attributes
      or ROLE_per_RT (3 4)
        and ATT_per_RT (0 0)
        and ONE_ROLE_per_RT (0 0)
      diagnosis "Rel-type &NAME has too many roles, or too few attributes"
    ID_per_RT (1 N)            % If RT have some identifiers, one is primary
        and PID_per_RT (1 1)
      or ID_per_RT (0 0)
      diagnosis "One of the identifiers of rel-type &NAME should be primary"
    EMBEDDED_ID_per_RT (0 0)   % Embedded identifiers are not allowed"
      diagnosis "Embedded identifiers should be removed in rel-type &NAME"
    ID_DIFF_in_RT (components) % All identifiers must have different components
      diagnosis "Ids made up of the same components should be avoided in &NAME"
    not SUB_ATT_per_ATT (1 1)  % Compound att must have at least two components
      diagnosis "Compound attribute &NAME has too few sub-attributes"
    ID_per_ATT (0 0)           % A compound attribute cannot have an identifier
      diagnosis "Multi-valued compound attribute &NAME should not have an id."
    COMP_per_GROUP (1 N)       % Every group must have at least one component
      diagnosis "Group &NAME should have components"
    ROLE_per_EID (0 0)         % An ET id. cannot be made up of a single role
        and COMP_per_EID (1 N)
      or ROLE_per_EID (1 N)
        and COMP_per_EID (2 N)
      diagnosis "ET Identifier &NAME should have another component"
    MULT_ATT_per_EID (1 1)     % If an ET id. contains a multi-valued attribute
        and COMP_per_EID (1 1) % it must be the only component.
      or MULT_ATT_per_EID (0 0)
      diagnosis "ET identifier &NAME should have no multi-valued attribute or
                                                         no other component"
    ONE_ROLE_per_EID (0 0)     % An ET identifier should not have a one-role
      diagnosis "One-roles should be removed from entity type identifier &NAME"
    OPT_ATT_per_EPID (0 0)     % Optional columns not allowed in primary ids.
      diagnosis "There should be no optional column in primary id &NAME."
    COMP_per_RID (1 1)         % If a RT identifier has only one component,
        and ROLE_per_RID (0 0) % it must be an attribute
      or COMP_per_RID (2 N)
      diagnosis "Rel-type identifier &NAME should have more components"
    MULT_ATT_per_RID (1 1)     % If a RT identifier contains a multi-valued
        and COMP_per_RID (1 1) % attribute, it must be the only component.
      or MULT_ATT_per_RID (0 0)
```

```
      diagnosis "RT identifier &NAME should have no multi-valued attribute
                                                     or no other component"
    ONE_ROLE_per_RID (0 0)     % A RT identifier should not have a one-role
      diagnosis "One-roles should be removed from rel-type identifier &NAME"
    OPT_ATT_per_RPID (0 0)     % No optional attribute in a rel-type identifier
      diagnosis "Optional att. should be removed from RT identifier &NAME"
end-model

schema-model LOG_SQL_SCHEMA
  title "Logical relational schema"
  description
    The logical relational schema model maps the generic entity/object-
    relationship (GER) model of DB-MAIN to a generic relational model, without
    any specific RDBMS in mind. Schemas compliant with this model are the one
    to give as a reference to people who need to write queries on the database.
  end-description
  concepts
    schema             "view"
    entity_type        "table"
    atomic_attribute   "column"
    user_constraint    "constraint"
    identifier         "unique constraint"
    primary_identifier "primary key"
    access_key         "index"
  constraints
    ET_per_SCHEMA (1 N)     % At list one table required
      diagnosis "Schema &NAME should have a table"
    RT_per_SCHEMA (0 0)     % No rel-type allowed
      diagnosis "Rel-type &NAME should not exist"
    COLL_per_SCHEMA (0 0)   % No collection/table space allowed
      diagnosis "The schema should have no table space"
    ATT_per_ET (1 N)        % At least one column per table
      diagnosis "Table &NAME should have at least one column"
    PID_per_ET (0 1)        % At most one primary key per ET
      diagnosis "Table &NAME has too many primary keys"
    KEY_per_ET (0 0)        % No access keys/indexes
      diagnosis "Table &NAME should not have an index"
    SUB_TYPES_per_ISA (0 0) % Is-a relations are not allowed
      diagnosis "Is-a relations are not allowed and &NAME has a sub-type"
    OPT_ATT_per_EPID (0 0)  % Optional columns not allowed in primary keys.
      diagnosis "There should be no optional column in primary key &NAME."
    DEPTH_of_ATT (1 1) and MAX_CARD_of_ATT (1 1)
                            % Columns are atomic and single-valued
      diagnosis "Column &NAME should be atomic and single-valued."
end-model

schema-model PHYS_SQL_SCHEMA
  title "SQL schema model"
  description
    The SQL schema model maps the generic entity/object-relationship (GER) model
    of DB-MAIN to an SQL relational model, including physical characteristics such
    as the setting of indexes and the definition of dataspaces. This is the schema
    model from which database creation scripts can be derived. This is the schema
    that can be used as a reference for the database administrator to fine tune
    the database.
  end-description
  concepts
    collection         "table space"
    schema             "view"
    entity_type        "table"
    atomic_attribute   "column"
    user_constraint    "constraint"
    identifier         "unique constraint"
    primary_identifier "primary key"
    access_key         "index"
  constraints
    ET_per_SCHEMA (1 N)     % At list one table required
      diagnosis "Schema &NAME should have a table"
    RT_per_SCHEMA (0 0)     % No rel-type allowed
      diagnosis "Rel-type &NAME should not exist"
    ATT_per_ET (1 N)        % At least one column per table
      diagnosis "Table &NAME should have at least one column"
    PID_per_ET (0 1)        % At most one primary key per table
      diagnosis "Table &NAME has too much primary keys"
    SUB_TYPES_per_ISA (0 0) % Is-a relations are not allowed
      diagnosis "Is-a relations are not allowed and &NAME has a sub-type"
    ID_NOT_KEY_per_ET (0 0) % Every unique constraint is an index
```

```
      diagnosis "Unique constraint &NAME should be an index"
    OPT_ATT_per_EPID (0 0)  % Optional columns not allowed in primary keys.
      diagnosis "There should be no optional column in primary key &NAME."
    DEPTH_of_ATT (1 1) and MAX_CARD_of_ATT (1 1)
      % Columns are atomic and single-valued
      diagnosis "Column &NAME should be atomic and single-valued."
    ALL_CHARS_in_LIST_NAMES (ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstu
                                                          vwxyz0123456789$_)
      and NONE_in_LIST_NAMES (_$,$$)
      and LENGTH_of_NAMES (0 31)
      and NONE_in_FILE_CI_NAMES (PHYSRDB.NAM)
      diagnosis "The name &NAME is invalid"
end-model

% Toolbox definitions
%%%%%%%%%%%%%%%%%%%%%

toolbox TB_ANALYSIS
  title "Analysis"
  description
    This toolbox allows you to draw a conceptual schema. You can create and edit
    entity types, relationship types, attributes, roles and integrity constraints.
  end-description
  add create-entity-type
  add create-rel-type
  add create-attribute
  add create-group
  add create-role
  add modify-entity-type
  add modify-rel-type
  add modify-attribute
  add modify-group
  add modify-role
  add delete-entity-type
  add delete-rel-type
  add delete-attribute
  add delete-group
  add delete-role
end-toolbox

toolbox TB_CONCEPTUAL_NORMALISATION
  title "Conceptual normalisation"
  description
    This toolbox allows you to enhance the readability of your conceptual schema
    without modifying its semantics. You can do it by applying some
    transformations on entity types, relationship types and attributes. You should
    be aware of some entity types that look like relationship types (the roles
    they play are all 1-1 and they are identified by all the roles they play),
    of some entity types that look like attributes (small, just a few attributes,
    and they play a single role in a single relationship type), of some entity
    types that are linked by a one to one relationship type and that have the same
    semantics, and of large entity types that do not have a clear semantic.
  end-description
  add tf-ET-into-att
  add tf-att-into-ET
  add tf-RT-into-ET
  add tf-ET-into-RT
  add tf-split-merge
  add modify-entity-type
  add modify-rel-type
  add modify-attribute
  add modify-group
  add modify-role
end-toolbox

toolbox TB_NAME_CONVERSION
  title "Name conversion"
  description
    The names of all objects of the schema should be transformed by removing
    white spaces, accents and other special symbols.
  end-description
  add name-processing
end-toolbox

toolbox TB_STORAGE_ALLOCATION
  title "Storage allocation"
  description
```

```
    Allows you to define what files have to create and which table goes in
    which file.
  end-description
  add create-collection
  add modify-collection
  add delete-collection
end-toolbox

toolbox TB_SETTING_PARAMETERS
  title "Setting coding parameters"
  description
    Allows you to update technical descriptions in order to specify a few
    database engine dependent parameters.
  end-description
  add modify-tech-desc
end-toolbox

% Process types definitions
%%%%%%%%%%%%%%%%%%%%%%%%%%%

process CONCEPTUAL_ANALYSIS
  title "Conceptual analysis"
  description
    On the basis of interview reports with the future users of the system that
    will be build, a conceptual schema of the database is drawn.
    It has to reflect the real world system.
  end-description
  input Interview_report[1-N] "Interview report" : TEXT_FILE
  output Conceptual_schema "Conceptual schema" : CONCEPT_SCHEMA
  strategy
    new(Conceptual_schema);
    toolbox TB_ANALYSIS [log off] (Conceptual_schema,Interview_report);
    toolbox TB_CONCEPTUAL_NORMALISATION [log all] (Conceptual_schema);
end-process

process RELATIONAL_TRANSLATION
  title "Relational design"
  description
    Transformation of a binary schema into a relational (SQL-compliant) schema.
  end-description
  update Logical_schema "Relational logical schema" : LOG_SQL_SCHEMA
  strategy
    % Transform is-a relations
    glbtrsf "Is-a relations" (Logical_schema,ISA_into_RT);
    % Transform all non-functional rel-types
    glbtrsf "Non-functional rel-types" (Logical_schema,
                                    RT_into_ET(ROLE_per_RT(3 N) or
                                      ATT_per_RT(1 N)),
                                    SPLIT_MULTIET_ROLE,
                                    RT_into_ET(N_ROLE_per_RT(2 2)));
    % Transform all compound and/or multi-valued attributes
    glbtrsf "Attributes"(Logical_schema,
                         LOOP,
                           ATT_into_ET_INST(MAX_CARD_of_ATT(2 N)),
                           DISAGGREGATE,
                         ENDLOOP);
    % Add technical identifiers where needed in order to be able to transform
    % all rel-types into referential constraints
    glbtrsf "Identifiers" (Logical_schema,SMART_ADD_TECH_ID);
    % Transform all rel-types into referential constraints
    glbtrsf "References" (Logical_schema,
                          LOOP,
                            RT_into_REF,
                          ENDLOOP)
end-process

process LOGICAL_DESIGN
  title "Logical design"
  description
    Logical design is the process of transforming a conceptual schema into a data
    model compliant schema, a relational model compliant schema in this case. In a
    first time, the conceptual schema will be simplified (transformed into a
    binary schema). It will be possible, in a second time, to optimise this
    simplified schema. In a third time, this optimised schema will be transformed
    into a relational schema. Finally, a few relational model specific
    optimisations can be performed.
  end-description
```

```
  input Conceptual_schema "Conceptual schema" : CONCEPT_SCHEMA
  output Logical_schema "Logical schema" : LOG_SQL_SCHEMA
  intern Raw_logical_schema "Raw logical schema" : weak LOG_SQL_SCHEMA
  strategy
    copy(Conceptual_schema,Raw_logical_schema);
    do RELATIONAL_TRANSLATION(Raw_logical_schema);
    copy(Raw_logical_schema,Logical_schema);
    toolbox TB_NAME_CONVERSION [log all] (Logical_schema);
end-process

process PHYSICAL_DESIGN
  title "Physical design"
  description
    Physical design is the process of updating a logical schema into a DBMS
    specific schema by adjunction of a series of specific structures like
    files, access keys,...
  end-description
  input Logical_schema "Logical schema" : LOG_SQL_SCHEMA
  output Physical_schema "Physical schema" : PHYS_SQL_SCHEMA
  strategy
    copy(Logical_schema,Physical_schema);
    % setting indexes
    glbtrsf "Setting indexes" (Physical_schema,
                               RENAME_GROUP,
                               GROUP_into_KEY(ID_in_GROUP(YES) or
                                 REF_in_GROUP(YES)),
                               REMOVE_PREFIX_KEY);
    toolbox TB_STORAGE_ALLOCATION(Physical_schema);
end-process

process CODING
  title "Coding"
  description
    Coding consists in setting a few database dependent parameters and
    generating an SQL DDL file.
  end-description
  input Physical_schema "Physical schema" : PHYS_SQL_SCHEMA
  intern Completed_physical_schema "Physical schema" : PHYS_SQL_SCHEMA
  output SQL_script "SQL database definition script" : SQL_FILE
  strategy
    copy(Physical_schema,Completed_physical_schema);
    toolbox TB_SETTING_PARAMETERS [log replay] (Completed_physical_schema);
    generate STD_SQL(Completed_physical_schema,SQL_script)
end-process

process FORWARD_ENGINEERING
  title "Forward engineering"
  description
    Forward engineering is the process of building a database from a conceptual
    schema. In this context, you will have to design an SQL database.
  end-description
  intern Interview_report "Interview report" : TEXT_FILE,
         Conceptual_schema "Conceptual schema" : CONCEPT_SCHEMA,
         Logical_schema "Logical schema" : LOG_SQL_SCHEMA,
         Physical_schema "Physical schema" : PHYS_SQL_SCHEMA,
         SQL_script "SQL database definition script" : SQL_FILE
  strategy
    repeat
      new(Interview_report);
    end-repeat;
    do CONCEPTUAL_ANALYSIS(Interview_report,Conceptual_schema);
    do LOGICAL_DESIGN(Conceptual_schema,Logical_schema);
    do PHYSICAL_DESIGN(Logical_schema,Physical_schema);
    do CODING(Physical_schema,SQL_script)
end-process

% Method definition
%%%%%%%%%%%%%%%%%%%

method
  title "Forward engineering"
  version "1.0"
  author "D. Roland"
  date "28-10-1998"
  perform FORWARD_ENGINEERING
end-method
```

F.2. The first case study: the interview report

This interview report is a text stored in the file "library.txt". It is used by the database engineer to design the conceptual schema of the database he designs.

```
A book is considered a piece of literary, scientific or technical
writing.

Every book has an identifying number, a title, a publisher, a
first published date, keywords, and an abstract (the abstracts are
being encoded), the names of its authors, and its bibliographic
references (i.e. the books it references).

For each book, the library has acquired a certain number (0, 1 or
more) of copies. The copies of a given book have distinct serial
numbers.  For each copy,  the date it was acquired is known, as
well as its location in the library (i.e. the store, the shelf and
the row in which it is normally stored), its borrower (if any),
the number of volumes it comprises.

It appears that one cannot borrow one individual volume, but that
one must borrow all the volumes of a copy. In addition, the copies
of a given book may have different numbers of volumes.  A book is
also characterised by its physical state (new, used, worn, torn,
damaged, etc), specified by a one-character code, and by an
optional comment on this state.

The author of a book has a name, a first name, a birth date, and
an origin (i.e. the organisation which (s)he came from when the
book was written).

For some authors, only the name is known. The employees admit that
two authors may have the same name (and first name), but such a
situation does not seem to raise any problem. Only the authors of
books known by the library are recorded.

A copy can be borrowed, at a given date, by a borrower. Borrowers
are identified by a personal id. The library records the name, the
first name, the address (name of the company, street, zip-code and
city name), as well as the phone numbers of each borrower.  In
addition, when (s)he is absent, another borrower (who is
responsible for the former) can be contacted instead.

When a copy is brought back, it is put in a basket from which it
is extracted at the end of the day to be stored in its location,
so that it is available again from the following day on. A copy is
borrowed on behalf of a project (identified by its name, but also
by its internal code). When a copy is brought back to the desk,
the "employee records the following information on this copy:
borrowing date, current date, borrower and project.
```

F.3. The first case study: the script of actions performed by the engineer

This script is the list of every actions performed by the database engineer using the DB-MAIN CASE environment. Following this script step by step, the reader should be able to perform exactly the same project too.

The following notational conventions are used in the script:

- **Bold characters** are used to show menu entries to select, or static text in dialogue boxes.
- *Italics* is used to show some text to be typed by the user.
- Square brackets **[...]** show a button to push.
- Quotes "..." surround a graphical object (process type, process, product type, product) that can be found in a window, or a file name.

Menu **File/New project**
Name: *Library*
Short name: *lib*
Methodology: *forward.lum*
[OK]

Menu **Windows/Tile** to display the method and the project windows side by side.

The engineer executes the "New" process type: in the method window, he clicks on the "New" process type with the mouse right button; a contextual menu appears, he selects **Execute**.

A **File open** dialogue box appears
The engineer selects the "library.txt" file.
And he validates by cliking **[OK]**

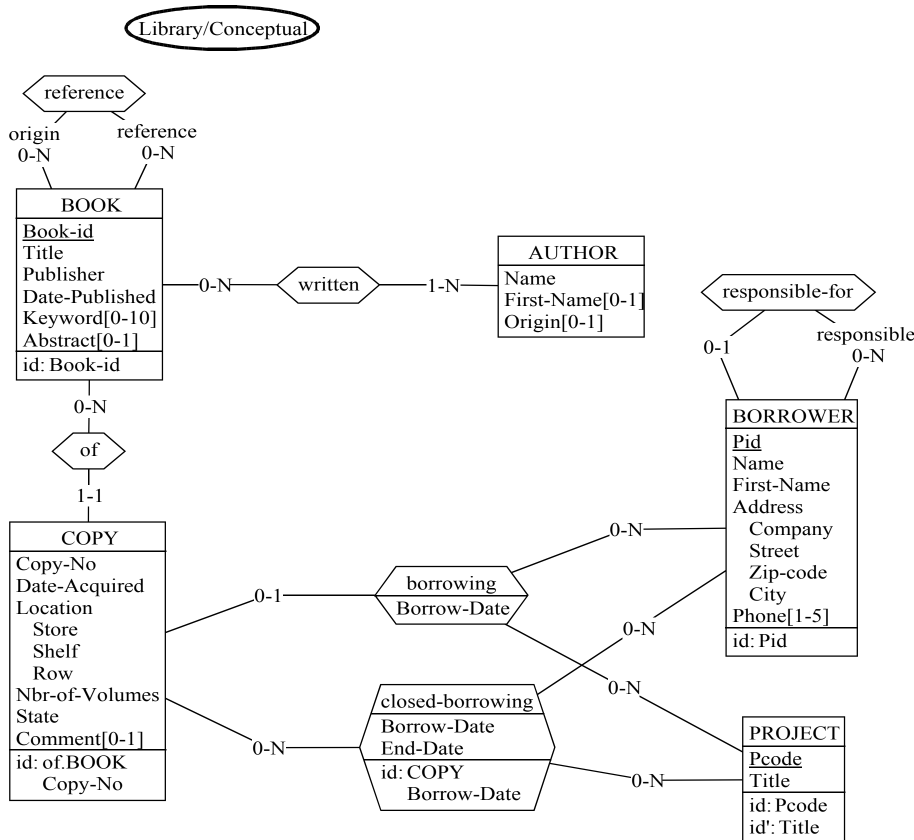
The engineer executes "Conceptual analysis".
He confirms the new proposed process name by clicking **[OK]**.

He executes the "New" process type.
Name: *Library*
Short name: *lib*
Version: *conceptual*
[OK].

He executes a process of the "Analysis" type.

The engineer opens the "Library.txt/IR" text file and the "Library/conceptual" schema.

He draws the following conceptual schema:



When finished, he closes both the schema and the text.

To signal the end of the job, in the project window, he selects the “Analysis” process.
Menu **Engineering/End use of primitives**.

And he terminates the “Analysis” process type: in the method window, he clicks on the “Analysis” process type, shown in the *running* state, with the mouse right button; a contextual menu appears, he selects **Terminate**.

The engineer executes “Conceptual normalisation”.

He opens the “Library/Conceptual” schema.

He sees that the schema is already normalised, so he immediately closes the window.

He selects “Conceptual normalisation” in the project window.

Menu **Engineering/End use of primitives**.
He terminates “Conceptual normalisation”.

The “Conceptual analysis” process is automatically terminated by the method engine. A dialogue box appears to allow the engineer to confirm the list of output products
He clicks on **[OK]** in the **End engineering process** dialogue box.

He terminates “Conceptual analysis”.

The engineer executes “Logical design”.

He executes “Copy”.
Version: *first logical*
[OK]

He executes “Relational design”.

The engineer executes “Process Is-a relations”.

He executes “Process non-functional rel-types”.

He executes “Process attributes”.

He executes “Process identifiers”.

He executes “Process references”

In the project window, the engineer selects menu **Engineering/End current process**, and he terminates “Relational design”.

The engineer executes “Copy”.
Version: *logical*

He executes “Name conversion”.

He opens “Library/logical”.

Menu **Transform/Name processing**
[Add] *"-"* -> *"_"* **[OK]**
[lower -> uppercase]
[OK]

The engineer closes the schema.

He selects “Name conversion” in the project window and the **Engineering/End use of primitives** menu entry.

He terminates “Name conversion”.

The **End engineering process** automatically appears. He clicks on **[OK]**

And he terminates “Logical design”.

He executes “Physical design”.

He executes “Copy”.
Version: *Physical*
[OK].

He executes “Setting indexes”.
[OK].

The engineer executes “Storage allocation”.

He opens schema “Library/Physical”, creates two collections and fills them:
- LIBRARY (AUTHOR,BOOK,COPY,KEYWORD,REFERENCE,WRITTEN)
- BORROWING(BORROWER,BORROWING,CLOSED_BORROWING, PHONE,PROJECT)

He closes the schema.

He selects “Storage allocation” in the project window and the **Engineering/End use of primitives** menu entry

He terminates “Storage allocation”.

The **End engineering process** automatically appears. The engineer confirms

He terminates “Physical design”.

The engineer executes “Coding”.

He executes “Copy”.
Version: *Implemented*
[OK]

He executes “Setting coding parameters”.

He opens the schema, decides to do nothing and closes the schema.

He selects “Setting coding parameters” in the project window and the **Engineering/End use of primitives** menu entry

He terminates “Setting coding parameters”.

The engineer executes “Generate”.
File Name: *LIBRARY.DDL*

The **End engineering process** automatically appears. The engineer confirms.

He terminates “Coding”.

F.4. The second case study: a reverse engineering method

```
% Product models definitions
%%%%%%%%%%%%%%%%%%%%%%%%%%%%

text-model COBOL_FILE
  title "COBOL file"
  description
    A COBOL program stored in an ASCII file.
  end-description
  extensions "COB"
end-model

schema-model PHYS_SCHEMA
  title "Physical schema model"
  concepts
    % No concepts declared here because this model must be inherited
  constraints
```

```
    ET_per_SCHEMA (1 N)     % At list one table required
      diagnosis "Schema &NAME should have a table"
    RT_per_SCHEMA (0 0)     % No rel-type allowed
      diagnosis "Rel-type &NAME should not exist"
    ATT_per_ET (1 N)        % At least one column per table
      diagnosis "Table &NAME should have at least one column"
    PID_per_ET (0 1)        % At most one primary ID per table
      diagnosis "Table &NAME has too much primary identifiers"
    SUB_TYPES_per_ISA (0 0) % Is-a relations are not allowed
      diagnosis "Is-a relations are not allowed and &NAME has a sub-type"
    ID_NOT_KEY_per_ET (0 0) % Every identifier is an access key
      diagnosis "Identifier &NAME should be an access key"
    OPT_ATT_per_EPID (0 0)  % Optional columns not allowed in primary ids.
      diagnosis "There should be no optional column in primary id &NAME."
end-model

schema-model PHYS_COBOL_SCHEMA is PHYS_SCHEMA
  title "COBOL schema model"
  concepts
    project             "database"
    schema              "view"
    entity_type         "table"
    atomic_attribute    "field"
    compound_attribute  "compound field"
    identifier          "key"
    primary_identifier  "primary_key"
    access_key          "key"
    user_constraint     "constraint"
  constraints
    ID_KEY_per_ET (1 N)               % At least one identifying AK per table
      diagnosis "Table &NAME has access keys that should also be identifiers"
    not KEY_PREFIX_in_ET (any_order)  % No key is a prefix of another one with
                                      % the fields in a different order.
      diagnosis "An invalid prefix key in &NAME should have its fields sorted"
    KEY_per_ET (0 0)                  % If there are several keys, at least one
      or KEY_per_ET (1 N)             % of them must be an identifier
         and ID_KEY_per_ET (1 N)
      diagnosis "In table &NAME, at least one key should be an identifier too"
    COLL_per_ET (1 1)                 % Each ET must be in one and only one file
      diagnosis "Table &NAME should be in one and only one file"
    COMP_per_EID (1 1)                % An identifier is made of a single
      and ATT_per_EID (1 1)           % single-valued field
      and MULT_ATT_per_EID (0 0)
      diagnosis "Id. &NAME should be made of a single single-valued field"
    COMP_per_KEY (1 1)                % An access key is made of a single
      and ATT_per_KEY (1 1)           % single-valued field
      and MULT_ATT_per_KEY (0 0)
      diagnosis "AK &NAME should be made of a single single-valued field"
    ALL_CHARS_in_LIST_NAMES (ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuv
                                                        vwxyz0123456789$_-)
      and ALL_in_LIST_CI_NAMES (*A*,*B*,*C*,*D*,*E*,*F*,*G*,*H*,*I*,*J*,*K*,*L*,
                          *M*,*N*,*O*,*P*,*Q*,*R*,*S*,*T*,*U*,*V*,*W*,*X*,*Y*,*Z*)
      and NONE_in_LIST_NAMES (^-,-$)
      and LENGTH_of_NAMES (0 31)
      and NONE_in_FILE_CI_NAMES (VAXCOBOL.NAM)
      diagnosis "The name &NAME is invalid"
end-model

schema-model LOG_SCHEMA
  title "Logical schema model"
  concepts
    % No concepts declared here because this model must be inherited
  constraints
    ET_per_SCHEMA (1 N)     % At list one table required
      diagnosis "Schema &NAME should have a table"
    RT_per_SCHEMA (0 0)     % No rel-type allowed
      diagnosis "Rel-type &NAME should not exist"
    COLL_per_SCHEMA (0 0)   % No collection allowed
      diagnosis "The schema should have no collection"
    ATT_per_ET (1 N)        % At least one column per table
      diagnosis "Table &NAME should have at least one column"
    PID_per_ET (0 1)        % At most one primary ID per ET
      diagnosis "Table &NAME has too many primary identifiers"
    KEY_per_ET (0 0)        % No access keys
      diagnosis "Table &NAME should not have an access key"
    SUB_TYPES_per_ISA (0 0) % Is-a relations are not allowed
      diagnosis "Is-a relations are not allowed and &NAME has a sub-type"
```

```
    OPT_ATT_per_EPID (0 0)  % Optional columns not allowed in primary ids.
      diagnosis "There should be no optional column in primary id &NAME."
end-model

schema-model LOG_COBOL_SCHEMA is LOG_SCHEMA
  title "Logical COBOL schema"
  concepts
    project            "database"
    schema             "view"
    entity_type        "table"
    atomic_attribute   "field"
    compound_attribute "compound field"
    identifier         "key"
    primary_identifier "primary_key"
    access_key         "key"
    user_constraint    "constraint"
  constraints
    not MIN_CARD_of_ATT (0 0)        % No optional field allowed
      diagnosis "Field &NAME should not be optional"
    COMP_per_EID (1 1)               % An identifier is made of a single
      and ATT_per_EID (1 1)          % single-valued field
      and MULT_ATT_per_EID (0 0)
      diagnosis "Id. &NAME should be made of a single single-valued field"
end-model

schema-model CONCEPT_SCHEMA
  title "Conceptual schema model"
  concepts
    project            "project"
    schema             "schema"
    entity_type        "entity type"
    rel_type           "relationship type"
    atomic_attribute   "attribute"
    compound_attribute "compound attribute"
    role               "role"
    group              "group"
    user_constraint    "constraint"
  constraints
    ET_per_SCHEMA (1 N)        % At list one ET required
      diagnosis "Schema &NAME should have an entity type"
    COLL_per_SCHEMA (0 0)      % No collection allowed
      diagnosis "The schema should have no collection"
    ATT_per_ET (1 N)           % At least one attribute per ET
      diagnosis "Entity type &NAME should have at least one attribute"
    KEY_per_ET (0 0)           % No access keys
      diagnosis "Entity type &NAME should not have an access key"
    REF_per_ET (0 0)           % No foreign key
      diagnosis "Entity type &NAME should not have a foreign key"
    ID_per_ET (1 N)            % If there are ids, one of them is primary
        and PID_per_ET (1 1)
      or ID_per_ET (0 0)
      diagnosis "One of the identifiers of entity type &NAME should be primary"
    EMBEDDED_ID_per_ET (0 0)   % Embedded identifiers are not allowed"
      diagnosis "Embedded identifiers should be removed in entity type &NAME"
    ID_DIFF_in_ET (components) % All identifiers must have different components
      diagnosis "Id. made of the same components should be avoided in &NAME"
    TOTAL_in_ISA (no)          % Total is-a relations should concern at least
      or TOTAL_in_ISA (yes)    % two subtypes
        and SUB_TYPES_per_ISA (2 N)
      diagnosis "Total is-a relations are not allowed with only one sub-type"
    DISJOINT_in_ISA (no)       % Disjoint is-a relations should concern at least
      or TOTAL_in_ISA (yes)    % two subtypes
        and SUB_TYPES_per_ISA (2 N)
      diagnosis "Disjoint is-a relations are not allowed with only one sub-type"
    ROLE_per_RT (2 2)          % 2 <=  degree of a rel-type  <= 4
      or ROLE_per_RT (3 4)     % if 3 or 4, the rel-type cannot have a one role
        and ATT_per_RT (1 N)   %  or it must also have attributes
      or ROLE_per_RT (3 4)
        and ATT_per_RT (0 0)
        and ONE_ROLE_per_RT (0 0)
      diagnosis "Rel-type &NAME has too many roles, or too few attributes"
    ID_per_RT (1 N)            % If RT have some id., one of them is primary
        and PID_per_RT (1 1)
      or ID_per_RT (0 0)
      diagnosis "One of the identifiers of rel-type &NAME should be primary"
    EMBEDDED_ID_per_RT (0 0)   % Embedded identifiers are not allowed"
      diagnosis "Embedded identifiers should be removed in rel-type &NAME"
```

```
    ID_DIFF_in_RT (components) % All identifiers must have different components
      diagnosis "Id. made of the same components should be avoided in &NAME"
    not SUB_ATT_per_ATT (1 1)  % Compound att. must have at least two components
      diagnosis "Compound attribute &NAME has too few sub-attributes"
    ID_per_ATT (0 0)           % A compound attribute cannot have an identifier
      diagnosis "Multivalued compound attribute &NAME should not have an id."
    COMP_per_GROUP (1 N)       % Every group must have at least one component
      diagnosis "Group &NAME should have components"
    ROLE_per_EID (0 0)         % An ET id. cannot be made of a single role
        and COMP_per_EID (1 N)
      or ROLE_per_EID (1 N)
        and COMP_per_EID (2 N)
      diagnosis "ET Identifier &NAME should have another component"
    MULT_ATT_per_EID (1 1)     % If an ET id. contains a multi-valued attribute
        and COMP_per_EID (1 1) % it must be the only component.
      or MULT_ATT_per_EID (0 0)
      diagnosis "ET id. &NAME should have no multi-valued attribute
                                                   or no other component"
    ONE_ROLE_per_EID (0 0)     % An entity type id. should not have a one-role
      diagnosis "One-roles should be removed from entity type identifier &NAME"
    OPT_ATT_per_EPID (0 0)     % Optional columns not allowed in primary ids.
      diagnosis "There should be no optional column in primary id &NAME."
    COMP_per_RID (1 1)         % If a rel-type id. has only one component,
        and ROLE_per_RID (0 0) % it must be an attribute
      or COMP_per_RID (2 N)
      diagnosis "Rel-type identifier &NAME should have more components"
    MULT_ATT_per_RID (1 1)     % If a RT id. contains a multi-valued attribute
        and COMP_per_RID (1 1) % it must be the only component.
      or MULT_ATT_per_RID (0 0)
      diagnosis "RT id. &NAME should have no multi-valued attribute
                                                   or no other component"
    ONE_ROLE_per_RID (0 0)     % A rel-type id. should not have a one-role
      diagnosis "One-roles should be removed from rel-type identifier &NAME"
    OPT_ATT_per_RPID (0 0)     % No optional attribute in a rel-type identifier
      diagnosis "Optional attributes should be removed from rel-type id. &NAME"
end-model

% Toolbox definitions
%%%%%%%%%%%%%%%%%%%%%

toolbox TB_ET_ID_SEARCH
  title "ET-ID search"
  description
    This process is aimed at searching for ids by analysis of the COBOL source
    code. Those that are found can be created and edited in the COBOL schema.
  end-description
  add create-group
  add modify-group
  add text-analysis
end-toolbox

toolbox TB_LONG_FIELDS_REFINEMENT
  title "Long fields refinement"
  description
    This toolbox allows you to refine long fields (several tenths of characters
    or more) which very often hide more complex structures. In the COBOL source
    codes, you may the genuine structure of some long fields. You can reproduce
    those structures in the database schema by creation of sub-attributes.
  end-description
  add create-attribute
  add modify-attribute
  add text-analysis
end-toolbox

toolbox TB_FK_SEARCH
  title "FK search"
  description
    This toolbox allows you to find out some foreign keys. This inquiry may be
    based on several methods like name analysis, field length analysis, source
    program analysis,...
  end-description
  add ref-key-search
  add create-group
  add modify-group
  add text-analysis
end-toolbox
```

```
toolbox TB_N_N_REFINEMENT
  title "N-N multiplicity refinement"
  description
    A column that has a minimum and a maximum multiplicity greater than one
    generally is the result of translation. Indeed, COBOL allows programmers to
    use arrays but with a fix number of columns only. So, a 10-10 multiplicity,
    for example, should in fact be a 0-10 or a 1-10 multiplicity. This should
    be found by analysis of the source programs.
  end-description
  add modify-attribute
  add text-analysis
end-toolbox

toolbox TB_ATT_ID_SEARCH
  title "Field-ID search"
  description
    Compound fields are sometimes an optimisation: a small table included in
    another one to increase access speed. In this case, this compound field very
    often has an identifier. This toolbox allows you to find it and to adapt the
    database schema.
  end-description
  add create-group
  add modify-group
  add text-analysis
end-toolbox

toolbox TB_RENAMING
  title "Renaming"
  description
    COBOL imposes that tables and fields be named with a reduced set of
    characters. Furthermore, programmers often use short-cuts or strange
    conventions to name tables and fields. The result is that table and field
    names can be unreadable to most people, making the schema unreadable too.
    Changing the names by using a broader character set (with spaces or accents,
    for instance) can be a good mean to improve a schema.
  end-description
  add name-processing
  add modify
  add text-analysis
  add change-prefix
end-toolbox

toolbox TB_BINARY_INTEGRATION
  title "Binary integration"
  description
    This toolbox allows you to integrate one schema into another.
  end-description
  add schema-integration
  add object-integration
end-toolbox

toolbox TB_TRANSFORM_FK
  title "Transform FK"
  description
    Foreign keys are the translation of rel-types. This toolbox allows you to
    untranslate foreign keys by transforming them back to rel-types.
  end-description
  add tf-ref-group-into-RT
end-toolbox

toolbox TB_REMOVE_FK
  title "Remove FK"
  description
    Foreign keys can be redundant. So they can be removed.
  end-description
  add delete-constraint
  add delete-group
end-toolbox

toolbox TB_EXTRACT
  title "Manual extract"
  description
    This toolbox allows you to extract data structures form COBOL files by
    upgrading existing schemas.
  end-description
  add text-analysis
  add create
```

```
  add delete
  add modify
end-toolbox

toolbox TB_ATT_INTO_ET
  title "Transform attributes into ET"
  description
    Complex attributes (compound, with identifier) are generally the result of
    an optimisation. They may be transformed back to a table.
  end-description
  add tf-att-into-ET
  add tf-disaggregate
end-toolbox

toolbox TB_ET_INTO_RT
  title "Trnasform ET into RT"
  description
    Entity types that plays only 1-1 roles and the identifier of which is made
    of all these roles looks like the translation of a complex rel-type. This
    toolbox allows you to find them and untranslate them.
  end-description
  add tf-ET-into-RT
end-toolbox

toolbox TB_ET_INTO_ATT
  title "Transform ET looking like attributes into attributes"
  description
    It can happen that an entity type seems to represent a property of another
    entity type, linked to it by a single rel-type, participating in no other
    rel-type, and partly identified by the role played by the other entity type.
    In that case, the schema may be more readable by integrating the entity type
    in the other one as an attribute, possibly compound or repetitive.
  end-description
  add tf-ET-into-att
end-toolbox

toolbox TB_RT_INTO_ISA
  title "Transform RT into is-a"
  description
    Rel-types with a generalisation/specialisation meaning can be transformed
    into more readable is-a relations
  end-description
  add tf-RT-into-isa
end-toolbox

% Process types definitions
%%%%%%%%%%%%%%%%%%%%%%%%%%%

process ENRICHMENT_EXPERT
  title "Physical schema enrichment expert"
  description
    When the database schema of a COBOL program has to be recovered, the
    DB-MAIN extractor can get all the information back from the data division
    but much more information can be found in the procedure division.
    A lot of indices can help to find that information. The expert can help
    to see them.
  end-description
  input COBOL_progs[1-N] "COBOL programs" : COBOL_FILE
  update COBOL_schema[1-1] "COBOL schema" : weak PHYS_COBOL_SCHEMA
  strategy
    if (exists(COBOL_schema,ID_per_ET(0 0))) then
      toolbox TB_ET_ID_SEARCH(COBOL_progs,COBOL_schema)
    end-if;
    if (exists(COBOL_schema,NONE_in_LIST_CI_NAMES(ID*,*ID))) then
      toolbox TB_ET_ID_SEARCH(COBOL_progs,COBOL_schema)
    end-if;
    if (ask "Are there long fields?") then
      toolbox TB_LONG_FIELDS_REFINEMENT(COBOL_progs,COBOL_schema)
    end-if;
    if (exists(COBOL_schema,REF_per_ET(0 0))) then
      toolbox TB_FK_SEARCH(COBOL_progs,COBOL_schema)
    end-if;
    if (exists(COBOL_schema,MIN_CARD_of_ATT(2 N))) then
      toolbox TB_N_N_REFINEMENT(COBOL_progs,COBOL_schema)
    end-if;
    if (exists(COBOL_schema,MAX_CARD_of_ATT(2 N)
        and SUB_ATT_per_ATT(1 N) and ID_per_ATT(0 0))) then
```

```
      toolbox TB_ATT_ID_SEARCH(COBOL_progs,COBOL_schema)
    end-if;
    toolbox TB_RENAMING(COBOL_progs,COBOL_schema)
end-process

process MANUAL_ENRICHMENT
  title "Physical schema enrichment"
  description
    Schema enrichment is the completion of an extracted COBOL schema
    with the information manually found in the procedure division of
    the COBOL files.
  end-description
  input COBOL_progs[1-1] "COBOL programs" : COBOL_FILE,
        COBOL_schema "COBOL schema" : weak PHYS_COBOL_SCHEMA
  output Complete_COBOL_schema "Complete COBOL schema" : weak PHYS_COBOL_SCHEMA
  strategy
    copy (COBOL_schema,Complete_COBOL_schema);
    some
      repeat
        do ENRICHMENT_EXPERT(COBOL_progs,Complete_COBOL_schema)
      end-repeat;
      repeat
        one
          toolbox TB_ET_ID_SEARCH(COBOL_progs,Complete_COBOL_schema);
          toolbox TB_LONG_FIELDS_REFINEMENT(COBOL_progs,Complete_COBOL_schema);
          toolbox TB_FK_SEARCH(COBOL_progs,Complete_COBOL_schema);
          toolbox TB_N_N_REFINEMENT(COBOL_progs,Complete_COBOL_schema);
          toolbox TB_ATT_ID_SEARCH(COBOL_progs,Complete_COBOL_schema);
          toolbox TB_RENAMING(COBOL_progs,Complete_COBOL_schema)
        end-one
      end-repeat
    end-some
end-process

process SCHEMA_INTEGRATION
  title "Schema integration"
  description
    The data extraction of all the COBOL files produce a set of small
    schemas (one for each COBOL file). They can all be integrated in
    a single larger schema.
  end-description
  input Physical_schema[2-N] : weak PHYS_COBOL_SCHEMA
  output integrated[1-1] : weak PHYS_COBOL_SCHEMA
  set master[1-1] "Master schema",
      secondary[1-N] "Secondary schemas",
      sec[1-1] "One secondary schema"
  strategy
    define (master,choose-one(Physical_schema));
    copy (master,integrated);
    define (secondary,minus(Physical_schema,master));
    for each sec in secondary do
      toolbox TB_BINARY_INTEGRATION(integrated,sec)
    end-for
end-process

process DATA_DIVISION_EXTRACTION
  title "COBOL data division extraction"
  description
    COBOL files data division extraction possibly with the automatic extractors
    or manually.
  end-description
  input COBOL_progs[1-N] "COBOL programs" : COBOL_FILE
  output COBOL_schema[1-N] "COBOL schemas" : PHYS_COBOL_SCHEMA
  set cobfil[1-1], cobsch[1-1]
  strategy
    for each cobfil in COBOL_progs do
      one
        extract COBOL(cobfil,COBOL_schema);
        sequence
          new(COBOL_schema);
          define(cobsch,last(COBOL_schema));
          toolbox TB_EXTRACT(cobfil,cobsch)
        end-sequence;
        sequence end-sequence
      end-one
    end-for
end-process
```

```
process SCHEMA_ENRICHMENT
  title "COBOL schema enrichment"
  description
    COBOL schema enrichment by analysis of COBOL programs.
  end-description
  input COBOL_progs[1-N] "COBOL programs" : COBOL_FILE,
        COBOL_schema[1-N] "COBOL schemas" : PHYS_COBOL_SCHEMA
  output Complete_COBOL_schemas[1-N] "Complete COBOL schemas" :
                                                              weak
PHYS_COBOL_SCHEMA
  set cobsch[1-1]
  strategy
    for each cobsch in COBOL_schema do
      do MANUAL_ENRICHMENT (COBOL_progs,cobsch,Complete_COBOL_schemas)
    end-for
end-process

process SCHEMA_EXTRACTION
  title "COBOL schema extraction"
  description
    COBOL files data structure extraction.
  end-description
  input COBOL_progs[1-N] "COBOL programs" : COBOL_FILE
  output Physical_schema[1-1] "Physical schema" : weak PHYS_COBOL_SCHEMA
  intern Raw_COBOL_schema[1-N] "Raw COBOL schemas" : PHYS_COBOL_SCHEMA,
         COBOL_schema[1-N] "COBOL schemas" : weak PHYS_COBOL_SCHEMA
  strategy
    do DATA_DIVISION_EXTRACTION(COBOL_progs,Raw_COBOL_schema);
    do SCHEMA_ENRICHMENT(COBOL_progs,Raw_COBOL_schema,COBOL_SCHEMA);
    if (count-greater(COBOL_schema,1)) then
      do SCHEMA_INTEGRATION (COBOL_schema,Physical_schema)
    else
      copy (COBOL_schema,Physical_schema)
    end-if
end-process

process SCHEMA_CLEANING
  title "COBOL schema cleaning"
  description
    Cleaning a schema by removing the physical elements in order to transform
    a physical schema into a logical one.
  end-description
  input Physical_schema[1-1] "Physical schema" : weak PHYS_COBOL_SCHEMA
  output Logical_schema[1-1] "Logical schema" : weak LOG_COBOL_SCHEMA
  strategy
    copy (Physical_schema,Logical_schema);
    glbtrsf "Remove files" log all (Logical_schema, REMOVE(ALL_COLL()));
    glbtrsf "Remove access keys" log all (Logical_schema, REMOVE_KEY);
end-process

process DEOPTIMISATION
  title "De-optimisation"
  description
    De-optimising a schema consists in removing all the constructs aimed at
    optimising database access. This can be done by removing redundant
    structures, extracting complex fields with identifier from tables,...
  end-description
  update Conceptual_schema[1-1] "Conceptual schema" : weak CONCEPT_SCHEMA
  strategy
    each
      if (exists(Conceptual_schema,ID_per_ATT(1 N))) then
        toolbox log all TB_ATT_INTO_ET(Conceptual_schema)
      end-if;
      if (exists(Conceptual_schema,TRANSITIVE_REF(yes))) then
        toolbox log all TB_REMOVE_FK(Conceptual_schema)
      end-if
    end-each
end-process

process UNTRANSLATION
  title "Untranslation"
  description
   Untranslating a schema consists in removing typical COBOL constructs, ie
   the constructs that make the difference between a COBOL schema and a
   conceptual one.
  end-description
```

```
  update Conceptual_schema[1-1] "Conceptual schema" : weak CONCEPT_SCHEMA
  strategy
    if (exists(Conceptual_schema,REF_per_ET(1 N))) then
      toolbox log all TB_TRANSFORM_FK(Conceptual_schema)
    end-if
end-process

process SCHEMA_CONCEPTUALISATION
  title "Schema conceptualisation"
  description
    Reverse engineering consists in recovering a possible conceptual database
    schema from a series of documents including source programs, DDL files,
    documentation,...
  end-description
  input Logical_schema[1-1] "Logical schema" : weak LOG_COBOL_SCHEMA
  output Conceptual_schema[1-1] "Conceptual schema" : weak CONCEPT_SCHEMA
  strategy
    copy (Logical_schema,Conceptual_schema);
    repeat
      one
        do DEOPTIMISATION(Conceptual_schema);
        do UNTRANSLATION(Conceptual_schema)
      end-one
    end-repeat
end-process

process CONCEPTUAL_NORMALISATION
  title "Conceptual normalisation"
  description
    Conceptual normalisation consists in transforming a logical schema into
    a good looking readable conceptual schema.
  end-description
  input Conceptual_schema[1-1] "Conceptual schema" : CONCEPT_SCHEMA
  output Normalised_schema[1-1] "Normalised conceptual schema" : CONCEPT_SCHEMA
  strategy
    copy (Conceptual_schema,Normalised_schema);
    toolbox log all TB_ET_INTO_RT(Normalised_schema);
    toolbox log all TB_ET_INTO_ATT(Normalised_schema);
    toolbox log all TB_RT_INTO_ISA(Normalised_schema)
end-process

process REVERSE_ENGINEERING
  title "Reverse Engineering"
  description
    Reverse engineering consists in recovering a possible conceptual database
    schema from a series of documents including source programs, DDL files,
    documentation,...
  end-description
  intern COBOL_progs[1-N] "COBOL programs" : COBOL_FILE,
         Physical_schema[1-1] "Physical schema" : weak PHYS_COBOL_SCHEMA,
         Logical_schema[1-1] "Logical schema" : weak LOG_COBOL_SCHEMA,
         Conceptual_schema[1-1] "Conceptual schema" : weak CONCEPT_SCHEMA,
         Normalised_schema[1-1] "Normalised conceptual schema" : CONCEPT_SCHEMA
  strategy
    repeat
      new (COBOL_progs)
    end-repeat;
    do SCHEMA_EXTRACTION(COBOL_progs,Physical_schema);
    do SCHEMA_CLEANING(Physical_schema,Logical_schema);
    do SCHEMA_CONCEPTUALISATION(Logical_schema,Conceptual_schema);
    do CONCEPTUAL_NORMALISATION(Conceptual_schema,Normalised_schema)
end-process

% Method definition
%%%%%%%%%%%%%%%%%%%

method
  title "Reverse-engineering"
  version "1.0"
  author "Didier ROLAND"
  date "24-10-2001"
  perform REVERSE_ENGINEERING
end-method
```

F.5. The Order.cob program analysed in the second case study

Order.cob is a COBOL program which is analysed in the second case study in order to reverse engineer the database it uses and to recover the design of this database.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. C-ORD.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
  SELECT CUSTOMER ASSIGN TO "CUSTOMER.DAT"
    ORGANIZATION IS INDEXED
    ACCESS MODE IS DYNAMIC
    RECORD KEY IS CUS-CODE.
  SELECT ORDERS ASSIGN TO "ORDER.DAT"
    ORGANIZATION IS INDEXED
    ACCESS MODE IS DYNAMIC
    RECORD KEY IS ORD-CODE
    ALTERNATE RECORD KEY IS ORD-CUSTOMER
    WITH DUPLICATES.
  SELECT STOCK ASSIGN TO "STOCK.DAT"
    ORGANIZATION IS INDEXED
    ACCESS MODE IS DYNAMIC
    RECORD KEY IS STK-CODE.

DATA DIVISION.
FILE SECTION.
FD CUSTOMER.
01 CUS.
  02 CUS-CODE PIC X(12).
  02 CUS-DESCR PIC X(80).
  02 CUS-HIST PIC X(1000).

FD ORDERS.
01 ORD.
  02 ORD-CODE PIC 9(10).
  02 ORD-CUSTOMER PIC X(12).
  02 ORD-DETAIL PIC X(200).

FD STOCK.
01 STK.
  02 STK-CODE PIC 9(5).
  02 STK-NAME PIC X(100).
  02 STK-LEVEL PIC 9(5).

WORKING-STORAGE SECTION.
01 DESCRIPTION.
  02 NAME PIC X(20).
  02 ADDR PIC X(40).
  02 FUNCT PIC X(10).
  02 REC-DATE PIC X(10).

01 LIST-PURCHASE.
  02 PURCH OCCURS 100 TIMES INDEXED BY IND.
    03 REF-PURCH-STK PIC 9(5).
    03 TOT PIC 9(5).

01 LIST-DETAIL.
  02 DETAILS OCCURS 20 TIMES INDEXED BY IND-DET.
    03 REF-DET-STK PIC 9(5).
    03 ORD-QTY PIC 9(5).

01 CHOICE PIC X.
01 END-FILE PIC 9.
01 END-DETAIL PIC 9.
01 EXIST-PROD PIC 9.
01 PROD-CODE PIC 9(5).
01 TOT-COMP PIC 9(5) COMP.
01 QTY PIC 9(5) COMP.
01 NEXT-DET PIC 99.

PROCEDURE DIVISION.

MAIN.
  PERFORM INIT.
  PERFORM PROCESS UNTIL CHOICE = 0.
```

```
  PERFORM CLOSING.
  STOP RUN.

INIT.
  OPEN I-O CUSTOMER.
  OPEN I-O ORDERS.
  OPEN I-O STOCK.

PROCESS.
  DISPLAY "1 NEW CUSTOMER".
  DISPLAY "2 NEW STOCK".
  DISPLAY "3 NEW ORDER".
  DISPLAY "4 LIST OF CUSTOMERS".
  DISPLAY "5 LIST OF STOCKS".
  DISPLAY "6 LIST OF ORDERS".
  DISPLAY "0 END".
  ACCEPT CHOICE.
  IF CHOICE = 1
    PERFORM NEW-CUS.
  IF CHOICE = 2
    PERFORM NEW-STK.
  IF CHOICE = 3
    PERFORM NEW-ORD.
  IF CHOICE = 4
    PERFORM LIST-CUS.
  IF CHOICE = 5
    PERFORM LIST-STK.
  IF CHOICE = 6
    PERFORM LIST-ORD.

CLOSING.
  CLOSE CUSTOMER.
  CLOSE ORDERS.
  CLOSE STOCK.

NEW-CUS.
  DISPLAY "NEW CUSTOMER".
  DISPLAY "CUSTOMER CODE: "
    WITH NO ADVANCING.
  ACCEPT CUS-CODE.
  DISPLAY "NAME OF CUSTOMER: "
    WITH NO ADVANCING.
  ACCEPT NAME.
  DISPLAY "ADDRESS OF CUSTOMER: "
    WITH NO ADVANCING.
  ACCEPT ADDR.
  DISPLAY "FUNCTION OF CUSTOMER: "
    WITH NO ADVANCING.
  ACCEPT FUNCT.
  DISPLAY "DATE: "
    WITH NO ADVANCING.
  ACCEPT REC-DATE.
  MOVE DESCRIPTION TO CUS-DESCR.
  PERFORM INIT-HIST.
  WRITE CUS
    INVALID KEY DISPLAY "ERROR".

LIST-CUS.
  DISPLAY "LIST OF CUSTOMERS".
  CLOSE CUSTOMER.
  OPEN I-O CUSTOMER.
  MOVE 1 TO END-FILE.
  PERFORM READ-CUS UNTIL (END-FILE = 0).

READ-CUS.
  READ CUSTOMER NEXT
    AT END MOVE 0 TO END-FILE
    NOT AT END
      DISPLAY CUS-CODE
      DISPLAY CUS-DESCR
      DISPLAY CUS-HIST.

NEW-STK.
  DISPLAY "NEW STOCK".
  DISPLAY "PRODUCT NUMBER: "
    WITH NO ADVANCING.
  ACCEPT STK-CODE.
```

```
  DISPLAY "NAME: "
    WITH NO ADVANCING.
  ACCEPT STK-NAME.
  DISPLAY "LEVEL: "
    WITH NO ADVANCING.
  ACCEPT STK-LEVEL.
  WRITE STK
    INVALID KEY DISPLAY "ERROR".

LIST-STK.
  DISPLAY "LIST OF STOCKS".
  CLOSE STOCK.
  OPEN I-O STOCK.
  MOVE 1 TO END-FILE.
  PERFORM READ-STK UNTIL END-FILE = 0.

READ-STK.
  READ STOCK NEXT
    AT END MOVE 0 TO END-FILE
    NOT AT END
      DISPLAY STK-CODE
      DISPLAY STK-NAME
      DISPLAY STK-LEVEL.

NEW-ORD.
  DISPLAY "NEW ORDER".
  DISPLAY "ORDER NUMBER: "
    WITH NO ADVANCING.
  ACCEPT ORD-CODE.
  MOVE 1 TO END-FILE.
  PERFORM READ-CUS-CODE UNTIL END-FILE = 0.
  MOVE CUS-DESCR TO DESCRIPTION.
  DISPLAY NAME.
  MOVE CUS-CODE TO ORD-CUSTOMER.
  MOVE CUS-HIST TO LIST-PURCHASE.
  SET IND-DET TO 1.
  MOVE 1 TO END-FILE.
  PERFORM READ-DETAIL
    UNTIL END-FILE = 0 OR IND-DET = 21.
  MOVE LIST-DETAIL TO ORD-DETAIL.
  WRITE ORD
    INVALID KEY DISPLAY "ERROR".
  MOVE LIST-PURCHASE
    TO CUS-HIST.
  REWRITE CUS
    INVALID KEY DISPLAY "ERROR CUS".

READ-CUS-CODE.
  DISPLAY "CUSTOMER NUMBER: "
    WITH NO ADVANCING.
  ACCEPT CUS-CODE.
  MOVE 0 TO END-FILE.
  READ CUSTOMER INVALID KEY
    DISPLAY "NO SUCH CUSTOMER"
    MOVE 1 TO END-FILE
  END-READ.

READ-DETAIL.
  DISPLAY "PRODUCT CODE (0 = END): ".
  ACCEPT PROD-CODE.
  IF PROD-CODE = 0
    MOVE 0
      TO REF-DET-STK(IND-DET)
    MOVE 0 TO END-FILE
  ELSE
    PERFORM READ-PROD-CODE.

READ-PROD-CODE.
  MOVE 1 TO EXIST-PROD.
  MOVE PROD-CODE TO STK-CODE.
  READ STOCK INVALID KEY
    MOVE 0 TO EXIST-PROD.
  IF EXIST-PROD = 0
    DISPLAY "NO SUCH PRODUCT"
  ELSE
    PERFORM UPDATE-ORD-DETAIL.
```

```
UPDATE-ORD-DETAIL.
  MOVE 1 TO NEXT-DET.
  DISPLAY "QUANTITY ORDERED: "
    WITH NO ADVANCING
  ACCEPT ORD-QTY(IND-DET).
  PERFORM UNTIL
    (NEXT-DET < IND-DET
      AND REF-DET-STK(NEXT-DET) = PROD-CODE)
    OR IND-DET = NEXT-DET
    ADD 1 TO NEXT-DET
  END-PERFORM.
  IF IND-DET = NEXT-DET
    MOVE PROD-CODE
    TO REF-DET-STK(IND-DET)
    PERFORM UPDATE-CUS-HIST
    SET IND-DET UP BY 1
  ELSE
    DISPLAY "ERROR: ALREADY ORDERED".

UPDATE-CUS-HIST.
  SET IND TO 1.
  PERFORM UNTIL
    REF-PURCH-STK(IND) = PROD-CODE
    OR REF-PURCH-STK(IND) = 0
    OR IND = 101
      SET IND UP BY 1
  END-PERFORM.
  IF IND = 101
    DISPLAY "ERROR: HISTORY OVERFLOW"
    EXIT.
  IF REF-PURCH-STK(IND)
      = PROD-CODE
    ADD ORD-QTY(IND-DET) TO TOT(IND)
  ELSE
    MOVE PROD-CODE
    TO REF-PURCH-STK(IND)
    MOVE ORD-QTY(IND-DET) TO TOT(IND).

LIST-ORD.
  DISPLAY "LIST OF ORDERS".
  CLOSE ORDERS.
  OPEN I-O ORDERS.
  MOVE 1 TO END-FILE.
  PERFORM READ-ORD UNTIL END-FILE = 0.

READ-ORD.
  READ ORDERS NEXT
    AT END MOVE 0 TO END-FILE
    NOT AT END
      DISPLAY "ORD-CODE "
        WITH NO ADVANCING
      DISPLAY ORD-CODE
      DISPLAY "ORD-CUSTOMER "
        WITH NO ADVANCING
      DISPLAY ORD-CUSTOMER
      DISPLAY "ORD-DETAIL "
      MOVE ORD-DETAIL TO LIST-DETAIL
      SET IND-DET TO 1
      MOVE 1 TO END-DETAIL
      PERFORM DISPLAY-DETAIL.

INIT-HIST.
  SET IND TO 1.
  PERFORM UNTIL IND = 100
    MOVE 0 TO REF-PURCH-STK(IND)
    MOVE 0 TO TOT(IND)
    SET IND UP BY 1
  END-PERFORM.
  MOVE LIST-PURCHASE TO CUS-HIST.

DISPLAY-DETAIL.
  IF IND-DET = 21
    MOVE 0 TO END-DETAIL
    EXIT.
  IF REF-DET-STK(IND-DET) = 0
    MOVE 0 TO END-DETAIL
  ELSE
```

```
    DISPLAY REF-DET-STK(IND-DET)
    DISPLAY ORD-QTY(IND-DET)
    SET IND-DET UP BY 1.
```

F.6. A small C program to clean log files

```c
#include <stdio.h>
#include <string.h>
#define SIZE 1000

void clean_sch_ent_rel(FILE* fin, FILE* fout, char* s, char** res)
{
  char lines[16][SIZE];
  int i;

  for (i=0; *res && i<16; i++) {
    strcpy(lines[i],s);
    *res = fgets(s,SIZE,fin);
  }
  if (strcmp(lines[4]+1,lines[10])
      || strcmp(lines[5]+1,lines[11])
      || strcmp(lines[6]+1,lines[14]))
    for (i=0; i<16; i++)
      if (fputs(lines[i],fout)==EOF) {
        *res = NULL;
        return;
      }
}

void clean_rol(FILE* fin, FILE* fout, char* s, char** res)
{
  char lines[16][SIZE];
  int i;

  for (i=0; *res && i<16; i++) {
    strcpy(lines[i],s);
    *res = fgets(s,SIZE,fin);
  }
  if (strcmp(lines[4]+1,lines[10])
      || strcmp(lines[5]+1,lines[13])
      || strcmp(lines[6]+1,lines[14]))
    for (i=0; i<16; i++)
      if (fputs(lines[i],fout)==EOF) {
        *res = NULL;
        return;
      }
}

void copy(FILE* fin, FILE* fout, char* s, char** res)
{
  do {
    if (fputs(s,fout)==EOF) {
      *res = NULL;
      return;
    }
    *res = fgets(s,SIZE,fin);
  } while (*res && *s!='*');
}

void clean(FILE* fin, FILE* fout)
{
  char s[SIZE];
  char* res;

  res = fgets(s,SIZE,fin);
  while (res==s) {
    if (!strcmp(s,"*MOD SCH\n"))
      clean_sch_ent_rel(fin,fout,s,&res);
    else if (!strcmp(s,"*MOD ENT\n"))
      clean_sch_ent_rel(fin,fout,s,&res);
    else if (!strcmp(s,"*MOD REL\n"))
      clean_sch_ent_rel(fin,fout,s,&res);
    else if (!strcmp(s,"*MOD ROL\n"))
      clean_rol(fin,fout,s,&res);
    else
```

```
      copy(fin,fout,s,&res);
  }
}

void main(int c, char**s)
{
  FILE* fin;
  FILE* fout;

  if (c!=3)
    return;
  fin = fopen(s[1],"rt");
  if (!fin)
    return;
  fout = fopen(s[2],"wt");
  if (!fout) {
    fclose(fin);
    return;
  }
  clean(fin,fout);
  fclose(fin);
  fclose(fout);
}
```