KATHOLIEKE UNIVERSITEIT LEUVEN
FACULTEIT WETENSCHAPPEN
FACULTEIT INGENIEURSWETENSCHAPPEN
DEPARTEMENT COMPUTERWETENSCHAPPEN
AFDELING INFORMATICA
Celestijnenlaan 200A - B-3001 Leuven

A CONSTRAINT-CENTRIC APPROACH FOR OBJECT-ORIENTED CONCEPTUAL MODELLING

Promotor:
Prof. dr. ir. Eric Steegmans

Proefschrift voorgedragen tot
het behalen van het doctoraat
in de wetenschappen: informatica
door
Stefan VAN BAELEN

Mei 2007

KATHOLIEKE UNIVERSITEIT LEUVEN
FACULTEIT WETENSCHAPPEN
FACULTEIT INGENIEURSWETENSCHAPPEN
DEPARTEMENT COMPUTERWETENSCHAPPEN
AFDELING INFORMATICA
Celestijnenlaan 200A - B-3001 Leuven

A CONSTRAINT-CENTRIC APPROACH FOR OBJECT-ORIENTED CONCEPTUAL MODELLING

Jury:
Prof. dr. Guido Dedene, voorzitter
Prof. dr. ir. Eric Steegmans, promotor
Prof. dr. ir. Wouter Joosen
Prof. dr. Tom Holvoet
Prof. dr. Hans-Gerhard Gross (TU Delft, Nederland)

Proefschrift voorgedragen tot
het behalen van het doctoraat
in de wetenschappen: informatica
door
Stefan VAN BAELEN

U.D.C. 681.3*D.2.1, D.2.4

Mei 2007

© Katholieke Universiteit Leuven, Faculteit Wetenschappen & Ingenieurswetenschappen
Arenbergkasteel, B-3001 Leuven, Belgium

Alle rechten voorbehouden. Niets uit deze uitgave mag worden vermenigvuldigd en/of openbaar gemaakt worden door middel van druk, fotokopie, microfilm, elektronisch of op welke andere wijze ook zonder voorafgaande schriftelijke toestemming van de uitgever.

All rights reserved. No part of the publication may be reproduced in any form by print, photoprint, microfilm, or any other means without written permission from the publisher.

D/2007/7515/54
ISBN 978-90-5682-820-2

Abstract

Object-oriented analysis, and more specifically conceptual modelling, is a software engineering activity that aims at studying, analysing, and capturing the knowledge about the universe of discourse for a system to be developed. This should result in the specification of a consistent and unambiguous model that describes all domain knowledge, facts, and rules, in which every element from the universe of discourse has a transparent one-to-one correspondence to an entity in the conceptual model.

We propose in this dissertation a constraint-centric approach towards object-oriented conceptual modelling. This is achieved by the usage of high-level constraint specifications as the core model structure for conceptual modelling. In particular, our approach enriches the conceptual model structure on two levels: by the definition of new structural concepts to express model constraints implicitly in the model structure, and by the introduction of constraints with supporting resolution mechanisms as a first-class model concept.

Concerning the definition of structural concepts, we developed new concepts with a dedicated applicability context attached in order to specify constraints implicitly in the model structure. The incorporation of model constraints in each methodological concept, the usage of existential dependency as the key modelling criterion, the introduction of explicit class archives, and the formal specification of model events and queries enrich the expressive power of a conceptual model structure.

Concerning the introduction of constraints as a first-class model concept, we developed a mechanism to specify model constraints using many-sorted first order logic. The constraint trigger concept attached to a constraint defines a generic constraint solver that can resolve constraint violations by injecting additional behaviour into an event or by firing an event due to progress of time.

Our approach has converged into the EROOS methodology of which two versions are proposed. A core version, the EROOS kernel, uses a constructional modelling approach in which information can only be added to a conceptual model instance. An extended version, the EROOS universe, provides additional support for recurrent EROOS kernel analysis patterns through advanced and more practical concepts using the core version as the underlying base.

Acknowledgments

During the process of performing and finishing my dissertation, I have received enormous support from a lot of colleagues, friends, family members, and loved ones, without whom this dissertation would never have been written.

First, I would especially like to thank my promoter Prof. Dr. ir. Eric Steegmans who provided me the drive to perform this research. I cannot thank him enough for all the time we have spent in numerous interesting discussions but also often heavy religious wars. Also a big word of thanks goes to Prof. Dr. ir. Johan Lewi, the head of the Software Development Methodology (SOM) research group at the time I started my research. He is the godfather of my research, and gave me the opportunity to perform this work in the first place. Together with Eric, he has been all these years a continuous source of lots of interesting remarks concerning my work.

My thanks also go to the members of the jury, Prof. Dr. Guido Dedene, Prof. Dr. ir. Wouter Joosen, Prof. Dr. Tom Holvoet, and Prof. Dr. Hans-Gerhard Gross, for taking the time to read this text and providing useful comments that helped to improve the quality of this text.

I also want to thank the Research Foundation of Flanders (FWO Vlaanderen) for providing me a doctoral research grant (aspirant NFWO) for 4 years that enabled me to execute this research.

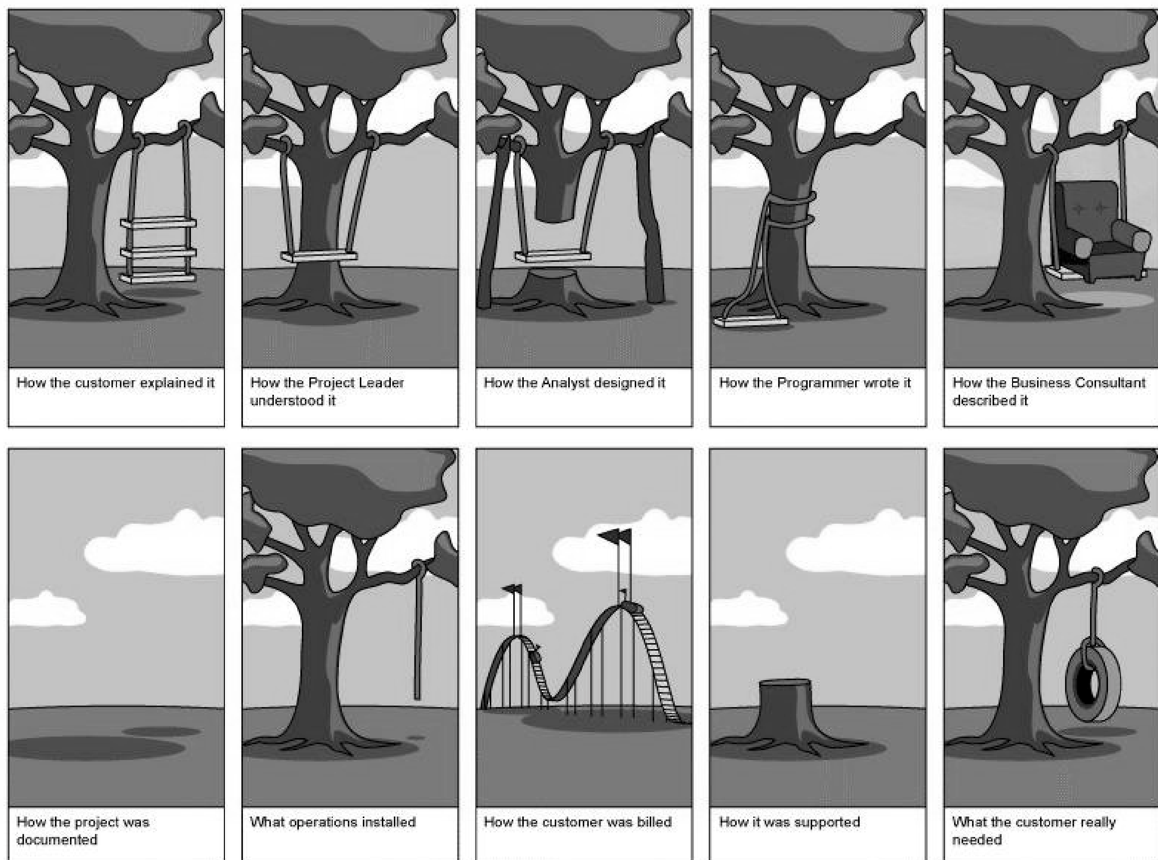
Of course, I specifically want to thank all members of the Software Development Methodology research group that came and go within the time I performed my research on this topic. Although I am probably going to forget to acknowledge someone, I especially want to mention Walid Al-Ahmad, Pieter Bekaert, Sam De Backer, Geert Delanote, Frank Devos, Maarten D’Haese, Jan Dockx, David Jehoul, Elke Malliet, Jamal Said, Bart Swennen, Eric Van Gestel, Peter Van Hirtum, and Helena Van Riel. They were excellent colleagues that always provided a critical platform to test new ideas and helped to identify potential contradictions and inconsistencies.

I also want to thank all people with whom I had a fruitful collaboration during the past and running European ITEA research projects. From K.U.Leuven, I especially want to thank Prof. Dr. ir. Karel De Vlaminck, Prof. Dr. ir. Yolande Berbers, Yvan Barbaix, Joris Gorinsek, Aram Hovsepyan, David Urting, Bert Vanhooff, and Andrew Wils. From Barco, I want to thank Lieven Demeestere, Andy De Mets, Jean-Christophe Monfret, Stijn Rammeloo, and Hans Van Genechten for our fruitful collaboration. From Jabil Circuit, formerly Philips Hasselt, I want to thank Linde Loomans. From E2S, I want to thank Michel Huybrechts en Guy Pauwels. I also want to thank all project partners with whom I came in contact with during the last years for providing me a rich scale of various perspectives on software and system engineering, and helping me to broaden the viewpoint from my originally rather theoretical ivory tower.

Last but not least, I want to thank my family and friends for their continuous encouragements, specifically my mother Godelieve and my sister Nina, my fellow students Mark, Antoon, Bruno, Frank, and Joris, my friends of the gymnastics association GymFed, and my friends of the board and society games club Speelduivel - De Leuvense Spellenclub. But utmost appreciation goes to my dearest spouse Marion Meeusen and wonderful daughter Zanna Van Baelen without whom my life would never have been such incredibly fantastic adventurous journey as today.

Modeling is hard -
Modeling skills are gained over years of experience
and only when a developer chooses to gain them

Scott W. Ambler
The Object Primer:
Agile Model-Driven Development
with UML2.0



Original author of cartoon unknown
(source: http://www.jroller.com/resources/behrangsa/software-project.jpg)

To Marion and Zanna

Contents

Abstract iii

Acknowledgments v

Contents xi

List of Figures xix

List of Tables and Scripts xxi

List of Definitions xxiii

Chapter 1 Introduction 1

1.1 Background 2
1.1.1 The Software Life Cycle 2
1.1.2 Object-Oriented Analysis 4
1.1.2.1 Scope and Goal of the Analysis phase 4
1.1.2.2 The Necessity of Analysis 6
1.1.2.3 Distinction between Analysis and Design Concerns 6
1.1.2.4 Classification of Analysis Views 8
1.1.2.4.1 Structural Analysis Methods 8
1.1.2.4.2 Behavioural Analysis Methods 8
1.1.2.4.3 UML as the Catalyst of Analysis Views 9
1.1.3 Problems and Open Issues for Object-Oriented Analysis 10
1.1.3.1 Modelling Notation 10
1.1.3.2 Model Consistency 12
1.1.3.3 Model Informality 13
1.1.3.4 Methodological Support 15

1.1.3.5 Analysis Demarcation and Further Transition 16

1.2 Goals 17

1.3 Contributions 18

1.4 Overview 19

Chapter 2 A Taxonomy for Model Constraint Formalisms in Object-Oriented Analysis 21

2.1 The Role of Model Constraints in Object-Oriented Analysis 21

2.2 Model Constraints versus Derivation Rules 23
2.2.1 Model Constraints in Object-Oriented Analysis 23
2.2.2 Constraint Logic Programming 23
2.2.3 Database Constraints 24
2.2.4 ECAA Rules in Active Databases 24

2.3 Example of the Library System 26

2.4 Specification of Model Constraints using Informal Text 26
2.4.1 Constraints using Informal Text for the Library Example 27
2.4.2 Evaluation of Constraints using Informal Text 28

2.5 Specification of Model Constraints using Operational Restrictions 30
2.5.1 Constraints using Operational Restrictions for the Library Example 30
2.5.1.1 Sequence Diagrams 31
2.5.1.2 Statechart Diagrams 33
2.5.1.3 Preconditions 34
2.5.2 Evaluation of Constraints using Operational Restrictions 34

2.6 Model Constraints as a First-Class Model Concept 37
2.6.1 Constraints as a First-Class Model Concept for the Library Example 38
2.6.2 Evaluation of Constraints as a First-Class Model Concept 39

2.7 Integration of Model Constraints in Existing Model Concepts 41
2.7.1 Integrated Model Constraints for the Library Example 42
2.7.2 Evaluation of Integrated Model Constraints 43

2.8 Model Constraints Implied by the Model Structure 44
2.8.1 Model-Implied Constraints for the Library Example 47
2.8.2 Evaluation of Model-Implied Constraints 48

2.9 Comparison and Conclusions 50

Chapter 3 Key Principles for Conceptual Modelling 53

3.1 Principle of Uniqueness 53

3.2 Principle of No Redundancy 54

3.3 Principle of Unambiguity 55

3.4 Principle of Completeness 55

3.5 Principle of Minimalism 56

3.6 Principle of Preciseness 56

3.7 Principle of No History 57

3.8 Principle of Model-Implied Constraints 57

3.9 Principle of Abstraction 58

3.10 Additional Considerations 58
3.10.1 Extendibility in Conceptual Modelling 58
3.10.2 Correctness in Conceptual Modelling 59

Chapter 4 A Methodological Kernel for Conceptual Modelling 61

4.1 Model, Model Instance, and Event Instance 62

4.2 Classes, Objects, and Static Classification 63
4.2.1 The Population of a Class 64
4.2.2 Model Constraints implied by the Class Concept 65
4.2.3 Specification of an EROOS Class 66
4.2.4 EROOS Kernel Classes for the Library Example 68
4.2.5 Contributions, Related Work, and Reflections 68

4.3 Attributes, Domains, Values, and Decoration 69
4.3.1 Value Domains 70
4.3.2 Attribute Values 72
4.3.3 Model Constraints implied by the Attribute Concept 73
4.3.4 Specification of an EROOS Attribute 74
4.3.5 Default Attributes 77
4.3.6 Implicit Attribute Queries 78
4.3.7 EROOS Attributes for the Library Example 78
4.3.8 Contributions, Related Work, and Reflections 79

4.4 Relations, Links, and Refinement 81
4.4.1 EROOS Relations and Object Links 81
4.4.2 Model Constraints implied by the Relation Concept 84

4.4.3 Specification of an EROOS Relation 85
4.4.4 Implicit Refinement and Participation Queries 87
4.4.5 Integrated Relationship Constraints on Connectivity 89
4.4.6 Integrated Relationship Constraints on Multiplicity 90
4.4.7 EROOS Relations for the Library Example 91
4.4.8 Contributions, Related Work, and Reflections 93

4.5 EROOS Constraints and Confinement 94
4.5.1 General Principles of Confinement in EROOS 95
4.5.2 Specification of an EROOS Constraint 98
4.5.3 Restrictions on EROOS Join Constraints 101
4.5.4 EROOS Constraints for the Library Example 102
4.5.5 Contributions, Related Work, and Reflections 104

4.6 Is-A Specialisations and Static Subdivision 107
4.6.1 Is-A Specialisation versus Subclassing versus Subtyping 108
4.6.2 Specialisation Partitions and Multiple Generalisations 109
4.6.3 Specification of an EROOS Specialisation 111
4.6.4 Model Constraints implied by the Specialisation Concept 112
4.6.5 Strengthening Constraints for a Specialisation 113
4.6.6 Causal Dependency for Specialisations 116
4.6.7 Implicit Specialisation Queries 117
4.6.8 EROOS Specialisations for the Library Example 117
4.6.9 Contributions, Related Work, and Reflections 119

4.7 Queries and Ornamentation 119
4.7.1 Specification of an EROOS Query 119
4.7.2 Examples of EROOS Queries 121
4.7.3 EROOS Queries for the Library Example 121
4.7.4 Contributions, Related Work, and Reflections 123

4.8 Events and Enrichment 123
4.8.1 Events in an EROOS Model 123
4.8.2 Specification of an EROOS Event 124
4.8.3 EROOS Events for the Library Example 125
4.8.4 Contributions, Related Work, and Reflections 126

4.9 Design Issues concerning Model Constraints 127
4.9.1 Design Issues for Model-Implied Constraints 127
4.9.2 Design Issues for First-Class Model Constraints 128

4.10 Evaluation of the EROOS Kernel 131
4.10.1 Achieving Uniqueness 131
4.10.2 Achieving No Redundancy 132
4.10.3 Achieving Unambiguity 133
4.10.4 Achieving Completeness 134
4.10.5 Achieving Minimalism 134

4.10.6 Achieving Preciseness 135
4.10.7 Achieving No History 135
4.10.8 Achieving Model-Implied Constraints 135
4.10.9 Achieving Abstraction 136

Chapter 5 Advanced Concepts for Conceptual Modelling 137

5.1 Class Archives and Object Destruction 137
5.1.1 EROOS Kernel Analysis Pattern for Class Archives 137
5.1.2 The Class Archive 138
5.1.3 Attributes for the Class Archive 140
5.1.3.1 Default Attributes for the Class Archive 141
5.1.4 Queries on the Class Archive 142
5.1.5 Class Archive as Relation Participant 143
5.1.6 EROOS Relations for the Library Example Revisited 149
5.1.7 Contributions, Related Work, and Reflections 150

5.2 Mutability of Attribute Values and Relation Participants 151
5.2.1 EROOS Kernel Analysis Pattern for Mutability 151
5.2.2 Specification of a Mutable EROOS Attribute 152
5.2.3 Specification of a Mutable EROOS Relation Participant 154
5.2.4 Attribute and Relation Mutation Events 155
5.2.5 Implicit Attribute, Refinement, and Participation Queries 157
5.2.6 EROOS Mutability for the Library Example 158
5.2.7 Contributions, Related Work, and Reflections 159

5.3 Compounds and Mutual Dependency 159
5.3.1 EROOS Compounds and Object Compound Links 160
5.3.2 Model Constraints implied by the Compound Concept 161
5.3.3 Integrated Compound Constraints on Connectivity 162
5.3.4 Integrated Compound Constraints on Mutability 162
5.3.5 Specification of an EROOS Compound 163
5.3.6 Implicit Compound Queries 165
5.3.7 Compound Mutation Events 166
5.3.8 Class Archive as a Compound Participant 167
5.3.9 EROOS Compounds for the Library Example 170
5.3.10 Contributions, Related Work, and Reflections 171

5.4 EROOS Constraint Triggers 172
5.4.1 Semantics of Functionality in EROOS 173
5.4.2 The Trigger Concept 174
5.4.2.1 Triggers and Model Validity 174
5.4.2.2 Addition Triggers versus Adaptation Triggers 175
5.4.2.3 Multiple Trigger Violations 176
5.4.2.4 Trigger Chains 177
5.4.2.5 Event Triggers versus Time Triggers 177
5.4.2.6 Time Triggers and Object Creation 178

5.4.3 Extending EROOS Model Concepts with Trigger Specifications 179
5.4.3.1 Trigger Specification for EROOS Constraints 179
5.4.3.2 Trigger Specification for Integrated Model Constraints..........................181
5.4.3.3 Trigger Specification for Implicit Model Constraints 182
5.4.4 Techniques for Describing the Overall Model Behaviour..........................183
5.4.4.1 Central Effect Descriptions..184
5.4.4.2 Modular Effect Descriptions..185
5.4.4.3 Distributed Effect Descriptions using Triggers 187
5.4.5 Using Nondeterminism in Functionality Specifications.............................189
5.4.6 EROOS Constraint Triggers for the Library Example 191
5.4.7 Contributions, Related Work, and Reflections 191

5.5 Derived Groups and Dynamic Subdivision 193
5.5.1 EROOS Analysis Pattern for Dynamic Specialisation 193
5.5.2 EROOS Groups and Dynamic Specialisation...196
5.5.3 EROOS Groups for the Library Example...198
5.5.4 Contributions, Related Work, and Reflections 199

5.6 Evaluation of the EROOS Universe 200
5.6.1 Achieving Uniqueness 201
5.6.2 Achieving No Redundancy 202
5.6.3 Achieving Model-Implied Constraints 203
5.6.4 Final Reflections 203

Chapter 6 Conclusions...205

6.1 Summary and Contributions 205
6.1.1 Advanced Methodological Concepts 206
6.1.2 Model Constraints implied by the EROOS Model Structure 207
6.1.3 Model Constraints as a First-Class Model Concept 208
6.1.4 Value Added for Model-Driven Development...209

6.2 Validation 210

6.3 Directions for Future Work 212

Bibliography...215

List of Publications 229

Biography 235

Summary in Dutch / Nederlandstalige samenvatting i

1 Inleiding ii
1.1 Problemen en vraagstukken betreffende objectgeoriënteerde analyse ii
1.2 Doelstellingen iv
1.3 Bijdragen..iv

2 Een taxonomie voor modelbeperkingsformalismen in objectgeoriënteerde analyse...v

3 Kernprincipes voor conceptuele modellering.. vii

4 Een methodologische kern voor conceptuele modellering viii
4.1 Klassen, objecten en statische classificatie...ix
4.2 Attributen, domeinen, waarden en decoratie ix
4.3 Relaties, verbanden en verfijning x
4.4 EROOS beperkingen en restrictie..x
4.5 *Is-A* specialisaties en statische onderverdeling..xi
4.6 Query's en ornamentatie xi
4.7 Gebeurtenissen en verrijking xii

5 Geavanceerde concepten voor conceptuele modellering.................................... xii
5.1 Klassenarchief en objectvernietiging... xii
5.2 Mutabiliteit van attribuutwaarden en relatieparticipanten.............................. xiii
5.3 Composieten en wederzijdse afhankelijkheid.. xiii
5.4 EROOS beperkingsreacties... xiii
5.5 Afleidbare groepen en dynamische onderverdeling xiv

6 Conclusies xiv
6.1 Toegevoegde waarde voor modelgedreven ontwikkeling xv
6.2 Validatie...xvi
6.3 Verder onderzoek.. xvii

List of Figures

Figure 2.1: A Basic UML Model for the Library System 27
Figure 2.2: Sequence Diagram for the *Fine* Constraint Realisation 31
Figure 2.3: Sequence Diagram for *Fine* introducing an Explicit Timer Object 32
Figure 2.4: Sequence Diagram for the *receive-check-accept* Constraint Realisation 33
Figure 2.5: State Diagram for the *receive-check-accept* Constraint Realisation 34
Figure 2.6: UML Model with Multiplicity Constraints for the Library System 42
Figure 2.7: Alternatives in UML for Modelling Associations 46
Figure 2.8: A Hierarchical Model for the Library System 47
Figure 4.1: Objects in an EROOS Class Population at Moment t 65
Figure 4.2: Graphical Representation of an EROOS Class 67
Figure 4.3: Objects decorated by an Attribute of Domain D at Moment t 73
Figure 4.4: Graphical Representation of an EROOS Attribute 74
Figure 4.5: Objects refined by a Binary and Unary Relation at Moment t 82
Figure 4.6: Decomposition of an n-ary Relation into Binary Relations 83
Figure 4.7: Graphical Representation of a Unary and Binary EROOS Relation 85
Figure 4.8: Graphical Representation of EROOS Multiplicity Constraints 90
Figure 4.9: EROOS Kernel Relations for the Library System 91
Figure 4.10: EROOS Constraint Specification from the Viewpoint of the Top Class 97
Figure 4.11: Graphical Representation of an EROOS Constraint 98
Figure 4.12: Forbidden EROOS Join Constraints 102
Figure 4.13: Allowed EROOS Join Constraint 102
Figure 4.14: Graphical Representation of an EROOS Specialisation 109
Figure 4.15: EROOS Specialisation Partitions 110
Figure 4.16: Strengthening a Participant for an EROOS Specialisation 114
Figure 4.17: Changing the Relation Arity for an EROOS Specialisation 115
Figure 4.18: Forbidden Causal Dependency between EROOS Specialisations 116
Figure 4.19: Forbidden Causal Model Dependency for an EROOS Specialisation 117
Figure 4.20: EROOS Specialisation for the Library System 118
Figure 4.21: Graphical Representation of an EROOS Query 120
Figure 4.22: Graphical Representation of an EROOS Event 125
Figure 5.1: EROOS Kernel Analysis Pattern for an Activity 138
Figure 5.2: Present and Past Population set for an EROOS Universe Class 139

Figure 5.3: Graphical Representation of an EROOS Archive Attribute 141
Figure 5.4: Graphical Representation of an EROOS Archive Query.............................142
Figure 5.5: Participation Types for an EROOS Class 144
Figure 5.6: EROOS Archive Participation Constraints 146
Figure 5.7: EROOS Relations for the Library System Revisited 150
Figure 5.8: EROOS Analysis Pattern for a Mutable Attribute 152
Figure 5.9: Graphical Representation of a Mutable EROOS Attribute..........................152
Figure 5.10: Graphical Representation of a Mutable EROOS Participant......................154
Figure 5.11: Graphical Representation of a Mutation Event..157
Figure 5.12: EROOS Mutability for the Library System..159
Figure 5.13: Objects involved in an EROOS Compound at Moment t 161
Figure 5.14: EROOS Compound and Alternative Constructs 161
Figure 5.15: Graphical Representation of a Mutable EROOS Compound......................163
Figure 5.16: EROOS Compound Participation Types..168
Figure 5.17: EROOS Compounds for the Library System 171
Figure 5.18: EROOS Analysis Pattern for a Time Trigger Creating Objects 179
Figure 5.19: Techniques for Describing the Overall Model Behaviour 183
Figure 5.20: Modelling Dynamic Specialisation using EROOS Queries........................194
Figure 5.21: EROOS Analysis Pattern for Dynamic Specialisation...............................195
Figure 5.22: Graphical Representation of an EROOS Group...196
Figure 5.23: Using EROOS Groups for Dynamic Specialisation...................................198
Figure 5.24: EROOS Groups for the Library System...199

List of Tables and Scripts

Table 2.1: OCL Specifications for Constraint Realisation using Preconditions 35
Table 2.2: OCL Specifications for Constraints as First-Class Model Concept 39
Table 2.3: Criteria in UML when Modelling Associations 45
Table 2.4: Overview of Specification Techniques for Model Constraints 51
Table 4.1: EROOS Class Script 67
Table 4.2: EROOS Class Script for the Library Example 68
Table 4.3: EROOS Attribute Script 75
Table 4.4: Implicit EROOS Script for the Default Attribute *Creation Timestamp* 77
Table 4.5: EROOS Attribute Script for the Library Example 79
Table 4.6: EROOS Kernel Relation Script 86
Table 4.7: EROOS Constraint Script..99
Table 4.8: EROOS Constraint for No Printing when in Liquidation 104
Table 4.9: EROOS Constraint for Single Copy Borrowing...104
Table 4.10: EROOS Specialisation Script 111
Table 4.11: EROOS Query Script...120
Table 4.12: Example of an EROOS Query Script for an Object Age 121
Table 4.13: Example of an EROOS Query Script for a Dual Participation Check 121
Table 4.14: EROOS Query for the Number of Borrowings 122
Table 4.15: EROOS Query for the Amount of the Fine 122
Table 4.16: EROOS Event Script 124
Table 4.17: EROOS Event of Deregistration for the Library Example...........................126
Table 5.1: EROOS Constraint for an Activity...138
Table 5.2: EROOS Universe Class Script 139
Table 5.3: EROOS Archive Attribute Script 141
Table 5.4: Implicit EROOS Script for the Default Attribute *Destruction Timestamp* 142
Table 5.5: EROOS Archive Query Script..143
Table 5.6: Alternatives for a Relation with Participation Restriction 147
Table 5.7: EROOS Universe Relation Script...148
Table 5.8: EROOS Universe Attribute Script..153
Table 5.9: EROOS Mutation Event Script...156
Table 5.10: Event Expression in an EROOS Event Script 157
Table 5.11: EROOS Compound-Part Script 163

Table 5.12: EROOS Compound Mutation Event Script..167
Table 5.13: Possibilities for a Compound with *Part* and *Whole* Restrictions 169
Table 5.14: Unsupported Restriction Cases in EROOS 170
Table 5.15: EROOS Compound Script Usage of Class Archive...170
Table 5.16: EROOS Constraint for a Time Trigger Creating Objects 179
Table 5.17: Trigger Specification for EROOS Constraints 180
Table 5.18: Trigger Specification for Integrated and Implicit Constraints 182
Table 5.19: Central Effect Description of *return book*..184
Table 5.20: Modular Effect Description of *return book*..186
Table 5.21: Distributed Effect Description of *return book*..188
Table 5.22: Nondeterministic Distributed Effect Description of *return book*..................190
Table 5.23: Time and Event Trigger for the Library Example..191
Table 5.24: EROOS Constraints for Simulating Dynamic Specialisation 195
Table 5.25: EROOS Group Script 196
Table 5.26: Time and Event Trigger for the Library Example..198

List of Definitions

Definition 4.1: Model, Model Instance, and Event Instance 63
Definition 4.2: EROOS Kernel Class 66
Definition 4.3: EROOS Creation Event....................68
Definition 4.4: EROOS Kernel Attribute 74
Definition 4.5: Extended Creation Event for an EROOS Attribute....................75
Definition 4.6: Implicit EROOS Kernel Attribute Query....................78
Definition 4.7: EROOS Kernel Relation 85
Definition 4.8: Extended Creation Event for an EROOS Relation 86
Definition 4.9: Implicit EROOS Refinement Query 88
Definition 4.10: Implicit EROOS Participation Query....................88
Definition 4.11: EROOS Constraint....................98
Definition 4.12: EROOS Specialisation 112
Definition 4.13: Implicit EROOS Specialisation Query....................117
Definition 5.1: EROOS Universe Class....................140
Definition 5.2: EROOS Archive Attribute 141
Definition 5.3: EROOS Universe Attribute....................153
Definition 5.4: EROOS Universe Relation....................155
Definition 5.5: EROOS Relation Mutation Event 156
Definition 5.6: EROOS Attribute Mutation Event 157
Definition 5.7: Implicit EROOS Universe Attribute Query 158
Definition 5.8: EROOS Compound....................164
Definition 5.9: Extended Creation Event for an EROOS Compound-Part 165
Definition 5.10: Implicit EROOS Compound Query 166
Definition 5.11: EROOS Compound Mutation Event....................167
Definition 5.12: EROOS Constraint with Trigger....................180
Definition 5.13: EROOS Group....................197

Chapter 1

Introduction

One of the real challenges for software engineering is, on the one hand, being able to obtain a good understanding of the needs and requirements for the software system to be built, and, on the other hand, the ability to deal with and respond to changing conditions and requirements throughout the development of the system and its further life cycle. Brooks already stated in 1987 that 'The hardest single part of building a software system is deciding precisely what to build.', while 'All successful software gets changed … The software product is embedded in a cultural matrix of applications, users, laws, and machine vehicles. These all change continually, and their changes inexorably force change upon the software product' [20]. Almost 20 years later, Ambler states that 'Domain knowledge is important … If you do not understand the problem domain there is very little chance that you can be effective'[4], while Martin states that 'It is the ability to respond to change that often determines the success or failure of a software product' [98].

In order to build a software system that conforms to the needs of the customers and end users, the software engineer must be able to obtain a clear insight into all issues regarding the software system and the context in which it must operate. In addition, modern software systems are far too complex to be built in an ad hoc way, but instead require clear and unambiguous methods and notations to develop qualitative systems that fulfil the needs and expectations of the customers. In order to master both the continuously changing requirements as well as the complexity of software systems, rigorous development methods, techniques, and notations are needed to model and structure the requirements in an optimal manner.

Object-oriented analysis, and more specifically conceptual modelling, is a key asset in dealing with changing requirements, since it enables to express these requirements in a consistent manner within the context of the universe of discourse and can as such

provide clear insight into the impact of a changing requirement on the software system and the environment in which it operates.

This introduction firstly discusses the background for this dissertation. Object-oriented analysis is positioned in the broader context of the overall software life cycle. The need and purpose of object-oriented analysis are stated, and its main characteristics are discussed. Hereafter, the major difficulties with current object-oriented analysis methods, and the main challenges for object-oriented analysis are identified. Next we state the goals addressed by this work. We conclude this introductory chapter with an overview of the text.

1.1 Background

1.1.1 The Software Life Cycle

The ever-increasing size and complexity of software demands for rigorous techniques and suitable tool support for the development of software. In software engineering, as in other engineering disciplines, the development process of a software system is structured into a number of phases in order to cope with the inherent complexity of it. Each phase is focussed on different aspect of the development process, and has its own purpose, goal, focus points, notations, and formalisms to express its results. Although there is no real consensus on the exact number and naming of the phases into which a software engineering process can be divided, the following phases are commonly identified:

- **Analysis Phase**: In this phase, the universe of discourse (UoD, also called problem domain, business domain, real world or system context) in which the software system will operate is studied, and modelled into **a conceptual model** (also called domain model, business model or analysis model). A conceptual model is a formal model in which every element from the universe of discourse has a transparent and one-to-one correspondence to an entity in the conceptual model. Its goal is to obtain a proper insight in the context in which the system will operate. A conceptual model provides a complete description of the current or envisaged universe of discourse expressed in one or more diagrams. In addition, the requirements of the system are identified and specified both on a functional and non-functional level. The specification of the requirements should be based on the facts and information that occur within the universe of discourse through the usage of the conceptual model. Sometimes a further distinction is made between the requirements phase [47][41][122], focussed on capturing the requirements and expressing them in a mostly textual format, and the domain analysis phase [89][7], focussing on studying the universe of discourse (context) and constructing a conceptual model for it (context realization).
- **Architectural Phase**: In this phase, an architecture for the software system is defined. An architecture describes the structure of the system, which comprises software elements, the externally visible properties of those elements, and the

relationships among them [10]. The result of this phase is **an architectural model**, which contains a complete description of the architecture of the system including its links with other systems and the external world.

- **Design phase:** In this phase, the software elements within the architecture are further elaborated. This includes a whole range of activities, going from taking decisions about the realisation of the software elements and the definition of their internal substructures, evaluating potential usage of software libraries and available (possibly off-the-shelf) components, incorporation and application of software patterns, definition of interfaces between software elements, to identification of implementation classes, and describing their interrelations and their internals. Sometimes a further distinction is made between high-level design, focussing on components, interfaces, classes and their interrelationships, and low-level design, focussing on class internals, such as state diagrams, operations, and instance variables [171]. The result of this phase is **a design model,** containing all the details about the system to be realised.
- **Implementation phase:** In this phase, the software system is implemented and executable code is produced according to its design. The result of this phase is **an executable program(s)** in a single or multiple programming languages.
- **Deployment phase**: in this phase, the system will be deployed in its environment in which it will operate. This includes providing the necessary run-time support for the system and establishing its interaction with the external world.

Although often a dedicated **maintenance phase** is identified, covering all tasks and activities that have to be performed to keep the system running and up-to-date with the expectations and requirements of the customers and the end users after system deployment, we do not consider this as a true phase on its own. In fact, the maintenance phase can better be seen as a continuous activity throughout all identified phases above, adding to and adjusting the outcomes of the software engineering phases in order to maintain the system according to the needs of its customers and end users. All tasks that have to be performed during maintenance can be categorised into one of the development phases that were mentioned above. In the same manner, a **testing phase** or more broadly a validation and verification phase could be identified. Also, this phase could be considered as not a true phase on its own, since validation, verification, and testing occurs to a certain extent during all phases of the software development process [11][98][12][149].

The process of developing a software system realises a certain path through a number of activities that will be performed in the phases defined above. Although it is possible that these phases are executed fully sequentially, the development of a software system often follows an iterative, incremental development process, focussing on the realisation of a part of the functionality after which the software system is further been extended [173][55][88].

Models are used to express the properties and structure of the software elements in order to reason and communicate about them. Throughout the software development

process, models can be further detailed by extending them with additional information that is relevant in a phase, transforming them into lower-level models using the same or different notations, and, eventually, generate code from them. Model-Driven Development (MDD) techniques [50][83] can be used to automate transformations from one model into another.

The object-oriented paradigm, which has been existing for almost 40 years, is nowadays widely accepted within the Software Engineering community as *the* paradigm to structure software systems in an optimal way. Although this paradigm originated from implementation languages, such as Simula [33] and Smalltalk [56], it is currently being used in all phases of the development process. The object-oriented paradigm focuses on identifying objects, clustering them into classes, and describing their characteristics, behaviour and interrelationships.

This work mainly focuses on the analysis phase. In addition, the relationship between the analysis phase and the design phase will be discussed to some extent. The implementation and deployment phase are beyond the scope of this work.

1.1.2 Object-Oriented Analysis

This section gives an overview of the most important aspects within object-oriented analysis. It will identify the main distinctions with object-oriented design and implementation, and will highlight the necessity of it for the overall software development process. Last, we will present and compare the different approaches to object-oriented analysis. In order to classify the object-oriented analysis methods in a number of methodological families, this section introduces the notion of *analysis views,* focussing on the models that a method produces. A large number of comparisons of object-oriented analysis and design methods have been made [6][35][102][32][45][68][69][164][66][23].

1.1.2.1 Scope and Goal of the Analysis phase

Although object-oriented analysis is mostly founded on object-oriented design and programming ideas, conceptual modelling is also heavily influenced by the data-oriented development methods. The roots of object-oriented analysis can actually be situated on four domains:

- The object-oriented programming paradigm [33][56] and object-oriented design approaches [17][16][29].
- Function-oriented and structured analysis (SA), design (SD), and programming approaches [54][37][169][170][116][81], such as Structured Systems Analysis and Design Method (SSADM) [40][123].
- The relational model [30], Entity-Relationship (ER) modelling [24][42], and data-oriented development methods [105][106].
- System design approaches such as Jackson Structured Programming (JSP) [72] and Jackson System Development (JSD) [73].

The goal of the analysis phase is to acquire an insight into the universe of discourse in which the software system will operate, and to express the requirements of the system to be applied within the universe of discourse according to the needs of the customers and the end users. The input to analysis is a universe of discourse and a problem statement. The output of analysis is an understanding of the problem domain, expressed in a conceptual model, and a requirements specification for a software system or a family of related software products [8]. During the analysis, the universe of discourse is thoroughly studied and the knowledge is captured in a conceptual model, also called a domain, business or analysis model.

An object-oriented conceptual model can actually be specified merely for a better understanding of the universe of discourse without necessarily having to be realised by a software system. It can as such be applied for Business Process Re-engineering (BPR) [78][19][67][53][77], in which an actual business process from the universe of discourse is modelled, studied, and improved, or being transformed into a totally new process. As such, the requirements in such a conceptual model will express the new procedures, rules and regulations that should be followed in the ameliorated process.

The object-oriented paradigm is highly suited for performing analysis and expressing the domain analysis results. Objects and classes, the primary modelling constructs of the object-oriented paradigm, are excellent means for performing a classification of relevant items from the universe of discourse, since the universe of discourse can be considered as a universe of related things that either are objects or subjects. Based on the core concept of a class, both structural elements, representing facts and interdependences from the universe of discourse, as well as behavioural elements, representing the ways in which these facts can evolve over time, can be used to construct a complete model of the universe of discourse that is relevant for the software system to be developed. Since an object-oriented model consists of an integration of both structure and behaviour, the resulting domain model will both cover the aspect of 'which information is present within the universe of discourse at a certain moment in time' as the aspect of 'how can the information within the universe of discourse evolve over time'.

The results of the analysis phase can be considered on three levels:

- A domain model of the present universe of discourse, containing a complete specification of the universe of discourse as it exists and operates at the current moment. It describes relevant elements that exist in the universe of discourse, the properties concerning them, and the events in which they can become involved. Such model is called by Ludewig [94] a *descriptive model.*
- The specification of the functional requirements, which will use the domain model as the context to which they refer.
- An analysis model expressing the solution model, describing the ultimate envisioned transmuted universe of discourse in which the functional requirements have been realised. Such model is called by Ludewig [94] a *prescriptive or explorative model.*

However, the analysis results cannot always be clearly separated according to the three levels above, since a number of elements cannot always be identified as belonging to a single level. In fact, the specification of the functional requirements will often be done by directly incorporating them into a prescriptive domain model. On the other hand, the distinction between the universe of discourse as it is and the universe of discourse as it should be is easier to make, although, in some cases, only the latter will be modelled during software development.

1.1.2.2 The Necessity of Analysis

According to Jackson [73], a software engineer should start by building a model of reality, after which the functional requirements should be described based on this model of reality. This improves extendibility, preciseness, and communication between the software engineers and the customers.

We claim that an accurate and precise description of the functional and non-functional requirements cannot be done without a clear insight into the universe of discourse and without being based on a detailed model of all aspects from the universe of discourse. Without the knowledge of how the universe of discourse is structured, and which inherent properties are embedded within it, it is impossible to describe how the envisioned system should behave in its context. For example, business processes that the system should support, or rules and regulations that the system should enforce cannot exhaustively be specified without having proper insight into the elements that they apply upon. This will lead to incomplete requirements elicitation and imprecise requirements definition.

A process that does not take into account object-oriented analysis will rest on an implicit image and understanding of the universe of discourse by the software engineers, each having to build an implicit conceptual model in their mind. Since these implicit conceptual models will never explicitly be reviewed, assessed or agreed upon, they will lead to incomplete, incorrect, and incompatible visions on the universe of discourse, giving rise to the introduction of errors and inconsistencies during later development stages, and customer dissatisfaction due to the discrepancy between their requirements for the system and the actual properties of the software system that is delivered.

1.1.2.3 Distinction between Analysis and Design Concerns

There are many visions on and definitions of object-oriented analysis. Object-oriented analysis is defined by Yuan [171] as ‘an activity of discovering, understanding, and describing facts about real-world objects and their behaviors in the problem to be solved. These facts are something system developers cannot change or invent.’, while object-object design is defined as ‘an activity of building the system architecture as well as each system class. This involves inventing and specifying classes and their properties in the solution domains.’ Monarchi [102] defines analysis as ‘modelling the problem domain by identifying and specifying a set of semantic objects that interact and behave according to system requirements.’, while object-oriented design

is ‘modelling the solution domain, which includes the semantic classes and interface, application, and base/utility classes identified during the design process.’

Since the analysis phase focuses on the universe of discourse, software- and hardware-related issues are not yet relevant at this stage. The transition to a software system will gradually introduce these issues during later stages of the development. Although a number of concepts seem to be quite similar due to the usage of the same object-oriented paradigm, there are some important major distinctions between, on the one hand, object-oriented analysis, and, on the other hand, object-oriented design and implementation:

- The concepts used in object-oriented analysis try to capture knowledge that is present within the universe of discourse. Although concepts as classes and objects are being used to represent them, these concepts try to grab information from the universe of discourse and thus refer to real-life items, people or facts. Although a real-life class in an analysis model can lead to the implementation of a similar class in the actual system, the concepts in a conceptual model are from a different nature and have no direct commonalities with software classes, software objects, database tables or database records.
- The structural elements in an analysis model try to capture the real-life information and dependencies from the universe of discourse. The information represented in an analysis model reflects facts present in the universe of discourse. Although a real-life fact in the analysis model can lead to a specific pointer or an instance variable value of a software object, the structures in a conceptual model have nothing directly in common with instance variables, pointers or data base entries. In addition, the use of inheritance in an analysis model reflects the fact that a certain real-life element type can be seen as a subtype of another real-life element type, which can but does not necessarily have to lead to an implementation inheritance relationship between software classes.
- The behavioural elements of an analysis model try to capture changes of real-life information within the universe of discourse. The represented events reflect occasions from the universe of discourse. Although a real-life event can lead to a specific call of a class method on a software object, the events in a conceptual model have nothing directly in common with methods, operations or stored procedures.
- The model constraints of an analysis model try to capture rules and regulations from the universe of discourse, whether they represent physical laws or human imposed restrictions. Although they can, and often also they will be enforced on the software level, the constraints of a conceptual model have nothing directly in common with the realisation of them in the software, and the decisions to be taken on how, where, and when to check and enforce them in the ultimate software system.
- Since an analysis model reflects the facts and occurrences from the universe of discourse, the information contained in the model is supposed to be correct, and

the events in synchronisation with the occurrences in the real-life world. Issues regarding the accuracy and validity of the stored information, and the time span between the occurrence of the event and its registration in the system are, therefore, only a concern of the actual software system, and are as such not treated in the analysis model.

Although object-oriented analysis uses the same kind of concepts as object-oriented design and implementation, there is a profound distinction on the kind of elements they model, the level of detail of the information in the model, and the quality criteria that are used during the construction of the model.

1.1.2.4 Classification of Analysis Views

Object-oriented analysis methods often offer a number of diverse analysis views on the universe of discourse. Therefore, the number of models they produce as a result of their activity can vary. Although most object-oriented analysis methods incorporate both structural and behavioural aspects to a certain extent, one can clearly distinguish two distinct families: structural-focussed analysis methods and behavioural-focussed analysis methods.

1.1.2.4.1 Structural Analysis Methods

Structural object-oriented analysis methods, such as Booch [15], OMT [93][126], OOA [28], OOIE [96][97], OOSA [43], and Fusion [31], are mainly focussed on discovering and identifying domain entities, their properties, and the interrelations between them. The domain model, expressed as a class diagram or a static structure diagram [128] providing a class centred description of the information, is the primary asset of this type of methods. Classes are used as the major modelling concept, and are further detailed using attributes and operations, and they are interconnected with each other using associations.

Although their main focus is to obtain a good capturing of the information structures present inside the universe of discourse, operations are nevertheless a specific point of interest for these methods. Descriptions are made based on the ways objects interact with each other to fulfil their responsibilities. Typical object interactions are identified and captured as operations applicable on the objects of the corresponding classes. This results in object interaction diagrams, such as collaboration diagrams and message sequence diagrams [128].

1.1.2.4.2 Behavioural Analysis Methods

Behavioural object-oriented analysis methods, such as SM [134][133], RDD [167], OBA [124], OOSE [76][75], BON [103][156], and UON [115], are mainly focussed on discovering and identifying inter-object behaviour. They focus on specifying the interface of the objects and defining the protocols between them. For instance, CRC cards can be used as the major modelling concept, defining the necessary Classes, their Responsibilities, and their mutual Collaborations. Although behavioural object-

oriented analysis methods also use classes as their core concept, the class is rather studied in isolation, focussing on the interaction with the class, while structural analysis methods study the class in its broader context, focussing on the interrelations with other classes. A number of behavioural analysis methods make a further distinction between different types of behavioural elements, such as events, operations, methods and signals.

Although class behaviour is the main focus of behavioural methods, class properties and interrelationships are, nevertheless, present in a slightly different form. Instead of defining properties and interrelationships as first-class entities, they can only be retrieved by interacting with the object, thereby obtaining the information indirectly using the provided operations for the object.

Since behavioural object-oriented analysis methods focus specifically on classes and the interaction with them, they tend to describe the software system rather than the universe of discourse. Therefore behavioural object-oriented analysis methods can be characterised as software analysis methods rather than domain analysis methods.

1.1.2.4.3 UML as the Catalyst of Analysis Views

The Object Management Group (OMG) has tried to standardise the notation used for the description of software artefacts during the overall software life cycle by defining the Unified Modeling Language (UML) [120][119][107][109][128]. Through the standardisation of a modelling notation, UML tries to establish a common vocabulary that can be understood by any software engineer. It establishes a common notation and semantics for software artefacts by defining a number of areas that can be modelled, views that can be defined for each area, models or diagrams that can be used to express each view, and concepts that can occur within these diagrams. Whereas UML 1.4 defines 9 diagram types (class, object, use case, sequence, collaboration, statechart, activity, component, and deployment diagram), UML 2.0 defines 4 major areas (structural, dynamic, physical, and model management area), 9 views (static, design, use case, state machine, activity, interaction, deployment, model management, and profile view), and 11 diagram types (class, internal structure, collaboration, component, use case, state machine, activity, sequence, communication, deployment, and package diagram).

The general adoption of UML as the de facto standard notation for the description of software artefacts from the analysis up to the implementation phase has led to a rather dominant position of structural object-oriented analysis methods. Although the emergence of Agile software development [11][98] gave rise to a revival of more behavioural object-oriented analysis techniques, the Agile Modelling approach [3][18] has redirected the Agile community again to the usage of UML and more structural based object-oriented analysis techniques.

In addition to the standardisation of the modelling notation, there are even efforts to standardise the software development process. The Unified Process (UP) [74] tries to

define a component-based software development process framework as a reference model that is use-case driven, architecture-centric, iterative, and incremental.

1.1.3 Problems and Open Issues for Object-Oriented Analysis

This section identifies a number of problems and open issues for object-oriented analysis. Although we provide an isolated description of the open issues, many of them are rather interrelated. As an example, since UML tries to integrate a large number of notations for software artefacts into a single modelling language, it is a very extensive language that offers the possibility of using a huge number of concepts. Therefore, it is unclear for an analyst to evaluate and identify the best-fitted modelling concept for expressing a specific item. So the modelling notation is related to the methodological approach. Another example is the fact that the usage of informal specifications and the usage of a complex notation with several separate models can easily give rise to an inconsistent analysis model.

1.1.3.1 Modelling Notation

Almost all current object-oriented analysis methods use the Unified Modeling Language as the notation for defining the outcomes of the analysis phase. Since the Unified Modeling Language is mainly defined as a unification of primarily object-oriented design notations, it has a clear focus on object-oriented design and implementation concepts. Although it is possible to express conceptual models using UML as the notation, the provided concepts will drive the analyst heavily to a more software focussed analysis model rather than a conceptual model. This can lead to several problems:

- Since the object-oriented paradigm originated from object-oriented programming languages, many object-oriented concepts are still largely biased towards a programming context. Therefore UML is better suited for building a computational model rather than a conceptual model. For instance, all functionality is designated on the object level in order to be invoked on a specific object, or on the class level in case of class methods. However, it could be useful to define functionality on a model level during the analysis phase. As another example, the value of attributes and associations can be changed, as an analogy to the fact that instance values and pointer sets are often manipulated on a programming level. However, the knowledge that an attribute had a particular value at a certain moment in time is a fact one should be able to reason about, without being concerned about how this value should be stored and referred to in the model.
- UML tries to cover the specification of all software artefacts during the whole software engineering life cycle. It must therefore allow the modeller to use the UML concepts on the analysis level, the high-level design level as well as the detailed design and implementation level. As such, each definition in UML is the result of a compromise between different viewpoints, allowing several interpretations in order to fit the needs of all levels on which it can be applied. From the standpoint of UML, it is useful to have a loose definition of the

concepts in order to enlarge its applicability. But from the standpoint of the analyst, it is very difficult to evaluate which concept should be used in which manner when there is no clear interpretation for it. This can lead to different interpretations of analysis models by several modellers, since the exact meaning of certain model constructs can be very ambiguous.

- UML contains a large number of concepts that are not aimed for being used during object-oriented analysis. Moreover, some concepts can be useful on the analysis level, but only on a rather abstract level without defining them in full details using all capabilities of the UML notation. But since these unsuited analysis concepts and unnecessary detailed descriptions are an integral part of UML, it is very confusing for an analyst for which purposes and to which extent these concepts should be used during the analysis phase. As an example, sequence and deployment diagrams are concepts that are more related to later phases of the software development process, while details such as visibility and navigability in a class diagram should not yet be addressed during the analysis phase. By using such a rich and complex notation as UML on the analysis level, the analyst gets overwhelmed by a huge variation of concepts spread out over several abstraction levels. The analyst must determine oneself which concepts could be useful for expressing analysis models. This is referred to by Wand [160] as *construct excess,* in which a notation offers constructs that do not correspond to a type of facts from the universe of discourse. This leads to analysis models incorporating certain low-level elements that should not have been defined yet, or providing a huge level of details that is not suited for the analysis level. In both cases, it gives rise to the fact that unnecessary decisions are taken much too early in the development process.
- Although UML offers a large number of concepts, it lacks concepts useful for object-oriented analysis. For instance, UML lacks the possibility to reason explicitly about dead or passive objects, expressing information that once was valid but has ceased to exist. This is referred to by Wand [160] as *construct deficit,* in which a type of facts from the universe of discourse cannot be represented by any of the modelling constructs. Although UML offers a profile mechanism in order to extend its notation using stereotypes and stereotype attributes (called tagged values in UML 1.x), it is impossible to add other kinds of extensions to UML. This leads to analysis models that try to express certain information by realising it using inadequate concepts offered by the notation.
- Since UML does not offer a clear-cut set of integrated and complementary concepts, a huge problem of overlapping modelling concepts arises. An analyst constantly has to evaluate during analysis which concepts to use for expressing certain information. For instance, a relation between two items can be modelled as a pure association, an association class, an association reified into an autonomous class having two assisting associations, or as referential attributes inside the items themselves. This is referred to by Wand [160] as *construct redundancy,* in which a type of facts from the universe of discourse can be represented by more than one modelling construct. This leads to different personal styles in analysis models, which obstruct the communication between

analysts and hamper the reuse of analysis models over time. As a result, discussions during model reviews will often focus on the modelling style rather than the information content that is expressed in the model. Analysis modelling guidelines could try to solve this problem to a certain extent. But a modelling notation that limits or even avoids analogous modelling constructs would be a better solution than offering modelling guidelines. Forcing the modeller to use a particular concept for modelling specific information improves the clarity and increases the communication power of the model.

As a conclusion, we state that object-oriented analysis needs a limited set of powerful concepts targeted to capturing knowledge and information during the analysis phase, and a corresponding modelling language to express this knowledge into a conceptual model. Although a restricted set of concepts offered by UML can be suitable for analysis, the usage of UML tends to drive the analysts to a computational view rather than a conceptual view. In spite of the fact that UML offers certain means for extending its notation using profiles [107][109][128], this is not sufficient to transform UML into a suitable analysis notation.

1.1.3.2 Model Consistency

Most object-oriented analysis methods produce different kinds of models as a result of the analysis activity. This allows multiple views on the universe of discourse, each focussing on specific aspects that are important for the software system. In addition to class diagrams, often used by structural analysis methods, other diagrams are used to express additional information, such as use case diagrams for a functional view on the system, and statechart diagrams, activity diagrams, and sequence diagrams for dynamic views on the system.

Although most methods incorporate a number of specific rules to enforce consistency in a model, limited attention is paid to the consistency between the models. For instance, an event that triggers a state transition inside an object must be part of the interface of that object, or called by a method that is part of its interface. Two solutions can be followed to obtain model consistency:

- Consistency rules can be defined between models in order to keep the information present in one model in line with the information defined in another model. As such, every element that is referred to inside a model should have a core model to which it belongs and in which it is completely defined. Concepts should be designated as such that there is a single specific model in which they must be defined, after which they can be used in other model types. However, such approach can lead to a number of other problems. On the one hand, it creates interdependencies between models, since certain models can only be constructed after other models have been composed. On the other hand, it can lead to a mutual dependency between models, where one model depends on concepts defined in another model and vice versa. In addition, one should take care that a model update is propagated through all other models that refer to the updated elements.

- Instead of supporting multiple models, a single model could be build that captures all information present in the different models. As an example, instead of modelling a statechart diagram, information from the universe of discourse that could lead to state transitions can be captured inside a class model. The state information that is contained inside the statechart diagram can then be derived from the information inside the conceptual model when necessary. The same approach can be made for activity diagrams by capturing knowledge about the start and end of an activity in the conceptual model, and deriving the ongoing activities afterwards from this model. Such approach will give rise to bigger class models, since all information that is normally spread out over a number of models will now be contained in the single class model. However, since consistency only has to be maintained in a single model, the effort of achieving consistency is reduced while the multi-model problems can be avoided.

1.1.3.3 Model Informality

Most object-oriented analysis methods offer informally defined concepts and notations as well as a mechanism for incorporating informal elements in a model. But such an approach leads to ambiguity and misinterpretation of the analysis outcomes. This ambiguity is situated on two levels: the model concept level and the model information level.

- Many object-oriented analysis methods only define the semantics of their concepts in an informal manner. The semantics are often stated in a textual description that can lead to misinterpretation and ambiguity. Examples are used to illustrate its meaning and usage, but are often very fragmentary and incomplete. Although methods try to express their semantics in a clear manner, a number of implicit assumptions are made that are not always explicitly stated. As such, an analyst starts to learn the modelling notation by example, and does not master the specificities and the full power of the offered notation. In fact, this is how novice modellers often acquire their knowledge of UML. In addition, precise model concepts with formal syntax and semantics enables model consistency checking on the analysis level in order to obtain error-free analysis results. Methods with informally defined concepts cannot perform model consistency checking, nor use the model as an input for model transformations. Models in such methods remain rather fancy but noncommittal pictures of the universe of discourse.
- Not only are the modelling concepts ill-defined, the information contained inside the ultimate analysis results is often incomplete and only paraphrased in natural language. For instance, UML offers the concept of a note to record comments or other textual information in a model. Notes are among others used for defining general restrictions on a model, and for defining the effect of events and methods inside a model. In addition, use cases are often informally specified using structured text descriptions. However, informal model entities can lead to a huge number of problems, such as model errors, incompleteness, contradictoriness, and ambiguity.

- **Errors** occur when the informal descriptions are wrong. It is possible that the offered concepts are inadequate to formulate the precise meaning. In such case, an analyst could try to formulate a closely related situation instead that could also not be fully correct. The model hereby wrongly specifies what must be valid in certain cases.
- **Incompleteness** occurs when the informal model entities do not completely describe the full set of cases on which they apply or the full set of rules that must be imposed. The model does not specify what must be valid in certain cases.
- **Contradictoriness** occurs when several informal model entities describe rules or situations that contradict each other. It is hereby impossible to achieve a valid model instantiation in certain cases.
- **Ambiguity** occurs when the informal model entities are not precisely described so that the intention of the description is not clear. One is obliged to interpret the descriptions in a certain manner. It is impossible to decide what must be valid in certain cases solely based on the information contained inside the model.

Moreover, it is impossible to validate and verify a model for **consistency and correctness** when informal descriptions are part of it, since informal model entities cannot automatically be checked or used as input for model transformations. Informal models cannot be the ultimate source of knowledge that can be shared between the customers, end users, analysts, designers, and implementers, but in contrary will be the cause of misconceptions and misunderstandings between all parties involved in a software development process.

In analogy to the CMMI [27] levels of the capability model for software engineering, Warmer [161] proposes the following six Modelling Maturity Levels (MML):

- Level 0 corresponds to having no specification at all of the software system. The results of this level is that (1) there are conflicting views among the developers, (2) it is only suitable for small applications, (3) the system can only be understood by the programmers themselves, and (4) may choices are made in an ad hoc fashion.
- Level 1 corresponds with a textual specification of the system in a number of documents. The results of this level is that (1) the specification is ambiguous, (2) the programmers must make business decisions based on their personal interpretation of the documents, and (3) it is impossible to keep the specification up to date.
- Level 2 corresponds with a textual specification of the system augmented with several high-level diagrams. The results of this level is that the drawbacks of level 1 are still present, although the documents are easier to understand because of the diagrams.

- Level 3 corresponds with a model specification of the system augmented with text. The results of this level is that (1) the diagrams are real representations of the software, (2) the transformation into code has still to be done manually, (3) it is still very difficult to keep the specification up to date, and (4) the programmers must still make business decisions themselves although it has less influence on the system architecture.
- Level 4 corresponds with precise models of the system. The results of this level is that (1) programmers do not make business decisions anymore, (2) keeping models and code up to date is essential and easy, and (3) iterative and incremental development are facilitated by the transition from model to code.
- Level 5 corresponds with precise models from which the code of the system can completely be generated. In this manner, software developers use models to express their software constructs whereupon code generators, in the same way as compilers do, generate the full executable code. At this level, the modelling language can be seen as a high-level programming language of its own.

As a conclusion, we can state that object-oriented analysis needs both a formal definition of the concepts used for modelling as well as a formal description of the knowledge captured inside the model.

1.1.3.4 Methodological Support

As its name indicates, the Unified Modeling Language (UML) is merely a modelling language that provides a notation for the specification of software models. It does not offer any guidelines for the software engineer on how to perform object-oriented analysis, and how to build analysis models. It is inadequate to offer only a modelling notation without offering a method and additional concrete guidance for the analyst on which models to build, which concepts to use, and how to transform facts from the universe of discourse into analysis model entities. It can be useful for offering an adequate toolbox of modelling concepts usable for developing an analysis model, but this should be accompanied by a concrete approach on how the universe of discourse should be modelled, and which criteria should be used for mapping facts from the universe of discourse into optimal model structures.

Preferably there should exist a single clear and unique path from the universe of discourse to the resulting model using the most adequate modelling concepts for each information fact from the universe of discourse. A good analysis method should provide analysts with clear criteria and guidance for analysing the universe of discourse and selecting the most suitable methodological concepts to capture its knowledge. These criteria should be unambiguous in order to avoid as much as possible uncertainty and doubt for the analyst in choosing between apparently equivalent variants within the modelling notation.

Moreover, the methodological guidance should focus on the analysis goals of capturing the knowledge from the universe of discourse without introducing design or implementation aspects at the analysis stage. All too often, analysis approaches tend

to focus on the software to be built rather than on the universe of discourse to be captured, thereby neglecting and obfuscating certain aspects from the universe of discourse, and inclining to let software related decisions determine the actual structure of the analysis model.

As argued earlier in Section 1.1.2.2, software development methods that neglect or minimise the analysis phase will give rise to a limited and informal vision on the universe of discourse and the system requirements, introducing potentially huge problems during or after the software development process. Therefore, only a sound and explicit methodological analysis approach with unambiguous modelling concepts, and a unique and univocal manner for modelling knowledge from the universe of discourse, will give proper support to the analyst for constructing analysis models in a suitable and efficient manner.

1.1.3.5 Analysis Demarcation and Further Transition

In Sections 1.1.1 and 1.1.2.3, we have stated the difference between object-oriented analysis, architecture, and design. Object-oriented analysis focuses fully on the universe of discourse, describing the facts, rules, and regulations from the universe of discourse, the functional requirements that must be realised, and the envisioned transmuted universe of discourse in which the functional requirements have been realised. From the architectural phase on, the focus shifts to software- and hardware-related issues concerning the system to be realised.

In a large number of object-oriented methods, the boundary between the analysis phase, and the architecture and design phase is very vague. From the analysis phase on, details concerning the software realisation creep in and cause a software bias in the description of the universe of discourse. As such, the analysis model is no longer a pure conceptual model, but contains a number of details and decisions that should better be postponed until a subsequent computational model.

This vague border between analysis and design is often seen as an advantage in order to obtain a smooth transition from analysis to design, without creating a large gap between these phases. Larman [89] claims that it might even be contra-productive to have a rigid definition and separation of these two phases. MOSIS [64] does not even distinguish analysis from design as separate phases. However, we claim that the nature and the concerns of analysis and design are so different that it is an absolute must to have a clear division of these phases and their kind of activities:

- The key condition for any successful activity is to have a clear focus on the goals and objectives of the activity. Regarding modelling, this means that the answer on the fundamental question of 'what must be described in an analysis model' should be crystal clear. A vague border between analysis and design impedes the analysis modelling activity, since one cannot make a clear assessment of which elements to include in an analysis model and which not. The analysis activity as such turns into an unconducted and unstructured activity with arbitrary decisions that are taken by the analyst, and noncommittal results that are obtained.

- A number of analysis methods promote *design by elaboration* [132], a kind of seamless transition from analysis into design. They claim this can be achieved by a continuous refinement of the analysis model, thereby gradually obtaining a suitable design model. Although it could be possible to refactor a high-level design model into a low-level design model, we claim that a seamless transformation of an analysis model into a design model is utopian, impracticable, and inadequate for developing suitable architectural and design models. Since an analysis model focuses solely on the universe of discourse, it can be expected that the analysis model structure is inadequate to represent a software system structure to be developed. As presented in Section 1.1.1, the main objective of the architectural phase that follows the analysis phase is, in fact, to obtain a suitable structure for the software system, based on the quality attributes to achieve and on the architectural drivers needed to realise these attributes [10]. It is a naïve and incorrect vision to suppose that an analysis model can seamlessly be translated into a suitable architectural model. Although the architectural phase is based on the knowledge gathered during the analysis phase, the architectural structure is build upon a set of criteria that are different from the ones used during the analysis phase.

As a conclusion, we can state that the object-oriented analysis phase should be clearly separated from the consecutive but much more software-focussed phases. Moreover, the transition of the analysis model, describing the universe of discourse and the functional requirements, into a software architecture is rather complex and far from evident, and therefore should not be considered as a mere model refinement activity. However this does not mean that the analysis models are just throwaway models that are merely constructed for achieving a better understanding of the universe of discourse, without providing any input for the consecutive software models. *Design by translation* [132] can be a successful approach for the transition from analysis to design. Design templates can provide means of transforming analysis constructs into design constructs [110][111][112]. Automated or semi-automated model transformations from analysis models into certain bigger or smaller portions of design models using Model-Driven Development (MDD) [50][83] techniques can be beneficial and can help to capitalise the analysis results during software development.

1.2 Goals

The goals of this dissertation are threefold.

- **Definition of key principles for conceptual modelling**. Current object-oriented analysis methods have a number of deficiencies regarding the modelling notation, the model consistency, the model informality, the methodological support, and the analysis demarcation. Based on this identification, the first objective of this dissertation is to formulate a number of key principles for conceptual modelling that are necessary in order to offer proper support for modelling the knowledge from the universe of discourse.

- **Evaluation and comparison of model constraint specification formalisms and notations.** Model constraints play an important role in object-oriented analysis. There exist several different specification formalisms to express model constraints. The usage of a specific formalism can cause a different impact of the model constraint on the resulting conceptual model. Several alternative modelling concepts for model constraints are even offered inside a single analysis method and notation. The second objective of this dissertation is to compare, evaluate, and build a taxonomy for model constraint specification formalisms and notations, and to investigate their suitability for representing knowledge from the universe of discourse.

- **Development of an appropriate object-oriented analysis method and accompanying notation for conceptual modelling.** Current analysis methods and notations, including UML, are not suited to describe conceptual models in an adequate manner. The third objective of this dissertation is to develop an appropriate object-oriented analysis method and a supporting notation for conceptual modelling in accordance with the identified key principles for conceptual modelling. This method should incorporate proper constraint specification formalisms. Such an analysis method is an indispensable asset to create a clear insight in the universe of discourse, capture its knowledge, properties, and structure in an appropriate format, and position the envisioned software system in its true real-world environment.

1.3 Contributions

The main contributions of this dissertation are.

- **Advanced methodological concepts for achieving the key principles for conceptual modelling.** We propose (1) a set of key principles for conceptual modelling that are necessary in order to create an adequate model of the universe of discourse, (2) a taxonomy for model constraint formalisms in object-oriented analysis, (3) a constructional conceptual model approach in which information can only be added to a model instance, (4) a querying mechanism to retrieve historical information regarding former attribute values, object links and creation and destruction times of objects, and (5) a formal notation for the semantics of queries and events that predates and is largely comparable with the Object Constraint Language (OCL).

- **The definition of new structural concepts to express model constraints implicitly within the model structure.** We propose (1) the incorporation of model constraints in each methodological concept by definition, (2) the usage of existential dependency as the key modelling criterion for constructing the conceptual model, resulting in a hierarchical relational model structure, and (3) the introduction of explicit class archives that can express lifetime dependencies between objects. These concepts enrich the expressive power of a conceptual model structure.

- **The introduction of constraints with supporting resolution mechanisms as a first-class model concept.** We propose a mechanism to specify model constraints as a first-class model concept, using a formal notation based on many-sorted first order logic. The constraint mechanism predates and is largely comparable with the Object Constraint Language (OCL). In addition, we propose the concept of a constraint trigger that can specify a generic constraint solver to resolve constraint violations, by injecting specific error handling behaviour into an event, or by firing an event due to progress of time

1.4 Overview

This dissertation contains six chapters. In addition to this introduction, the remainder of this dissertation is organised in the following chapters:

Chapter 2 proposes a taxonomy for model constraint formalisms in object-oriented analysis. We present an overview and a comparison of approaches for dealing with model constraints in object-oriented analysis. The role of model constraints in object-oriented analysis is situated, and different approaches for the specification of model constraints are presented and compared. We argue that model constraint specifications should form the core model structure for a conceptual model.

Chapter 3 proposes the key principles for conceptual modelling, being Uniqueness, No Redundancy, Unambiguity, Completeness, Minimalism, Preciseness, No History, Existential Dependency, and Abstraction. We claim that these principles are of utmost importance during analysis in order to obtain the most suitable conceptual model.

Chapter 4 proposes the EROOS kernel for conceptual modelling, which is developed according to the key principles for conceptual modelling. The EROOS kernel is based on a backbone of model-implied constraint specifications, using existential dependency as the key criterion for obtaining the most suitable conceptual model structure. It proposes a constructional approach for a conceptual model in which information can only be added to the conceptual model instance.

Chapter 5 defines an advanced methodology built on top of the EROOS kernel. Although the EROOS kernel concepts are actually sufficient to build a conceptual model, it is more practical to have additional suitable concepts at one's disposal to simplify the specification of recurrent EROOS analysis patterns. The EROOS universe proposes advanced concepts for modelling the universe of discourse. Key contributions of this dissertation that are proposed in this chapter include class archives, compound structures for modelling mutual dependency, and constraint triggers for automatic event triggering and constraint violation resolution.

Last, we conclude this dissertation in Chapter 6 by a summary and an overview of the major contributions of our work, and directions for future research.

Chapter 2

A Taxonomy for Model Constraint Formalisms in Object-Oriented Analysis

This chapter introduces the notion of *model constraints* in object-oriented analysis, and provides a taxonomy for model constraint notations.[1] The comparison of different notations for model constraints highlights the importance of proper support for the specification of model constraints during the analysis phase, and identifies deficiencies in current object-oriented analysis methodologies and their notations.

2.1 The Role of Model Constraints in Object-Oriented Analysis

Model constraints play a key role in object-oriented analysis. All object-oriented analysis methodologies incorporate the notion of model constraints in their notations somehow, each in their own manner. By means of model constraints, intrinsic properties of the system to be modelled can be described in a very elegant way. Model constraint specifications are used to express business rules, legal laws, social rules, physical limitations, undesired behaviour, and invalid situations within the universe of discourse in the conceptual model, as such restricting the potential valid instances of the model. Model constraints can be seen as general rules of the universe of discourse restricting certain events or services, or forcing certain business policies. They describe normal or wanted situations within the universe of discourse, excluding undesired, inadmissible, and forbidden situations. A model constraint is an analysis

[1] A part of the work presented in this chapter has been published in [152] and [151].

model concept that can be used to specify these kinds of rules, thereby formalising and capturing the actual semantics of the rule into the conceptual model.

Within an analysis model, model constraints are a means to express general properties of model entities. Model constraints must remain valid during the entire lifetime of a model execution or simulation. As such, model constraints describe properties that must be true at each moment in time, without necessarily determining how they are to be preserved. The number of potential valid instances of the specified model is diminished because the information present in the system at a certain moment in time, expressed by an instantiated model, must obey all model constraint rules. In formulating model constraints at the analysis level, only the aspect of '*what* properties must be satisfied by a model instance' is covered, thereby abstracting from '*how* these properties can be achieved' and '*when* they must be controlled'. These aspects should be deferred to a later phase of the software life cycle.

The specification of model constraints and business rules is not a major concern of most object-oriented analysis methodologies. Although the *Unified Modeling Language* (UML) [120][119][107][109][128] provides support for model constraint specifications through its *Object Constraint Language* (OCL) supplement [108][161], the integration of OCL with other UML model concepts is rather minimal. In fact, UML2.0 even excluded OCL from its core definition, and repositioned it as a separate add-on to UML. Model constraints are too often treated as a kind of non-formal or semi-formal documentation and comments rather than a distinct and important model concept on its own. When introducing model constraints, a large number of analysis methodologies present them as a kind of patch glue that can optionally be used to bring more consistency into the analysis specification, besides stressing that it is often either too obvious or too complicated and as such not needed in practice. When constraints are introduced in an analysis method, the interaction between the constraints and the object behaviour is often neglected. It is not clear in which manner an event that violates a certain constraint could be refused without putting the whole model instance in an invalid status. For instance, UML states that the condition of a constraint must be maintained as true. Otherwise, the system is invalid with consequences outside the scope of UML. In this way, constraints are not truly imposed on a model and its potential behaviour, but only serve as requirements validation rules for the model state at a certain moment in time.

The ways in which model constraints are introduced in the model differ from methodology to methodology, and even differ between different types of model constraints within a single methodology. Existing object-oriented analysis methodologies mostly use a mixture of different specification techniques. However, some techniques do not always reflect the importance of certain model constraint types, while others are rather improper notations that cannot be applied consistently to analogous cases. There exist different formalisms in which model constraints can be described, each having their distinct benefits and drawbacks. The importance of model constraints in a methodology is, in fact, reflected in its offered specification notation, which can vary from informal textual descriptions to treating model constraints as a first-class model concept.

In the remainder of this chapter, we categorise the different kind of model constraint specification mechanisms in a taxonomy, examine and compare them, and describe their appropriateness for conceptual modelling. We use the term *model concept* to indicate the concepts that are part of a methodology and defined on a meta-level, e.g., class, association or attribute, and the term *model element* or *model entity* to indicate instantiations of these concepts that are defined in the model, e.g., specific classes, associations or attributes.

2.2 Model Constraints versus Derivation Rules

The term *constraint* is a rather overloaded concept in computer science. Therefore, we give a definition of the term as it is being used in object-oriented analysis, in general, and in this text, in particular. Moreover, we try to position the concept of *model constraints* in object-oriented analysis, as a specification formalism for modelling restrictions, next to analogous concepts in the domain of logic programming and database design. Last, the difference between model constraints and Event-Condition-Action-Alternative (ECAA) firing rules is described.

2.2.1 Model Constraints in Object-Oriented Analysis

Constraints play a significant role in Object-Oriented Analysis [65][113][156][125][58]. In object-oriented analysis, a *model constraint* is a kind of *restriction* on the analysis model. Model constraints are declarative specifications of logical rules that must be fulfilled by each concrete instance of the analysis model, without specifying where and when the checks occur. In other words, model constraints are properties of an analysis model that must be satisfied by its instances at each moment in time. As a consequence, model constraints have an impact on the model state, diminishing the valid instances of an analysis model, as well on the model behaviour, forbidding certain model instance transitions. So, the main purpose of model constraints in object-oriented analysis is to maintain the validity of the instantiated model during the entire lifetime of a model execution or simulation.

2.2.2 Constraint Logic Programming

In Constraint Logic Programming (CLP), constraints are used to describe high-level computation and derivation rules [5][36][52]. Given a set of known (bound) domain values, the ultimate goal is to find proper values for a number of free domain variables, using a set of constraints as boundary conditions. So the specified constraints do not define the solution algorithm, but give a declarative specification of all conditions that must be fulfilled by a correct solution. The underlying system uses computational rules to derive suitable results for the free variables starting from the values of the bound variables and fulfilling all stated constraints. So, the purpose of constraints in CLP is to specify boundary conditions that must be valid during calculation of the free problem variables.

2.2.3 Database Constraints

Databases constraints can be divided in three main categories:

- Inherent model-based constraints, which are inherently present in the data model of the database. They express basic characteristics of relations that can never be changed.
- Schema-based constraints that can be expressed directly in the schemas of the data model, typically by specifying them in the Data Definition Language (DDL). Examples of such constraints are domain constraints, restricting the range of allowed attribute values, key constraints, expressing the uniqueness of a combination of attributes, constraints on null values, and entity and referential integrity constraints.
- Application-based constraints, such as semantic integrity constraints consisting of state and transition constraints. They cannot be expressed directly in the data model. This kind of constraints have to be expressed and enforced in the application logic that resides on top of the database, or by using a constraint specification language. In fact, these constraints will not belong to the database but will be realised throughout the application logic using the database.

Integrity constraints define whether a certain database state is a valid or invalid. Integrity constraints specified on a database schema are expected to remain valid on every database state of that schema. The Database Management System (DBMS) is responsible for ensuring that the integrity constraints are not violated. Entity integrity specifies that no primary key value can be a null value, while referential integrity specifies that a foreign key must refer to an existing tuple. Semantic integrity constraints can be specified using the *CHECK* clause for a table definition or a create assertion. These constraints will be checked only whenever a tuple is inserted or updated.

Database constraints are very closely related to analysis model constraints, since they are defined also on a database schema and pose constraints on the validity of each database state. However, a number of database constraint constructs already define the precise moments at which the constraints will be checked, hereby neglecting the fact that constraints are expected to be valid in every valid database state at any time.

2.2.4 ECAA Rules in Active Databases

Model constraints are quite different from Event-Condition-Action-Alternative (ECAA) rules [117][163], although there is a kind of similarity between them. Model constraints are general restrictions on the analysis model, restricting its potential model instances during the whole lifetime. This means that at each moment in time, the model instance must fulfil all specified model constraints. ECAA rules follow a totally different approach. ECAA rules are stimulus-response based. They specify actions to be performed automatically whenever particular events happen or specific conditions occur. The *event* part defines the set of operations that can serve as a

stimulus for the action to be fired. The *condition* part defines specific conditions that must be valid in order to activate the *action*, or the conditions that must be false in order to activate the *alternative* action. When a specific event occurs, several actions could be fired automatically based on the conditions specified in the ECAA rules. ECAA rules are triggered whenever a particular event is generated at the moment a specific condition is valid, regardless of how the event was generated. As such, it is a technique to inject additional behaviour into a model, without specifying the exact details about the places where the behaviour must be executed. It is even possible to omit the condition part of an ECAA rule, so that the action will be fired each time the specified event occurs. In the same manner, it is possible to omit the event part of an ECAA rule, so that an action will be fired each time the condition becomes valid irrespectively of the events that have led to this condition. UML2.0 defines a change event, which can be compared with a condition-action sequence of an ECAA rule.

Seen from a more process-oriented viewpoint, model constraints can be used to define *rule-constrained processes* while ECAA rules can be used to define *rule-based processes*.

- For rule-constrained processes, the potential processes are described independently from the rules that act upon them. Model constraint rules are then imposed on the business processes, limiting the outcomes and the allowed paths for these processes. Model constraints are unconditional, since they must remain valid at each moment in time for all instance models, regardless of the underlying process that created the instance model
- For rule-based processes, the processes themselves are defined throughout the application of the constraint rules. ECAA rules are interrelated, and ultimately define the final outcomes of the processes. This requires reasoning to be applied on the model, constructing the process given the events that occur, the conditions that are valid at a certain moment in time, and the triggers that will be fired. ECAA rules must not be valid at each moment in time for an instance model. In fact, whether a condition is valid or not at a certain moment is actually irrelevant for the instance model but it is only important for the construction of the underlying process. ECAA rules define conditions that are used to specify and extend a process, and execute additional functionality when appropriate, but do not allow to reason about properties of instance models.

Since ECAA rules can be used to describe the injection of additional functionality into the model at a certain moment in time, this mechanism is excellent to be applied in cases where a separation of concerns is needed, such as for the modelling of certain specific cases or exceptional situations within an object-oriented model. ECAA rules can introduce crosscutting behaviour into a model, and can therefore be seen as a kind of Aspect-Oriented Software Development (AOSD) [46] technique. We propose to apply a mixed approach, enabling the specification of both rule-constraint as well as rule-based processes. In order to achieve this, we enrich the rule-constrained processes using model constraints with rule-based processes using ECAA-like rules.

2.3 Example of the Library System

We will illustrate the different model constraint specification types with the running example of a library system. The informal description of the universe of discourse regarding the library system is as follows: '*The library system offers book copies that can be borrowed by its clients.*' Additional model constraints that must be valid in the universe of discourse are the following:

{1} The system may never lend books to persons that are not registered at the library.

{2} A person may never borrow more than one copy of the same book.

{3} A person may never borrow more than a specified number of books at the same time.

{4} A person may never borrow a book longer than a certain restricted period of time.

{5} When a person does not return the borrowed books within a certain time period, a fine must be raised. The size of the fine is dependent on the number of days overdue.

{6} A person may not borrow additional books while existing fines are still unpaid.

Notice that although the present tense has been used in the formulation of this informal description, it does not impose on the model how and when to perform the necessary checks and actions in order to keep its consistency. The informal description could also have been described in a past tense, which would suggest a more retroactive instead of a proactive reaction pattern.

{1} The system may only have lent books to persons that have been registered at the library.

{2} A person may only have borrowed no more than one copy of the same book.

{3} A person may only have borrowed a specified number of books at the same time.

{4} A person may only have borrowed a book for a restricted period of time.

{5} When a person has returned the borrowed books after a certain time period, a fine must have been raised. The size of the fine is dependent on the number of days overdue.

{6} A person may only have borrowed new books while all existing fines have been paid.

Although the rules only state what must remain valid, and not how and when checks are being performed in order to enforce the validity of these rules, the realisation in the actual software system must be as such that these rules are valid at any moment in time and can never be violated. There is a certain freedom of choice on how to actually implement constraint checks, but the software engineer is nevertheless restricted due to the fact that these properties must always remain valid.

2.4 Specification of Model Constraints using Informal Text

Most object-oriented analysis methodologies and notations only have informal support for specifying generic model constraints on the model instance structure. Moreover, certain methodologies such as OOA [28], RDD [167] and SM [134][133], neglect almost totally the importance of model constraints. Properties of the universe of discourse cannot be expressed explicitly, but have to be expressed in separate texts

as an additional part of the documentation set for the object-oriented analysis model, or as textual notes within the analysis model. Instead of incorporating the identification and specification of model constraints as a distinct part in the analysis phase, these methods consider them to be a minor point of interest for the model. This leads to a negligence of the important role of model constraints in the universe of discourse, and in its representation within the conceptual model. Although model constraints are generally of utmost importance, whether they express business rules, rules of logic, rules of physics, or human-defined laws and regulations, expressing them informally will never lead to the same amount of impact on the conceptual model as they have in the universe of discourse.

2.4.1 Constraints using Informal Text for the Library Example

We illustrate model constraint specifications using informal text with the running example of the Library System that was introduced in Section 2.1. In Figure 2.1, a UML model of the basic classes and associations is presented. This model is capable to capture the following facts:

- A number of copies of a certain book can be printed (association *print*).
- A library can possess book copies (association *possession*).
- A person can be registered at a library (association *registration*).
- A person can borrow a book copy (class *Borrowing* with associations *borrower* and *borrowed item*).
- A library can apply a fine for a person (association *fine*).
- The library can state the maximum lending period {4}, the maximum number of lending items for an individual borrower {3}, and the amount of the fine to impose for each day overdue {5} (attributes for class *Library*).

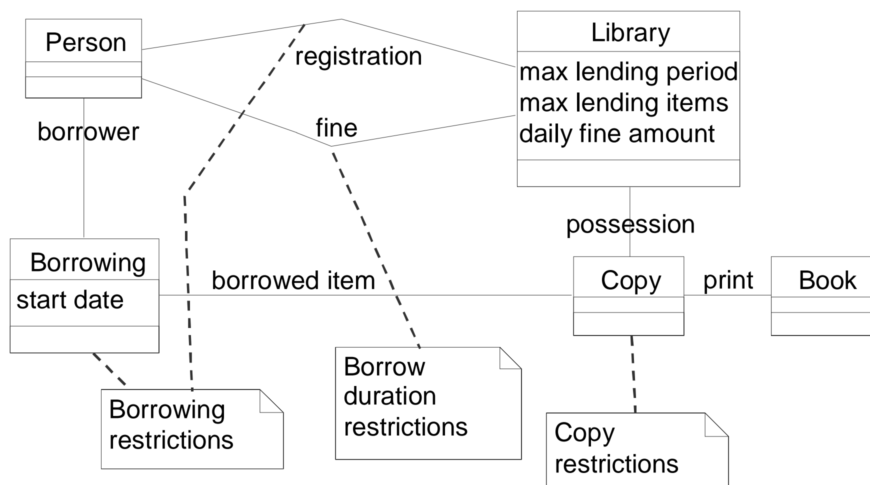


Figure 2.1: A Basic UML Model for the Library System

The informal model constraints that have to be added to this model are the following:

- Copy restrictions: [2]
  - A book copy can only exist if exactly one book is associated with it.
  - A book copy can be lent to no more than one person.
  - A book copy can be in possession of no more than one library.
- Borrowing restrictions:
  - A borrowing can only exist if one person and one copy are associated with it.[3]
  - A person that borrows a book copy must be registered at the same library as the one that is in possession of that book {1}.
  - A person may only borrow no more than one copy of the same book {2}.
  - A person may only borrow a certain number of books at the same time {3}.
  - A person may not borrow new books at a library while existing fines from that library are still unpaid {6}.
- Borrow duration restrictions:
  - When a person did not return the borrowed books after the maximum lending period, a fine must have been raised {4}{5}.

These informally specified model constraints can be attached to the model by means of textual comments in UML, although they will not have a specific semantic meaning for the model. In fact, a UML comment (called note in UML1.x) is just a notational element for rendering various kinds of textual information. The content of a note is merely basic text or a text document. Although textual clarifications inside a comment could potentially have a huge semantic impact on the UML model, this is not as such defined within the UML meta-model. Moreover, the allowed syntax for including comments and other notes within model entities is not further specified by UML. For the specification of model constraints, and partly also for the specification of functions and operations, the Object Constraint Language (OCL) [108][161], which is further discussed in section 2.6, could be used. However, UML does not oblige the modeller to specify all possible model constraints using OCL, nor does it make a distinction between textual notes and OCL notes.

2.4.2 Evaluation of Constraints using Informal Text

The usage of formal versus informal specifications is an actual controversy in the object-oriented analysis research area. One of the main reasons stated for using informal specifications in object-oriented analysis is to stimulate the creative process of analysis by avoiding to impose strict rules on the analysis process. As a consequence, a strict formal description of the outcomes of this process is rejected.

---

[2] In fact, UML associations provide means to express lower and upper bound values, which are called multiplicity values. We discuss multiplicity constraints in section 2.7.

[3] In fact, this is only true in the assumption that every borrowed copy gives rise to a new borrowing object. A single borrowing object could also contain more than one copy, grouping all book copies that have been borrowed at the same time.

On the other hand, Ambler [3] indicates ‘you can often learn more in five minutes drawing a diagram with your users than you can in five hours discussing it or reading about it in corporate manuals.’ We argue that, although the analysis process must keep its flexibility and creativity, its outcomes must be formal.

First, informally specified model constraints will never achieve to obtain the same amount of impact on the analysis model as the original constraints and rules have on the universe of discourse. Formal specification techniques are better suitable to impose such model constraints since they can enforce them explicitly on the analysis model instance. Model constraints that are attached to the analysis model by means of UML notes can easily be overlooked, diminishing their importance and their impact on the modelled universe of discourse. In the same way as classes, associations, attributes, and inheritance constructs are used to express knowledge in an analysis model, model constraints need and deserve suitable modelling support in order to be specified correctly and consistently.

A second reason to use formal model constraint specification techniques is the ambiguity of natural languages. Language in nature allows a certain degree of interpretation. Even when one tries to specify a rigorous textual explanation, misconceptions by the reader are difficult to avoid, whether it is not understanding the total set of restrictions that must be applied, reading more restrictions than intended to or misinterpreting the restriction. An informal specification is always exposed to human interpretation. This will often not correspond to the intention of the analyst who formulated it. Too much is left to the interpretation of the reader, whether the person is an expert of the universe of discourse, a model reviewer or a designer. When the design phase must start with an informal analysis description as a base, it will almost certainly be inevitable for errors to creep in, leading to a system that does not comply with its intended requirements. In case a formal analysis description with clear, well-defined semantics is produced, the following phases of the software life cycle have a reference model for its intended behaviour, and as such a solid base for the development of the software system.

As an example, the textual specification of the model constraints above does not say exactly whether a person can borrow a copy of the same book at two different libraries at the same time. In fact, the current formulation suggests that this situation is forbidden, although one could expect that the library system would apply such restrictions only locally. Although the modeller had a specific restriction in mind during the formulation of the model constraint, the underlying suppositions that seemed obvious and self-evident to the modeller were not captured by the textual specification of the model constraint. An outsider who receives such model and must try to comprehend its semantic meaning likely has a different background as the modeller, and will not be able to reconstruct all underlying implicit intentions. Because the textual model constraint specifications will never capture all implicit rules in a precise manner, the model specification is always ambiguous and will give rise to a false interpretation of the model. This will often lead to misconceptions and the introduction of logical errors into the ultimate system to be built.

A third reason to use formal model constraint specification techniques above informal ones is that verification techniques can be used to verify the obtained analysis model before going into system development. This will prevent logical errors and inconsistencies in requirements from the analysis phase on. Although the outcomes of the analysis process do not necessarily have to be directly executable, it will be a good thing to make them interpretable. As such, efficient checking, testing, and prototyping can be done at the analysis level. The UML community is also currently undergoing this evolution to executable UML models [99][141], which has led to the UML extension proposal regarding action semantics.

A fourth reason to use formal model constraint specification techniques is that they can be used as an input for further model transformations. The Model-Driven Development (MDD) [50][83] approach advocates semi-automatic model transformations, gradually introducing more detail and platform-dependency in the lower-level models. Models can only be used within an MDD approach when they contain their information in a formal notation that can be investigated, evaluated, and transformed into a different format. Even a straightforward implementation could be produced automatically in order to do a sort of simulation and rapid prototyping of the conceptual model. Therefore, formal descriptions of behaviour effects of the events and methods, and formal definitions of model constraints are advisable.

As a conclusion, we can state that a more precise and formal formulation of model constraints in object-oriented analysis is definitely necessary. Textual descriptions are inadequate as an analysis result. Different ways in which model constraints can be specified more formally are presented in the next sections.

2.5 Specification of Model Constraints using Operational Restrictions

More formal model constraints can be incorporated in the analysis model by regulating and controlling the allowed event occurrences, the sending of messages and the execution of methods. As such, model constraints can be enforced and unwanted model instance transitions can be prohibited. Most object-oriented analysis methodologies provide concepts to model such execution restrictions for events. By means of local execution restrictions on classes, e.g., using state transition diagrams for a class and preconditions for its methods, or global execution restrictions on the model, e.g., using interaction diagrams, the allowed occurrences and execution orders of messages and methods can be controlled, avoiding violations of the model constraints to be maintained.

2.5.1 Constraints using Operational Restrictions for the Library Example

We illustrate model constraint specifications using operational restrictions with the running example of the Library System as introduced in Section 2.1. The UML model

of the basic classes and associations as presented in Figure 2.1 will form the base model that is extended with operational restrictions. To realise all model constraints of the example through controlling the execution of methods, almost every method of each class will have to be controlled. Each class in the model can possibly contain a method that could give rise to a violation of a specified model constraint. Since a class has the right to change the association links[4] in which it takes part, each method of that class can be the cause of a potential model constraint violation due to a change of its association links.

2.5.1.1 Sequence Diagrams

A sequence diagram is a diagram that shows object interactions arranged in time sequence, indicating the objects participating in the interactions and the sequences of messages exchanged. The sequence diagram of the model constraint {5} concerning the fine is presented in Figure 2.2. It expresses the fact that a fine must automatically be generated when a borrowing is overdue.

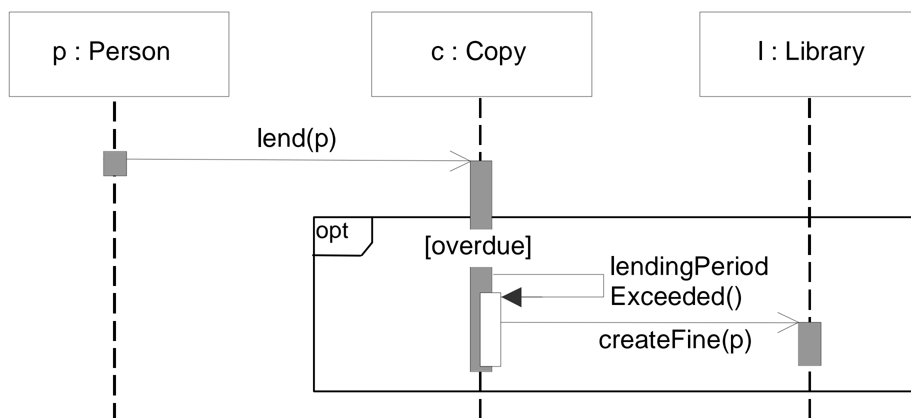


Figure 2.2: Sequence Diagram for the *Fine* Constraint Realisation

The interpretation of the diagram for the model constraint {5} is as follows:

- The first arrow indicates that a person starts a borrowing of a specific book copy. The person object sends the *lend* message directly to the book copy object. An alternative would be to model it as an event sent to the library indicating the book copy to be borrowed.
- The book copy object checks whether the stated lending period has not been exceeded. If the book copy is overdue, a '*lendingPeriodExceeded*' message will be generated to trigger the creation of a fine.

[4] A link is a tuple, an instance of an association containing an individual connection between 2 objects.

- When the '*lendingPeriodExceeded*' message is being generated, the book copy object sends a consecutive '*createFine*' message to the library object, indicating the responsible person of the violating borrowing as a parameter.

Another approach to represent the same model constraint concerning the fine is by the explicit introduction of a timer object as presented in Figure 2.3. As such, a start message is sent to a timer object whenever a new lending has started. The timer is then responsible to send a '*timeOut*' notification message back to the originator indicating the elapse of the requested time period. After receipt of the '*timeOut*' message, the '*lendingPeriodExceeded*' message can be generated and the fine can be created. When the book copy is returned on time, the timer must be stopped explicitly in order to avoid an erroneous overdue notification and consecutive fine creation.

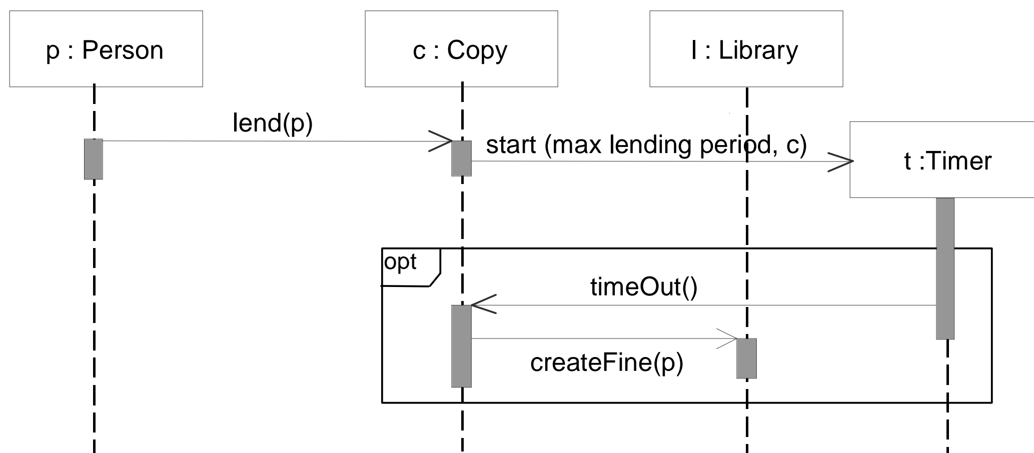


Figure 2.3: Sequence Diagram for *Fine* introducing an Explicit Timer Object

The sequence diagrams for the other model constraints stated in Section 2.4 can be built using a generic *receive-check-accept* specification pattern for constraint realisation. This pattern can be used whenever the validity of a message has to be checked, for example according to the state of the instantiated model at the moment of occurrence. This pattern can be seen as a kind of design pattern for testing a number of preconditions of a certain message before the message is actually executed. The interpretation of the sequence diagram for this pattern is as follows:

- The sender generates a message and passes it to the receiver. This message triggers the *receive-check-accept* pattern.
- The receiver checks whether the received message is valid or not according to the stated acceptance criteria. These criteria can amongst others be based on the current state of the instantiated model.

- When the acceptance criteria are fulfilled, the message will be accepted and processed by the receiver. If not, the message will be ignored.[5]

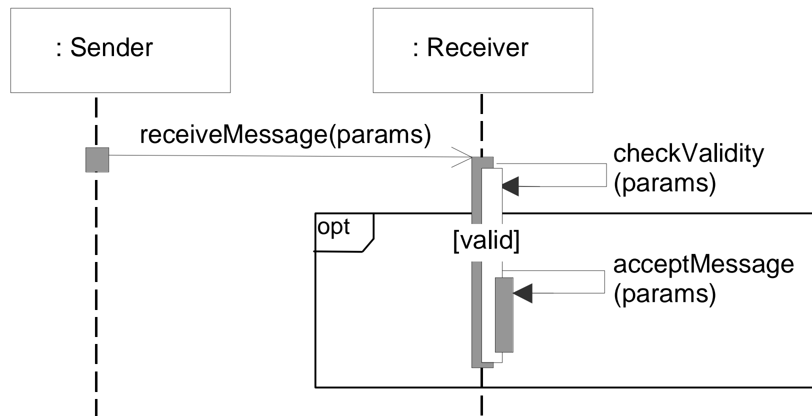


Figure 2.4: Sequence Diagram for the *receive-check-accept* Constraint Realisation

2.5.1.2 Statechart Diagrams

A statechart diagram [61][60][59][135], which actually is an extension of a Final State Machine (FSM) [2], is a diagram that shows a state machine for a class, indicating the sequences of states that an object goes through in response to events during its lifetime. The previous model constraints could also be expressed by means of statechart diagrams. A generic state diagram pattern applying the same *receive-check-accept* behaviour is presented in Figure 2.5. This pattern can be used as an alternative realisation whenever the validity of a message has to be checked. The interpretation of the state diagram for this pattern is as follows:

- Whenever an object is passive, thus able to receive a message, it is in the '*ReceivingMsgs*' state.
- When the object receives a message, the validity of the message is tested.
  - If the message is not valid, the object will ignore it (or send an error message if needed).
  - If the message is valid, the object accepts the message and performs the necessary actions to process the message correctly ('*AcceptingMsg*' state). Afterwards, the object returns to the '*ReceivingMsgs*' state.

---

[5] In case an error message has to be sent when an invalid message arrives, the pattern can easily be extended to incorporate such behaviour.

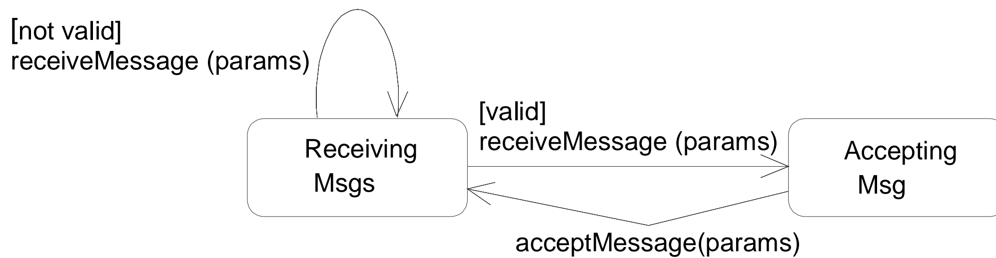


Figure 2.5: State Diagram for the *receive-check-accept* Constraint Realisation

2.5.1.3 Preconditions

A precondition is an expression for an operation that must be valid before the operation can be invoked. The stated model constraints can also be expressed by means of preconditions on the involved methods. Especially the methods manipulating the links of the involved associations and the values of the involved attributes are of utmost importance. The informal model constraints defined in Section 2.4.1 are expressed in Table 2.1 using preconditions in the Object Constraint Language (OCL) [108][161]. The OCL keywords are hereby indicated in bold.

Each OCL expression firstly defines its context in the UML model. This is a reference to an element in the UML model to which the OCL expression belongs. In the case of Table 2.1, the context defines the method to which the precondition belongs by means of defining the class, the method name, the parameters, and the return type. Hereafter, the corresponding precondition is expressed. Since this precondition applies for every method invocation of each object of the involved class, the object on which the method is being applied can be indicated by using the OCL keyword '*self*'. The OCL expression can be formed using basic values and types, logical and collection expressions, attribute values, association navigation, and query methods.

2.5.2 Evaluation of Constraints using Operational Restrictions

The approach of modelling constraints using operational restrictions by transforming the model constraints into action control constructs, whether they are interaction diagrams, state transition diagrams or preconditions, cause several problems for object-oriented analysis models. A first problem is the gap that is introduced between the universe of discourse and the resulting analysis model. Instead of describing *which* rules apply in the universe of discourse, the analysis model describes *how* they must be realised within the model. The model constraints themselves are not specified as such, but they are transformed into restrictions on messages and method executions for the involved objects.

A book copy can be lent to no more than one person

```
context Copy::lend (p : Person)
pre: self.borrowed item->isEmpty()
```

A book copy can be in possession of no more than one library

```
context Library::addPossession (c : Copy)
pre: c.possession->isEmpty()
```

A person that borrows a book copy must be registered at the same library as the one that is in possession of that book {1}

```
context Copy::lend (p : Person)
pre: p.registration->intersection(self.possession)->notEmpty()
```

A person may only borrow no more than 1 copy of the same book {2}

```
context Copy::lend (p : Person)
pre: p.borrower.borrowed item->select(copy |
        (copy.possession = self.possession) and
        (copy <> self)).print->excludes(self.print)
```

A person may only borrow a certain number of books at the same time {3}

```
context Copy::lend (p : Person)
pre: p.borrower.borrowed item->select(copy |
       copy.possession = self.possession)->size()
       < self.possession.max lending items
```

A person may not borrow new books at a library while existing fines from that library are still unpaid {6}

```
context Copy::lend (p : Person)
pre: self.possession.fine->excludes(p)
```

Table 2.1: OCL Specifications for Constraint Realisation using Preconditions

The way in which model constraints will be checked and maintained must certainly be specified at some point during the development life cycle, but at the design level rather than at the analysis level. The analysis phase must be centred on conceptual modelling, and should support a direct mapping of information from the universe of discourse into the conceptual model. Specifying model constraints by means of controlling method execution introduces a gap between the universe of discourse and the conceptual model. When constraints must be implemented using lower-level concepts instead of being treated as model concepts of their own, the conceptual model will no longer be a proper reflection of the universe of discourse. The specification of model constraints using operational restrictions is therefore a rather artificial and unsuited approach for conceptual modelling.

A second problem of specifying the model constraint realisations instead of the basic model constraints themselves concerns revisions and future modifications. The drawback of specifying model constraints using operational restrictions becomes

visible when the lifetime of an analysis model is taken into consideration. An analysis model is a development artefact that later will be subject to human auditing and maintenance. Since the model constraints are not present in the analysis model as individual entities but only through their realisation in terms of operational restrictions, it is very hard to preserve the consistency of these model constraint realisations. Each extension or change of the analysis model can influence the validity of the realised model constraints by introducing unforeseen and unwanted side effects.

For instance, if a new event must be introduced that can manipulate an association directly, such as transferring a book copy from one library to another, the diagrams that realise the operational restrictions have to be extended in order to incorporate the cases introduced by the new event. After any addition of a new event for a class, one is obliged to review the entire set of interaction diagrams to be able to maintain their correctness and consistency. Because these diagrams are already realisations of the model constraints, they have to be corrected after the slightest change within the model that has an impact on them, since every addition of an event can lead to a potential violation of the model constraints. In the case that preconditions are used for realising the constraints, every event that will be added to the model must take care that all constraints remain valid. The analyst must provide an additional precondition for each constraint that can possibly be violated by the new event. When a precondition is forgotten, the model will no longer enforce the intended constraints in a correct manner and, thus, will be incorrect. Therefore, it is very hard to maintain the consistency of model constraint realisations by means of operational restrictions during the software life cycle, which in nature will consist of several consecutive revisions, modifications, and adaptations.

When a model constraint is specified as a concept of its own, it will remain present as such in revised and modified versions of the analysis model. An analyst cannot break the realisation of the constraint, since it is present in the model as a single entity. When the model constraint is realised in an operational way, the semantic meaning of the model constraints is scattered around the whole model through a number of operation restrictions that enforce the constraint. It will be hard to assess the consequences of model additions and model changes on the model constraint realisation scheme. The analyst must perform a kind of reverse engineering activity by trying to extract the high-level model constraints from their lower-level realisation patterns. When adding or changing the model, the analyst must update the constraint realisations in order to preserve the implemented constraints. When model constraints would be formulated in a single place and could remain present in the model as a distinct concept, they would be highly visible and better comprehensible. Additions and changes to the model would not have a direct impact on the stated constraints, since their specification will remain unaffected.

As a conclusion, we can state that the analysis phase should be centred on the description of the universe of discourse, providing a direct mapping of it into a conceptual model. Although realising model constraints by means of operational restrictions results in a formal description of the model constraints, this approach is

actually inadequate for conceptual modelling. The specification of model constraints in a conceptual model should be formal, explicit, consistent, unscattered, and independent from issues regarding how they ultimately will be realised.

2.6 Model Constraints as a First-Class Model Concept

Model constraints should be treated as a first-class concept in an object-oriented analysis model, since they are of the same importance level as classes, attributes, associations, and events. To overcome the difficulties of transforming high-level constraints into low-level specification mechanisms and to get a more formal specification mechanism for constraint, a distinct notation and specification formalism for model constraints is needed.

When early object-oriented analysis methods were used in practice, people realised that there was a large need for having a more formal way of specifying model constraints in order to obtain consistency within analysis models. The Object Constraint Language (OCL) [108][161], which originated in 1995 within IBM, was presented as an add-on for UML in order to express model constraints and invariant conditions that must be valid for the system being modelled. Since OCL is a pure expression language, it does not have any side effects that can alter the state of the instantiated model. Although OCL can also be used to specify postconditions, it merely describes the state change that arises from a method execution instead of explicitly triggering the state change in the model.

UML constraints are attached to one or more model entities.[6] A UML constraint contains a Boolean expression in textual form (natural language, OCL, mathematical notation, programming language, et cetera.) that must be valid for each instance of a model. More precisely, the expression must always yield true when evaluated for the instances of the constrained elements at any time when the system is stable, i.e. after execution of an operation. Although constraints are introduced as a first-class model concept in UML, constraints do not have a specific graphical notation but use the comments symbol (🗋). The only difference between a constraint and a textual comment is that the string of the constraint expression is placed between curly brackets, e.g. '{constraint}'.[7] Notice that since the UML meta-class *Constraint* is derived from the meta-class *PackageableElement*, constraints can be given a name, although this feature is rarely used in UML.

---

[6] Constraints can also be attached to stereotypes or added to UML profiles. Such constraints do not have an impact on the instantiated model, but impose contraints on the model itself. These kinds of constraints belong to the meta-model and impose rules on the model regarding the correct usage of a stereotype or the allowed formulation of the model. Since this kind of meta-model constraints does not have a direct impact on the model instance, they will not further be treated in this text.

[7] Notice that UML does also provide some other specific ways of specifying constraints, such as the text in brackets following a single element or aligned with a dashed arrow from one element to another, expressing constraints on 2 elements.

2.6.1 Constraints as a First-Class Model Concept for the Library Example

We illustrate the specification of model constraints as a first-class model concept with the running example of a Library System as introduced in Section 2.1. The UML model as presented in Figure 2.1 will form the base model on which the model constraints will be added. The informal model constraints that were added to this model can be described in OCL as presented in Table 2.2.

Copy restrictions:

A book copy can only exist if exactly one book is associated with it

```
context Copy
inv: self.print->size() = 1
```

A book copy can be lent to no more than one person.

```
context Copy
inv: self.borrowed item.borrower->size() <= 1
```

A book copy can be in possession of no more than one library.

```
context Copy
inv: self.possession->size() <= 1
```

Borrowing restrictions:

A borrowing can only exist if exactly one person and one copy are associated with it.

```
context Borrowing
inv: ( self.borrower->size() = 1) and
     ( self.borrowed item->size() = 1 )
```

A person that borrows a book copy must be registered at the same library as the one that is in possession of that book {1}

```
context Copy
inv: self.borrowed item.borrower.registration
       ->intersection(self.possession)->notEmpty()
```

A person may only borrow no more than 1 copy of the same book {2}

```
context Copy
inv: self.borrowed item.borrower.borrower.borrowed item
       ->select(copy | (copy.possession = self.possession) and
        (copy <> self)).print->excludes(self.print)
```

A person may only borrow a certain number of books at the same time {3}

```
context Person
inv: self.registration->forAll(library | self.borrower.borrowed item
         ->select(copy | copy.possession = library)->size()
          < library.max lending items)
```

A person may not borrow new books at a library while existing fines from that library are still unpaid {6}

```
context Copy
inv: self.possession.fine->intersection(self.borrowed item.borrower)
         ->isEmpty()
```

Borrow duration restrictions:

When a person did not return the borrowed books after the maximum lending period, a fine must have been raised {4}{5}.

```
context Copy
inv: if now -self.borrowed item.start date) >
          self.possession.max lending period
     then self.borrowed item.borrower.fine
      ->intersection(self.possession)->notEmpty()
     endif
```

Table 2.2: OCL Specifications for Constraints as First-Class Model Concept

Although this approach allows a formal notation of model constraints as a first-class model concept, there is some arbitrariness introduced in the formulation of a model constraint, since the model constraint specification is described as an invariant starting from a specific class chosen between all involved classes. In order to abolish this asymmetry totally, a hierarchy between classes and associations could be introduced. One can then specify every model constraint starting from the highest class or classes of the hierarchy. Such approach has been developed in the EROOS methodology and is presented in Chapter 4.

2.6.2 Evaluation of Constraints as a First-Class Model Concept

The approach of treating model constraints as a first-class concept leads to a consistent, unambiguous, and formal model constraint specification. However, the cases that are not well supported by the specification technique of using constraints as a first-class model concept are threefold. First, a number of model constraints are very closely related to specific model entities, e.g., the cardinality constraints for the associations. Instead of separating such constraints from the model entity to which they relate, it is better to integrate these kinds of constraints in the model entity. On the one hand, this integration allows a better organisation of the model, creating more cohesion in the model by obtaining a clustering of closely related specifications. Instead of creating a model scattered with a lot of small and quite unrelated model entities, the entities should better be clustered together in a logical manner. On the other hand, this approach forces analysts to focus on these kinds of constraints whenever they introduce such model entities. By separating model constraints from the model entities to which they belong, one introduces the danger of overlooking these important constraints during the development of the analysis model. Integrating them into their related entities focuses the attention of the modeller on these constraints each time a model entity is defined.

Second, classes that are actually reifications of higher-level associations always lead to the specification of additional model constraints in order to specify all details concerning these associations. For example, giving the *Borrowing* class in the library

example, the model constraint on the existence of a person and a copy for a borrowing object expresses the fact that a borrowing is actually an association between a person and a copy object. They emerge due to the fact that a high-level association object (a *Borrowing* object) cannot exist without the knowledge of the participants it relates to (a *Person* and a *Copy* object). When the higher-level association is broken down into an association reified as a class having two assisting lower-level associations, for example the *Borrowing* class and the *borrower* and *borrowed item* associations, the characteristics must be enforced by means of explicit additional model constraints.

A final drawback appears when one object or association link is dependent on the existence of another object or link, e.g., the model constraint on the existence of a *registration* link in order to allow the existence of a *borrowing* object. This constraint expresses the existential dependency for a borrowing object on a corresponding registration link. This kind of constraints can be seen as model glue that keeps the model entities consistent with respect to the rules of the universe of discourse. When a flat, non-hierarchical analysis model is being developed, many structural dependency constraints have to be enforced explicitly, since they cannot be expressed in the model structure.

Such approach is favourable neither from the viewpoint of the model engineer nor from the viewpoint of the model reader, reviewer, or re-user. This is due to the fact that the structural dependencies between classes are only specified by means of additional model constraints and not by the model structure. The modeller has to make an explicit transition from the logical structure of the information within the universe of discourse to a different representation of it in the model. The reader of such loose model with many additional structural model constraints attached will have to put the pieces of the puzzle together before that person gets insight in the actual model structure. Instead of highlighting the basic structure of the model, one of the important elements of an analysis model, it is neglected and shifted to additional constraints. Moreover, it is possible for an analyst to construct a model without having to consider the structural constraints that are present in the universe of discourse. Since a good model should capture many constraints directly in its structure, a flat model structure is inadequate for conceptual modelling.

To conclude, we can state that the notation of constraints as a first-class model concept leads to a consistent, unambiguous, and formal model constraint specification. However, this approach has two important drawbacks. On the one hand, model constraints related to a specific model entity are better integrated with them in order to obtain a complete and consistent model, and retaining a better overview on the overall model. On the other hand, important structural dependencies become hidden in constraint specifications instead of forming the core of the model structure. Therefore, it is more appropriate to express certain kinds of model constraints directly in the model structure, reflecting the logic structural dependencies within the universe of discourse implicitly in the internal model structure.

2.7 Integration of Model Constraints in Existing Model Concepts

Most object-oriented analysis methods that incorporate constraints in their model in a formal and explicit manner, integrate them with other concepts of the method. For example, constraints concerning associations and attributes are integrated in the definition of the association and the attribute, while constraints about objects of a class are specified as part of the class description. The specification of such constraints is mostly restricted to a single entity of a model concept.[8] This can be very useful and suitable for certain types of constraints. However, other constraint types are forced into a single concept or a single model entity despite the fact that they can spread out over several of them.

A typical example of a model constraint that is integrated in an existing model concept is the definition of multiplicity constraints for association ends. This kind of constraint is almost always integrated in the association definition. Obviously, such constraint is a basic part of an association. Separating the multiplicity from the association end definition will introduce the danger of overlooking this important aspect concerning associations during the development of the analysis model. Such constraints are of utmost importance for the model entity on which they interact. If these constraints are not integrated in their related model entities, they can too often be neglected, which give rise to errors, misunderstandings, and deficiencies within the application.

Other examples of useful integration of model constraints in existing model concepts are the multiplicity of attributes, which indicated the possibility of having a single or many attribute values for an attribute of an object, attribute range restrictions, limiting the allowed range of an attribute between a lower and an upper bound, and the changeability properties of an attribute and an association end. For example, UML1.x allows defining the changeability of an association role or an attribute as *changeable*, *addOnly* or *frozen*. It is useful to specify whether certain attributes may only be defined at creation time of the object or can change during the life cycle of the object. For instance, the date of birth of a person may only be defined at the time of birth of that person and may afterwards never been changed. Also, the name of a person,[9] the account number of a bank account, and the approval date of a loan are examples of immutable attributes. On the other hand, the address of a person, and her or his length and weight are examples of attributes for which the actual value will vary during the lifetime of a person. The absence of a mutator for the attribute does not prevent any change to the attribute, because such mutator can always be added later to the model. Therefore, the property of attribute changeability has to be defined in order to prevent future changes.

---

[8] There is a slight difference between constraints that spread out over more than one concept and constraints that spread out over more than one entity of a single concept. An example of the former is a constraint that includes both an attribute and an association, whereas an example of the latter is a constraint that deals with link restrictions of more than one association.

[9] We are hereby neglecting the legal possibilities to change one’s name.

2.7.1 Integrated Model Constraints for the Library Example

We illustrate the approach of integrating model constraints in existing model concepts with the running example of the Library System as introduced in Section 2.1. Figure 2.6 presents the basic UML model of Figure 2.1 extended with attribute multiplicity and multiplicity constraints for the association ends.

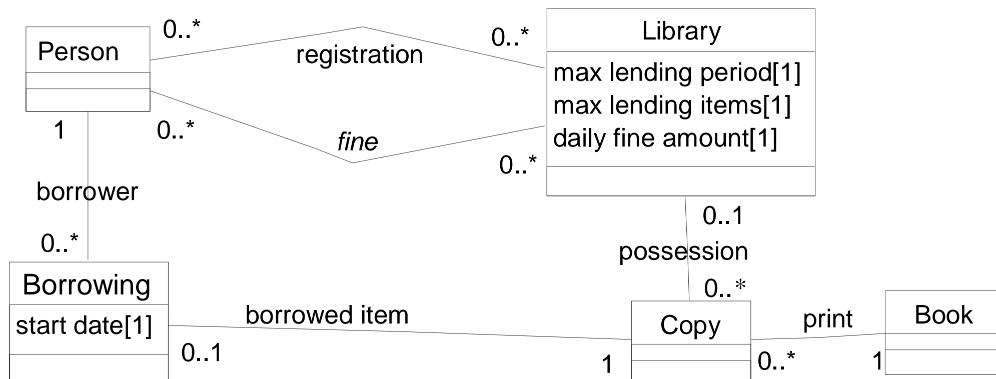


Figure 2.6: UML Model with Multiplicity Constraints for the Library System

The following model constraints have been expressed using integrated constraints:

- Association end restrictions:
  - A book copy can only be borrowed no more than once at the same time.
  - A book copy can only exist if exactly one book is associated with it.
  - A Book copy can be in possession of no more than one library.
  - A borrowing can only exist if one person and one copy are associated with it.
  - There are no restrictions on other association ends, e.g., a person can have several registrations at different libraries, or can even have no registration at all.
- Attribute restrictions:
  - A library has exactly one value for its attributes *max lending period*, *max lending items*, and *daily fine amount*.[10]
  - A borrowing has exactly one value for its attribute *start date*.[10]
- Changeability restrictions:
  - No changeability restrictions have been added to the model. This means that every association and attribute in the model is changeable. Due to the fact that the property of being changeable is the default value in UML, it cannot be stated explicitly. In contrast with this, the fact that an association or attribute is unchangeable after object creation can be stated with the stereotype *{frozen}* as the property string for the association end or attribute.

[10] Notice that a multiplicity of [1] is the default value for attributes in UML, expressing that each attribute must always have exactly one value.

However, the model constraints introduced in Section 2.1 cannot be expressed by means of model constraints integrated in a model entity:

- Model constraints {1}, {2}, and {6} deal with restrictions between several classes and associations. For instance, model constraint {1} involves classes *Library*, *Person*, *Copy,* and *Borrowing*, and associations *registration, borrower*, *borrowed item,* and *possession*.
- Model constraints {3}, {4}, and {5} deal with three associations, namely *borrower, borrowed item,* and *possession*, and a number of attributes, namely *max lending items* for {3}, and *max lending period* and *start date* for {4} and {5}. Model constraint {5} even involves the association *fine*.

The problem of integrating these constraints in a model entity is the selection of the most suitable model entity to integrate with. As an example, model constraint {3} can be defined in the classes *Person, Library* or *Copy*, but also in the associations *borrower, borrowed item* or *possession* or even in the attribute definition of *max lending items*. Constraint {1} is a typical example of a join constraint in an association ring. A join or anti-join constraint is a constraint that states 'if an object a1 of class A is connected through successive associations with a2, also of class A, then a1 must be equal to a2, respectively different from a2'. In this case, one could choose between four classes, namely *Person*, *Library, Copy*, and *Borrowing*, and four associations, namely *borrower, borrowed item*, *possession*, and *registration*. The choice between these alternatives will have to be made rather arbitrarily, since there is no good criterion to select one over another. No matter which one is chosen, this will lead to the introduction of arbitrariness and asymmetry in the obtained model.

2.7.2 Evaluation of Integrated Model Constraints

Constraints that easily can be integrated in existing model concepts only bear upon a single entity of a concept of the analysis method, such as a single association or a single attribute definition. However, if a constraint can spread out over several entities of the same concept, or, even worse, over several concepts, it is impossible to decently integrate the constraint in a single concept. Constraints that spread out over several associations cannot be placed consistently with one particular association. A method may decide to place rules between attributes of the association participants directly in the association definition, e.g., as in OMT [93][126]. But, for instance, rules between attributes of objects connected by two or more consecutive associations, or join and anti-join constraints in an association ring cannot be adjudged to a particular dedicated class or association.

These kinds of constraints spread out over a large part of the model instead of being localised to some instances of a single concept. When these constraints are integrated in a single class or in a single association, arbitrariness will have a huge impact on the model. We could have chosen an alternative viewpoint for placing and specifying the same constraint. It would even be hard to see that two constraints are actually identical when they are specified from a different viewpoint. In addition, the information distribution in the obtained model will be very asymmetrical. Useful

information concerning classes is hidden in the definition of other classes, associations or attributes. Another possibility, next to placing constraints in one particular class, is to place a copy of the constraint in every class that is influenced by it. However, this will give rise to an enormous amount of consistency problems and information duplication.

A second problem of inconsistently integrating constraints in model entities is that a bad placement of constraints in the model will lead to a diminishing of reuse capabilities. When constraints are scattered over the whole model and inappropriately integrated in rather arbitrary chosen model entities, it will be very hard to get a proper insight in the existing model structures and the superimposed rules. Such approach will encourage the analyst to rather start all over again from scratch instead of to reuse parts of the existing model. A separate notation mechanism for constraints influencing more than one specific model entity is therefore appropriate.

As a conclusion, we can state that some constraint types are strong related to existing model concepts. Therefore, it would be advisable to integrate them in the concept they belong to. However, a large number of constraints may be spread out over a variety of model entities and can therefore not be placed properly in a single entity. In such cases, a mechanism to specify constraints formally and explicitly, as introduced in Section 2.7, would be more appropriate.

2.8 Model Constraints Implied by the Model Structure

To diminish the gap between the logical information structure of the universe of discourse and the actual conceptual model structure, the expressive power of the model structure elements should be enriched. This can be done in several manners: through introducing new structural model concepts, through strengthening the semantics of the existing concepts or through defining strict usage rules for each concept. Two examples of constraints implied by the model structure, are (1) the change from a flat association structure to a hierarchical association structure, which treats associations as classes themselves, and (2) the obligation for class attributes to have a meaningful value at all times, thus excluding the null value.

Treating associations as first-class entities by reification of an association link into an object, introduces a hierarchical model structure based on existential dependency between objects. Analysis methods and notations, such as OMT [93][126], OSA [43], and UML [120][119][107][109][128], provide association classes that offer the possibility to model an association as a class (although they are often only used in exceptional cases and not supported by many UML modelling tools). On a meta-level, *AssociationClass* inherits from both *Association* and *Class*, and thus inherits all potential properties that can be defined for an association as well as a class. However, it is merely a technique of objectifying links in order to allow attributes to be specified for the link. Association classes are not treated as ordinary classes, with objects having their own identity, but remain, in essence, associations with class-like

properties. The identity of an object from an association class is defined in UML1.x as a combination of the identities of the objects taking part in the association link. Thus, an association class can be considered as a kind of a derived class, for which the derived objects are determined by the association link between the two associated objects.

Such approach has important consequences. For instance, duplicates are impossible, since it would introduce two association class objects having the same identity when they associate the same two objects. UML1.x explicitly states 'There are not two links of the same association that connects the same set of instances in the same way'. This applies for ordinary links as well as links of an association class. However, UML2.0 [108][128] has introduced the possibility of labelling the multiplicity of an association as a '{*bag}*', which allows the duplication of association links. Furthermore, an analyst already has to make a choice at the analysis level whether a relationship from the universe of discourse is going to be modelled as (1) a straight association, for which the analyst has the choice between an ordinary association, a qualified association,[11] an aggregation or a composition, (2) an association class, or (3) an association reified into a class having two assisting associations for linking the original participants. This choice will often depend on the fact whether association attributes should be expressed, whether duplication should be possible within the association, and whether other associations should be able to refer to the association links. Table 2.3 expresses the choices an UML modeller has to make and the criteria the modeller will mostly use. The different styles of modelling, presented in Figure 2.7, are from top to bottom an ordinary association, an association class, an association reified into a class, a qualified association, and an aggregation.

Link attributes	**Duplicate links**	**Link as participant**	**UML model entity most suitable**
No	No	No	Association
No	No	Yes	Association with other associations redirected to one of the participating classes (no * to * multiplicity)[12]
No	No	Yes	Association Class (* to * multiplicity)[13]
No	Yes	No	Association or Reified Association
No	Yes	Yes	Association Class or Reified Association
Yes	No	No	Association Class or Qualified Association
Yes	No	Yes	Association Class or Reified Association
Yes	Yes	No	Association Class or Reified Association
Yes	Yes	Yes	Association Class or Reified Association

Table 2.3: Criteria in UML when Modelling Associations

[11] A qualifier for an association end is an attribute whose values serve to partition the set of associated objects. They can be considered as attributes of the association link.
[12] In the library example, for instance, additional associations to the borrowing object can be redirected to the book copy object, since there is only a single borrowing object attached to a book copy object.
[13] When the association has a * to * multiplicity, an additional association to the link cannot be redirected to one of the objects participating in the link, since these objects can participate in more than one link.

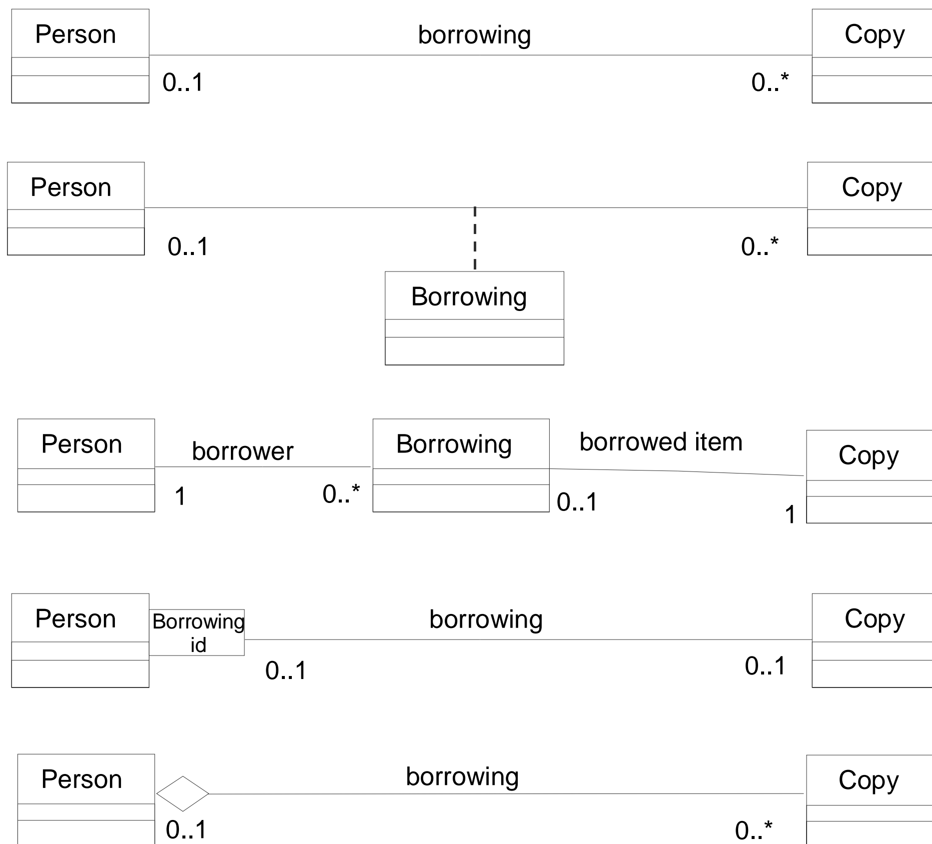


Figure 2.7: Alternatives in UML for Modelling Associations

Other approaches to express certain types of model constraint directly in the model structure, are the use of obliged values for class attributes and the modelling of dependencies between object states as explicit existential dependency associations.

- The obliged presence of an attribute value results in a reduction of alternative model variants for modelling certain facts, guiding the analyst to a clearer and more expressive model. Instead of modelling an attribute with a potential undefined value, an additional class must be introduced for such attributes. This newly introduced class represents the fact that an attribute value is actually present and defined at a certain moment.

- By explicitly modelling dependencies between object states as existential dependency associations, a better insight in these dependencies can be established. Since a large part of the model constraints deals with specifying state consistency and state dependency between objects, such constraints can be expressed directly in the model structure. By reifying states into classes, state consistency and state dependency constraints can be transformed into existential dependency constraints between reified state objects. In fact, reifying states into

objects results in more expressive and extendible structural models. State consistency and state dependency constraints can also be specified in statechart diagrams. However, their specification should best not be hidden inside a statechart diagram, but should be highlighted in the model structure.

2.8.1 Model-Implied Constraints for the Library Example

Although UML is not very suited to express a hierarchical association structure, it is possible to imitate such structure using association classes. The model of Figure 2.1 can be transformed into a hierarchical model using association classes as shown in Figure 2.8. By using existential dependency as the main criterion for specifying associations, the model structure can highlight the dependencies between objects. For instance, a borrowing object can only exist if a person is registered at a library, expressed by the association class *Registration*, and if a library is in possession of a book copy, expressed by the association class *Possession*. An analyst can easily express that a condition has to be fulfilled before a certain service can be requested, as indicated in the example where a person must firstly be registered at a library before that person can borrow a book. In order to obtain hierarchical model structures in UML, a modelling rule could be stipulated that obliges to transform all associations with multiplicity lower bounds of zero into association classes. This forces the modeller to reify most associations into association classes.

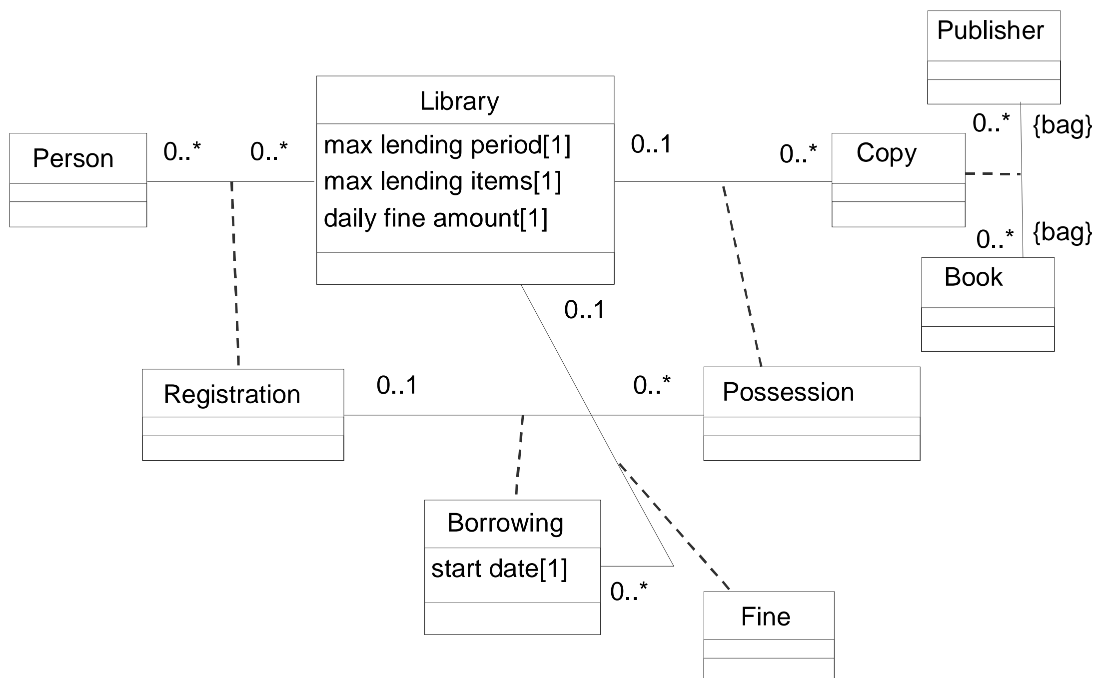


Figure 2.8: A Hierarchical Model for the Library System

The hierarchical model structure designates classes to different hierarchical dependency levels:

- Classes on level 0, namely *Person, Library, Book, and Publisher* in the example, are not directly dependent on any other class. Objects of these classes can come into existence without any additional restrictions.
- Classes on level 1, namely *Registration* and *Copy* in the example, are dependent on classes of level 0. In fact, since every object expresses an association link between two other objects, an object of level 1 is existentially dependent on two objects of level 0. As an example, a copy link object cannot exist without a book object and a publisher object.
- Classes on level 2, namely *Possession* in the example, are dependent on two other objects of a lower level, namely a copy object of level 1 and a library object of level 0.
- Classes on level 3, namely *Borrowing* in the example, are dependent on two other objects of a lower level, namely a possession object of level 2 and a registration object of level 1.
- Last, classes on level 4, namely *Fine* in the example, are dependent on two other objects of a lower level, namely a borrowing object of level 3 and a library object of level 0. A fine link object expresses the fact that a fine can only be given to a borrowing that is overdue. Notice that this additional condition for the fine object, namely the fact the borrowing must be overdue, is not yet expressed in the presented model. In fact, an additional constraint must be added to the model in order to express this condition. The existential dependency only forbids fines to exist without arising from a borrowing.

The definition of the '*{bag}*' property string in UML2.0, namely that 'the association end represents an object collection that permits the same element to appear more than once', and the fact that the bag property is associated to the association end instead of the association, indicates that its usage is more directed to the specification of implementation issues rather than the modelling of the association property. Therefore, associations should better be made first-class entities in UML instead of being both an association and a class at the same time. By a true encapsulation of every association into a class of its own, the choice to model a certain relationship of the universe of discourse as a direct association, an association class or an association reified into a class will disappear, since it would always be modelled as an association encapsulated into a class. The decision whether an association encapsulated into a class will be implemented by means of a class or an ordinary association could be deferred to the design phase.

2.8.2 Evaluation of Model-Implied Constraints

The specification of model constraints implied by the model structure provides a number of advantages. On the one hand, information dependencies in the model become clearly highlighted. Since constraints are expressed directly in the model

structure, the information dependencies from the universe of discourse play a core role in the corresponding conceptual model. When important constraints can be implied by the model structure, the logical structure of the universe of discourse is directly reflected in the conceptual model structure. This allows people to get better and faster insights in the information represented by the analysis model, thereby easily obtaining a view on the information structures in the universe of discourse.

A second advantage is that the number of constraints that has to be added to the model will diminish, since a number of these constraints will already be expressed in the model structure. As an example, model constraint {1} is directly expressed in the model structure that is represented in Figure 2.8. The association class *Borrowing* expresses that a borrowing can only occur by a person that has been registered at the library. The registration participant for the association captures this fact, expressing that a borrowing object is existentially dependent on a registration object, which, in turn, is existentially dependent on a person and a library object.

The representation of existential dependency should best not be restricted to binary associations, expressing that a link object is dependent on 2 association objects. It should also be possible to express that a link object is dependent on only a single other object. This can currently be simulated in UML using an ordinary association between the dependent object and the object on which it depends, having a multiplicity lower bound of 1 at for the latter. An alternative representation could be a reification of a 1-tuple represented by a unary association into an object. Although UML offers both binary and n-ary associations, it does not allow the direct modelling of an unary association. In addition, since UML associations have been extended in order to allow duplicate links, it should also be possible to constrain the number of duplicate links that are allowed to exist at the same moment. In Section 4.3.7, we propose the concept of unary associations and duplication occurrence constraints in the EROOS methodology as a solution for these UML restrictions.

Existential dependency among objects may seem too restrictive for the ultimate software system to be built. A large deal of run-time flexibility, e.g., in populating the model with instances, would be lost. However, object-oriented analysis is basically concerned with building an abstraction of the universe of discourse, expressing information, facts, and dependencies present in the universe of discourse without considering how to express this information in the system at run-time. Therefore, focusing on the universe of discourse in its normal appearance should have priority over the unavailability of information to the system at run-time.

As a conclusion, we can state that the specification of model constraints implied by the model structure diminishes the gap between the logical information structure of the universe of discourse and the corresponding model structure, since the constraints are highlighted directly in the model structure. However, UML and other analysis methods do not fully support such high-level structural concepts in their notation. Therefore, the expressive power of the model structure concepts should be enriched in order to obtain methods that can produce suitable models expressing the constraints from the universe of discourse directly in the model structure.

2.9 Comparison and Conclusions

In this chapter, we have presented a taxonomy for model constraints. We have shown that model constraints can be specified in a number of manners, more specifically:

- as informal text, expressing the constraint in natural language as an informal addendum to the model specification,
- as operational restrictions, realising the constraint using method execution control,
- as a first-class model concept, introducing model constraints as a basic building block of an analysis model,
- integrated in existing model concepts, specifying a model constraint in the definition of the model entity on which it applies,
- and implied by the model structure, using existential dependency, obliged attribute values, and reified object states in order to enrich the model structure.

We have argued about the advantages and disadvantages of each approach, using the gap between the logical information structure of the universe of discourse and the corresponding model structure as the most important criterion. A summary of the advantages and disadvantages of the discussed specification techniques for model constraints is presented in Table 2.4.

After comparison of the different approaches for model constraint specification, our conclusions are the following:

- Specifying constraints as informal text is too informal as an outcome of the analysis phase. This will give rise to the introduction of human interpretation errors during later stages of the development.
- Specifying constraints explicitly by operational restrictions is useful during the design stage, but too low level on the analysis level. Such approach is not advisable because it introduces a huge gap between the universe of discourse and the analysis model. Instead of describing which rules apply in the universe of discourse, the analysis model describes how they are enforced. In addition, constraints must always be converted from their conceptual meaning to their operational implementation and vice versa.
- Constraints can be considered as independent model entities, and, in general, need to be modelled as a first-class model concept. As such, the importance of constraints in an analysis model is highlighted to the right extent. However, other constructs are sometimes better suited in certain cases. First, constraints closely related to certain model entities should better be directly integrated in these elements in order to achieve a clear focus on these constraints during analysis. Second, existential dependency and other structural model constraints should be expressed directly in the model structure instead of being specified as independent constraints. Instead of highlighting the basic structure of the model, the structure would be neglected and hidden into the specified constraints.

Constraint specification	Advantages	Disadvantages
Informal text	• Expressivity of natural language	• Limited impact on model • Imprecise descriptions • No verification possible
Operational restrictions	• Formal technique • Clear insight in places where to check constraints	• Low-level specification mechanism • Gap between analysis model and universe of discourse • Model extension and revision problems
First-class model concept	• Consistent, unambiguous, formal and general applicable model constraint description	• Gap between model entities and related constraints • Unsuited for reified associations • No reflection of logical domain structure
Integrated in model concepts	• Focus on specific constraint types in the concept definition • Useful for constraints on a single model entity	• Arbitrariness in constraint placement • Improper description when constraints spread out over several model entities • Limited reuse
Implied by model structure	• Model highlights logical structure of the universe of discourse	• Change in logical structure has huge impact on the model structure

Table 2.4: Overview of Specification Techniques for Model Constraints

- Constraints closely related to certain model entities, such as attribute and association end multiplicity, attribute range restrictions, and general changeability properties of attributes and associations, could easily be integrated in these model entities. However, when constraints can spread out over several model entities, it is not advisable to integrate them in a single entity. This leads to asymmetry and arbitrariness in the constraint specification.
- Existential dependency and other structural model constraints should best directly be implied by the model structure. A hierarchical association structure can capture existential dependency constraints implicitly in the model structure. This reduces the number of additional constraints, and highlights and

incorporates the logical structure of the universe of discourse directly in the corresponding analysis model.

UML does not provide suitable support for specifying constraints in the right manner in an analysis model. The expressive power of the UML model structure must be enriched in order to obtain suitable conceptual models expressing the structures from the universe of discourse directly in the model structure. In Chapter 4 and Chapter 5, we propose two versions of the EROOS methodology, the EROOS kernel and the EROOS universe, that define suitable notations and formalisms to specify a large number of constraints from the universe of discourse directly in the conceptual model structure.

Chapter 3

Key Principles for Conceptual Modelling

Before we present the EROOS methodology, we first propose the key principles for conceptual modelling that have led to certain methodological decisions in EROOS, and provide arguments as to why these principles are of utmost importance during conceptual modelling in order to obtain the most suitable models.

3.1 Principle of Uniqueness

The *Principle of Uniqueness* states that every fact from the universe of discourse must result in a unique model element in the corresponding conceptual model. There should exist no alternatives in modelling facts from the universe of discourse in order to avoid different conceptual models that are somehow equivalent. Instead, the model concepts offered by a methodology should be such that the analyst is guided from the universe of discourse to be modelled to the most appropriate conceptual model that represents these facts.

Although this principle is lacking in current analysis methodologies, we argue that it is of utmost importance to incorporate this principle in an analysis methodology. On the one hand, the analyst has a huge and difficult task of mapping the information from the universe of discourse into an appropriate conceptual model. It should not be the responsibility of the analyst to decide on choosing an appropriate style for modelling certain information in the conceptual model. Instead, the analyst should primarily focus on deciding whether or not facts related to the universe of discourse are relevant according to the requirements that must be incorporated in the model. A good analysis methodology should offer conceptual support for the transformation of

facts from the universe of discourse into a suitable conceptual model, having clear criteria in leading the analyst toward the most appropriate model concept. It is the task of the methodology to investigate and evaluate all potential alternative modelling concepts and notations that can be used in expressing certain facts, and to force the analyst in using the most suitable concept.

In addition, the Principle of Uniqueness leads to a true standardisation on the analysis level. It is not enough to standardise the notation, as the Object Management Group (OMG) has done with the Unified Modeling Language (UML) as a general software development notation. There are still a number of flavours or personal preferences possible in UML for expressing certain information. As an example, we have identified in Section 2.8 that an association can be represented as an ordinary association, a qualified association, an aggregation, a composition, an association class, and even an association reified into a class. This creates confusion for the analyst during the development of a conceptual model about which concept to choose in order to model certain information. In addition, the intention of the analyst using a specific model concept must be reconstructed during model revisions. Considering the lifetime of a conceptual model and the number of people involved in creating, extending, adapting, and reviewing these models, it is beneficial in terms of time and complexity of having a single model for a single universe of discourse when models are frequently passed between people.

Notice that the Principle of Uniqueness is also known under different names, amongst other, as *construct redundancy* [160], in which a type of facts within the universe of discourse can be represented by more than one modelling construct, or as *No-Choice* [38].

3.2 Principle of No Redundancy

The *Principle of No Redundancy* states that every single information item that is represented in a conceptual model must have a distinct added value of its own, and should not be derivable from the other items present in the conceptual model. Each fact from the universe of discourse should directly be reflected in the conceptual model by means of a model entity that can be traced back to the universe of discourse.

Models incorporating a large degree of redundancy are much more difficult to keep consistent than models without any redundancy. In addition to consistency problems that can arise from the mapping of the universe of discourse to the conceptual model, model redundancy introduces additional consistency problems within the conceptual model. This creates an extra level of complexity inside the conceptual model.

Another disadvantage of model redundancy is that the information captured in a model is more difficult to grasp by model readers, reviewers, and re-users. Instead of focussing on the information in the model, they have to filter the model in order to detect and reduce the redundant information, or are puzzled about the difference

between 2 model entities that actually represent the same information. A model without redundancy is easier to comprehend than a model containing redundant information.

Notice that the Principle of No Redundancy is also addressed as *uniqueness* [38].

3.3 Principle of Unambiguity

The *Principle of Unambiguity* states that each conceptual model element must result from a distinct fact in the universe of discourse. There should exist no two different situations in the universe of discourse that result in the same conceptual model element.

When a conceptual model can be interpreted in many ways, it can mentally be mapped back onto different situations in the universe of discourse. In this manner, a single conceptual model can express different realities. If this is the case, the conceptual model is ambiguous about which facts from the universe of discourse are actually covered by the model. This can cause confusion, misunderstanding, and misinterpretations. To reduce this source of confusion and diminish the threat of discrepancy between the required and the actual delivered software system, a conceptual model element should be traceable to a single and unique fact from the universe of discourse.

Notice that the Principle of Unambiguity is also addressed, amongst other, as *unambiguous* [62], or as *construct overload* [160], in which the same concept represents several types of facts from the universe of discourse.

3.4 Principle of Completeness

The *Principle of Completeness* [172] states that all relevant information from the universe of discourse must also be reflected in the conceptual model. A conceptual model cannot reflect a certain universe of discourse when a number of facts are not represented in the model. If some facts are not described explicitly, and they are only present in the mind of the analyst or domain expert, the conceptual model is not complete, and can lead to errors, misunderstandings, confusion and arbitrary decisions during later stages of the development process. Although it is acceptable that certain technical elements of the solution domain are not expressed in a conceptual model, the universe of discourse should be modelled to its full extent.

When incomplete conceptual models are used as a kind of sketch of the universe of discourse, details but also important or even crucial information could have been omitted. It is conceivable, that a software engineer faced with missing information, will neglect certain important elements, or give her or his own personal interpretation that can differ from the facts within the universe of discourse. When important

development decisions are based on imprecise, incomplete, and non-exhaustive information, this can lead to serious problems and failures in the system being developed.

The Principle of Completeness could be compared with *construct deficit* that have been defined by Wand [160], which states that a fact from the universe of discourse cannot be represented by any modelling construct. Although construct deficit implies incompleteness, incompleteness does not imply construct deficit, since incompleteness does not necessarily arise from the fact that it is impossible to model certain facts from the universe of discourse. These facts could somehow be omitted by the modeller.

3.5 Principle of Minimalism

The *Principle of Minimalism* states that only the relevant information in the universe of discourse must be reflected in the conceptual model. A conceptual model should not contain any surplus or irrelevant information that cannot be connected to the universe of discourse to be modelled and the requirements for the actual system. When information cannot directly be linked to a relevant knowledge fact within the universe of discourse, it is superfluous and should be omitted from the conceptual model. An analyst should be aware of the boundaries of the universe of discourse and should not try to model unimportant or unrelated facts.

Software engineers are often inclined to anticipate on a large number of potential future extensions to the system to be built, or to construct an oversized system that can be reused in other applications operating within the same or a related universe of discourse. This is also postulated by the agile software development community and expressed in the Agile Manifesto [11][98] as the principle of '*Simplicity is Essential*'. It is the task of the analyst to construct a complete conceptual model of the whole universe of discourse, while it should be constricted within the boundaries of the universe of discourse.

Notice that the Principle of Minimalism is also addressed as *Abstract* [62], as *Abstraction* [38], as *Pertinency,* as *Noise,* or as *Parsimony.*

3.6 Principle of Preciseness

The *Principle of Preciseness* states that all facts and information of the universe of discourse must be modelled in a formal way using suitable concepts that are offered for this purpose by the supporting analysis methodology. No text elements or notes in natural language should be part of the conceptual model without having a corresponding formal model representation of the facts they intend to express.

As previously argued in Section 2.4, formal specifications are preferable over informal, textual specifications in a conceptual model due to unambiguity, the possibility of performing model validation and verification, and the impact that can be achieved on the other model elements [34].

3.7 Principle of No History

The *Principle of No History* states that the resulting conceptual model must be independent of the order in which the facts from the universe of discourse have been modelled. The conceptual model should only be dependent on the total set of information from the universe of discourse that has to be modelled, and not on the order in which these pieces of information have been added to the model. A conceptual model should be a representation of the universe of discourse, and should not represent any history information concerning the construction of the model.

In fact, the Principle of Uniqueness that was stated above, already implies the Principle of No History, since if only a single conceptual model can result from a set of facts from the universe of discourse, it definitely cannot contain any history information concerning the construction of the conceptual model. When history information concerning the construction could have an influence on the resulting model, the same set of facts from the universe of discourse can lead to a variety of models depending on the order in which the facts have been modelled, which is in contradiction with the Principle of Uniqueness.

Nevertheless, we find it important to stress the Principle of No History as a distinct conceptual modelling principle. Since it is possible that an analysis method does not comply with the Principle of Uniqueness, such method could in addition be assessed regarding its compliance with the Principle of No History.

3.8 Principle of Model-Implied Constraints

The *Principle of Model-Implied Constraints* states that constraints arising from rules and regulations in the universe of discourse must be reflected in the structure of the conceptual model. This means that the concepts offered by an analysis methodology should be able to express these important constraints directly in the model structure. In addition, information in the universe of discourse that is dependent on other kind of core information as a prerequisite for its existence, should as such be reflected in the conceptual model. This means that the model entity expressing the conditional information should also be modelled as being dependent on the model entity that presents the core information.

The reason behind this principle is to reflect and preserve the implicit structures and the existential dependency relations from the universe of discourse in the core

structure of the resulting conceptual model. We have extensively argued in Chapter 2 why this principle is of utmost importance for conceptual modelling.

3.9 Principle of Abstraction

The *Principle of Abstraction* states that complex information in a conceptual model, due to the intrinsic complexity of the universe of discourse, must be presented in its full detail in the resulting conceptual model. However, a conceptual model can offer model views in a more abstracted form for the ease of the model reader.

Producing abstract views on the universe of discourse should not be one of the main concerns of conceptual modelling, since a conceptual model must be able to capture the universe of discourse in its full detail. However, for interaction with customers and end users, it can be useful to build summary models and condensed views on the possibly complex overall conceptual model.

When abstract views on a conceptual model are not considered to be a basic part of the conceptual model, the Principle of Abstraction is not contradictory to the Principle of Uniqueness and the Principle of No Redundancy as presented in Sections 3.1 and 3.2. Abstract model views are not considered as true alternatives for modelling the universe of discourse, but merely offer a condensed and more understandable view on the conceptual model.

3.10 Additional Considerations

Extendibility and correctness with respect to the universe of discourse are sometimes postulated as important issues for conceptual modelling. However, we have not incorporated them in our proposed set of key principles for conceptual modelling.

3.10.1 Extendibility in Conceptual Modelling

Devos [38] proposes *extendibility* as one of the principles for conceptual modelling. The proposed principle states 'It must be possible to extend a model with a set of real-world facts without modifying existing specifications.' However, we argue that this principle is inadequate for conceptual modelling due to a number of reasons.

First, this proposition could be contraproductive in realising the Principle of Uniqueness and Model-Implied Constraints. In realising the Principles of Uniqueness and Model-Implied Constraints, a methodology tries to guide the analyst to a unique model that expresses all essential constraints in its core model structure. If one wants to adhere to the proposed principle of extendibility, the consequence is that the core model structure that was constructed during the first modelling iteration cannot be altered anymore to reflect the additional information that must be captured during the second modelling iteration. This leads to the situation in which the information from

the second iteration is modelled using unintegrated model structures that are artificially connected with the model structures that were specified during the first iteration. Since a conceptual model should be a unique representation of the universe of discourse containing all dependencies in its core model structure, it should be possible to review the originally developed model structures when additional information must be added to the model during the second iteration.

Second, this proposition is in contradiction with the Principle of No History when being applied to conceptual modelling. On the one hand, according to the proposed principle of extendibility, models must be extendible without having to alter previously defined model elements. But on the other hand, according to the Principle of No History, models must be independent of the order in which elements have been added to the model. It is only possible to satisfy both principles at the same time when model elements are fully independent from each other. In this manner, additional model elements would not have an impact on previously defined model elements. However, model elements in conceptual models are often heavily interrelated and dependent on each other. It is therefore almost impossible to add new model elements while at the same time keeping them isolated from existing model elements.

Therefore, we consider the proposed principle of e*xtendibility* as inadequate, and do not adopt it as a key principle for conceptual modelling.

3.10.2 Correctness in Conceptual Modelling

Although correctness is a sound principle to strive for, it is very difficult to achieve in practice. Correctness can be situated on two levels, namely (1) external correctness, which is the correctness of the conceptual model in relation to the universe of discourse, and (2) internal correctness in the model.

Regarding external correctness, Ludewig [94] points out that this can never be achieved completely. Every person has a distorted view on the world. In order to approach correctness, we have to improve our models constantly through a comparison with the reality. Whenever the conceptual model and the universe of discourse do not agree, the reality is always right and the model is always wrong.[14] It is therefore impossible to prove that a model is correct in relation to the universe of discourse. One can only prove that a model is incorrect. A model is correct as long as no evidence to the contrary can be provided. The goal of conceptual modelling is thus to achieve a conceptual model that is the best estimation of the universe of discourse given the facts and information that we know, and for which no evidence to the contrary can be provided.

[14] In the case of Business Process Re-engineering (BPR), the opposite is true. The newly defined business process is the ultimate goal that must be realised by enforcing it in reality.

Internal correctness, also called *Consistency*, is a necessity for any specification formalism. Concerning the conceptual modelling, the modelling methodology must state the methodological rules and guidelines to which a model must adhere. Furthermore, by complying with the principle of No Redundancy, sources of internal incorrectness can be removed since the redundant information must no longer be kept consistent with its counterparts. The principle of Preciseness enables model verification and validation, since formal notations can be interpreted and checked on internal correctness.

Chapter 4

A Methodological Kernel for Conceptual Modelling

The development of the EROOS[15] conceptual modelling methodology[16] was led by the conclusions of our study on model constraint formalisms in object-oriented analysis in Chapter 2, and the key principles for conceptual modelling that were proposed in Chapter 3. The EROOS methodology wants to guide the analyst to a unique conceptual model for a specific universe of discourse. In the EROOS methodology, constraints play a crucial role in the modelling process. First, EROOS introduces the usage of existential dependency as the main criterion to determine the core model structure, thereby expressing model constraints implicitly in the EROOS model structure. Second, the impact of model constraints on every model concept is carefully considered, integrating model constraints in certain model concepts when appropriate. Third, model constraints in EROOS are treated as a first-class model concept linked to the model entities that they affect.

Chapter 4 and Chapter 5 provide a detailed description of the EROOS methodology. We propose two version of the EROOS methodology: A core version in this chapter, the EROOS kernel, in which information can only be added to the conceptual model instance, and an extended version in Chapter 5, the EROOS universe, in which additional support for recurrent EROOS kernel analysis patterns is provided through advanced and more practical concepts, using the core version as the underlying base.

[15] EROOS was originally an acronym for 'Entity-Relationship Object-Oriented Specifications', but is currently considered to be a proper noun.
[16] A part of the work presented in this and the following chapter has been published in [154], [153], [150], [90], [91], [142], and [143].

4.1 Model, Model Instance, and Event Instance

The main goal of conceptual modelling is to develop a model of the universe of discourse in which the ultimate system will actually operate. The resulting *conceptual model* expresses the analyst's perception of the universe of discourse, and serves as the major means of communication between the customers, which are the prime contractors having ordered the system, the end users, the experts on the universe of discourse, and the software engineers responsible for the actual development of the system. The goal of conceptual modelling is to capture all facts and knowledge from the universe of discourse into a conceptual model that will serve as a reference for it, specified using the concepts offered by the analysis methodology. A model is a meta-representation in the sense that it does not contain any specific facts from the universe of discourse at a particular moment in time. A model only describes potential structures that can exist between elements in the universe of discourse. It is a description of all capabilities of the universe of discourse without describing any instantaneous exposure within the universe of discourse.

However, it is possible to represent a snapshot of the universe of discourse, which is the description of an actual situation in the universe of discourse at a particular moment in time, using a *model instance*. A model instance contains specific objects, each having their own properties and concrete relationships with other objects. A model instance expresses information about a concrete situation in the universe of discourse. A model instance can only contain information that is situated within the boundaries of the allowed structures as defined in its conceptual model.

A model instance can remain valid for a certain period. However, at any moment in time, a model instance can change due to a set of events that occur in the universe of discourse. Since the model instance is a representation of the information in the universe of discourse, any occurrence that causes a change of certain information is reflected by a change of the model instance representing this information. The set of concrete events that cause a transformation of a model instance is indicated as an *event set instance*. The definitions of Model, Model Instance, Model Instance Universe, Event Set Instance, and Event Universe can be found in Definition 4.1.

A **Model** is a set of methodological concept instances (classes, attributes, relations, etc.) representing potential information structures in the universe of discourse. It is a meta-representation describing all potential situations that can occur in the universe of discourse.

A **Model Instance** is an instantiation of a model at a particular moment in time, representing a concrete situation in the universe of discourse at that time. Although a model instance is a representation of a situation at a particular moment in time, it can remain valid for a certain period.

The **Model Instance Universe** is the collection of all model instances that can exist at a certain moment. It is the set of all potential instantiations of a model.

An **Event Set Instance** is a concrete set of events that defines a transition at particular moment in time from an existing model instance, which was valid until that moment, to a new model instance, which will become valid starting from that moment. The new model instance is obtained by adding information to the existing model instance.

The **Event Universe** is the collection of all events that can occur in a model. It is the union of all events that already have occured in the past and all events that could occur in the future.

Given

Model M; Model Instance Universe MIU; Model Instance MI;

Event Universe EU; Event Set Instance E;

M^{17} = (MIU, MI, OU, EU, E, t, M_{ccl}, M_{acl}, M_d, M_a, M_{br}, M_{ur}, M_{ct}, M_p, M_q, M_{co}, M_g) | $M_{cl} = M_{ccl} \cup M_{acl} \wedge M_r = M_{br} \cup M_{ur}$; (model structure)

MI: TIME → MIU | $\forall$ t $\in$ TIME: $MI_t \subseteq MI_{t+1}$ (growing model instance)

E: TIME → $\mathcal{P}$(EU) (model behaviour)

t: MIU x $\mathcal{P}$(EU) → MIU | $t(MI_t, E_{t+1}) = MI_{t+1}$ (new instance)

Definition 4.1: Model, Model Instance, and Event Instance

4.2 Classes, Objects, and Static Classification

When modelling facts and knowledge from the universe of discourse into a conceptual model, information will be represented in the model as *objects*, also called *instances.* An object is a model representation of an observable fact in the universe of discourse. An object does not necessarily map to a tangible item in the universe of discourse, but can also map to an abstract item, a piece of knowledge, or an important occurrence in the universe of discourse. An object can represent any kind of thing that comes into existence at a certain instance of time as a representation of a fact that arises in the universe of discourse. In the EROOS kernel, an object can only be created. The EROOS universe offers advanced concepts in which an object can also cease to exist at a certain moment in time. This will lead to the possibility of object destruction, which is treated in section 5.1. Objects are distinguished from values, whose lifetime is infinite, i.e., a value has always existed in the past and will always exist in the future while an object comes into existence at a specific moment in time. An event is an occurrence that can be observed in the universe of discourse in connection with the appearance of a fact, or the creation of an item.

We introduce the notion of a *class* as a concept for structuring the set of objects into collections of objects having the same properties. In this manner, all properties common to a collection of objects can be specified once for the whole class. In

[17] The different elements that compose a model will be presented in the next sections.

EROOS, a class is a static classification of a collection of objects with the same properties, defining a specific type for the objects that is shared by all objects in the collection. Classes constitute the basic building blocks of a conceptual model in EROOS. All model elements that introduce additional properties, such as relations, attributes, and constraints are defined on top of one or more classes.

In developing conceptual models, both structural and behavioural aspects must be considered. The structural aspects of a conceptual model comprise a specification of how the universe of discourse might look like at a particular moment in time. The behavioural aspects specify how the universe of discourse may evolve in the course of time. All elements of a conceptual model will describe the structural aspects of the information that can be contained in a model instance, and the behavioural aspect of how a model instance can be transformed into a consecutive model instance.

From a structural point of view, classes serve to cluster objects with the same properties. Instead of having to define every single object that can come into existence in the analysis instance model, a class allows to define a skeleton description for an object only once as a part of the conceptual model, whereupon a number of objects can be introduced in the instance model based on this unified description. The structural aspect in the definition of a class is confined to the clustering of objects and the introduction of a single name to indicate the whole cluster. The behavioural aspect in the definition of a class consists of the specification of an event, indicated as a *creation event,* which creates a new object inside the model instance as the representation of an occurrence in the universe of discourse.

This section further describes the properties concerning classes and their objects in detail, together with the model constraints implied by the class concept. Hereafter, the different aspects involved in the introduction of a class are surveyed, namely the structural and behavioural aspects in terms of initial functionality for the class, and the rules to be obeyed when introducing a new class. Last, the EROOS classes that can be identified in the running example of the library system are presented.

4.2.1 The Population of a Class

In human life, it is common practice to cluster elements with similar properties in order to perform a classification of elements as part of a mental model. Such classifications are a means to master complexity by clustering elements, finding commonalities between these elements, and identifying common relationships between them. This clustering will be reflected in a conceptual model by means of classes. The characteristics and behaviour common to the collection of objects associated with a class can then described once and for all at the level of that class. In contrary with a set, which is a static and fixed collection of elements that cannot be changed anymore, an EROOS class is a dynamic collection of objects, since at all times new objects can be added to a class. At each moment in time, a class collection corresponds to a fixed set of objects that are part of the collection at that moment. The collection associated with a class is named the *population* of that class. The process of defining classes is named *static classification*. Figure 4.1 presents a population of a

class at a particular moment t, having fifteen objects within the class population at that moment.

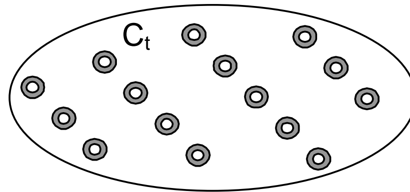


Figure 4.1: Objects in an EROOS Class Population at Moment t

4.2.2 Model Constraints implied by the Class Concept

EROOS incorporates important model constraints directly in its methodological concepts. The following constraints are directly implied by the class concept:

- **Disjunctness**: Different concrete classes in EROOS are assumed to divide the universe of objects into disjoint collections. In other words, each object is and will always be associated with a single concrete class. Different concrete classes are not allowed to share any of their objects.[18]
- **Immutability**: The bond of a given object with its class is static. In particular, at the moment an object is created, it is bound to a class and it will keep that bond for its entire lifetime. Objects cannot switch from one class to another.
- **Finiteness**: The collection of objects associated with a class is finite.
- **Uniqueness:** Whenever a new object of a class C is created, that object will be different from any already existing objects of the class C or of any other classes. From its creation on, each object will have its own and unique object identity [82] so that it can be differentiated from all other existing objects. The object identity is encapsulated, but can be observed using equality and inequality:
  - Given two expressions e_1 and e_2, both expressions referring to an object, the assertion '$e_1 = e_2$' is true, if and only if both expressions refer to the same object, i.e., the identity of the objects referred to by e_1 and e_2 is the same.

  This means that, although all external observable properties concerning two objects can be the same, the objects can still be totally different in nature due to their unique object identity.

The definition of a class can be found in Definition 4.2.

[18] Notice that the implied constraint of disjunctness only holds for concrete classes. Section 4.6 introduces abstract class that do not comply with the implied constraint of disjunctness.

A **Concrete Class** is a model entity defining, at each moment in time, a disjoint object population set, which is element of the corresponding model instance. This population set can only be extended over time.

The **Object Universe** is the representation of the entire collection of objects that can exist for a model. It is the union of all objects that came into existence in past model instances and all objects that could come into existence in future model instances.

Given

Model M; Object Universe OU; Concrete class $C, C' \in M_{ccl}$;

$C: TIME \rightarrow \mathcal{P}(OU) \mid$ (finiteness)

$\forall\ t \in TIME: C_t \subseteq C_{t+1}$ (immutability)

$\forall\ t,u \in TIME: C_t \cap C'_u = \varnothing$ (disjunctness)

Definition 4.2: EROOS Kernel Class

4.2.3 Specification of an EROOS Class

The definition of a class in EROOS is given in a class script and comprises both structural and behavioural aspects. From a structural point of view, classes serve to cluster objects with the same properties into disjoint collections. The structural aspect in the definition of a class is confined to the introduction of a single and proper name for the objects. The name of a class must be an expression in some natural language, rooted in some culture, and refer to a cultural entity. The name for a class must reflect as good as possible the entity in the universe of discourse that it is supposed to model. There is no need to restrict oneself to English names, nor to ASCII or even to the Roman alphabet. Classical computer-oriented issues, such as the use of digits as a first character or the use of spaces in a name, are purely technical and of no importance in a conceptual model. For readability purposes, EROOS class names have to be represented as singular nouns and in uppercase. Moreover, all classes introduced in a conceptual model must have different names in order to be distinguishable.

A class script also includes behavioural aspects. Each class must at all times be complemented with a specification of the events in which its objects can become involved. When a new class is defined, the functionality to be introduced is restricted to the specification of a single creation event. A creation event is an event by means of which a new object of the given class comes into existence and is added to the population of that class. It reflects the fact that, in the universe of discourse being modelled, a new object with an identity distinct from the identity of any already existing objects has come into existence. The specification of a creation event in a class script is limited to the definition of a proper name for the event, which must reflect as good as possible the occurrence in the universe of discourse that it is

supposed to model. For readability purposes, EROOS event names have to be non-conjugated verbs, i.e., verbs in infinitive form, and represented in small letters. All events for a single class must have different names.

An event in EROOS occurs at a specific moment in time and it is instantaneous. An event thus has no duration. Whenever the duration of a creation event should be considered as a period, because its duration is important for the universe of discourse, this fact should be modelled as two objects, each having its own distinct creation event. The composite creation event must as such be split in two basic creation events, namely a first one to express the start of the creation activity, and a second one to express the end of the creation activity. Notice that the modelling of an occurrence can be dependent on its relevance in the universe of discourse. Whereas in one model a certain occurrence, whose duration is considered to be irrelevant, is modelled as an event of a single object, that same occurrence could be modelled as two distinct events belonging to different objects as soon as aspects of its duration turn out to be important. The EROOS approach concerning modelling events allows the analyst to make a clear distinction between, on the one hand, the modelling of an event or occurrence that is instantaneous, which can be represented by a creation event for a single object, and, on the other hand, the modelling of an activity lasting for a certain period, which must be represented by two creation event representing the start and end of the activity.

In EROOS, all elements of a conceptual model can be represented both textually and graphically. Whereas the textual representation fully defines all details concerning a certain model element, the graphical representation provides a condensed view on the element. The structural and behavioural aspects involved in the definition of a class are defined in a class script, as given in Table 4.1. As presented in Figure 4.2, a class is graphically represented in the form of a rectangle with the name of the class specified inside. The definition of a creation event can be found in Definition 4.3.

```
<EROOS kernel class script> =
"class" <CLASS NAME>
  "creation event"
    <creation event name>
"end class" <CLASS NAME>
```

Table 4.1: EROOS Class Script

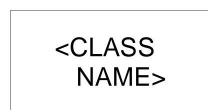


Figure 4.2: Graphical Representation of an EROOS Class

A **creation event** is an event of a class that, if applied on a model instance at a certain time, adds a new object to the object population set for that class.

Given

Model M; Object Universe OU; Event Universe EU;

Event Set Instance E; Concrete Class $C \in M_{ccl}$;

Creation Event $c_1,\ldots,c_n \in EU$;

$c_1,\ldots,c_n \in E_{t+1} \Rightarrow \exists\ o_1, \ldots, o_n \in OU$:

$(o_1, \ldots, o_n \notin C_t) \wedge (C_{t+1} = C_t \cup \{o_1, \ldots, o_n\}$ (Uniqueness)

Definition 4.3: EROOS Creation Event

4.2.4 EROOS Kernel Classes for the Library Example

Given the example of the library system that was presented in Section 2.3, the following EROOS kernel classes can be identified: PERSON, DEATH (of a person), LIBRARY, DISSOLUTION (of a library), REGISTRATION, DEREGISTRATION, BOOK, DISMISS (a book that has been taken out of print), PUBLISHER, LIQUIDATION (of a publishing company), COPY, DESTRUCTION (of a book copy), POSSESSION, WRITE-OFF (of a book possession), BORROWING, RETURN (of a borrowing), FINE, and PAYMENT (of a fine). Since an EROOS class script do not provide any additional information next to the definition of a creation event name for the class, we only give the EROOS class script for the class of REGISTRATION in Table 4.2 as an example.

```
class REGISTRATION
  creation event
    register
end class REGISTRATION
```

Table 4.2: EROOS Class Script for the Library Example

4.2.5 Contributions, Related Work, and Reflections

The EROOS class concept is largely comparable with the class concept in UML. Our contributions concerning the class concept are the following:

- The **constructional model approach,** in which model instances can only grow and information can only be added to a model instance, is a crucial property of the EROOS kernel in achieving the Principle of Uniqueness. Objects cannot be destroyed, but instead the destruction of an object must be reified into the creation of a distinct object representing this destruction event.

- The methodological approach using **instantaneous events** obliges the modeller to split an occurrence with a relevant duration into two model events. This allows a proper guiding of the modeller to a unique conceptual model for the universe of discourse to be modelled.

An observation that can be made is that the application of the EROOS kernel methodology gives rise to a class model with a huge number of classes. In order to model activities in the EROOS kernel, the activity must be split into two events characterising the activity, namely the start of the activity and the end of the activity. A large part of the knowledge in the universe of discourse has the form of an activity, in which, for instance, a temporal validity or verity can be seen as an activity starting at the moment the fact becomes valid and ending at the moment the fact expires or becomes invalid. Therefore, most information is modelled using two related classes, in which a first class indicates the start and a second dependent class indicates the end of the activity or validity. The EROOS universe, presented in Chapter 5, will offer advanced modelling concepts to merge these two objects into a single object that represents the whole period and has both a creation event and a destruction event.

A second observation concerns the synchronicity between the occurrences in the universe of discourse and the events in the conceptual model. There is sometimes a need for a clear definition of the exact moment in time when a model event must be activated, e.g., the moment that the person signs the application form in the example of the library registration.

4.3 Attributes, Domains, Values, and Decoration

In addition to objects, which capture facts about events occurring in the universe of discourse, *values* are used to describe specific properties and relevant information related to these events. As such, classes can be decorated with attributes, allowing objects to have specific attribute values containing relevant information regarding their occurrence. *Attributes* will be introduced in this section to model properties in terms of relationships involving objects and values. Attributes are said to *decorate* classes, providing a description of the relevant information that is related to the object and its creation event. When an attribute is defined for a class, each object of the decorated class will have to be associated at all times with a value of the proper *domain*, which describes all values of a specific type.

The introduction of an attribute decorating a given class must always be complemented with decoration functionality. This forces the analyst to specify the impact of the decoration on the creation event, in addition to the static properties underlying the decoration of a class. The creation event introduced in the class script must be extended with specific information regarding the attribute, establishing a binding of the new object with a proper value of the corresponding domain. Notice that the EROOS kernel does not allow changing any information that is captured inside a model instance. Only additional information can be added to a model

instance. However, a change of an attribute can be modelled by introducing a change object, reflecting the fact that an attribute value has been changed in the universe of discourse. In addition to the extended creation event, the decoration functionality also automatically includes an implicit *attribute query* returning the information contained in the attribute, which can be used for later retrieval.

First, this section introduces the differences between values and objects, and the general principles underlying the decoration of a class. The model constraints implied by the attribute concept, as well as the structural and behavioural component of an attribute script, are described thereafter. Third, the default attribute *Creation Timestamp* and the implicit attribute query '→' are introduced. Last, the attributes that can be identified in the running example of the library system are presented.

4.3.1 Value Domains

In EROOS, objects are clearly distinguished from values. Any kind of thing that could come into existence at a certain moment in time, can be modelled as an object. Contrary to objects, values are information descriptions from the universe of discourse for which the lifetime is considered to be infinite. Consequently, whereas objects of a class are to be created explicitly using a creation event, EROOS assumes that values of a domain have always existed in the past and will always exist in the future. Objects capture facts and information about events in the universe of discourse, whereas values are used to describe specific properties and relevant information related to these events. The basic differences between objects and values lead to different concepts for modelling them. Whereas objects having similar properties are clustered into classes, domains are introduced for describing values of the same type using type descriptions. Domains include a specification of functions that can be applied to its values, expressing calculations or equations that can be applied to them.

EROOS defines a number of commonly used domains as part of the methodology. A conceptual model developed in EROOS can include additional domain specifications, defining values belonging to the given domain and functions applicable to these values. However, not just any set of values can be used in decorating classes. A newly introduced domain must either fulfil the rules of a basic domain or must be composed from already existing valid domains. Each domain will need to specify (1) a mapping to a mathematical set, (2) a standard domain unity specified for the unity element of the set, (3) a set of additional domain units and their mapping to the domain unity, and (4) a set of functions that map values of certain domains to other values of the same or other domains. We define 4 categories of domains in EROOS:

- **Magnitude domains** describe physical magnitudes in the universe of discourse. Examples of such domains are:
  - The domain of *MASS* offers a set of values for expressing the weight of material objects. Values of this domain can be expressed in different units, such as grams, kilograms, pounds, and tons. The standard unity of the domain is *gram (g)*.

– The domain of *LENGTH* offers a set of values for expressing the size of material objects. Values of this domain can be expressed in different units, such as metres, kilometres, yards, foots, and miles. The standard unity of the domain will be *metre (m)*.

– The domain of *TIME* offers a set of values for expressing moments in time. Values of this domain can be expressed in different ways using a specific era as a kind of unit. An era is a system for dating events from a specific reference point in time. This reference point refers to a particular event or moment in history, such as the birth of Christ or the coronation of the last Japanese Emperor. The standard unity of the domain will be *Common Era* (CE), also called Anno Domini (AD). In addition, the domain of TIME offers a dynamic function, denoted as **now**, which returns the current time at the moment the expression is evaluated.

– The domain of *DURATION* offers a set of values for expressing the duration of certain activities. Values of this domain can be expressed in different units such as milliseconds, minutes, hours, days, and years. The standard unity of the domain will be *second (s)*.

– The domain of *TEMPERATURE* offers a set of values for expressing the temperature of objects and fluids. Values of this domain can be expressed in different units, such as degrees Celsius, degrees Fahrenheit, and degrees Kelvin. The standard unity of the domain will be *degree Celsius (ºC)*.

– Other domains include *CURRENT,* having *ampère (A)* as standard unity, *RESISTANCE,* having *ohm (Ω)* as standard unity, and *LUMINOSITY,* having *candela (cd)* as standard unity.

All magnitude domains at least include addition, subtraction, and comparison as functions applicable to their values, in addition to multiplication and division, that will result in a value of a composed domain, as explained further on.

- **Reference domains** describe textual references in the universe of discourse. These reference domains are represented using strings, i.e., sequences of Unicode characters. A particular domain may impose restrictions on the alphabet it covers, and on the length of the sequences it supports. All reference domains will therefore be defined as subsets of the all-embracing domain of STRING, offering all sequences of characters of any length that can be represented in Unicode.

- **Denomination domains** describe dedicated pricing and money references in the universe of discourse. Although the currency conversion between distinct denominations is actually undefined, currency conversions are often performed using a predefined conversion scheme or using an approximation of the actual conversion rates defined by the global money market at a particular moment in time. Currencies defining a domain are for instance EUR VALUE (euro as the unity), USD VALUE (US Dollar as the unity), and GBP VALUE (GB Pound as the unity). Due to the fact that it is impossible to define a static transformation between all available currencies, the currencies are not units of an encompassing

domain of VALUE, but instead define their own distinct domain having currency-specific denominations.

- **Composed domains** describe values obtained by applying mathematical calculations on a number of values from several domains. Although, in principal, the obtained values could be represented by couples, e.g., (10m, 20m) as a value of LENGTH x LENGTH, a composed value describes only the end result of the calculation, e.g. 200 m^2 as a value of the composed domain $LENGHT^2$.
  - The composed domain of *VOLUME* offers a set of values for expressing dry and liquid measures of capacity. Values of this domain can be expressed in different units, such as litres, pints, and gallons. The standard unity of the domain will be *litre (l)*. Notice that the domain of VOLUME is actually a composed domain, since it is equal to $LENGHT^3$, and the unity of litre is equal to $decimetre^3$.

Notice that it is forbidden in EROOS to use Boolean and integer attribute types. Although such attributes are commonly used in object-oriented analysis, design and implementation, we claim that there is a better way to represent the information contained in a Boolean and integer attribute. In EROOS, the analyst is forced to reify a Boolean and integer attribute into a class, thereby explicitly modelling the facts that are concealed behind these attributes. A Boolean attribute expresses a fact that can be true or false. This can also be represented using a relation participation, in which a participant object can be participating in a relation link (true) or not (false). Another representation could be a static subdivision into two specialization classes (true and false class). An integer represents a number as the outcome of a specific count. EROOS forces the analyst to explicitly model the elements that have been counted, instead of modelling it in a summary version using a count attribute.

In addition, attributes can only represent a single domain value in EROOS. If there is a need to model multi-valued attributes, the analyst must reify the multi-valued attribute into a set of object, each having a single attribute value attached.

4.3.2 Attribute Values

Associations involving objects of a class and values of a domain are treated differently from associations involving objects only. Whereas the latter are modelled by means of relations, associations involving objects and values will be modelled by means of attributes. Attributes are restricted to model binary associations. The structuring of the objects of a class C decorated by an attribute involving values of a domain D, is at each moment in time a function from C to D, as illustrated in Figure 4.3. In order to emphasise the difference between objects, which have internal properties, and values, which do not have internal properties, values are shown using a textual representation. The defined characteristics for objects resulting from the decoration of their class C are represented by lines connecting the decorated objects with values of the decorating domain.

Permanent binding of an object with an attribute value may seem to be too restrictive for the ultimate system to be developed. These values are often unknown to the software system, or must be measured in order to obtain an approximation of the value. However, object-oriented analysis is basically concerned with building an abstraction of the universe of discourse, expressing information present in the universe of discourse without considering how this information can be obtained by the system at run-time. Therefore, focusing on the information from the universe of discourse in its normal appearance should have priority over implementation issues concerning the software system. Implementation issues regarding observations and measurements of attribute values are described by Fowler [49].

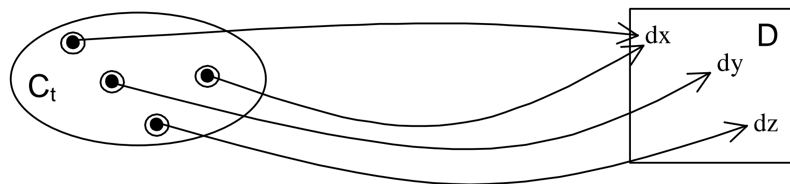


Figure 4.3: Objects decorated by an Attribute of Domain D at Moment t

4.3.3 Model Constraints implied by the Attribute Concept

EROOS incorporates important model constraints directly in the methodological concepts. The following constraints are directly implied by the attribute concept:

- **Permanent binding**: Each attribute decorating a given class implies the permanent binding of every decorated object with a specific value of the decorating domain. This means that each object of the decorated class must at all times be associated with a value of the decorating domain. If it should be the case that certain objects could exist without being associated with such a value, it is clear that these objects do not share the attribute as a common property for the class. In such case, an additional class must be introduced, expressing the fact that an attribute value has been associated to an object of the original class.
- **Immutability**: The attribute association of a given object with its value is considered to be static. In particular, at the moment an object is created, it must directly be bound to a value of the proper domain, and it will keep that bond for its entire lifetime. Objects cannot switch from one attribute value to another during their lifetime. If it is needed to change an attribute value, an additional class must be introduced expressing the fact that an attribute value has been changed.[19]

The definition of an attribute can be found in Definition 4.4.

[19] In the EROOS universe, an attribute value can be changed into another value of the attribute domain.

An **attribute** is a model entity defining a property for a class for which, at each moment in time, every object of the class, called a decorated object, must be associated with a specific value of the domain defined for the attribute.

A **domain** is a collection of values that refer to static and unchangeable properties in the universe of discourse. A domain can be a magnitude, reference, denomination, or a composed domain.

Given

Model M; Class $C \in M_{cl}$; Attribute $CA \in M_a$; Domain $D \in M_d$;

$CA: TIME \rightarrow (C_t \rightarrow D) \mid$ [20] (permanent binding)

$\forall t \in TIME: CA_t \subseteq CA_{t+1}$ (immutability)

Definition 4.4: EROOS Kernel Attribute

4.3.4 Specification of an EROOS Attribute

The specification of an attribute decorating a class is represented in an attribute script. This script introduces the definition of the attribute along with an extension of the creation event. Since every object must at all times be associated with a specific attribute value of the proper domain, the creation event must define how the associated value will be determined. This is done by (1) defining a parameter for the creation event by which the proper attribute value can be set, or (2) by defining a default domain value for all objects of the class.[21] The syntax of an attribute script is given in Table 4.3. As presented in Figure 4.4, an attribute is graphically represented in the form of an ellipse containing the attribute name, attached to the corresponding class. The definition of an extended creation event can be found in Definition 4.5.

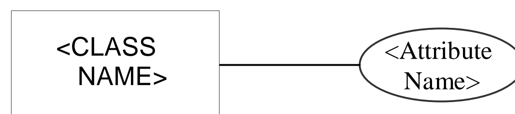


Figure 4.4: Graphical Representation of an EROOS Attribute

[20] As an alternative, the definition can be given as:
$CA: TIME \rightarrow (\mathcal{P}(OU) \rightarrow D) \mid \forall t \in TIME: dom(CA_t) = C_t$ or as $CA: C \rightarrow D$.

[21] Since attribute values cannot be changed in the EROOS kernel, defining a default value for an attribute is not very meaningful since all objects of the class would share the same value in such case. Default values are more useful in the EROOS universe, where objects can change their attribute values over time.

```
<EROOS kernel attribute script> =
"class" <CLASS NAME>
  "definition"
    "decorated by" [ "unique" ] "attribute"
        <Attribute Name> ":" <DOMAIN NAME>
        [ "constrained by" [ <lower bound> ( "<" | "≤" ) ]
          <Attribute Name> [ ( "<" | "≤" ) <higher bound> ] ]
  "creation event"
    <creation event name>
      [ "(" <parameter name> ":" <DOMAIN NAME> ")" ]
    "effect"
      ( "new self→"<Attribute Name> "=" <parameter name>
      | "new self→"<Attribute Name> "=" <domain expression> )
"end class" <CLASS NAME>
```

Table 4.3: EROOS Attribute Script

An **extended creation event** for a class decoration, is an event of a class that, if applied on a model instance at a certain time, in addition to adding a new object to the object population set for that class, will define a value of the appropriate domain as the attribute value for the object.

Given

Model M; Object Universe OU;, Event Universe EU;

Event Set Instance E; Class $C \in M_{cl}$; Attribute $CA \in M_a$;

Domain $D \in M_d$; Attribute Value $a \in D$; Creation event $c \in EU$;

$$c(a) \in E_{t+1} \Rightarrow \exists\ o \in OU : (o \notin C_t) \wedge (o \in C_{t+1}) \wedge (CA_{t+1}(o) = a)$$

Definition 4.5: Extended Creation Event for an EROOS Attribute

An attribute can only serve to decorate a single class. If there is a need to define an attribute for more than one class, each class will have to define its own distinct attribute, possibly all sharing the same attribute name. However, it is also possible to define an attribute for a generalisation class, as presented in Section 4.6, which is then, by default, part of each specialisation class that is derived from this class. A class can be decorated by a number of attributes, in which each attribute defines a certain property for all objects of the class. A domain can be used to decorate a number of classes, decorating each class a number of times, for which each time a distinct attribute for that class is defined. The resulting overall class description script is a combination of all individual attribute scripts for a class. The different components in the specification of an attribute script, as shown in Table 4.3, are the following:

- The domain of the attribute value and a name for the attribute must be specified. In the same manner as class names, the name of an attribute must be an expression in some natural language and must refer to a cultural entity. For readability purposes, EROOS attribute names must be singular and represented in title case. Moreover, all attributes decorating a single class must have different attribute names in order to distinguish them.
- In most cases, the binding between objects of a decorated class and values of the attribute domain will be a many-to-one binding, in which an object has a single attribute value, whereas that value can decorate an unrestricted number of objects. However, it is possible to define an integrated constraint that imposes the uniqueness of an attribute value among all objects of the class, using the **unique** keyword. The integrated uniqueness constraint imposes a one-to-one binding between the class and the domain, in which a value can only be used as an attribute value for a single object. For instance, this allows the modelling of registration numbers for an institute, or passport numbers for a government. Notice that, due to the property of permanent binding, each object of the decorated class must be associated with a specific value. The attribute value for an object cannot be left undefined.
- An integrated constraint can be defined that imposes restrictions on the allowed domain values by defining a lower and/or higher bound. As such, a restricted range of attribute values can be defined for the attribute.
- The specification of the creation event in an attribute script must correspond to its definition in the class script. In particular, the name of the creation event in the attribute script must be identical to its name as defined in the class script.
- A formal argument serves as a symbolic name for a value of the decorating domain. Each time the creation event occurs, a concrete argument has to be supplied for each formal argument. The formal argument will be used in establishing the binding of the new created object with a value of the decorating domain. Typically, the argument name is but does not have to be identical to the name of the attribute. Argument names must be represented in lowercase.
- The final component in the definition of a decorated creation event specifies the new binding of the decorated object with a value of the domain. For that purpose, an assertion will be included. The assertion states that, if the given implicit query →<attribute name>, defined in section 4.3.6, will be applied to the newly created object, referred to as **self**, at the moment the creation has occurred, referred to as **new**, the value described by the expression on the right-hand side must be returned as a result. A more detailed treatment of assertions can be found in section 4.7.4 on events. Broadly speaking, the domain expression defining the actual domain value can be built using the formal argument along with constants of the domain. The expression may involve functions and operators applicable to values of the decorating domain.

4.3.5 Default Attributes

EROOS attributes serve to model properties shared by all the objects of a single class. Attributes shared by all objects of every class are referred to as default attributes. Their definition is an integral part of the EROOS methodology. The *Creation Timestamp* for an object is defined as a default attribute in EROOS. The Creation Timestamp is used to specify the exact time at which an object has come into existence. The Creation Timestamp will be fixated at the time of creation of the object, i.e., at the moment of occurrence of the creation event. Although the Creation Timestamp does not have to be defined explicitly, its semantics can be defined in an implicit default EROOS attribute script, as presented in Table 4.4.

The definition of the default attribute Creation Timestamp introduces its name and domain, the TIME domain, as well as the extension of the creation event specifying that the default attribute Creation Timestamp has to be initialised with the current time, expressed using the keyword **now**. Because the attribute Creation Timestamp is available by default for every class, it would be incorrect to explicitly introduce an attribute expressing the same information as the Creation Timestamp, since this would lead to two equal attributes decorating the same class, namely, the explicit attribute and the default attribute Creation Timestamp.

```
<EROOS default creation timestamp> =
"class" <CLASS NAME>
  "definition"
    "decorated by attribute Creation Timestamp : TIME"
  "creation event"
    <creation event name>
    "effect"
      "new self→Creation Timestamp = now"
"end class" <CLASS NAME>
```

Table 4.4: Implicit EROOS Script for the Default Attribute *Creation Timestamp*

Moreover, it is not permitted in EROOS to model an attribute that can be derived from other attributes present in the model [131]. For example, when modelling a time period, the start time, end time, and duration cannot be modelled together. In such case, only the start time and duration should be modelled, since the end time can be derived. Although it seems that we could also have chosen to model the start time and the end time as attributes, this is however not the case. EROOS obliges the modeller to take those attributes that do not give rise to additional constraints in the model. Since the choice of start time and end time as attributes would introduce an additional constraint stating that the end time must be larger than the start time, while the choice of start time and duration as attributes would not introduce any additional constraint, EROOS oblige the analyst to model these attributes, whereas the end time can be

modelled as a query, as presented in Section 4.6.9. Tasker [146] identifies a number of types of situations to help analysts recognise derivable attributes.

4.3.6 Implicit Attribute Queries

The definition of an attribute decorating a class is automatically complemented with an implicit query for retrieving the attribute value. In general, a query offers the ability to inspect the properties of an object. Given an attribute A decorating a class C and involving the domain D, the implicit query '→A', applicable to each object c of the decorated class C, returns the value of the decorating domain D that is associated with the object c. This implicit query '→A' is used in (1) specifying the semantics of the creation event, (2) specifying queries in order to retrieve information from a model instance, (3) specifying model constraints for which the attribute is relevant, and (4) specifying the semantics of mutation events in the EROOS universe, as presented in Section 5.1.7. The definition of the implicit query '→A' can be found in Definition 4.6.

An **implicit query →A** or **→C/A** for an attribute A of a class C is a query that can be applied on an object of class C at a moment t, and that returns the attribute value bound with the object on moment t.

Given

Model M; Class $C \in M_{cl}$; Attribute $CA \in M_a$; Domain $D \in M_d$;

Query $\rightarrow C/A \in M_q$;

$\rightarrow C/A$: TIME $\rightarrow$ ($C_t \rightarrow D$) |

$\forall t \in TIME, \forall o \in C_t : \rightarrow C/A_t (o) = CA_t (o)$

Definition 4.6: Implicit EROOS Kernel Attribute Query

4.3.7 EROOS Attributes for the Library Example

Given the example of the library system that was presented in Section 2.3, two attributes can be identified for the class *LIBRARY*, namely *Maximum Lending Period* and *Amount Of Daily Fine*. The resulting creation event for the library class can be composed through a combination of all parameters and effects of the individual attribute scripts, as presented in Table 4.5.

Notice that, as modelled in the UML model presented in Figure 2.1 on page 27, it is forbidden in EROOS to model

- an attribute that represents the maximum number of items that may be lent from a library, since it is not allowed to use integer attributes. A reification of this attribute into a class has to be made. This is presented further in Section 4.4.7.

- an attribute that represents the start date of a borrowing, since this is already expressed by means of the default attribute Creation Timestamp for the class *BORROWING*.

```
class LIBRARY
  definition
    decorated by
      attribute Maximum Lending Period : DURATION
      attribute Amount Of Daily Fine : EUR VALUE
  creation event
    establish (lending period: DURATION,
               fine amount: EUR VALUE)
    effect
      new self→Maximum Lending Period = lending period
      new self→Amount Of Daily Fine = fine amount
end class LIBRARY
```

Table 4.5: EROOS Attribute Script for the Library Example

4.3.8 Contributions, Related Work, and Reflections

Our contributions concerning the attribute concept are the following:

- The **constructional model approach,** in which model instances can only grow and information can only be added to a model instance, is a crucial property of the EROOS kernel in achieving the Principle of Uniqueness. Attribute values cannot be changed, but instead the mutation of an attribute value must be reified into the creation of a distinct object representing this mutation event. It allows modellers to focus on the information of the universe of discourse that must be modelled. A modeller does not have to decide on which information will be kept inside a model and which information could be overridden, since the set of knowledge and facts inside a model instance can only be enlarged.
- The **default attribute *Creation Timestamp*** for all objects of every class, enables the modeller to reason about the moment at which an object has come into existence. This attribute does not have to be modelled explicitly, but it is automatically available for every object in EROOS. A modeller often has to reason about the time a certain event occurred, for example, to reconstruct the order in which certain requests were made, to determine the age of a certain object or to calculate the duration of a certain activity. The modeller does no longer have to model these attributes, nor is it necessary to decide whether such attributes are needed in the model. The EROOS methodology automatically exposes this kind of information for all objects.

- The **prohibition of using Boolean attributes and integer attributes** in EROOS, the fact that **attribute values cannot be undefined**, and **the prohibition of derived attributes**, forces the analyst to model explicitly a number of facts in the model using classes, specialisation hierarchies, and queries, rather than hide this information in a compact form inside an attribute. Such incorporation of implicit model constraints in each methodological concept provides dedicated semantics for each model concept, thereby limiting its usage to a specific context and forcing the analyst to use the most adequate concepts in all situations.

The EROOS attribute concept is largely comparable with the attribute concept in UML. However, our methodological approach that drives the analysis to a single and unique conceptual model, is a rather novel vision on conceptual modelling. Most analysis methods use modelling guidelines and metrics [118][26][25][44], and analysis patterns [48] in order to steer the analyst to a resulting conceptual model of sufficient quality. We claim that such approaches are rather informal and noncommittal, and do not offer the analyst suitable methodological support for performing conceptual modelling. Although a casual analysis methodology seems rather attractive in providing sufficient freedom for the modelling process, a strict and rigid approach, incorporating well-defined outcomes, are of more avail to the analyst.

The constructional model approach in which model instances can only grow through the addition of information, thereby enlarging the set of knowledge and facts that are stored in a model, is largely comparable with the *evolution monotonicity* concept in the MOOSE framework [155]. Information additions to the model substitute model mutations. Afterwards, the latest and most relevant information can easily be retrieved from the model instance at any moment using querying mechanisms.

An observation that can be made is that the application of the EROOS kernel methodology gives rise to a class model with a huge number of classes. Since the EROOS kernel focuses on achieving the Property of Uniqueness, all information in the EROOS model is specified as individual objects. This leads to a huge number of classes present in a conceptual model. While other analysis models allows a modeller to specify Boolean attributes, integer attributes and undefined attributes, the EROOS methodology forces the modeller to introduce additional objects for representing these facts. Especially transforming integer attributes into classes give rise to a huge increase in the number of objects present in a model instance. For instance, considering a show with a limited number of allowed attendants. The EROOS methodology forces the analyst to model every single possible attendance, e.g., represented as an entrance ticket, as an object on its own instead of modelling the maximum number of attendants for the show.

A second observation is that attribute values tend to have a limited validity. Since the EROOS kernel offers only mechanisms for extending the information contained in a model instance rather than changing this information, attribute updates have to be modelled using explicit update objects. Therefore, objects representing facts with related attribute values often have update objects attached to them for modelling changed attribute values.

4.4 Relations, Links, and Refinement

Objects, as a representation of facts from the universe of discourse, usually do not exist on their own. Typically, a fact in the universe of discourse is related to many other facts, and may even be dependent on a number of facts in order to be valid. A *relation* in EROOS serves to describe relationships among objects as they can be observed in the universe of discourse. First, this section introduces the principles underlying the refinement of a class, namely the encapsulation of relations within classes and the property of existential dependency for the modelling of relations. Hereafter, the model constraints implied by the relation concept are defined. Third, the notion of a relation script is introduced, the implicit relation queries '↓' and '↑' are defined in order to retrieve information about the relation, and the integrated constraints on connectivity and multiplicity are presented. Last, the EROOS relations that can be identified in the running example of the library system are presented.

4.4.1 EROOS Relations and Object Links

One of the basic characteristics of EROOS, distinguishing it from UML and other methods and notations for Object-Oriented Analysis, concerns the methodological rule that relations cannot exist on their own. In EROOS, every relation is encapsulated in a class. It is only in the design phase that we have to decide if a relation will be implemented by means of a class or an ordinary association. As such, the choice to model a certain relational thing as a relation or a class, will totally disappear. It is always be modelled as both a class and relation at the same time.

A relation expresses a number of facts from the universe of discourse, and therefore must always be contained inside a class representing and materialising these facts as objects. A relation encapsulated in a class is said to refine the class, defining additional characteristics for each object of that class by encapsulating a link between objects of other classes within the refined object. Each object of the refined class will at all times be associated with exactly one object of each of the classes involved in the relation, which are called participants. This property is referred to as existential dependency, whereby objects of the refined class are existentially dependent on objects of the participating classes. At the same time, the relation expresses that objects of each participating class have the potential to become associated with objects of the other participating class, via refined objects that encapsulate the association links.

Relations in EROOS are restricted to model unary and binary relations. A relation in EROOS is thus either a binary relation, i.e., a relation involving 2 participating classes, or a unary relation, i.e., a relation involving a single participating class. The structuring of the objects of a class R, refined by a binary relation involving classes A and B, and a class S, refined by a unary relation involving class C, at a certain moment t, is illustrated in Figure 4.5.

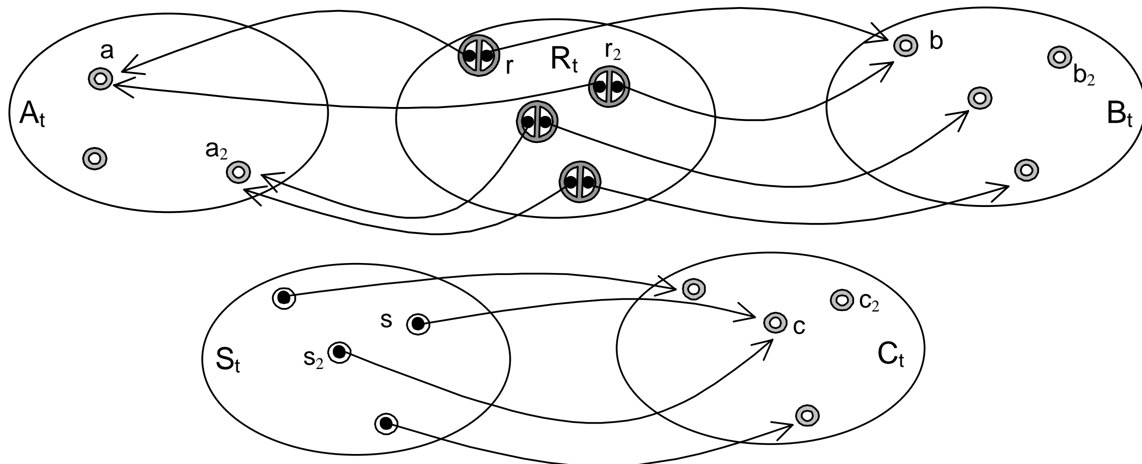


Figure 4.5: Objects refined by a Binary and Unary Relation at Moment t

The characteristics resulting from the refinement of a class are represented by lines, connecting refined objects with objects of the participating classes. As such, the object referred to as r is associated with object a of class A and object b of class B. In fact, the object r is associated with the tuple (a,b) expressing a link of the relation for class R between classes A and B. In the same way, the object referred to as s in is associated with the object c, or more precisely associated with the tuple (c) expressing a link of the relation for class S involving class C. Notice that several objects of a refined class can be associated with a single object of a participating class or even with the same tuple of objects, for example r and r_2, and also s and s_2. Such objects are called duplicates, since they share the same link. Duplicate objects differ from one another due to the unique object identity assigned to each of them. Each object of a refined class must at all times be associated with an object from each of the classes participating in the relation underlying the refinement. On the other hand, objects of the participating classes can exist without being associated with any objects of the refined class, for example b_2 and c_2.

EROOS restricts relations to be unary or binary. Relations of a higher degree are to be modelled as a combination of unary and binary relations. Rumbaugh [128] indicated that associations with an arity higher than two are usually not useful unless the multiplicity is many on all ends. Even in such case, EROOS obliges to decompose an n-ary relation into a number of binary relations. The reason behind the restriction to unary and binary relations, is to force the analyst to look for underlying dependencies that are often not directly visible at first sight. Instead of modelling an n-ary relation, the analyst has to study the relation between each pair of participating classes in order to find potential dependencies that are unrelated to the original n-ary dependency. If one wants to model, for instance, a relation R=(A,B,C) between classes A, B, and C,

the analyst has to investigate the universe of discourse to assess whether there exist preceding dependencies between (A,B), (A,C) or (B,C).

Depending on the universe of discourse, one of the following binary decompositions, presented in Figure 4.6, must be chosen: ((A,B),C), ((A,C),B), (A,(B,C)), ((A,B),(A,C)), ((A,B),(B,C)), or ((A,C), (B,C)).[22] The criterion for making the proper binary decomposition is to investigate whether there exists a preceding dependency between objects of two participating classes independently from the relation R. For instance in the case of R=((A,C),B), details from the universe of discourse could show that objects of classes A and C can be related in some manner without directly having to be linked to objects of class B in the relation R.

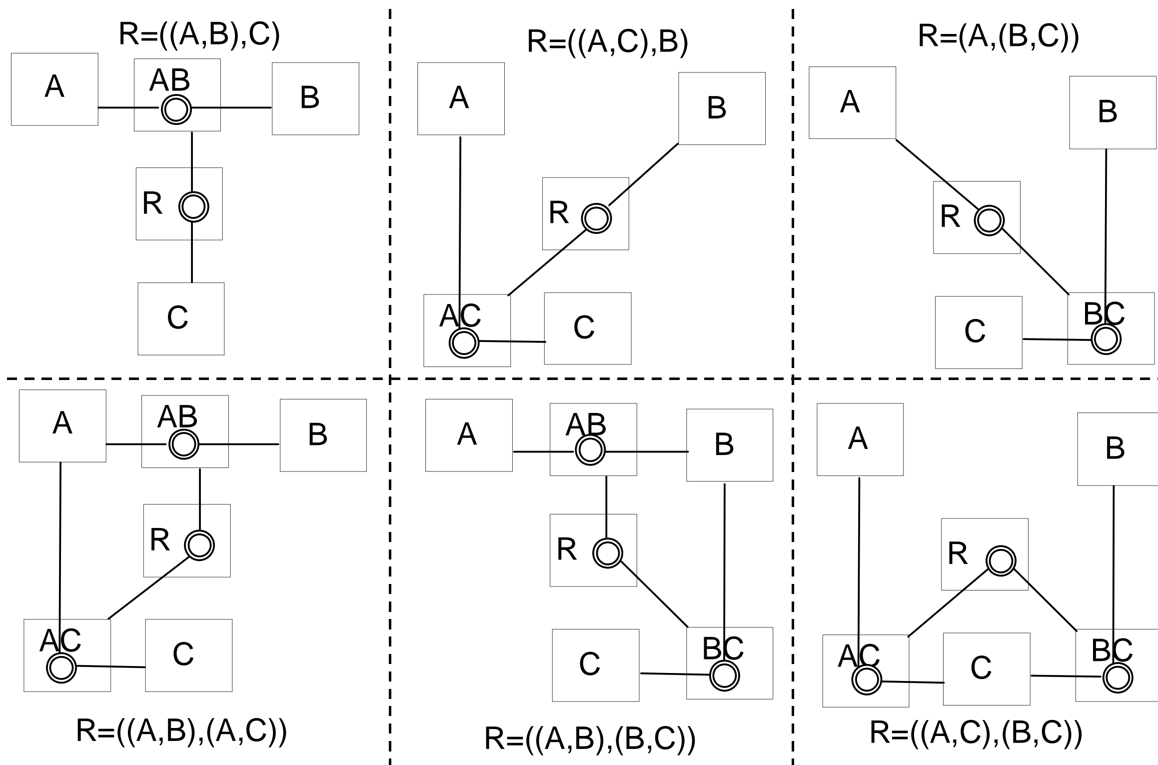


Figure 4.6: Decomposition of an n-ary Relation into Binary Relations

In EROOS, the refinement of classes by means of unary and binary relations must result in a pure hierarchical structuring of the classes. A class cannot be refined with a relation directly or indirectly involving its own objects. Due to the property of existential dependency, such self-refinements would make it impossible to create the

[22] Notice that this is not an exhaustive list, since more complex structures, involving intermediate classes, could be identified. The ultimate relation R will however always exist of two participating classes that are directly or indirectly dependent on classes A, B, and C.

first instance of that class. More precisely, for each chain of refinements consisting of relations R_i refining a class C_i with C_{i-1} as a participating class ($1 \leq i \leq n$), $C_j \neq C_k$, for all $0 \leq j < k \leq n$.[23] In addition, each of the classes must somehow be related to each of the other classes in the model by means of a number of model entities. This enforces the modelling Principle of Minimalism, as defined in Section 3.5. The Principle of Minimalism guides the analyst in determining which aspects of the universe of discourse are important to be modelled, and which of them can be ignored in the context of the model to be developed. As such, if a class is not related to other classes in a conceptual model, either an aspect of reality has been modelled that is of no relevance, or some aspects, e.g., a number of relations, have been overlooked.

Existential dependency among objects may seem too restrictive for the ultimate system to be developed. A large deal of run-time flexibility, in populating the implementation classes with instances, would be lost. As already mentioned, focusing on the structures from the universe of discourse in its normal appearance should have priority over implementation issues concerning the software system. Implementation issues regarding object-oriented conceptual models are discussed in Section 4.8.4.

4.4.2 Model Constraints implied by the Relation Concept

EROOS incorporates important model constraints directly in the methodological concepts. The following constraints are directly implied by the relation concept:

- **Existential dependency**: A relation refining a class implies the existential dependency of the refined objects on exactly two objects in the case of a binary relation, and on one object in the case of a unary relation. This means that each object of the refined class must, at all times, be associated with an object of each of the participating classes. If it should be possible that certain objects must exist without being associated with such participating objects, the relation property is not shared by all objects of the class. In such case, an additional class must be introduced, expressing the fact that certain objects are dependent on participation objects while other objects do not comply with such obliged dependency.
- **Immutability**: The association of a given object with its participating objects is considered to be static. In particular, at the moment an object is introduced, it must be associated with an object of each of the participating classes, and it will keep that association for its entire lifetime. Objects cannot switch from one participating object to another during their lifetime. If it is needed to change a participating object, an additional class must be introduced expressing the fact that a link to a participating object has been changed.[24]

The definition of a relation can be found in Definition 4.7.

---

[23] As explained later, specialisation structures can be used to model recursive existential dependency hierarchies. However, in such case there must always be a specialised class that is unrefined and thus can serve as a kind of sentinel for the recursive dependency structure.
[24] In the EROOS universe, a link of a relation can be redirected to another object of the participating class.

A **relation** is a model entity defining a property for a class for which, at each moment in time, every object of the class, called a refined object, must be associated to a specific object, called a participant object, of the participating class defined for the relation.

A relation can either be a **binary relation,** defining exactly 2 participating classes for the refined class, or a **unary relation,** defining exactly 1 participating class.

Given

Model M; Class C,D,E,F,G $\in M_{cl}$; Binary Relation CB $\in M_{br}$;

Unary Relation CU $\in M_{ur}$;

CB: $TIME \rightarrow (C_t \rightarrow (D_t \times E_t)) \mid$ (existential dependency)

$\forall t \in TIME: CB_t \subseteq CB_{t+1}$ (immutability)

CU: $TIME \rightarrow (F_t \rightarrow G_t) \mid$ (existential dependency)

$\forall t \in TIME: CU_t \subseteq CU_{t+1}$ (immutability)

Definition 4.7: EROOS Kernel Relation

4.4.3 Specification of an EROOS Relation

The specification of a relation refining a class is represented in a relation script. This relation script identifies the participating class or classes of a relation. The property of existential dependency influences the creation of objects of a refined class. Each time a new refined object is to be created, its binding with objects of each of the participating classes must be established as well. Therefore, the creation event must be extended in order to establish the bindings of the new refined object with an object of each of the participating classes. The syntax of a refinement script is given in Table 4.6. As presented in Figure 4.7, a relation is graphically represented in the form of a double circle within the refined class, and attached to the classes that participate in the relation. The double circle expresses the possibility of duplicate links. Duplicate links are possible when links are encapsulated in objects, since the object identity will distinguish the objects even if two objects contains an identical link between the same participating objects. The definition of the extended creation event can be found in Definition 4.8.

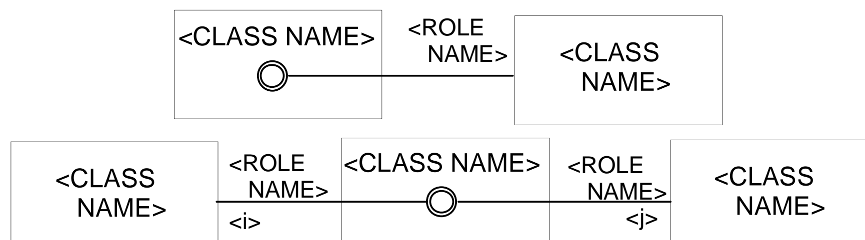


Figure 4.7: Graphical Representation of a Unary and Binary EROOS Relation

```
<EROOS kernel relation script> =
"class" <CLASS NAME>
  "definition"
    ( "refined with binary relation"
    ( <positive number> | "unlimited" | "∞")[25] <CLASS NAME>
        [ "as" <ROLE NAME> ] ","
    ( <positive number> | "unlimited" | "∞") <CLASS NAME>
        [ "as" <ROLE NAME> ]
    | "refined with unary relation"
      <CLASS NAME> [ "as" <ROLE NAME> ] )
     ( ( "unlimited" | "∞" ) "occurrences" | "one occurrence"
     | <positive number larger than 1> "occurrences" )[25]
  "creation event"
    <creation event name>
      "(" <parameter name> ":" <CLASS NAME>
        [ "," <parameter name> ":" <CLASS NAME> ] ")"
    "effect"
      "new self↓"<Participant Name> "=" <parameter name>
      [ "new self↓"<Participant Name> "=" <parameter name> ]
"end class" <CLASS NAME>
```

Table 4.6: EROOS Kernel Relation Script

An **extended creation event** for a class refinement, is an event of a class that, if applied on a model instance at a certain time, in addition to adding a new object to the object population set for that class, will define a link to objects of the appropriate participating classes as the relation link for the object.

Given

Model M; Object Universe OU;, Event Universe EU;

Event Set Instance E; Class B1,B2,U $\in M_{cl}$; Object p $\in$ B1;

Object q $\in$ B2; Object r $\in$ U; Creation event c1,c2 $\in$ EU;

Binary Relation CB $\in M_{br}$;Unary Relation CU $\in M_{ur}$;

$c1(p,q) \in E_{t+1} \Rightarrow \exists\ o \in OU$:

$\quad (o \notin C_t) \land (o \in C_{t+1}) \land (CB_{t+1}(o) = (p,q))$

$c2(r) \in E_{t+1} \Rightarrow \exists\ o \in OU$:

$\quad (o \notin C_t) \land (o \in C_{t+1}) \land (CU_{t+1}(o) = r)$

Definition 4.8: Extended Creation Event for an EROOS Relation

[25] See Section 4.4.5 for the definition of connectivity constraints, and Section 4.4.6 for multiplicity constraints.

A relation can only serve to refine a single class, whereas a class can be refined by only one relation. If there is a need to define a specific relation for more than one class, each class will have to define its own distinct relation, possibly all sharing the same role names. However, it is possible to define a relation for a generalisation class, as presented in Section 4.6, which is then, by default, part of each specialisation class that is derived from this class. The different components in the specification of a relation script, as shown in Table 4.6, are the following:

- The definition of a relation essentially identifies the participating class or classes. In addition, a role name for each participating class can be defined as a reference to it, in order to emphasise the role that a participating class assumes in the relation. For readability purposes, a role name must be a singular noun in uppercase, and must be unique for the relation in order to distinguish them. Role names are mandatory in binary relations that model links involving two objects of the same class, i.e., relations from a participating class to itself. In such relations, the participating class has two distinct roles in the relation, which must be distinguished using role names.
- The specification of the creation event in a relation script must correspond to its definition in the class script. In particular, the name of the creation event in the relation script must be identical to its name as defined in the class script.
- A formal argument serves a symbolic name for an object of the participating class. Each time the creation event occurs, a concrete argument has to be supplied for each formal argument. The formal argument will be used in establishing the binding of the new object with an object of each participating class. Typically, the argument name is but does not have to be identical to the role name or the name of the participating class. Argument names must be represented in lowercase.
- The final component in the definition of a refined creation event specifies the binding of the refined object with an object of the participating class. For that purpose, an assertion will be included. The assertion states that, if the given implicit query ⇂<Participant Name>, defined in section 4.4.4, will be applied to the newly created object, referred to as **self**, at the moment of creation, referred to as **new**, the object on the right-hand side must be returned as a result.

4.4.4 Implicit Refinement and Participation Queries

The definition of a relation refining a class is automatically complemented with one or two implicit refinement queries applicable to all objects of the decorated class, and an implicit participation query applicable to all objects of each participating class. These queries offer the ability to inspect and retrieve information concerning the current binding of a refined object with objects of the participating classes. Given a relation refining a class C involving the participating classes P and Q,

- the implicit refinement queries '⇂P' and '⇂Q', or '⇂PR' and '⇂QR' in case that PR and QR are role names given to P and Q in the relation, applicable to each

object c of the refined class C, returns the object of the participating class P and Q contained in the relation link for the refined object c.

- the implicit participation query '↑C' or '↑PR/C', applicable to each object p of the participating class P, and '↑C' or '↑QR/C', applicable to each object q of the participating class Q, returns the set of objects of class C that encapsulates a relation link in which the object p, respectively q, participates.

The relation between '↓P' and '↑C' is: $\forall c \in C: \forall p \in P: c{\downarrow}P = p \Leftrightarrow c \in p{\uparrow}C$

```
An implicit refinement query ↓P or ↓C/P for a refined class C
having a participant P, or ↓R, ↓C/R, ↓R/P, or ↓C/R/P when
the participant P has a role name R, is a query that can be
applied on an object of the refined class at a moment t, and
that returns the participant object contained in the link for
that object on moment t.
Given
Model M; Binary Relation CB ∈ M_br; Unary Relation CU ∈ M_ur;
Class C,D,P,Q,R ∈ M_cl; Query ↓C/P,↓C/Q,↓D/R ∈ M_q;
↓C/P: TIME → ( C_t → P_t ) |
   ∀ t ∈ TIME: ∀ c ∈ C_t : ↓C/P_t(c) = P(CB_t(c))
↓C/Q: TIME → ( C_t → Q_t ) |
   ∀ t ∈ TIME: ∀ c ∈ C_t : ↓C/Q_t(c) = Q(CB_t(c))
↓D/R: TIME → ( D_t → R_t ) |
   ∀ t ∈ TIME: ∀ d ∈ D_t : ↓D/R_t(d) = CU_t(d)
```

Definition 4.9: Implicit EROOS Refinement Query

```
An implicit participation query ↑C or ↑P/C for a participant
P of a refined class C, or implicit query ↑R/C or ↑P/R/C when
the participant P has a role name R, is a query that can be
applied on an object of the participating class at a moment
t, and that returns the set of all refined objects that
contains a link in which the object is involved on moment t.
Given
Model M; Binary Relation CB ∈ M_br; Unary Relation CU ∈ M_ur;
Class C,D,P,Q,R ∈ M_cl; Query ↑P/C,↑Q/C,↑R/C ∈ M_q;
↑P/C: TIME → ( P_t → 𝒫(C_t) ) | ∀ t ∈ TIME:
   ∀ p ∈ P_t : ∀ c ∈ C_t : c ∈ ↑P/C_t(p) ⇔ P(CB_t(c)) = p
↑Q/C: TIME → ( Q_t → 𝒫(C_t) ) | ∀ t ∈ TIME:
    ∀ q ∈ Q_t : ∀ c ∈ C_t : c ∈ ↑Q/C_t(q) ⇔ Q(CB_t(c)) = q
↑R/D: TIME → ( R_t → 𝒫(D_t) ) | ∀ t ∈ TIME:
    ∀ r ∈ R_t : ∀ d ∈ D_t : d ∈ ↑R/D_t(r) ⇔ CU_t(d) = r
```

Definition 4.10: Implicit EROOS Participation Query

The notation supports the view of zooming into the elements that are an existential part of an object (projection ↓), and zooming out to the elements in which an object is contained (election ↑). These implicit queries are mainly used in (1) specifying the semantics of the creation event, (2) specifying queries in order to retrieve information from a model instance, (3) specifying relation navigation paths from an object to a set of related objects in a model instance, and (4) specifying the semantics of mutation events in the EROOS universe, as presented in Section 5.1.7. The definition of the implicit refinement and participation queries can be found in Definition 4.9, respectively Definition 4.10.

4.4.5 Integrated Relationship Constraints on Connectivity

The definition of a binary relation can be complemented with connectivity constraints, restricting the existence of certain combination of links. The definition of an EROOS relation also specifies how many different objects of one participating class can be associated with a single object of the other participating class at a moment in time, through objects of the refined class encapsulating the links, and vice versa. This type of constraint, referred to as connectivity constraint, is integrated into the EROOS relation concept. The specification of a connectivity constraint consists of the specification of a connectivity value for each class participating in the binary relation. When no restriction is placed on the number of related objects, the value for the connectivity constraint is defined as '∞' or **unlimited**.

The specification of a connectivity constraint is integrated in the participant clause of the relation specification. In particular, as outlined in Table 4.6, the description of each participant clause must start with a positive number written in a numerical or verbose style, referred to as the connectivity value of the participant, or '∞' or **unlimited** to specify an unrestricted value. As presented in Figure 4.7, a connectivity constraint is graphically represented by noting the connectivity value in the neighbourhood of the participating class.

- 'i' represents the connectivity value for the participating class on the left-hand side, i.e., the maximum number of objects of that class that can be associated with a single object of the participating class on the right-hand side, through objects of the refined class encapsulating the links.
- 'j' represents the connectivity value for the participating class on the right-hand side, i.e., the maximum number of objects of that class that can be associated with a single object of the participating class on the left-hand side, through objects of the refined class encapsulating the links.

The value '∞' or **unlimited** will not be included in graphical representations since it does not stand for an actual restriction. In such cases, the line connecting the refined class with the participating class will not be having a connectivity value.

4.4.6 Integrated Relationship Constraints on Multiplicity

The definition of a unary relation or a binary relation can be complemented with a multiplicity constraint, restricting the number of identical links that can be encapsulated in objects of the refined class. Since relation links in EROOS are always encapsulated in a refined object, the same link can be used several times to refine different objects of the refined class. Such links, called duplicates, can be distinguished from each other by means of the intrinsic object identity of the object in which the link is encapsulated. A multiplicity constraint for a binary relation specifies how many times a single object of the participating class on the left-hand side can be associated with the same object of the participating class on the right-hand side, through objects of the refined class that encapsulate the links. A multiplicity constraint for a unary relation specifies how many times a single object of the participating class can be used as a participant for objects of the refined class.

The specification of the multiplicity constraint, presented in Table 4.6, is specified in a separate clause in the definition of a relation. The definition of a relation includes a multiplicity clause in which the multiplicity value for the relation must be defined in the form of a positive number in a numerical or verbose style, or be defined as unrestricted, using '∞' or **unlimited**. The graphical representation of multiplicity constraints for a unary and binary relation is illustrated in Figure 4.8.

- If the number of duplicates is bounded to a specific value 'i' (i > 1), the multiplicity value is noted inside a double circle, representing the relation.
- If an unlimited number of duplicates are allowed ('∞' or **unlimited** as multiplicity value), a double circle is drawn for the relation.
- If the number of duplicates is set to 1, thus actually when no duplicates are allowed, a single circle is drawn for the relation.

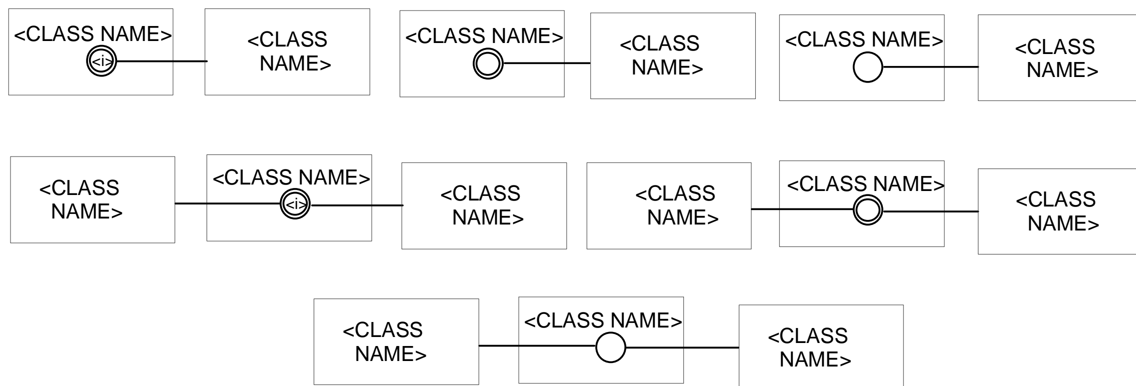


Figure 4.8: Graphical Representation of EROOS Multiplicity Constraints

4.4.7 EROOS Relations for the Library Example

Given the example of the library system that was presented in Section 2.3, a large number of relations can be identified. Based on the classes for the library system that were defined in Section 4.2.4, the relations between these classes are represented in Figure 4.9. Since the EROOS relation scripts do not provide any additional information next to the extension of the creation event, we have omitted them.

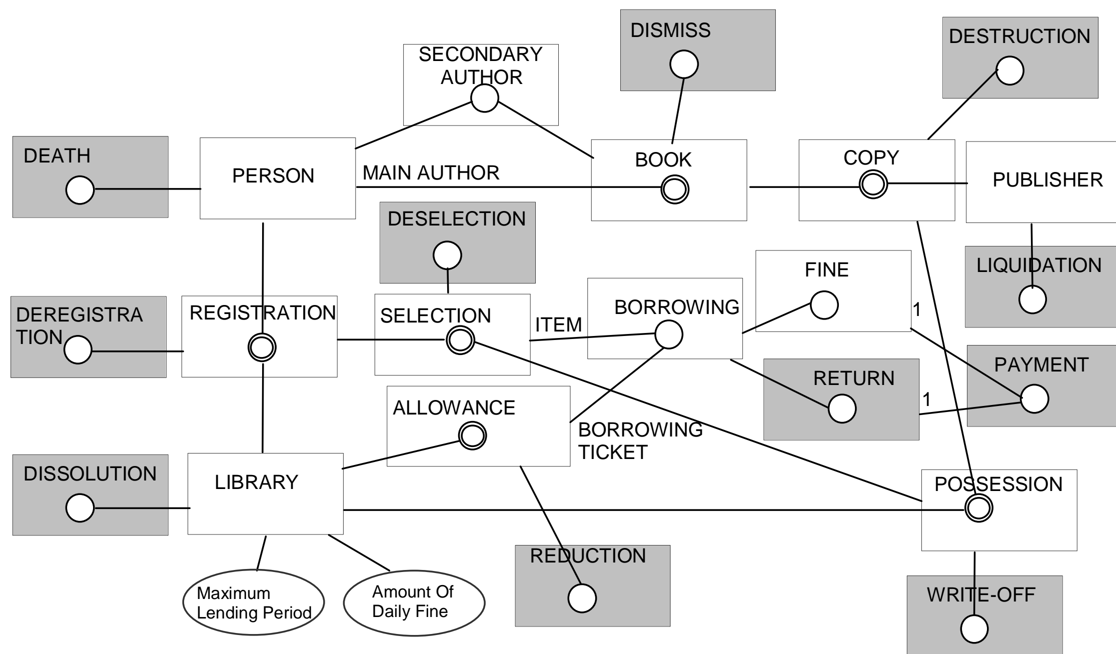


Figure 4.9: EROOS Kernel Relations for the Library System

The following observation can be made:

- The relation structure can get quite complex when all relevant relations are added. This is due to the split-up of activities into two object that represent the start and the end of the activity. The EROOS universe, presented in Chapter 5, will enable to merge these two objects into a single object representing the whole activity. The darker shaded classes can then be merged with their counterparts to whom they are attached.
- The refinement for the class PAYMENT using a binary relation between FINE and RETURN expresses the fact that a person can only pay her or his fine for a book after the borrowed book has been returned. It is impossible to calculate the amount of the fine as long as the borrowing has not been ended.
- The identification of relations give rise to the discovery of new classes for the library system, such as SECONDARY AUTHOR, representing the secondary authors of a book, ALLOWANCE, representing the fact that a registered person

has the possibility of borrowing a certain number of books, REDUCTION, representing the fact that a library can decrease the maximum number of books that can be lent to a single person, SELECTION, representing the fact that a person can have chosen a certain book for borrowing, and DESELECTION, representing that a person has chosen not to borrow a certain selected book. In fact, the class ALLOWANCE is a reification of an integer attribute representing the maximum number of lending items, as indicated in Figure 2.1 on page 27. Because the EROOS methodology does not allow to model integer attributes, the modeller is forced to make explicit an ALLOWANCE object that represents the possibility of borrowing a book. The class REDUCTION represents a diminishing of the attribute. The constraint expressing that a person can only borrow a maximum number of books, is now expressed by means of an existential dependency from the borrowing object on an allowance object. The reason behind this restriction is that the modeller should have a clear view on such reified attribute in order to being able to utilise it when necessary, e.g., when the borrowing period of a book differs between different lent item.

- The multiplicity constraints seem to be very loosely formulated. For example, the fact that a person can have many registrations at the same library seems to be not in correspondence with the universe of discourse. It is impossible that a person has more than one registration at the same time at a library. But one must also take into account that a deregistration does not cause the registration object to disappear, but merely denotes that the registration object is participating in a deregistration object. The model is thus correct, since a number of finished registrations can exist together with a single running registration. What is currently missing in the model, is a constraint stating that there can exist only one active registration for a person at a library. However, a multiplicity constraint for the deregistration, restricting the possibility for a person to deregister more than once for a certain registration, is nevertheless appropriate for the current model.
- One would expect the specification of more connectivity constraints. For instance, one could express that a book copy can be in possession of at most one library. However, since the EROOS kernel offers a constructional model approach, it does not make sense to introduce these kinds of connectivity constraints. Since objects cannot be destroyed in the EROOS kernel, a connectivity constraint would restrict the existence of links to the first link that has been created. So, when a book copy is written-off by one library and is afterwards acquired by another library, it cannot come into possession of the second library, since the connectivity constraint only allows a copy to be related to at most one library. The writing-off of the possession only creates an object of the class WRITE-OFF, and does not destroy the possession object. Regarding the fines, one can nevertheless state that a fine can be paid only once using the appropriate return object, and that the return of a borrowing can give rise to the payment of at most one fine.
- Although a distinction has been made between the main author of a book and the secondary authors of a book, the order in which the secondary authors are ranked is not captured in the model. However, we will introduce this ordering in Section

4.6, when specialisation has been introduced for the EROOS kernel. In addition, there is no class representing the end of the lifetime of a secondary author object, since they are considered to exist as long as a book has not been dismissed.

4.4.8 Contributions, Related Work, and Reflections

Our contributions concerning the relation concept are the following:

- The systematic usage of **existential dependency** as the main criterion to determine the core model structure, is a key contribution of our work. Such approach leads to a hierarchical object dependency structure that gives a clear insight in which information is dependent on certain other information. It leads to a powerful model that implies a large number of model constraints directly through its model structure. Relations in EROOS are explicitly and uniquely modelled, since they are always encapsulated in a refined class. In contrast to that, UML offers only a number of possibilities to model relations, such as associations, association classes, qualified associations, aggregates, compositions, and an association reified into a class.
- The **constructional model approach,** in which model instances can only grow and information can only be added to a model instance, is a crucial property of the EROOS kernel in achieving the Principle of Uniqueness. Relation participants cannot be changed, but instead the mutation of a participant must be reified into the creation of a distinct object representing this mutation event. It allows modellers to focus on the information of the universe of discourse that must be modelled. A modeller does not have to decide on which information will be kept inside a model and which information could be overridden, since the set of knowledge and facts inside a model instance can only be enlarged.

The EROOS relation concept is somewhat comparable with the association concept in UML. Similarities between classes and relations are also identified by Rumbaugh [129]. However, in this approach that has evolved into OMT [93][126], there remains a strict distinction between classes and relations. Using existential dependency as the key modelling criterion to construct the conceptual model structure, has also been applied by the M.E.R.O.DE. methodology [138][136][137]. M.E.R.O.DE. defines existential dependency as 'the total embedding of the life of a so-called *marsupial* object occurrence into a *mother* object occurrence'. Since objects come only into existence in the EROOS kernel and are never been destroyed, we define existential dependency as 'the obligation of the existence of a participant object (corresponding to the mother object in M.E.R.O.DE.) at the moment the refined object (corresponding to the marsupial object in M.E.R.O.DE.) is created'. The EROOS universe, which is defined in Chapter 5, introduces the concept of a class archive, and enables to specify a specific restriction between the destruction timestamps of the objects. This allows for a refined (marsupial) object to outlive its participant (mother) object, which is impossible in M.E.R.O.DE. In UML1.x [120][119][107], it is possible to simulate existential dependency using a restricted form of multiplicity for associations, obliging that at least one participant in an association must have a lower bound multiplicity higher than zero. In UML2.0 [109][128], existential dependency

can be expressed using association classes, since the restriction on association link duplicates can be removed using the '*{bag}*' property string for an association end. However, such approach is considered to be on the same level as modelling guidelines, since the UML notation does not enforce such rules on the model.

An observation that can be made is that the application of the EROOS kernel methodology gives rise to a class model with a huge number of classes. Every relation will introduce a distinct class that encapsulates the association. However, we don't consider the introduction of this class as a disadvantage, because it creates a hook in the model that enables the attachment of future properties, and that can be used for the expression of additional existential dependency constraints.

A second observation is that connectivity and multiplicity constraints seem to be inadequate for the EROOS kernel. This is due to the fact that that constructional model approach only allows adding new objects to a model instance. However, the EROOS universe, presented in Chapter 5, introduces the ability to destroy object. This enables a proper usage of connectivity and multiplicity constraints for relations.

A third observation is that the EROOS kernel does not allow the specification of mutual dependency between objects. The only possibility of modelling mutually dependent objects is to merge them into a single object. The EROOS universe introduces the concept of compounds to model mutually dependent objects.

4.5 EROOS Constraints and Confinement

In observing and modelling the universe of discourse, objects, attributes, and relations must comply with a lot of human-imposed, physical or legal laws, rules, and regulations, restricting certain characteristics and their evolution. As discussed in Chapter 2, this results in model instance restrictions delimiting the valid instances of a conceptual model in order to comply with the universe of discourse. Constraints reflect rules from the universe of discourse that cannot be broken and thus must be satisfied at all times.

The EROOS concepts introduced in the previous sections, namely classes, attributes, and relations, already incorporate a number of implied and integrated constraints. However, as discussed in Chapter 2, not all restrictions occurring in the universe of discourse are suited to be expressed using implied or integrated constraints. Therefore, the explicit specification of constraints as a first-class model concept is introduced in EROOS in order to enable the description of all kind of restrictions on the conceptual model that apply in the universe of discourse. To make a clear distinction with other kinds of constraints present in a model, such as implied constraints, e.g. existential dependency constraints for relations, and integrated constraints, e.g. lower and higher bound constraints on attributes, we explicitly call this type of constraint an *EROOS constraint.* Although EROOS constraints are specified separately from the definition of other concepts, they remain dependent on a

number of structural model items that are being confined by the constraint. The model elements upon which the EROOS constraint acts are indicated as its *context*.

EROOS constraints impose hard restrictions on a model. At each moment in time, a model instance is obliged to fulfil all defined EROOS constraints. Events that will lead to a constraint violation should be forbidden and, thus, are rejected by the model. In such a case, the model instance will remain in the state it was at the moment before the event occurred.[26] One of the basic characteristics of EROOS, distinguishing it from a number of other methods for object-oriented analysis that are using informal annotations to the model, is the fact that constraints are expressed formally, using many-sorted first order logic (MSFOL) [95]. As explained in Chapter 2, the formalisation of EROOS constraint expressions must lead to the avoidance of ambiguities and misunderstandings in a conceptual model. The expressivity of EROOS constraints is largely comparable with invariants specified in the Object Constraint Language (OCL) [108][161] for UML.

Contrary to the other modelling concepts, EROOS constraints do not define new model structures, in which additional characteristics of objects are described, or new behaviour, in which new events applicable to objects are specified. EROOS constraints merely describe rules that restrict the already defined structures and behaviour of the model. This affects the set of potential states for the model instance, and the events in which objects can be involved at a specific moment in time.

This section introduces the general principles underlying EROOS constraints and confinement, and presents the notion of constraint scripts to define the EROOS constraints in a model. Hereafter, specific restrictions on EROOS join constraints are presented as a means to avoid information duplication in a relation hierarchy. Last, the types of EROOS constraints that can be identified in the running example of the library system are presented.

4.5.1 General Principles of Confinement in EROOS

An EROOS constraint enforces a logical rule on a certain part of the conceptual model, called the context of the constraint, which must always be kept valid. The context can include a number of classes, relations, attributes, or other EROOS model concepts that will be defined later in this text. Notice that the context is not restricted to a single class, as is the case for OCL invariants.

Since many classes are potentially involved in the specification of a single EROOS constraint, a large number of equivalent formulations of an EROOS constraint could be possible. For instance, each involved class has the possibility of specifying the given EROOS constraint as it can be observed from the viewpoint of that class. However, equivalence among conceptual models tremendously compromises their re-

[26] In the EROOS universe, we propose constraint triggers that can resolve certain constraint violation by adding additional functionality to the event in order to regain a valid model instance. As such, a constraint trigger serves as a general problem solver for the constraint it belongs to.

use for an analogous universe of discourse. In addition, understandability of the conceptual model for customers, end-users, and other software engineers decreases when different notations for a single aspect from universe of discourse are possible. Therefore, the EROOS method enforces the analyst to formulate an EROOS constraint from a specific viewpoint, namely the *top class* or *top classes* of the constraint, whereas other equivalent formulations for the constraint are forbidden. The top classes of a constraint are those classes in the relation hierarchy from which all other involved classes, mentioned in the context of the EROOS constraint, can be reached using only a refinement query ($\downarrow$). The EROOS constraint is specified as a rule that must be valid for all top objects of the top classes, namely

$$\forall\ tc_1 \in TC_1,\ldots,\ \forall\ tc_n \in TC_n\text{: <constraint expression for } tc_1,\ldots,tc_n\text{>}$$

- When the EROOS constraint must be true for all objects of a specific class, it can be specified from the viewpoint of that class.
- When the EROOS constraint must only be true for the objects of a class P that are participating in a certain relation link, encapsulated in class R, it must be specified from the viewpoint of the refined class R, since class R is the highest involved class for the EROOS constraint.
- If the EROOS constraint must only be valid for the objects of class P that are *not* participating in a certain relation link encapsulated in class R, it cannot be defined from the viewpoint of objects of class R. In this case, the constraint must be defined directly from the viewpoint of class P, but is restricted to the collection of objects that are not participating in any relation link of R. We indicate such constraint as a *not participating constraint*, namely

  $$\forall\ p \in P \text{ not participating in R: <constraint expression for p>}$$

- When the EROOS constraint must only be true for the objects of a class P that are participating in two relation links, encapsulated in the classes R and S, it must be specified from the viewpoint of the pair of refined classes R and S, since these two classes are the highest involved classes for the EROOS constraint.

The determination of the top class(es) for a constraint and the corresponding viewpoints for specifying EROOS constraints are illustrated in Figure 4.10.

The formalism for expressing EROOS constraints is based on many-sorted first order logic (MSFOL) [95]. However, there are some restrictions for constructing valid constraint expressions that apply (1) to the predicates allowed for an expression, and (2) to the operators allowed for combining predicates in order to obtain complex expressions. The idea behind the imposed restrictions is to force analysts to use implied constraints whenever appropriate, and force a single and unique manner for specifying EROOS constraints. In principal, the formalism provided for expressing EROOS constraints should not be too powerful enough such that it becomes possible to express implied constraints using the concept of an EROOS constraint. The advantage of a unique specification manner for a constraint concerns the fact that it provides solid criteria in developing conceptual models. This leads to a single

common model among all analysts involved in the development of a conceptual model. In order to reach the objective of a unique EROOS model for a specific universe of discourse, as stated by the conceptual modelling Principles of Uniqueness and Minimalism, EROOS also explicitly inhibits the definition of an EROOS constraint that can logically be derived from other constraints already present in the conceptual model. As such, the specification of constraints in an EROOS model can be restricted to the set of relevant constraints, and does not include a huge set of rather trivial derived constraints. The uniqueness of a model constraint expression is the key point that differentiates EROOS constraints from OCL invariants.

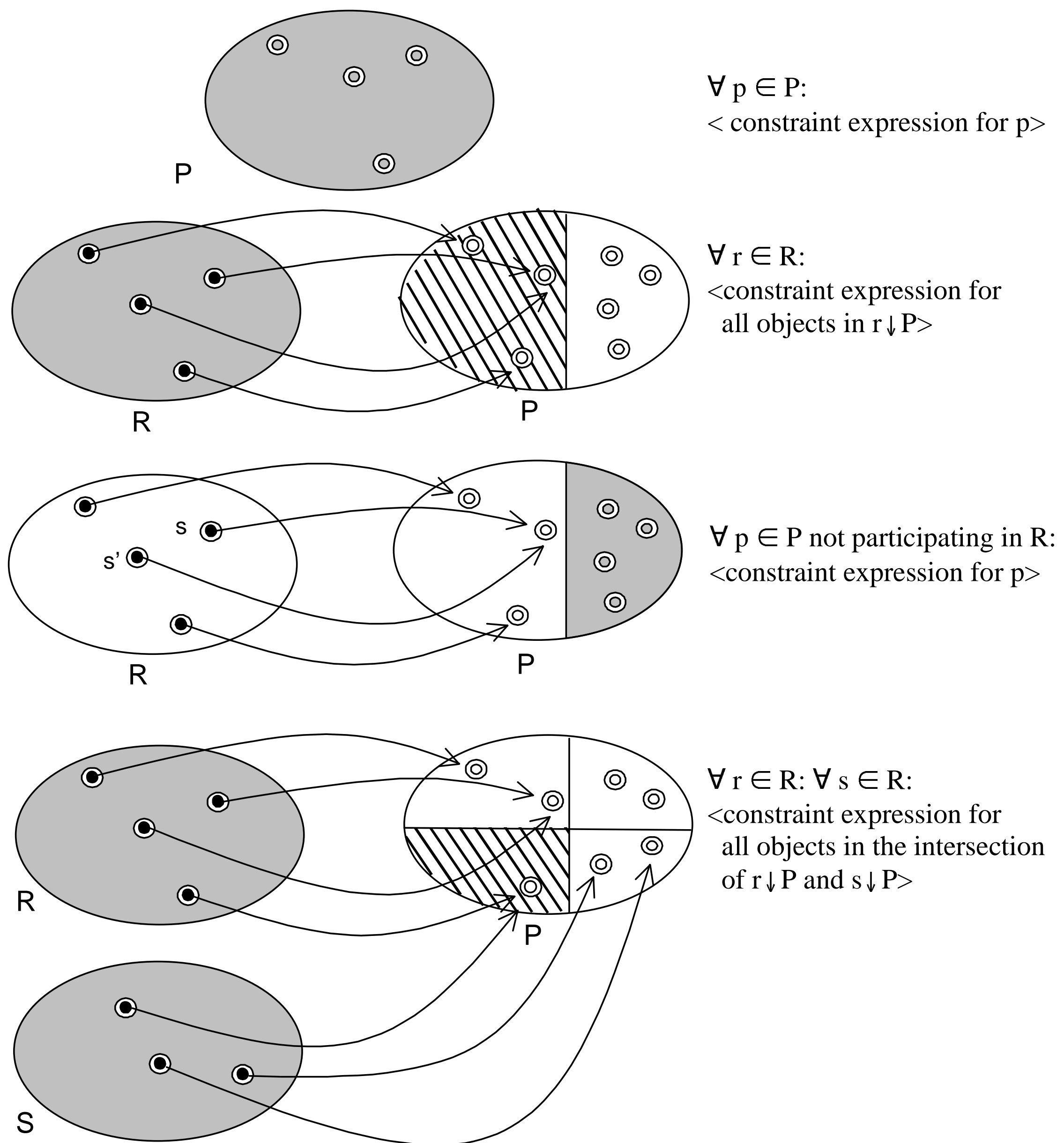


Figure 4.10: EROOS Constraint Specification from the Viewpoint of the Top Class

4.5.2 Specification of an EROOS Constraint

EROOS constraints restrict the defined structures and possible behaviour of the model. The definition of an EROOS constraint is presented in Definition 4.11. The syntax of an EROOS constraint script is given in Table 4.7. As presented in Figure 4.11, an EROOS constraint is graphically represented in the form of a triangle attached to the top class(es) of the constraint, and annotated with the constraint name or a reference number.

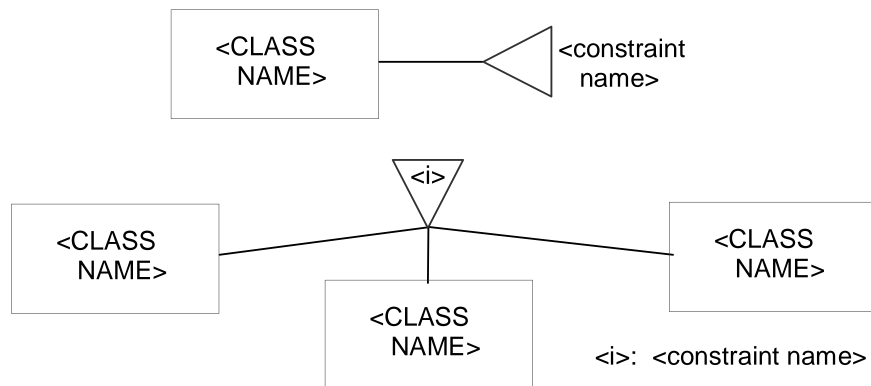


Figure 4.11: Graphical Representation of an EROOS Constraint

An **EROOS constraint** is a model entity restricting the set of possible instances of a model by defining rules that must be valid for a model instance at each moment in time
Given
Model M; Model Instance Universe MIU; Model Instance MI;
Constraint CT $\in M_{ct}$;
CT: TIME → MIU | $\forall$ t $\in$ TIME: $MI_t \in CT_t$ (constraint validity)

Definition 4.11: EROOS Constraint

The different aspects involved in the definition of an EROOS constraint are:

- The lowercase constraint name refers to the logical constraint in the universe of discourse. The constraints that share a same top class must have different names.
- The top classes enumerate the classes from which the constraint is formulated. They must be the highest classes of the relation hierarchy from which the constraint can be formulated. Using these top classes, the constraint expression may not contain any participation query ↑C, either directly in its specification or indirectly through the use of assistance queries.
- The context clause enumerates the model entities upon which the constraint interacts. Each constraint enforces a logical rule on a certain part of the model

that is affected by the constraint. The context of a constraint can be automatically derived from its logical clause. Each element that is used in the logical expression of the constraint is mentioned in its context.

```
<EROOS kernel constraint script> =
"constraint" <constraint name>
  ( "top class" <TOP CLASS NAME>
  | "top classes" <TOP CLASS NAME> ("," <TOP CLASS NAME> )* )
  "context"
    ( <TOP CLASS NAME> <context clause> )+
  "definition"
    ( "for all" <identifier> ( "," <identifier> )*
      "in" <TOP CLASS NAME>
      ["not participating in" <CLASS NAME> ("↑"<CLASS NAME>)*
        ( "," < CLASS NAME> ( "↑"<CLASS NAME> )* )* ] ":" )+
      <logical clause>
"end constraint" <constraint name>

<context clause> =
  [ "having" ( "attribute" <ATTRIBUTE NAME>
    | "attributes" <ATTRIBUTE NAME> (","<ATTRIBUTE NAME>)+ )]
  [ "having"
    ( "query" <QUERY NAME>
      "returning" <return type> [ <context clause> ]
    | "queries" < QUERY NAME>
      "returning" <return type> [ <context clause> ]
      ( "," <QUERY NAME> "returning" <return type>
        [<context clause>] )+ )]
  [ "having"
    ( "participant (" <descending path> [<context clause>]")"
    | "participants ("<descending path> [ <context clause> ]
      ( "," <descending path> [ <context clause> ] )+ ")" )]
  ["being participant of ("
    <CLASS NAME> [ "/"<ROLE NAME ] [ <context clause> ]
      ( "," <CLASS NAME> [ "/"<ROLE NAME ]
        [ <context clause> ] )* ")" ]

<descending path> =
  ( [ <CLASS NAME> "/" ] [ <ROLE NAME "/" ] <CLASS NAME> |
    [ <CLASS NAME> "/" ] <ROLE NAME>
```

Table 4.7: EROOS Constraint Script

- The '*for all*' clause introduces a number of formal identifier objects of the top classes for the constraint, and should include at least one identifier for every top class. Each formal identifier ranges over all object of its top class. The logical constraint clause is formulated as viewed from these formal identifier objects. As such, the logical clause must be valid for every possible object of the top class, or, in the case that more than one top class are involved, for every possible combination of actual objects of the top classes. Notice that a logical clause using more than one formal identifier of the same top class must also be valid in case that these formal identifiers are substituted with the same actual object of the top class. This means that the constraint must also be valid for *condition(c,c)* in

$$\forall \text{ c1, c2} \in \text{C : condition(c1, c2)}$$

  However, an explicit expression (c1 ≠ c2) could be added to the condition.

- Both a mathematical style, i.e., ∀ c ∈ C, and a verbose style, i.e., **for all** c **in** C, of specification are allowed. To specify a constraint on objects not participating in a certain relation, the *not participating* clause is used. It is possible to specify a '*not participating*' clause involving an indirect participation. For instance, when a constraint is specified for a class A not participating in B ↿ C ↿ D, all objects of class A that do not participate indirectly in D are confined. It is possible to split such constraint in a number of parts, namely (1) all objects of A not participating in B, (2) all objects of A participating in B but not further in C, which must be expressed from the viewpoint of 'B not participating in C', and (3) all objects of A participating in B and further in C but not participating in D, which must be expressed from the viewpoint of 'C not participating in D'. However, such scattering of the constraint is only allowed when the conditions imposed on these different subsets of class A vary.

- It is not allowed to specify a logical clause that can never be satisfied. Therefore, it is also not allowed to specify a constraint with a *not participating* clause and a condition that always fails. Such type of constraint would express an existential dependency from the participant object to the refined object. The constraint

$$\forall \text{ p} \in \text{P not participating in R: false}$$

  would express that an object p of class P that does not participate in a certain relation link r of class R, always results in a constraint violation. This constraint actually obliges that every object p of class P must participate in at least one refined object r of class R. Such constraint would violate the existential dependency hierarchy in EROOS, which expresses that a refined object cannot exist without a participant object, but that an object of a participating class can always exist without having to participate in a refined object. Therefore, it is allowed to specify a constraint on objects that do not participate in a certain relation, but only if such object can fulfil this condition in a certain manner.

- The logical clause of a constraint is a many-sorted first order logic expression that expresses the rule enforced by the constraint on the model. This logical clause is constructed by combining Boolean expressions and other logical clauses, using logical operators such as **not** (¬), **and** (∧), **or** (∨), **xor** (⊕),

if…then… (⇒), **if and only if…then...** (⇔), and **if** a **then** x **else** y ((a ⇒ x) ∧ (¬a ⇒ y)). A Boolean expression is constructed from primitive components, such as objects, values, bags, queries, and primitive operations.

The logical clause however is severely restricted to avoid interference with other EROOS concepts. For instance, it is forbidden to use the participant query ‘↑’ in the logical clause of a constraint, either (1) directly using the participant query in the logical expression, (2) indirectly by using queries that use the participant query themselves, or (3) implicitly by specifying sets of objects ranging over a class that is no direct participant of a top class. In addition, it is not allowed to use the **and**-operator on the top level of the logical clause in order to combine two unrelated constraints. Such constraints must be split into separate constraints, one for each operand. Combining several unrelated constraints in a composite constraint diminishes the reusability and extendibility of a model. It is easier to grasp a number of small individual constraints than to get insight into a complex composite constraint. However, related constraints should best be combined. It is therefore improper to split a constraint requiring ‘a⇔b’ into two constraints ‘a⇒b’ and ‘b⇒a’, or to split a constraint ‘**if** a **then** x **else** y’ into two constraints ‘a ⇒ x’ and ‘¬a ⇒ y’.

4.5.3 Restrictions on EROOS Join Constraints

As stated earlier, the logical clause of an EROOS constraint is severely restricted to avoid interference with other EROOS concepts. Constraint types that must be avoided are certain forms of join constraints. Join constraints put an equality on a certain number of their (mostly indirect) participants. Join constraints state that objects obtained by following different relational paths through the model, starting from a certain class, must be equal. We can identify three kinds of join constraints:

- A join constraint for a class C that puts a restriction on objects of a pure indirect participating class, meaning that the class is no direct participant of C, is allowed.
- Constraints on binary relations having two identical participating classes, and that are expressing an obligation for the two participant objects to be equal, are forbidden. Such binary relations must be transformed into unary relations.
- Constraints on binary relations for equality between a direct participant object and one of their indirect participants are also forbidden in EROOS. Such binary relations must also be transformed into a unary relation.

The reason behind the restriction of these kinds of constraints is to avoid (1) information duplication, and (2) interference between binary and unary relations. In such cases, EROOS forces the analyst to make use of the unary relation concept instead of using a binary relation with a corresponding join constraint. As illustrated in Figure 4.12, the second participant of the relation adds no additional information to the model, and thus is superfluous, while both participants in Figure 4.13 express specific information that cannot be omitted. This restriction on join constraints complies with the key conceptual modelling principles of Uniqueness, No Redundancy, and Model-Implied Constraints, because it forces the analyst to a unique

conceptual model that is as small as possible, using existential dependency as the core criterion for determining its structure.

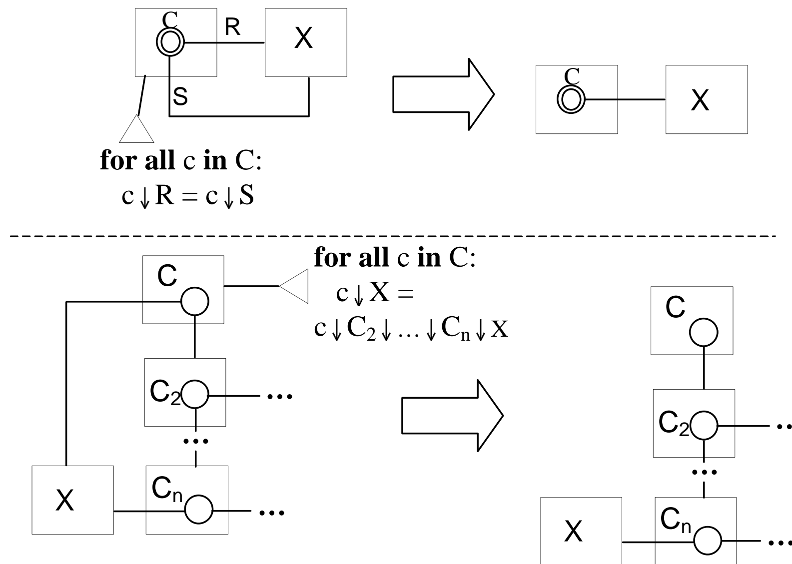


Figure 4.12: Forbidden EROOS Join Constraints

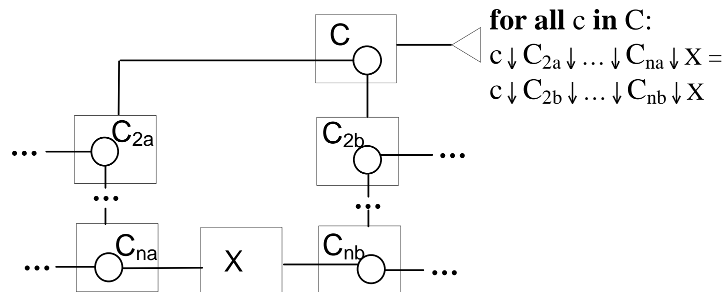


Figure 4.13: Allowed EROOS Join Constraint

4.5.4 EROOS Constraints for the Library Example

Given the example of the library system presented in Section 2.3, and the relation hierarchy defined in Section 4.4.7, a large set of constraints must be added to this model. There are two types of constraints that must be added to the model:

- *Logical rules that are a reflection from impossibilities in the universe of discourse.* There are a lot of constraints that have to be added to the conceptual model in order to enforce consistency that is embedded in the nature of the universe of discourse. Examples of such constraints are the fact that a person cannot perform any borrowings when that person has died, or the fact that no new copies of a book can be printed when the publisher has gone into liquidation,

as presented in Table 4.8. This type of constraints poses restrictions on the creation of certain objects when the participant object is already participating in another relation.

- *Logical rules that have to be imposed in the universe of discourse.* Such constraints impose rules that restrict unwanted behaviour. There are a number of subtypes that can be identified:
  - Constraints that restrict the creation of certain objects with a participant object that is already, or, on the contrary, is not yet participating in another relation. Examples of such constraints are the fact that a person cannot perform any borrowing when that person has been deregistered from the library, or that a person cannot perform any borrowings when that person has unpaid fines.
  - Constraints that restrict the creation of certain objects because specific objects already exist. Examples of such constraints are the fact that a book copy can only be in possession of a single library, the fact that a book copy can only be borrowed when there is no other active borrowing for that copy, the fact that for each person only a single active borrowing can be attached to an allowance object, or the fact that a person cannot borrow a second copy of the same book, as presented in Table 4.9.
  - Constraints that restrict the existence of objects based on certain attribute values. Examples of such constraints are the fact that a fine must exist when the maximum lending period has been exceeded. Notice that it is needed to explicitly model the progress of time in the EROOS kernel in order to be able to specify this constraint. Since the fine constraint obliges the creation of a fine when the deadline is exceeded, the progress of time would stop when this constraint would be violated. The only manner, in which a modeller can automatically create a fine, is through the explicit modelling of the progress of time. The EROOS universe offers constraint triggers in Section 5.3.10 in order to specify such time-triggered behaviour.
  - Join constraints that request the equality of certain participants. Examples of such constraints are the fact that a person can only select and borrow a book from the same library as where that person is registered, and the fact that a person can only pay the fine of a borrowing if that person already has returned the book copy.

An observation that can be made is that the constraint regarding the number of allowed borrowings cannot be expressed using an EROOS constraint, since such constraint would need an integer attribute representing the maximum number of lending items. Because EROOS does not allow to model integer attributes, the modeller is forced to reify the attribute into a class that represents the possibility of borrowing a book. The constraint regarding the number of allowed borrowings is transformed into an existential dependency restriction from a borrowing object on an allowance object.

```
constraint no printing when in liquidation
  top classes
    COPY, LIQUIDATION
  context
    COPY having participant (PUBLISHER),
    LIQUIDATION having participant (PUBLISHER)
  definition
    for all copy in COPY:
    for all liquidation in LIQUIDATION:
     if copy↓PUBLISHER = liquidation↓PUBLISHER then
       copy→CreationTimestamp
         < liquidation→CreationTimestamp
end constraint no printing when liquidation
```

Table 4.8: EROOS Constraint for No Printing when in Liquidation

```
constraint single copy borrowing
  top class
    BORROWING
  context
    BORROWING having participant (
      SELECTION having participants (REGISTRATION,
        POSSESSION having participant (
          COPY having participant (BOOK))))
  definition
    for all b1,b2 in BORROWING:
     if (b1 ≠ b2) and
       b1↓SELECTION↓REGISTRATION = b2↓SELECTION↓REGISTRATION
     then b1↓SELECTION↓POSSESSION↓COPY↓BOOK
          ≠ b2↓SELECTION↓POSSESSION↓COPY↓BOOK
end constraint single copy borrowing
```

Table 4.9: EROOS Constraint for Single Copy Borrowing

4.5.5 Contributions, Related Work, and Reflections

Our contributions concerning the constraint concept are the following:

- In addition to a large number of constraints that are implied by the EROOS model structure, EROOS constraints offer the possibility of modelling **constraints as a first-class model concept**. Using a formal notation, model

constraints can be superimposed on a model in order to express rules and regulations of the universe of discourse. Our work, first published internationally in 1992 [153], predates and is largely comparable with OCL, which originated in 1995 within IBM [161].

- Contrary to OCL, EROOS forces a **single and unique manner for specifying EROOS constraints.** This is achieved by (1) the obligation to formulate constraints from the top class(es) in the relation hierarchy, and (2) the introduction of the *not participating* clause. The advantage of a unique specification manner for a constraint concerns the fact that it provides solid criteria in developing conceptual models. This leads to a single common model among all analysts involved in the development of a conceptual model. EROOS also explicitly inhibits the definition of an EROOS constraint that can logically be derived from other constraints already present in the conceptual model. As such, the specification of constraints in an EROOS model can be restricted to the set of relevant constraints, and does not include a huge set of rather trivial derived constraints.
- Contrary to OCL, EROOS forces analysts to use implied constraints whenever appropriate. The formalism provided for expressing EROOS constraints is developed as such that **it is impossible to express implied constraints using the concept of an EROOS constraint**. This is achieved by the prohibition of using the participation query ($\uparrow$) in the formulation of an EROOS constraint.

The introduction of formal model constraints as a first-class model concept in EROOS is largely comparable with the Object Constraint Language (OCL) [108][161]. A major difference between OCL and EROOS is the viewpoint from which the constraint can be formulated. In the EROOS methodology, each constraint has a single and unique viewpoint from which it is formulated, namely the top class or top classes in the relation hierarchy that are involved in the constraint. EROOS forces the modeller to use this strict constraint specification viewpoint in order to obtain uniqueness for conceptual modelling. OCL has very loose specification rules, and puts no restrictions on the constraint expressions, or on the specification viewpoints. A constraint in OCL can be formulated from the viewpoint from any involved class.

An observation that can be made is that for the specification of events, as introduced in Section 4.8, an analyst often has to interpret and to circumvent the model constraints that must be obliged at all times. When specifying the effect the event has on the model, the analyst must take care that all model constraints remain valid. Otherwise, the event will violate a model constraint and will be refused. This leads to a recurring pattern of (1) describing the standard behaviour of an event, (2) checking whether the state of the new model instance remains valid, and (3) providing a constraint exception handling mechanism that tries to resolve the constraint violation. As such, the model contains a lot of duplication of constraint checking and resolving descriptions inside the event specifications. A generic mechanism to detect and react to constraint violations would be appropriate in order to avoid the repetition of this kind of constraint checking and handling specifications. As part of the EROOS

universe, Section 5.3.9 presents a mechanism for the specification of constraint triggers that can intervene and extend the event behaviour whenever a constraint could become violated, and a mechanism for nondeterminism in EROOS specifications in order to facilitate the selection of appropriate model instance items that comply with all stated model constraints.

A second observation is that for the specification of time related restrictions in the EROOS kernel, an analyst is obliged to model the progress of time explicitly. Although it is possible to express time related constraints, there is no possibility to capture a violation of a time related constraint caused by the progress of time. One could use the indication of the current time at the moment of evaluation, indicated as **now,** and express an explicit condition that this value must be smaller than a certain moment in time. But it is impossible to specify that an event will be triggered when the specified constraint would be violated. For example, it is impossible for the library system to specify that a fine object must automatically be created when the borrowing reaches its expiry date. Such constraint would result in a time standstill in the model, which would correspond with an erroneous situation. A possible solution to model this kind of time triggered behaviour in the EROOS kernel would be the explicit modelling of the progress of time. As such, at each moment the time progresses in the model, one could check whether a certain time related constraint could be violated, and respond to it by triggering the behaviour that must be executed when reaching the deadline, e.g., raising a fine, switching to an alarm level, starting corrective or repossession measures, et cetera. The constraint triggering mechanism of the EROOS universe presented in Section 5.3.9 can facilitate the specification of such kind of time-triggered behaviour.

A third observation is that, while EROOS constraints impose restrictions on a conceptual model that must always be satisfied, some situations could demand for a set of rather contradicting rules applicable on certain information. This is often the case in planning and scheduling systems. Fur such systems, only a limited number of constraints must be satisfied at all times, called crisp constraints, while a large set of rules are used to define satisfaction levels for evaluating the obtained solutions. This type of rules is called soft constraints [51][14], which define a level of preference or a level of importance concerning the satisfaction of the rules. These rules are not real constraints in the sense that they must be satisfied at all times, but they define satisfaction selection rules that can discriminate the set of possible solutions complying with all crisp constraints. Integrating such approach of soft constraints into EROOS would provide a better support for planning and scheduling systems. Although the universe of discourse concerning a planning and scheduling system is quite easy to model in EROOS, a model for the calculation of the most optimal solution is much harder. However, it is possible to obtain an elegant solution of a planning system using constraint triggers and nondeterminism, as is demonstrated in the case studies of the electronic agenda system.

4.6 Is-A Specialisations and Static Subdivision

In addition to concrete classes as presented in Section 4.2, EROOS introduces another type of classes for classification purposes, namely abstract classes. Contrary to concrete classes, abstract classes do not have objects on their own. The collection of objects associated with an abstract class is instead obtained by joining the respective collections of objects of other classes for which the abstract class is an abstraction. The concept of specialisation is used in EROOS to express an '*is a (kind of)*' meta-relationship between two classes, namely a generalised abstract class and a specialised class. We call a specialisation between classes a meta-relationship since it relates the classes on a meta-level rather than relating the objects of the class. This meta-relationship models the fact that objects of a number of classes resemble each other. The abstract class is statically subdivided into a number of specialised classes. Notice that a specialised class can be a leaf class in the inheritance hierarchy, in which it is a concrete class, but also an abstract class in turn, having again a number of specialised classes as its descendants. Next to specialisation, which concretises abstract classes into specialised classes, one could also identify the inverse concept of generalisation, which abstracts specialised classes into generalised classes. The resemblance of classes can come from two sources: (1) observable features, and (2) common sense knowledge.

- According to the first source, objects of two classes resemble each other if they share common structural and behavioural features. These features can be defined on the level of the abstract class. For instance, shared structural features can be a number of attributes, relation refinements, relation participations, and constraints. These features are called observable because an analyst can detect them in the universe of discourse, by closely observing and looking for commonalities.
- The second source of detecting resemblances between classes is common sense and background knowledge of the universe of discourse. Two objects resemble each other because they are known to do so in the universe of discourse. Although resemblance can be a feature or property on its own, this kind of resemblance often is, but does not have to be supported by a number of common observable features.

By using the concept of specialisation, the common features of classes can be expressed once only, without introducing redundant specifications in the model. The specialised class inherits all features from the generalised class. In addition, it is possible that a single class is a specialisation of more than one generalised class at the same time, inheriting all features from each generalised class. Each specialisation hierarchy can be looked upon from two viewpoints. On the one hand, one can consider the generalised class as the entity in the focus of attention. The specialisations are then special cases of this generic concept. On the other hand, when the specialisation classes are considered to be the entity in the focus of attention, the generalisation class is the description of the common features of all specialised classes. Which of these viewpoints is commonly used, depends on the context or on

the information that the observer uses to reason about the universe of discourse. However, both approaches lead to the same specialisation hierarchy in EROOS.

4.6.1 Is-A Specialisation versus Subclassing versus Subtyping

As argued by Lalonde [87], the '*Is-A (kind of)*' meta-relationship in EROOS is different from subclassing, also called inheritance in object-oriented programming languages, and from subtyping. The difference lies in the criteria to be used when evaluating whether a class should be a descendant of another class. What is more general in one viewpoint could be described as more specialised in another viewpoint:

- Subclassing, also known as inheritance, is an implementation mechanism that allows a programmer to share code and representation by letting a class inherit all code, including methods and instance variables, from another class. The subclass can hide or overwrite certain methods in order to fine-tune its own implementation. It is a form of implementation reusability by inheriting rather than copying code.
- Subtyping is a behavioural substitutability relationship that follows the Liskov Substitution Principle (LSP) [92]. It uses the criterion of *substitutability*, namely that an instance of a supertype can be substituted by an instance of a subtype without noticing any differences. How the subtypes are implemented is totally irrelevant, as long as they have the right interfaces and behaviour to be substitutable. A class is thus a special case of another class if it provides at least the same services, including the same interface and behaviour, as the original class, without violating any additional explicit or implicit suppositions. It is possible that the specialised class provides more services than the general class, or that existing services are extended such that they cover more cases. As long as the original class does not rely on certain extensions not being covered, e.g., by relying on the occurrence of certain exceptions to be thrown, additional extensions can be made while adhering to the criterion of substitutability.
- The '*Is-A (kind of)*' meta-relationship, as being used in EROOS, follows the rules of the logical specialisation relationship between classes. A class is a kind of other class when it complies with the specification of that class, both on a structural as well as on a behavioural level. This means that its objects can be seen as objects of the other class in the universe of discourse. A specialised class must therefore comply with all the features defined for the generalised class, such as attributes, relation refinements, relation participants, constraints, and functionality. Structural elements and functionality correspond to a number of implicit and integrated constraints that are incorporated in the methodological concept. For instance, if a class is decorated by an attribute, all objects of that class must have a proper domain value for that attribute. Likewise, all objects of a refined class must have an object of each of its participating classes associated with them. Therefore, a class is a specialisation of another class if it satisfies at least all constraints, both implicit, explicit, as well as EROOS constraints, specified for the generalised class. It is possible that the objects of the more specialised class satisfy more constraints than the objects of the more general

class. Specialisation thus corresponds to the process of more strictly defining a specific class by restricting it with more constraints through the definition of additional model entities.

Because objects of the specialised class must satisfy more constraints, they are likely to be less general in use. Therefore, they can probably provide less functionality than the more general class. On the other hand, objects of the specialised class can have more structural features attached to them and provide additional functionality related to these features. Notice that '*is a (kind of)*' is sometimes also referred to as subtyping, for example by Wegner [162] which defines it as the addition of predicates that constrain the structure of expressions.

4.6.2 Specialisation Partitions and Multiple Generalisations

As presented in Section 4.2, each concrete EROOS class is associated with a collection of objects. The creation event of a concrete class adds a new object to the population of the class. Objects of a specific concrete class cannot be created, queried or manipulated by the functionality of another concrete class. Therefore, the collections of objects associated with concrete classes are disjoint. Moreover, each object belongs to exactly one collection of a concrete class.

In contrast, abstract classes are not associated with an object collection, because they do not have objects of their own. They merely describe common features of objects from a number of specialised classes. Therefore, an abstract class is said to be associated indirectly with a collection of objects, namely the union of the direct and indirect associated collections of all classes it generalises. A class that is specialised in a number of other classes is always an abstract class in EROOS, while a class that has not been specialised, is always a concrete class. Figure 4.14 illustrates how the collections of objects that are associated with classes, are related to each other. Parts of the ellipses representing the collections are hatched to indicate that these parts of the collection are empty and do not contain any objects.

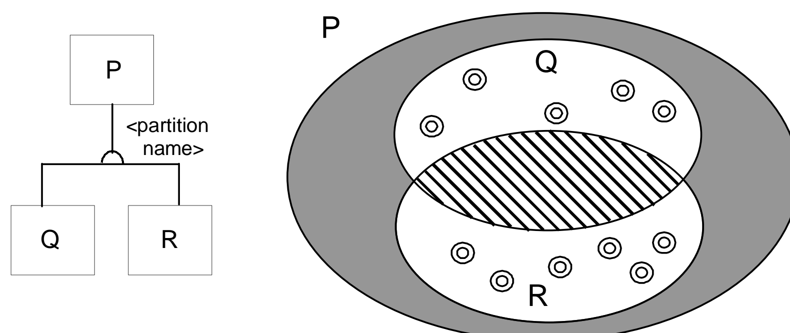


Figure 4.14: Graphical Representation of an EROOS Specialisation

Not only is the collection of objects associated with the generalised class equal to the union of the collections of the specialised classes, but also no object may belong to two or more specialised classes in a specialisation hierarchy. Specialisation classes that are specialised from the same generalised class must completely partition the objects described by that generalised class into disjoint subsets. It is possible to specify more than one partition for a generalised class. When two specialised classes must share some objects, they must be defined in different specialisation partitions. When more than one partition is defined for a specialisation, they must be orthogonal partitions, since each must fully partition the same collection of the generalised class in disjoint subsets. All objects described by the generalised class must be fully qualified for each partition, i.e., they must be assigned to a specific specialised class according to all partitions of that generalised class. Therefore, when more than one partition is specified for a specific class, all direct specialisations of that class must be abstract classes since an object may belong only to a single concrete class. Moreover, all further concrete specialisation classes lower in the specialisation hierarchy must be specialisations of one class from each partition. Partitions can be named or nameless. Specialisation relationships with no partition name are assumed to belong to the same default partition. Figure 4.15 shows two specialisation partitions P and Q for a single class C, and the corresponding collections of objects. Notice that both partitions P and Q are complete classifications of the objects in class C, but neither describe the objects completely. Therefore, the classes P1, P2, Q1, and Q2 are abstract classes, while only the classes C11, C12, C21, and C22 are concrete classes.

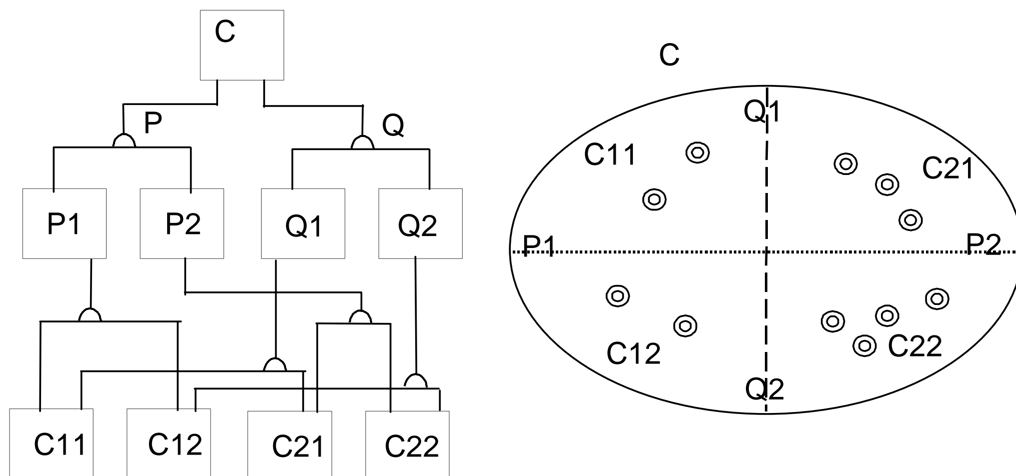


Figure 4.15: EROOS Specialisation Partitions

A class can be a specialisation of more than one generalised class at the same time, but only when the generalised classed belong to a different partition. When more than one partition is present, it even must be specialised from one generalised class of each partition. In order to prevent multiple specialisations coming from rather unrelated classes, a multiple specialisation has the restriction that all generalised classes must be specialised from a single most general class.

The EROOS methodology explicitly prohibits the definition of identical partitions, since they do not provide any added value to the conceptual model. In addition, specialisation classes in a partition that cannot contain any objects, e.g., due to the fact that its generalised classes have contradictory constraints attached so that an object can never belong to both classes, are also forbidden.

4.6.3 Specification of an EROOS Specialisation

The syntax of an EROOS specialisation script is given in Table 4.10. A specialisation script can be formulated from two viewpoints: The viewpoint of the generalised class, for which all partitions and its specialised classes are defined, and the viewpoint of the specialised class, for which all involved partitions and its generalised classes are defined. As presented in Figure 4.15, an EROOS specialisation partition is graphically represented in the form of a semi-circle with the generalised class attached to the curve and all specialised classes attached to the bottom.

```
<EROOS specialisation script> =
"class" <CLASS NAME>
  "definition"
    "specialisation"
    "(" <generalisation clause>
      ( "," <generalisation clause> )+ ")"
"end class" <CLASS NAME>

<generalisation clause> =
"of" <CLASS NAME> ["according to partition" <partition name>]

<EROOS generalisation script> =
"class" <CLASS NAME>
  "definition"
    "specialised"
    "(" <specialisation clause>
      ( "," <specialisation clause> )+ ")"
"end class" <CLASS NAME>

<specialisation clause> =
[ "according to partition" <partition name> ] "in"
  "(" <CLASS NAME> ( "," <CLASS NAME> )+ ")"
```

Table 4.10: EROOS Specialisation Script

4.6.4 Model Constraints implied by the Specialisation Concept

EROOS incorporates important model constraints directly in the methodological concepts. The following constraints are directly implied by the specialisation concept:

- **Abstractness**: Generalised classes do not have objects associated with them that are not also associated to a concrete class. Generalised classes are considered abstract. They are only indirectly associated with a collection of objects through the union of the collections of all direct and indirect specialisation classes. Each object is directly associated with a concrete class and can be indirectly associated to a number of generalised classes.
- **Immutability**: The association of a given object with its generalisation classes is considered to be static. In particular, at the moment an object is to be created, it will be associated with a single concrete class and it will keep that association for its entire lifetime. In addition, the associations with the generalised classes are also established at the moment of object creation and cannot be altered in a later stage of the object lifetime.
- **Finiteness**: The collection of objects associated with any generalised class will always be finite, since it is the union of the finite object collections that are associated with its specialisation classes.

A **specialisation** is a model entity defining an 'is a kind of' meta-association between classes, in which the objects of the specialised class derive all model entities that have been defined for the generalised class.

An **abstract class** is a class that does not define its own object population set. Instead, it has a derived object description set that is equal to the union of the object description sets associated to its specialised classes. The object description set of a concrete class is equal to its object population set.

A **partition** is a set of specialisations for a single generalisation class, in which the object description set of the generalisation is equal to the complete and non-overlapping union of the object description sets of the specialisation classes in the partition.

Given

Model M; Object Universe OU; Class $C^1,\ldots,C^n \in M_{cl}$;

Abstract Class A $\in M_{acl}$; Partition P $\in M_p$;

A: TIME $\rightarrow \mathcal{P}$(OU) |

$\forall$ t $\in$ TIME: P(A) = $\{C^1,\ldots, C^n\} \Rightarrow A_t = C^1_t \cup \ldots \cup C^n_t$ (partition

$\forall$ t $\in$ TIME: $\forall$ C, C' $\in$ P(A): $C_t \cap C'_t = \emptyset$ disjunctness)[27]

Definition 4.12: EROOS Specialisation

[27] This definition also implies abstractness, immutability and finiteness, since an abstract class is the union of its specialised classes, each of them conforming to the properties of immutability and finiteness.

- **Partition Disjunctness**: In EROOS, different specialised classes in the same partition are assumed to divide the universe of objects into disjoint collections. Each object associated with a generalised class must be associated with exactly one specialised class in a partition. Whenever two specialised classes must share a number of objects, they should be defined in distinct partitions. Different specialised classes in the same partition are not allowed to share objects.

The definition of a specialisation can be found in Definition 4.12.

4.6.5 Strengthening Constraints for a Specialisation

Specialisation can have an impact on other entities present in an EROOS model. Functionality and constraints associated with attributes and relations can be restricted for the specialised class. New structural elements can be specified for a specialised class, and a specialised class also inherits the features from its generalisation classes. All attributes, relations, participations, EROOS constraints, events, and queries of the generalised class are inherited by its specialisation classes. Consequently, amongst others, if the more general class has been refined, no new relation can be specified for its specialised classes.

Although abstract classes have a creation event defined for them, it can never be applied directly because an abstract class cannot have objects of its own. Therefore, the creation event only serves as a contract for the specialised classes to be obeyed. The effect of a creation event for an abstract class is to add a new object to the derived population associated to that class. Since the specialised creation event adds an object to the population of its class, and since the population of an abstract class is the mere union of the populations of its specialised classes, a new object will thus be indirectly added to the derived object collection of the abstract class.

The model entities and functionality of a specialised class come from two sources: (1) the own model entities and functionality that is directly specified for the specialised class, and (2) the functionality defined for all direct and indirect generalised classes from which the specialised class is derived. As such, all constraints specified for the generalisation class are guaranteed to be satisfied by all objects of the specialisation class. However, objects of the specialisation class cannot guarantee to behave exactly as specified for the generalised class, due to the fact that extra constraints can be introduced on the level of the specialised class. Therefore, certain events defined on the level of the generalised class could possibly violate the rules specified for the specialisation class, and can be refused whenever they occur.

Adding new constraints on the specialisation level can lead to strengthening attributes and participants of a relation. Strengthening an attribute can be done by adding an integrated '*unique*' constraint to the specialised version, or strengthening the lower and upper bounds for the allowed attribute values, thus increasing the lower bound or decreasing the upper bound.

As illustrated in Figure 4.16, strengthening a participant can be done in three manners:

- *The strengthened participant can have the original participant as its direct or indirect participant.*

  A participating class A can be replaced by a refined class C in which A participates. This participation does not have to be direct, as long as the new participating class is somehow dependent on the original participating class. Strengthening a participating class corresponds to limiting the kind of objects that can participate in the relation. The restriction states that only objects of the new participating class C, which are guaranteed to be dependent on an object of the original participating class A, are now allowed to participate in the restricted relation. It adheres to the constraint on the generalised level, since the newly refined object of class C guarantees to have the original participant object of class A as one of its own participants. A conformance rule for the strengthening must be defined, but is mostly automatically deducible.

- *The strengthened participant can be a direct or indirect specialisation of the original participant.*

  A participating class X can be replaced by one of its specialised classes Y. This specialised class can be a direct or an indirect specialisation of X. The restriction corresponds to limiting the kind of objects that can participate in the relation, namely, only objects that belong to the specialised class. It adheres to the constraint on the generalised level, since the newly specialised object of class Y guarantees to adhere to the original generalised class X.

- *The strengthened participant can have the original participant as its own direct or indirect specialised participant through a number of relations and specialisations.*

  This is a combination of the previous two cases, in which the specialisation can be applied on any of the intermediate classes between the new participating class and the old one.

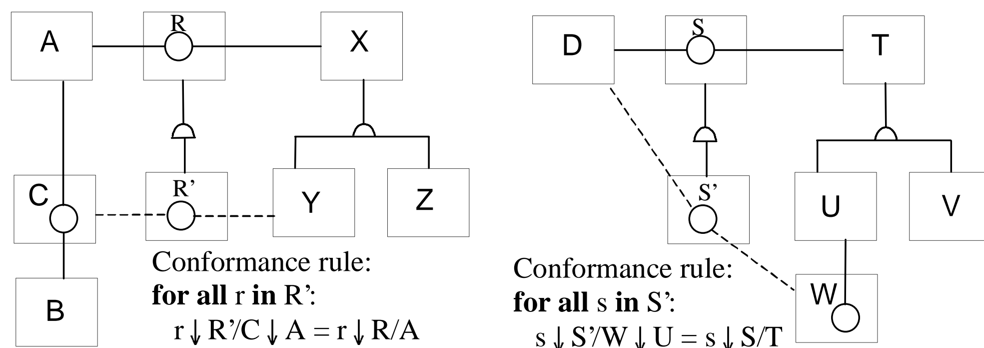


Figure 4.16: Strengthening a Participant for an EROOS Specialisation

Note that strengthened model entities are distinguished from newly defined model entities by using a dashed line to represent them. In this way, it is directly visible that these model entities are not newly defined for the specialised class, but are actually strengthened features inherited from a generalised class. All model entities that have not been strengthened will be derived automatically in their most restrictive form.

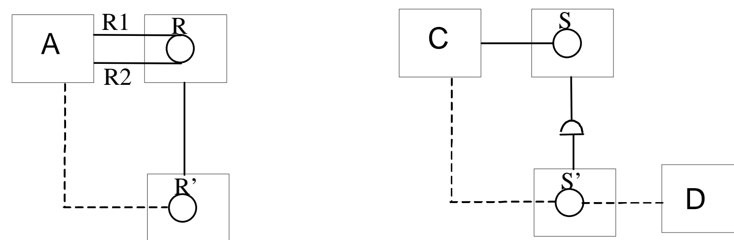
As illustrated in Figure 4.17, strengthening a relation by changing the arity of it can be done in two manners:

- *A binary relation can be restricted to a unary relation.*

  A binary relation R that relates the same participating class A twice, can be strengthened to a unary relation when the participant objects must be equal at all times for the strengthened relation. In fact, this is the only valid specification to model such rule in EROOS given the specific restrictions on join constraints as described in Section 4.5.3. It would therefore be forbidden to specify an additional EROOS constraint for expressing this restriction. An example of such restriction is a manager-subordinate relationship, for which a specialisation could be defined for a CEO, expressing that this person manages oneself.

- *A unary relation can be extended to a binary relation.*

  Unary relations can be extended to binary relations by specifying a new, second participant for them. This participant does not have somehow to be related to the first participant, since it is an additional element that is introduced on the level of the specialised class. Such specialisation of a unary into a binary relation can be seen as an extension, since an additional participant D is added to the relation for S having participant C, but also a restriction, since objects of the specialised class cannot exist anymore without being dependent on a specific object of class D. When it is not obvious which participant is the original one, a conformance rule for the strengthening must be defined.



Conformance rule:
for all r **in** R':
$r \downarrow R'/A = r \downarrow R1 = r \downarrow R2$

Figure 4.17: Changing the Relation Arity for an EROOS Specialisation

Although implied and integrated constraints for attributes and relations can be strengthened, an EROOS constraint cannot be strengthened. We could imagine

introducing EROOS constraint specialisations in order to obtain a stronger rule on the specialised level than on the generalised level, but the distinction between strengthening an EROOS constraint and introducing a new EROOS constraint for the specialised level is unclear and ambiguous. Therefore, EROOS always treats the strengthening of an EROOS constraint as the introduction of a new EROOS constraint on the specialised level.

4.6.6 Causal Dependency for Specialisations

When evaluating the appropriateness of the EROOS methodology during case studies and student projects, as explained in Section 6.1.4, we noticed that a number of analysts often introduce a large number of specialisations that could be deduced from (1) a specific root specialisation, or (2) other information inside the conceptual model. When such causally dependent specialisations can be defined in an EROOS model, the methodology would violate the Principle of No Redundancy as defined in Section 3.2. Causal dependency between specialisations is illustrated in Figure 4.18. A class R can be refined with a relationship having participant X. When X is specialised in 2 specialisation classes Y and Z, it implicitly partition the class R into 2 collections of objects, namely (1) the objects having a participant object belonging to the specialised class Y, and (2) the objects having a participant object belonging to the specialised class Z. When class R would also be partitioned to express this partitioning in an explicit manner, the partitioning for class R can be deduced from the partitioning of its participating class X. In order to avoid such derived partitions, it is forbidden in EROOS to specify a partition for a refined class that can be deduced from a partition for one of its direct or indirect participating classes whenever the partition does not include any additional information in the model. In Section 5.4.7, we describe the concept of groups to specify a collection of objects that can automatically be selected based on existing information contained in the model.

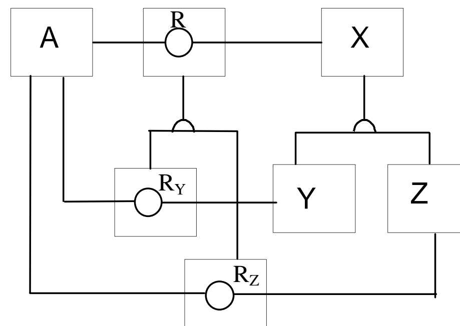


Figure 4.18: Forbidden Causal Dependency between EROOS Specialisations

Analogous to the causal specialisation dependency, it is also forbidden in EROOS to introduce a specialisation that is causally dependent on other information inside the

model. These kinds of causal model dependencies can be identified by the usage of two EROOS constraints, stating a specific condition for the one subclass and its negation for the other subclass. Also in this case, a group should be used to divide objects based on existing information contained in the model.

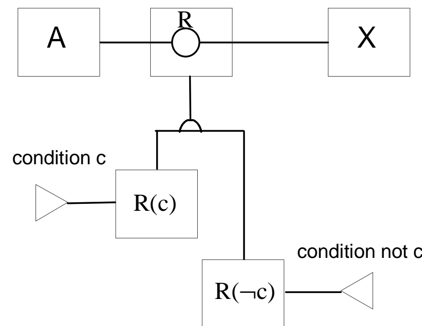


Figure 4.19: Forbidden Causal Model Dependency for an EROOS Specialisation

4.6.7 Implicit Specialisation Queries

The definition of a specialisation introduces an implicit query **in**, also called **element of** or '∈', to check whether an object of the generalised class is an actual member of a specialised class. The definition of this implicit query can be found in Definition 4.13.

```
An implicit query 'in', 'element of', or '∈' for a concrete
or abstract class is a query that can be applied on an
object, and that returns the fact whether this object belongs
to the object description set of that class.
Given
Model M; Object Universe OU; Query in ∈ Mq;
in: TIME → ( (OU x Mcl ) → Boolean) |
   ∀ t ∈ TIME: ∀ o ∈ OU: ∀ C ∈ Mcl :
     o in C ⇔ o ∈ Ct
```

Definition 4.13: Implicit EROOS Specialisation Query

4.6.8 EROOS Specialisations for the Library Example

Given the example of the library system that was presented in Section 2.3, and the relation hierarchy that was defined in Section 4.4.7, a few specialisation partitions can be identified. However, one important specialisation that can be defined is the generalisation of the main author and secondary author into a single class AUTHOR, as presented in Figure 4.20. In addition to the grouping of all authors for a book, the

order in which the authors are ranked can now be captured explicitly in the model. The following observations can be made:

- The class AUTHOR represents all authors for a book. It is refined with a unary relation having PERSON as the participating class, representing the persons that are authors of a book. Since a person can be author of many books, the multiplicity value is defined as 'many'.
- The class BOOK is a specialisation of AUTHOR, thereby inheriting the relation to PERSON. This relation represents the main author for a book. Since a book and its main author are mutually dependent, the EROOS kernel forces the analyst to merge these two facts into a single object called BOOK. A book represents the concept of a book and the main author at the same time.
- The class of SECONDARY AUTHOR is also a specialisation of AUTHOR, thereby inheriting the relation to PERSON. This participant is a person that is one of the secondary authors of a book. However, we have specialised the relation from a unary to a binary relation, expressing the fact that the secondary author follows another author, who, in turn, can be the main author, or another secondary author.

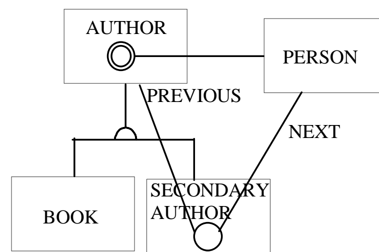


Figure 4.20: EROOS Specialisation for the Library System

The specialisation hierarchy give rise to a dependency chain between the authors:

- The main author, expressed by the class BOOK, is dependent only on a single person, being the main author
- The second author, expressed by the class SECONDARY AUTHOR, is refined with a relation link between the person being the second author, and the main author, expressed by the class BOOK that is generalised as AUTHOR.
- The third author, expressed by the class SECONDARY AUTHOR, is refined with a relation link between the person being the third author, and the second author, expressed by the class SECONDARY AUTHOR that is generalised as AUTHOR.

The result of such model structure, is that the last author is dependent on its predecessor, etc., the second author is dependent on the first author, while the first author, being merged with the book object, is not dependent on any other authors.

4.6.9 Contributions, Related Work, and Reflections

The EROOS specialisation concept is largely comparable with the generalization concept in UML. Our contributions concerning the specialisation concept are:

- The mechanism of **strengthening constraints for a specialisation** is a key contribution of our work. A relation participant class, expressing existential dependency of a refined object on a participating object, can be strengthened to a class that has the participant class as a direct or indirect specialised participant class through a number of relations and specialisations. This enables the modeller to express more stringent dependencies for a specific subset of a refined class.
- The systematic approach to specialisation, obliging (1) **partition disjunctness** for every specialisation hierarchy, (2) the strict separation between abstract generalisation classes and concrete leaf classes, and (3) the prohibition of causal dependency, forces the analyst to modelling clean specialisation structures and surveyable multiple inheritance trees.

An observation that can be made is that in order to express dynamic specialisation, a query must be defined returning the fact whether an object belongs to a certain dynamic subset or not. If this information cannot be derived from the information already contained in the model, an explicit class must be added to the model reflecting this fact. As an example, a dynamic class of adults cannot be derived from a class of persons, but it should be modelled as a query for a person returning the fact whether the person is an adult or not. In addition, it is impossible to derive a dynamic class of students from a class of persons, but it should be modelled as a class of enrolments refined with a person and an institute. A query for a person can return the fact whether the person is registered as a student or not. Thus, dynamic subgroups are only implicitly present in a model and cannot be made explicit. It would be convenient to highlight such dynamic subsets directly inside a model.

4.7 Queries and Ornamentation

In previous sections, we already introduced a number of implicit EROOS queries in order to retrieve information about the model instance at a specific moment in time. As such, an attribute automatically introduces a decoration query (→), a relation defines a number of refinement (↓) and participation (↑) queries whereas a specialisation gives rise to the introduction of a specialisation query for each class (***in***, **element of** or ∈). This section introduces the general concept of a query for enlarging the information retrieval capabilities in a model and defining facilities for extracting derived information from a model instance.

4.7.1 Specification of an EROOS Query

In general, a query offers the ability to inspect the properties of objects in a model instance. In order to be able to inspect the model instance at a certain moment in time,

the analyst can define EROOS queries. Queries are said to *ornament* a class. An EROOS query is specified in a query script, as given in Table 4.11.

```
<EROOS query script> =
"class" <CLASS NAME>
  "context"
    <context clause>
  "query"
    <query name> [ "(" <parameter name> : <TYPE NAME>
         ( "," <parameter name> ":" <TYPE NAME> )* ")" ]
    "returns" <TYPE NAME>
    "result" <query expression>
"end class" <CLASS NAME>
```

Table 4.11: EROOS Query Script

The different components in the specification of a query are the following:

- The query name, represented in lowercase, should provide a good description of the information to be returned. Query names are not restricted to verbs. However, the name of a query must differ from the names introduced for other queries of the class, as well as from the names used for the events associated with that class.
- A query can introduce, by means of successive formal arguments, symbolic names for values and objects to be supplied each time the query is instantiated.
- The final component of a query script specifies the result type and the actual result to be returned by the query. The query expression must return an object, value, or set, according to the defined return type. The expression is evaluated at the time the result of the query is needed. The expression determining the result of a query, is to be built from the object on which the query is applied (**self**), the values and objects that serves as actual arguments for the query, the default EROOS attributes, domain values, and domain functions.

Since the focus of this text is on the constraint-centric approach in EROOS, we refer to the EROOS Reference Manual [143] for a complete and in-depth description of EROOS queries. As presented in Figure 4.21, queries are represented graphically by means of a circle connected to the class of the object on which the query applies. The name of a query is preceded by a question mark.

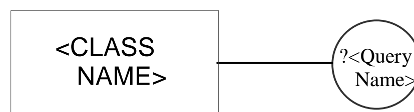


Figure 4.21: Graphical Representation of an EROOS Query

4.7.2 Examples of EROOS Queries

As an example of an EROOS query, the age of an object can be defined by comparing the time at the moment the query is evaluated with the Creation Timestamp of the object. The specification of such query is given in Table 4.12. When the age of an object is an important semantic property in the universe of discourse, it should be introduced in the model and be given a proper name for describing this property.

```
class C
  context
    having attribute Creation Timestamp[28]
  query
    age returns Duration
      result now - self→Creation Timestamp
end class C
```

Table 4.12: Example of an EROOS Query Script for an Object Age

As another example of a query for a relation, one can define a query for a class participating in two relations. The query, given in Table 4.13, has to check whether the object participates in at least a relation link of each relation.

```
class P
  context
    being participant of (R, S)
  query
    full participation returns Boolean
      result (self↑R ≠ empty set) and (self↑S ≠ empty set)
end class P
```

Table 4.13: Example of an EROOS Query Script for a Dual Participation Check

4.7.3 EROOS Queries for the Library Example

Given the example of the library system presented in Section 2.3, and the relation hierarchy defined in Section 4.4.7, we could define a large number of useful queries. We restrict ourselves to two examples. A query returning the number of books that a person has borrowed at a certain moment, is expressed in Table 4.14. A query returning the amount of the fine at a certain moment, is expressed in Table 4.15.

---

[28] Notice that since the attribute **Creation Timestamp** is a default attribute, the context could have been omitted. Howeve, we explicitly show it in the example for didactical purposes.

```
class PERSON
 context
   being participant of (REGISTRATION
     being participant of (SELECTION
       being participant of (BORROWING
         being participant of (RETURN))))
 query
  number of borrowings
    returns Natural
    result
      let borrowings = self↑REGISTRATION↑SELECTION↑BORROWING
      let current borrowings =
          {b in borrowings | b↑RETURN = empty set}
        #(current borrowings)29
end class PERSON
```

Table 4.14: EROOS Query for the Number of Borrowings

```
class FINE
  context
    having participant (BORROWING
      being participant of (RETURN)
      having participant (ALLOWANCE
        having participant (LIBRARY
          having attribute Amount of Daily Fine)))
  query
    amount
      returns EUR VALUE
      result
        let duration =
          (if self↓BORROWING↑RETURN = empty set
           then now
            else self↓BORROWING↑RETURN→Creation Timestamp)
           - self↓BORROWING→Creation Timestamp
       self↓BORROWING↓ALLOWANCE↓LIBRARY→Amount of Daily Fine
         * days(duration)
end class FINE
```

Table 4.15: EROOS Query for the Amount of the Fine

---

[29] In EROOS, the cardinality of a set can be denoted using #(), **card**(), or **cardinality**().

4.7.4 Contributions, Related Work, and Reflections

The EROOS query concept is comparable with the definition of query operations in OCL. Our contribution concerning the query concept is the definition of a **formal notation for expressing the semantics of queries**. This allows a complete and precise description of the behaviour part of a model. As such, the conceptual model can be used for simulation, which leads to a better validation of the model by the customers, and for model transformation to more software focussed models at a lower abstraction level. Our work predates and is largely comparable with OCL. Since the focus of this text is on the constraint-centric approach in EROOS, we refer to the EROOS Reference Manual [143] for a complete and in-depth description of EROOS queries.

4.8 Events and Enrichment

In Section 4.2, creation events were introduced as a means to create objects of a class. This section introduces the concept of a general event as a clustering of a number of other events, being creation events and other general events, in order to model the behaviour of a more complex change that can occur in the universe of discourse.

4.8.1 Events in an EROOS Model

An EROOS model consists of structural aspects, such as classes, attributes, relations, constraints, and specialisations, as well as behavioural aspects, such as queries and events. EROOS events provide the means to create a new instantiation of the EROOS model, by extending the model instance with a number of new objects. As such, events define changes that can be applied upon the model instance. A creation event is a reflection in the conceptual model of changes that occur in the universe of discourse. Events are said to *enrich* a class with additional functionality. A creation event, introduced in Section 4.2, allows the specification of an event that introduces a new object for a class. But an analyst often wants to describe a clustering of events or a conditional event based on a number of properties that are important for the event. The EROOS concept of a general event offers the possibility for the specification of such more complex events.

An event consists of a description of the effect it has on the model instance when it is successful, i.e., when it violates no constraints that have been stated in the model. This effect description is totally declarative and states what happens instead of how it happens. As such, a new model instance is defined based on the state of the existing model instance. An event can be atomic, or composed by clustering a number of other events. An event is instantaneous and timeless, which means that there exists no time period between the moment the event is initiated and the moment the effect is visible in the model. When queries are used in an event, they obtain information about the model instance at the time the event occurred, i.e., just before the change of the event has been applied. In fact, the changes caused by an event are directly visible after the

event has taken place. When an event takes place at time t, the event is visible in the timeframe]t, ∞[. So queries that are mentioned in the event are evaluated at time t and will not see the changes caused by the event yet.

Whenever the duration of an event should be considered as a period, because its duration is important for the universe of discourse, the composite event must be split in two basic events, namely a first one to express the start of the activity, and a second one to express the end of the activity. In this manner, the EROOS approach allows the analyst to make a clear distinction between the modelling of an event or occurrence that is instantaneous, represented by a single event, and the modelling of an activity that lasts for a certain period, represented by two events.

4.8.2 Specification of an EROOS Event

In general, an event offers the ability to cluster a number of events into a single event. This allows the specification of functionality that creates a number of objects from several classes using a single event, or that conditionally creates an object. An event is applied on an existing object (**self**) that forms the reference point for expressing the event.[30] An EROOS event is specified in an event script, as given in Table 4.16.

```
<EROOS event script> =
"class" <CLASS NAME>
  "context"
    <context clause>
  "general event"
    <general event name>
      [ "(" <parameter name> ":" <TYPE NAME>
        ( "," <parameter name> ":" <TYPE NAME> )* ")" ]
    "effect"
      <event expression>
"end class" <CLASS NAME>

<event expression> =
 ( ["let" <mnemonic> "=" ] <CLASS NAME>"."
     <creation event name>"("<parameter expression>")"
 | <object expression>"."<general event name>
     "(" <parameter expression> ")" )+
```

Table 4.16: EROOS Event Script

[30] One can consider the object on which the event is applied as a default argument for the event.

The different components in the specification of an event script are the following:

- The name of an event should provide a good description of the occurrence from the universe of discourse that it represents. The event name must be a lowercase verb to express the change it applies onto the model instance, and should be unique in the context of the class for which it is defined.
- An event can introduce, by means of successive formal arguments, symbolic names for values and objects to be supplied each time the event is instantiated.
- The final component of an event script specifies its effect on the model instance. The event expression is an (eventually conditional) enumeration of an event collection, consisting of other EROOS general events and basic creation events from a variety of classes. The expression that determines the event collection, is to be built from the object on which the event is applied (**self**), the values and objects that serve as actual arguments for the event, the default EROOS attributes, domain values, and domain functions. The effect description of an event will be evaluated at the time the event is activated. In the case that a random choice has to be made between elements from a certain set, the EROOS selection operator **random one of** can be used. This selection operator makes a random selection of a single element from a set of potential elements.

Since the focus of this text is on the constraint-centric approach in EROOS, we refer to the EROOS Reference Manual [143] for a complete and in depth description of EROOS Events. As presented in Figure 4.22, EROOS events are represented graphically by means of a circle connected to the class of the object on which the event applies. The name of an event is preceded by an exclamation mark.

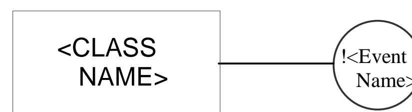


Figure 4.22: Graphical Representation of an EROOS Event

4.8.3 EROOS Events for the Library Example

Given the example of the library system that was presented in Section 2.3, and the relation hierarchy that was defined in Section 4.4.7, we could define a large number of useful events. We restrict ourselves to the example of a deregistration of a person at a library, which is expressed in Table 4.17. In order to perform a deregistration, a person must (1) return all its current borrowings, (2) pay all its open fines, and (3) deselect all her or his selected books.

```
class REGISTRATION
  context
    having participant (LIBRARY
      having attribute Maximum Lending Period)
    being participant of (DEREGISTRATION,
      SELECTION being participant of (DESELECTION,
        BORROWING being participant of (
          RETURN being participant of (PAYMENT),
          FINE being participant of (PAYMENT))))
  general event
    deregister complete
      effect
        DEREGISTRATION.create(self)
        for all s in self↑SELECTION:
          s↑DESELECTION = empty set ⇒ DESELECTION.create(s)
        let returned books = self↑SELECTION↑BORROWING:
          b↑RETURN = empty set
        for all b in returned books:
          let return = RETURN.create(b)
          if now - b→Creation Timestamp >
             self↓LIBRARY→Maximum Lending Period
          then let fine = FINE.create(b)
               PAYMENT.create(fine,return)
        for all f in self↑SELECTION↑BORROWING↑FINE:
          f↑PAYMENT = empty set ⇒
            PAYMENT.create(f,f↓BORROWING↑RETURN)
end class REGISTRATION
```

Table 4.17: EROOS Event of Deregistration for the Library Example

4.8.4 Contributions, Related Work, and Reflections

Our contributions concerning the event concept are the following:

- The definition of a **formal notation for** expressing the semantics of **events**. This allows a complete and precise description of the behaviour part of a model. Our work predates and is largely comparable with OCL.
- The methodological approach using **instantaneous events** obliges the modeller to split an occurrence with a relevant duration into two model events. This allows a proper guiding of the modeller to a unique conceptual model for the universe of discourse to be modelled.

The EROOS event concept is largely comparable with the definition of operations in OCL. The formal specification of events, in contrast with the common approach of informal and textual event descriptions in most analysis methods, can be compared with formal specification formalisms [39], such as Z [63][140], the Vienna Development Method (VDM) [79], and the B-method [1], and the use of formal specifications in programming languages, such as Eiffel [100]. In addition to Z that has a state-based transition formalism, defining a model instance based on the previous model instance, EROOS extends this approach with a time-based transition scheme, defining a possibly new model instance based on the previous model instance at each moment in time. Since the focus of this text is on the constraint-centric approach in EROOS, we refer to the EROOS Reference Manual [143] for a complete and in depth description of EROOS Events.

4.9 Design Issues concerning Model Constraints

Conceptual modelling must be focused on the universe of discourse. The final outcomes of the analysis phase results in a complete description of the universe of discourse. Aspects of the software solution domain are not incorporated in the conceptual model. Therefore, constraints in a conceptual model are a high-level specification mechanism for rules and regulations from the universe of discourse, without incorporating any decisions on how and when they are going to be checked and enforced. Since constraints are an important part of the universe of discourse and therefore deserve to play an important and influential role in a conceptual model, they are mostly of such importance that they also have to be enforced in the actual software system. The design phase is the right place and time to take decisions on the actual details regarding the enforcement of the specified model constraints. We present an overview of techniques for realising constraint checking for implied model constraints and first-class model constraints at the design and implementation level. Additional design considerations have been described by Said [130].

4.9.1 Design Issues for Model-Implied Constraints

A hierarchical structuring of relations results in a larger number of classes and a more complicated association structure to implement. Therefore, it is advisable to transform the hierarchical structures into a simpler, flat structure in order to implement them. UML associations are preferred at the design level for reasons of simplicity and implementation ease. An association do not have an identity or attached functionality. It is rather straightforward to transform the developed hierarchical model into a bipartite, flat model, consisting of classes and associations. Each EROOS binary relation can be transformed into two associations, each connecting a participating class to the refined class. We have developed a model transformer from EROOS models to UML models, in which the EROOS hierarchical model structure is flattened into a UML model.

It is also possible to optimise a model by reversing the reification of a relation into a class. As such, a refined class can be optimised by replacing it with a plain UML association. However, the consequence is that all functionality defined for the class must be shifted to one of its participating classes. This can be done easily when the relation contains no duplicate relation links, and has a connectivity of '1' for at least one of the participants. In that case, the functionality can be shifted to the participant class. Considering the library system, since a book can be borrowed by at most one registered person at the same time, the borrowing functionality can be incorporated in the book object. This is due to the fact that there is a one-to-one correspondence between a borrowing and a book, although it is optional from the viewpoint of the book. By performing such optimisation transformation, the number of classes contained in the analysis model can be diminished at the design level. The design level is the right place to decide which refined classes have to be optimised and which ones should be implemented as classes. The main concern for such optimisation process is to find a good balance between the data (memory) and the procedural (processing) part of a system.

Existential dependency among objects may seem too restrictive for the ultimate system to be developed. A large deal of run-time flexibility, in populating the implementation classes with instances, would be lost. Since the conceptual model is focussed on the universe of discourse in its normal appearance, issues regarding unavailability of information to the system at run-time were not yet taken into account. Therefore, it is possible that some constraints present in the conceptual model must be relaxed for implementation reasons.

4.9.2 Design Issues for First-Class Model Constraints

Constraints that are specified as a first-class model concept, have to be enforced in some manner in the actual software system. At the design phase, several topics arise concerning the constraint enforcement. The main issues are concerned with *when* and *how* to perform the constraint checking. The software engineer must determine the place and time that the system must perform the necessary checks for detecting possible constraint violations. In addition, the system must also determine the actions that must be taken when a constraint violation is detected that is going to occur or has already occurred. Two distinct approaches can be distinguished, namely a proactive and a retroactive approach.

- The proactive approach consists of preventing the occurrence of a constraint violation. First of all, the set of operations that can be the source of a constraint violation must be determined. For each operation, a precondition must be derived that can detect possible constraint violations. When all preconditions are satisfied for an operation, the operation can be executed without violating any constraints. Notice that such precondition must be an active precondition that must always be checked before the action may be executed. It is an obliged condition, and not a kind of design contract that is only checked during debugging, e.g., as in Eiffel [100]. The preconditions prevent the system from entering a wrong state. Given a certain system state, the operations that bring the system in an erroneous state are

prevented before they could have been executed. This approach causes a loss of efficiency due to a high number of tests, but keeps the system in a highly consistent state at each moment in time.

- The retroactive approach consists of detecting incorrect system states, whereupon the system will perform either (1) a sort of rollback to the previous valid state, or (2) the invocation of an error recovery procedure that tries to fix the system in some manner. Before an actual change is performed in the system, the necessary measures are taken in order to enable an undoing of the change. For instance, the old value of an instance variable that must be changed, can be stored temporarily until a valid system state has been reached. This results in an important gain of efficiency, but can leave the system in an inconsistent state during a certain time. In addition, a mechanism to detect invalid system states must be put in operation.

The choice between these two approaches is often situation specific. A trade-off has to be made between efficiency and consistency, depending on the criteria that are of utmost importance for the ultimate system.

Regarding the technical realisation of constraint enforcement, a number of solutions can be identified:

- A constraint checking meta-layer can be developed that intercepts operation calls, and performs the necessary checks and measures to enforce the constraint. As such, the constraint checking meta-layer governs and controls the normal execution of operations, and intervenes when necessary.
- A software library for EROOS constraints can be developed in order to support constraint checking at run-time. Such library can provide facilities for evaluating constraints expressions, so that they can be checked at run-time. An EROOS constraint interpreter can be developed that parses constraint expressions, computes their validity, and triggers error handling code when necessary.
- Aspect-Oriented Software Development (AOSD) techniques [46] can be used to weave constraint-related behaviour into the normal system behaviour. As such, the constraint checking facilities are specified as a separate entity, and can be woven into the system functionality at the places where constraint checking and error handling code must be injected.
- Constraint logic programming techniques, including supporting languages and libraries, could be integrated in order to detect constraint violations and support rule deduction. Wu [168] discusses different approaches to combine logic programming and object-oriented programming.

As part of the validation for the EROOS methodology, we have developed a generator for a retroactive run-time constraint meta-layer that can recover from constraint violations. The decisions that have to be made concerning the constraint validity, are ordered in several successive levels in order to obtain separation of concerns. We generate code to locate where and when each constraint can be violated. At these violation checkpoints, a meta-layer implementing a constraint

control mechanism is triggered, which traces back to each constraint that could have been violated at that place. This reduces the moments and the objects to be checked. The realisation is done in four steps:

- First, code is introduced to preserve the old system values for each change that can occur in the system, using the memento design pattern. This design pattern forces the encapsulation of an object state into a memento object, in order to enable the manipulation of its internal state. As such, the object memento is cloned before an actual change is executed on any of the object's state variables. When a rollback has to be performed, it is sufficient to restore the old memento object for the changed object and discard the erroneous one.
- Then, the classes that are involved in each constraint are determined. Each constraint will be checked on all objects of its involved classes after the invocation of each of its operations. Since all involved classes will be checked, operations on other classes cannot be of any influence on the validity of the constraint. This will decrease the moments on which a constraint has to be checked. Notice that our involvement identification is done rather roughly on a class-based level, triggered by every operation of a class. However, operations are often restricted to manipulate certain specific characteristics of an object, such as attributes or association references. Such operations will not violate constraints about other characteristics of the class, and, therefore, need not to be checked at all times. The determination of the violation checkpoints could be made more fine-grained, triggering only at the moment when certain specific operations of the class that can violate the constraint, are executed. They do not have to be triggered at those places where the constraint cannot be violated. However, this would need a deeper analysis to determine which operations can violate which constraints. The moments when a constraint has to be checked, could heavily decrease when performing such advanced, complex operation analysis.
- Third, we determine the set of objects of the involved classes that have to be verified after an operation on an object of the class is executed. Mostly, it will not be necessary to check each object of the class when a certain operation has been executed. It is mostly sufficient to check only the object on which the operation has been applied. Those common situations will give rise to a decrease of the objects on which a constraint has to be checked.
- Last, code is injected to implement a violation checkpoint. Such checkpoint triggers the constraint control meta-layer and passes the set of constraints together with the set of objects that have to be checked. This component will verify the validity of the identified constraints for the given objects. When a constraint has been violated, a rollback mechanism is invoked in order to restore the old state of all changed objects. In case that all constraints have been preserved, the state change can be committed and no further actions are needed by the constraint control meta-layer.

4.10 Evaluation of the EROOS Kernel

The EROOS kernel provides a number of basic concepts to perform conceptual modelling, and to capture the knowledge and information of the universe of discourse in a conceptual model. Moreover, the concepts offered by the EROOS kernel are fine-tuned and restricted in their applicability, in order to obtain the key principles for conceptual modelling stated in Chapter 3. In this section, we evaluate the EROOS kernel according to these key principles, argue how the EROOS kernel succeeds in achieving the principles, and give an overview of related work. Although it is not our goal to perform a thorough evaluation of UML according to the key principles for modelling, Opdahl [114] indicates a number of major problems in UML, such as failing to comply with the key principles of uniqueness, unambiguity, completeness and preciseness. While developing the EROOS methodology, we tried to adopt and integrate all key principles for conceptual model that have been identified.

4.10.1 Achieving Uniqueness

The *Principle of Uniqueness* is a key principle ingrained in the EROOS kernel. It has a huge impact on the precise definition of the EROOS concepts, and the delimitation of their applicability. The EROOS kernel achieves to create a single and unique conceptual model due to three important factors:

- The **incorporation of model constraints** in each methodological concept provides a dedicated meaning to each model concept, thereby limiting its usage to a specific context and forcing the analyst to use certain concepts in specific situations. As an example, the prohibition of using Boolean and integer attribute types, forces the analyst to introduce a specific class or specialisation hierarchy to model this kind of information. As another example, the prohibition of using the participation query inside an EROOS constraint specification, forces the analyst to use relations to express the existential dependency, or a different expression viewpoint for the constraint using specific top classes.
- The usage of **existential dependency** as the core model structure, forces the analyst to use a specific structure for every situation to be modelled. The usage of an alternative model structure will introduce a number of existential dependency constraints that are different from the rules in the universe of discourse, and therefore lead to the description of a different situation.
- The fact that **information can only be added** to a model in the EROOS kernel, leads to the property that there is no information loss inside a model. This means that the modeller does not have to be concerned with weighing up the advantages and disadvantages of modelling the full information in all its details, versus modelling an optimised representation of the information that contains only the data that is strictly needed. As such, all fundamental information is modelled as core facts. Derived information can be expressed as queries, and evaluated whenever necessary. As an example, no computable attribute will be modelled and kept up-to-date in the EROOS kernel. Instead, all values that compose the

basic elements of the computable attribute will be modelled as single facts, enabling the calculation of the exact attribute value at any moment in time.

There are some cases in which additional methodological criteria are needed in order to enforce uniqueness in a model. As an example, during the modelling of a period, the necessity of modelling the start time, end time, and duration can lead to the definition of three attributes, of which two are actually sufficient. Moreover, by choosing to model the start time and end time, the analyst is even obliged to add an additional EROOS constraint in order to express the fact that the end time must be greater than the start time. Due to the fact that the EROOS methodology (1) prohibits the modelling of derived information, and (2) obliges to model those attributes that do not give rise to additional constraints, an analyst is forced to model the start time and duration as attributes, and transforming the end time into a query.

4.10.2 Achieving No Redundancy

The *Principle of No Redundancy* is achieved by the fact that each EROOS concept captures specific information inside the conceptual model. The extension of a model with additional concepts therefore introduces more information inside the model and leads to a different model. When the model is extended with information that could be derived from information already present in the model, the analyst cannot specify the equality of the information. For every concept that could be duplicated inside a model, it is impossible to define such equality constraint. Therefore, the modeller cannot express derived information in a model, but has to represent it using queries.

- In the case of class duplication, the objects of both classes express different occurrences in the universe of discourse. In order to model derived objects, they should be made mutually dependent. Since the EROOS kernel only allows the specification of unidirectional existential dependency, it is impossible to express that two objects are a single unity. Such model always expresses a different situation, in which both objects are not mutually dependent and can exist independent from each other. Therefore, they both express a specific and distinct fact, and cannot be considered as redundant.
- In the case of attribute duplication, it could be possible that the value for a specific attribute can be derived from values of other attributes, as illustrated in the previous section. Therefore, the EROOS methodology prohibits the modelling of derived attribute information, forcing the analyst to a model with the least number of attributes and EROOS constraints. Derived attributes must be transformed into EROOS queries.
- In the case of relation duplication, the fact that a relation is always encapsulated in a class would introduce both an additional relation and a corresponding class for a duplicated relation. Therefore, this case is identical as the one of class duplication. However, relations can give rise to a second kind of duplication, namely participant duplication. Participant duplication means that a participant object in a relation link has unnecessarily multiple presences in the existential dependency path for a refined object. Such situations are revealed through the

presence of EROOS join constraints, which put an equality constraint on a certain number of direct or indirect participant objects. EROOS explicitly inhibits this kind of join constraints, thereby avoiding participant derivation.

- In case of EROOS constraint derivation, EROOS explicitly inhibits the definition of EROOS constraints that can be derived from other kind of model constraints. Therefore, it is not allowed to specify constraints that can be derived from other constraints in the model.
- In case of EROOS specialisation, the well-formedness rules regarding partitions explicitly inhibits the definition of identical specialisations in a partition, since the specialised classes must completely partition the generalised class into disjoint subsets. In addition, identical partitions as well as specialised classes that cannot contain any objects are forbidden. Last, derivable specialisation partition structures are avoided due to the specific rules regarding causal specialisation dependency.

4.10.3 Achieving Unambiguity

The *Principle of Unambiguity* is achieved by a precise definition of the EROOS methodological concepts. As such, the information contained inside a conceptual model has a well-defined meaning that expresses a specific fact from the universe of discourse. It is therefore impossible that distinct situations in the universe of discourse result in the same conceptual model. It is possible that a certain subset of two different universes of discourse results into the same EROOS conceptual model when they both have certain knowledge and facts in common. Due to the *Principle of Uniqueness*, the shared subset of the two universes of discourse will definitely result in the same conceptual model.

Since every single fact in a conceptual model has a clearly defined meaning that expresses a certain fact in the universe of discourse, there will be a one-to-one mapping from the conceptual model to a real or envisioned universe of discourse. A problem that can arise, is a discrepancy between events that occur in the universe of discourse, and the view of the corresponding modelled event in the mind of a person who tries to obtain an understanding of the modelled universe of discourse. This potential discrepancy arises due to the fact that the correspondence between the universe of discourse and the conceptual model cannot formally be defined, since there exists no global reference model of the universe of discourse to refer to. In fact, the conceptual model has exactly the purpose of being the reference model of the universe of discourse for the software system to be built. Therefore, it is of utmost importance to use a proper naming scheme to facilitate a correct and precise mapping between elements in the universe of discourse, and elements in the conceptual model. Besides the context of an event, which includes its properties, dependencies and related knowledge, the event name is the only information available to map the modelled event onto a real-world event in the universe of discourse.

4.10.4 Achieving Completeness

The *Principle of Completeness* is achieved by the fact that the EROOS methodology only offers well-defined formal concepts for modelling knowledge of the universe of discourse. It is impossible to add additional informal documents or descriptions to an EROOS conceptual model in order to attach this kind of information to the conceptual model. The EROOS conceptual model is the sole source of knowledge that is captured in the conceptual model. It is the task of the analyst to accomplish a complete model of the universe of discourse. The analyst has to assess whether the resulting conceptual model covers all elements of the universe of discourse or not.

An analysis methodology can only guide the analyst in achieving completeness, but cannot enforce the completeness of a conceptual model regarding a certain universe of discourse. The EROOS methodology assists the analyst by emerging concealed or latent facts, and forces the analyst to model them explicitly in the conceptual model. For instance, the fact that relations always must be encapsulated in a class, forces the relation to manifest in the conceptual model. As another example, the prohibition of Boolean and integer attribute types forces the analyst to highlight the hidden facts behind these attributes, and explicitly reify them as objects or specify them inside the specialisation hierarchy.

4.10.5 Achieving Minimalism

The *Principle of Minimalism* is achieved by the fact that an EROOS model must be a coherent model, without any classes that are unrelated to other classes in the model. Moreover, every modelled element must be related to a functional requirement or to an element of the real or envisioned universe of discourse. However, this is not enforceable, since it is largely the responsibility of the analyst, in consultation with the clients and the end users, to decide whether an element belongs to the envisioned universe of discourse or not. Software metrics for Object-Oriented Models [118][26][25][44], which propose measurements for coupling and cohesion, could assist the modeller in deciding whether certain elements are loosely coupled. But such metrics are often a poor indicator to measure the quality criterion of minimalism. Therefore, it is the task of the analyst to accomplish a minimal model of the universe of discourse. The analyst has to assess whether certain knowledge belongs to the universe of discourse, or whether it falls beyond its boundaries. An analysis methodology can only guide the analyst in achieving minimalism, but cannot enforce the minimalism of a conceptual model regarding a certain universe of discourse.

The *Principle of Minimalism* should not be viewed in isolation, but should be considered in relationship with the *Principle of* Completeness. Certain model optimisations to obtain an apparently smaller model, thereby achieving a higher degree of minimalism, could conceal or obfuscate certain important elements that should be highlighted to obtain model completeness. In achieving the right balance between minimalism and completeness, the EROOS methodology favours completeness through highlighting hidden facts, above minimalism through model reductions and optimisations.

4.10.6 Achieving Preciseness

The *Principle of Preciseness* is achieved by a complete formalisation of the analysis results in EROOS. No textual specifications can be made inside an EROOS model. All model entities, event descriptions, model constraint specifications in an EROOS conceptual model are formally specified. As such, the information in the conceptual model can fully be verified at each moment in time, in order to check its correctness and validity. Not only the structural part of the conceptual model, but also the behavioural part is formally specified.

A formal specification of the structure and behaviour of a conceptual model allows formal verification of the analysis results, and rapid prototyping in order to achieve early customer feedback. We have validated our behaviour specification by building a generator for model simulations that automatically generates a C++ [144] or Java [57] application with an accompanied user interface, in order to support rapid prototyping and early model validation.

4.10.7 Achieving No History

Since an EROOS conceptual model only represents basic facts and their interdependencies, the *Principle of No History* is fulfilled by the EROOS methodology. No historical information regarding the model creation process can be captured in the resulting EROOS conceptual model. A change in the model will have a direct impact on the model structure, and cannot be superimposed on the old version of the model.

An EROOS model captures a lot of historical information regarding the event occurrences, but this kind of historical information is an inherent part of the universe of discourse. Information about the gradual materialisation of a conceptual model cannot be retrieved from the resulting EROOS model.

4.10.8 Achieving Model-Implied Constraints

The *Principle of Model-Implied Constraints* is a key principle ingrained in the EROOS kernel. It is achieved through the extensive and sound definition of the concepts offered by the EROOS methodology. A large number of model constraints are directly implied by the EROOS concepts. This chapter has extensively described the model-implied constraints for each EROOS concept, such as disjunctness, immutability, finiteness, and uniqueness for classes, permanent binding and immutability for attributes, existential dependency and immutability for relations, and abstractness, immutability, finiteness, and partition disjunctness for specialisations.

Through the encapsulation of every relation in a class, an existential dependency hierarchy is established as the core model structure. Every dependency of an item on the existence of other items will be reflected in the model by a direct or indirect relational dependency between (1) the objects expressing these refined items and (2) the objects expressing the dependent items. This establishes an implicit and automatic

enforcement of the dependency constraints that exist in the universe of discourse. However, the EROOS kernel lacks the possibility to make a separate description of two items that are mutually dependent. In order to reach model uniqueness, mutually dependent items can only be modelled by merging them into a single unity object. The EROOS universe, presented in Chapter 5, offers advanced concepts that facilitate the separate modelling of mutually dependent items.

4.10.9 Achieving Abstraction

Regarding the *Principle of Abstraction*, the EROOS kernel only offers a full detailed view on a conceptual model. EROOS does not offer concepts for generating model views, in order to present a model in an abstracted form to a model reader. The only concern of the EROOS kernel is to offer means to create a complete and detailed conceptual model for the universe of discourse. However, it is possible to generate abstract views from an EROOS model, using for instance Model-Driven Development (MDD) [50][83] techniques to transform an EROOS model into a more abstract model that is better focussed to the needs of the model reader. As such, a detailed conceptual model can be translated into a suitable customer interaction model using MDD model transformations.

Chapter 5

Advanced Concepts for Conceptual Modelling

An EROOS model is composed of both structural elements, such as classes, attributes, relations, specialisations, and constraints, as well as behavioural elements, such as queries and events. In the previous chapter, the concepts and notations of the EROOS kernel have been presented. Although these core concepts are sufficient to build a conceptual model that complies with the key principles for conceptual modelling, it is useful to have better suitable concepts at one's disposal in order to simplify the specification of recurrent patterns. Based on common analysis patterns that have been detected for the EROOS kernel, the EROOS universe offers the analyst advanced and more practical concepts for modelling the universe of discourse.

5.1 Class Archives and Object Destruction

This section introduces the EROOS kernel analysis pattern that has identified the necessity of introducing the concept of a class archive. The definition of the class archive, the specification of attributes and queries for a class archive, and the usage of the class archive as relation participant, are described thereafter. Last, the class archive concept is applied on the running example of the library system.

5.1.1 EROOS Kernel Analysis Pattern for Class Archives

Objects often represent properties from the universe of discourse having a temporarily meaningful lifetime. The property can represent an activity, or it can have a limited physical time of existence, a limited time of validity, or a specific duration in which the property is active or enabled. This results in a specific EROOS kernel

analysis pattern that is presented in Figure 5.1 and Table 5.1. The class '*LIVING/ACTIVE*' represents the property when it is active, while the class '*DEAD/PASSIVE*' models the expiration or end of the property. All meaningful attributes and relations are attached to the class '*LIVING/ACTIVE*'. A number of constraints explicitly check that the property has not yet expired for a valid participation in a refined class, e.g., '*INVOLVEMENT*'. This EROOS kernel analysis pattern can also be observed in the example of the library system, presented in Figure 4.9 on page 91. In order to simplify the specification of this recurring EROOS kernel analysis pattern, the EROOS universe provides notational support by means of an extension of the class concept.

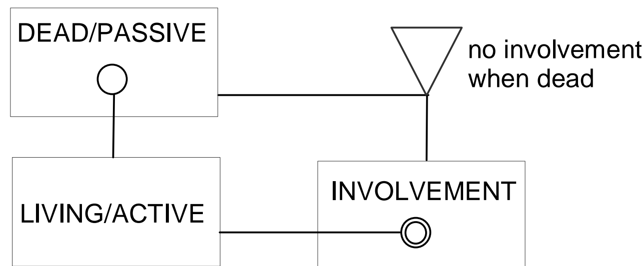


Figure 5.1: EROOS Kernel Analysis Pattern for an Activity

```
constraint no involvement when dead
  top classes DEAD/PASSIVE, INVOLVEMENT
  context
    DEAD/PASSIVE having participant (LIVING/ACTIVE),
    INVOLVEMENT having participant (LIVING/ACTIVE)
  definition
    for all d in DEAD/PASSIVE:
    for all i in INVOLVEMENT:
      d↓LIVING/ACTIVE ≠ i↓LIVING/ACTIVE
end constraint no involvement when dead
```

Table 5.1: EROOS Constraint for an Activity

5.1.2 The Class Archive

In order to support the modelling of an object with a temporary lifetime, object destruction is introduced in the EROOS universe as an extension for the class concept. As such, all objects of a class automatically have an active lifetime in which most of their events and activities will occur. Objects that have passed their active lifetime, indicated by the occurrence of a destruction event for the object, are put into the class archive. The destruction event reflects the fact that a property or an item in the universe of discourse corresponding with the object has ceased to exist, or stopped

to be of any importance. This corresponds with the existence of a DEAD/PASSIVE object for the class LIVING/ACTIVE in Figure 5.1

The population of a class, representing the set of objects associated with that class, will be split in two disjoint collections, namely a present population set and a past population set. The present population set represents the living objects, which are those objects for which the destruction event has not yet occurred. The past population set, also called archive, is an object collection of the class that contains all dead objects, which are those objects that have been involved in a corresponding destruction event. The method will offer possibilities for checking whether an object is '*alive*' or '*dead*', as well as for retrieving information about the final attribute values of the dead object and the relation link it has been representing. Figure 5.2 shows the detailed representation of a class, together with the basic state diagram expressing the course of life for an object, and the partitioning of the population set of a class into a present and past part at a moment t. The extended class script and definition of a class is given in Table 5.2 and Definition 5.1. Notice that the definition allows a new object to be involved both in a creation and destruction event at the same moment t, in which case the object will be added to the overall population set as well as the past population set of the class. In fact, the life span of such stillborn objects will be of zero length.

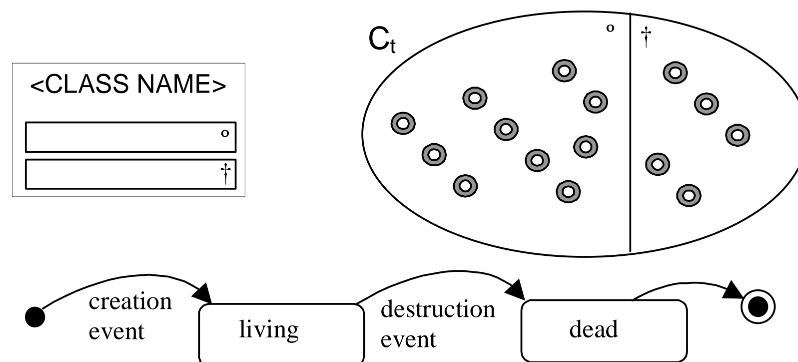


Figure 5.2: Present and Past Population set for an EROOS Universe Class

```
<EROOS universe class script> =
"class" <CLASS NAME>
  "creation event"
    <creation event name>
  "destruction event"
    <destruction event name>
"end class" <CLASS NAME>
```

Table 5.2: EROOS Universe Class Script

A **concrete class** is a model entity defining, at each moment in time, a disjoint object population set, which is element of the corresponding model instance. This total population set can only be extended in time, and is split into two disjoint parts: a present population set that represents the living objects of the class, and a past population set, called archive, that represents the dead objects. The archive set can only be extended in time, whereas the population set can grow due to new objects but also shrink due to the migration of objects to the archive.

A **creation event** is an event of a class that, if applied on a model instance at a certain time, adds a new object to the object population set for that class. The object will be added to the present population set, unless it is at the same moment involved in a destruction event, in which case it is directly added to the past population set.

A **destruction event** is an event of a class that, if applied on an object of a model instance at a certain time, removes an object from the present population set of the class, and add it to the the past population set of the class.

Given

Model M; Object Universe OU; Event Universe EU;

Event Set Instance E; Concrete class $C \in M_{ccl}$;

Object $o_1,\dots,o_n \in OU$; Destruction Event $d_1,\dots,d_n \in EU$;

$C^\dagger: TIME \rightarrow \mathcal{P}(OU) \mid \forall t \in TIME: C^\dagger_t \subseteq C_t$

$\forall t \in TIME: C^\dagger_t \subseteq C^\dagger_{t+1}$

$C^o: TIME \rightarrow \mathcal{P}(OU) \mid \forall t \in TIME: C^o_t = C_t \setminus C^\dagger_t$

$o_1,\dots,o_n \in C^o_t \wedge o_1.d_1, \dots, o_n.d_n \in E_{t+1}$

$\Rightarrow C^\dagger_{t+1} = C^\dagger_t \cup \{o_1, \dots, o_n\}$

Definition 5.1: EROOS Universe Class

5.1.3 Attributes for the Class Archive

In EROOS, an attribute can be specified for a class archive, meaning that the attribute will only have to be defined for dead objects, thus belonging to the archive. The permanent binding for an attribute is preserved, but since the attribute is attached to the class archive, only dead objects can and must have an attribute value attached to them. Obviously, the definition of an archive attribute influences the destruction event of an object, since the destruction event must establish the binding with a value of the decorating domain in order to fulfil all implied constraints concerning the archive attribute. The definition of an archive attribute can be found in Definition 5.2, whereas the specification of an archive attribute script is given in Table 5.3 and the graphical notation is presented in Figure 5.3.

An **archive attribute** is a model entity defining a property *for a class archive* for which, at each moment in time, every object of the *past population* of a class must be associated to a specific value of the domain defined for the attribute.

Given

Model M; Class C $\in M_{cl}$; Attribute $CA^{\dagger} \in M_a$; Domain D $\in M_d$;

$CA^{\dagger}: TIME \rightarrow (C^{\dagger}_t \rightarrow D) \mid \forall\, t \in TIME: CA^{\dagger}_t \subseteq CA^{\dagger}_{t+1}$

Definition 5.2: EROOS Archive Attribute

```
<EROOS archive attribute script> =
"class" <CLASS NAME>
  "definition"
    "decorated by" [ "unique" ] "archive attribute"
        <Attribute Name> ":" <DOMAIN NAME>
        [ "constrained by" [ <lower bound> ( "<" | "≤" ) ]
          <Attribute Name> [ ( "<" | "≤" ) <higher bound> ] ]
  "destruction event"
    <destruction event name>
      [ "(" <parameter name> ":" <DOMAIN NAME> ")" ]
    "effect"
      ( "new self→"<Attribute Name> "=" <parameter name>
      | "new self→"<Attribute Name> "=" <domain expression> )
"end class" <CLASS NAME>
```

Table 5.3: EROOS Archive Attribute Script

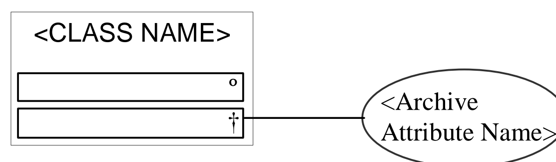


Figure 5.3: Graphical Representation of an EROOS Archive Attribute

5.1.3.1 Default Attributes for the Class Archive

When introducing the class archive in the EROOS universe, a default attribute is introduced for the class archive, namely the *Destruction Timestamp*. The default attribute Destruction Timestamp, implicitly decorating each past object of every class, is used to specify the exact time at which an object has ceased to exist. The Destruction Timestamp will be fixated at the time of the destruction of the object, i.e.,

at the moment of occurrence of the destruction event. Although the Destruction Timestamp does not have to be defined explicitly, its semantics can be defined in an implicit default EROOS attribute script, as presented in Table 5.4.

```
<EROOS default destruction timestamp> =
"class" <CLASS NAME>
  "definition"
    "decorated by"
      "archive attribute Destruction Timestamp : TIME"
  "destruction event"
    <destruction event name>
    "effect"
      "new self→Destruction Timestamp = now"
"end class" <CLASS NAME>
```

Table 5.4: Implicit EROOS Script for the Default Attribute *Destruction Timestamp*

5.1.4 Queries on the Class Archive

Since EROOS queries serve to retrieve information about the model instance, and the properties of the objects within it, at a specific moment in time, they could also be applied on objects of the class archive. There are no restrictions for using queries that were originally specified on a class, since they can be used for retrieving the properties of living as well as dead objects.

In addition to queries applicable on all objects, a specific set of queries can only be applied on objects of the class archive, namely those that involve the usage of an archive attribute or the Destruction Timestamp. Since these attributes only obtain a value at the moment the destruction event occurs, such queries cannot be applied on objects that are still living.

As presented in Table 5.5, the specification of an archive queries is analogous to the specification of an ordinary query with two distinction, namely (1) the keyword *archive query* that is used to define a query for the class archive, and (2) the capacity of using archive queries and the Destruction Timestamp (→**Destruction Timestamp**). The graphical representation of an archive query is presented in Figure 5.4.

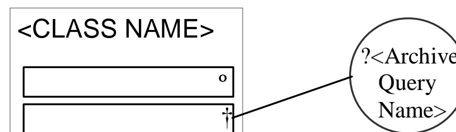


Figure 5.4: Graphical Representation of an EROOS Archive Query

```
<EROOS archive query script> =
"class" <CLASS NAME>
  "context"
    <context clause>
  "archive query"
    <query name> [ "(" <parameter name> ":" <TYPE NAME>
        ( "," <parameter name> ":" <TYPE NAME> )* ")" ]
    "returns" <TYPE NAME>
      "result" <archive query expression>
"end class" <CLASS NAME>
```

Table 5.5: EROOS Archive Query Script

5.1.5 Class Archive as Relation Participant

The introduction of an archive for a class leads to two kinds of objects: the living objects, belonging to the present population of the class, and the dead objects belonging to the past population. This fact can now be exploited when using classes within other EROOS concepts. Instead of using the present population of a class as a participant, the class archive or even the total population set, which is the union of the present and the past population, can be used when dependency relationships between refined objects and participant objects are defined. This allows the formulation of additional restrictions between the lifetimes of a refined object and a participant object. There exist three types of relation participation for a class. All three types include the core existential dependency property that must be valid for every refined object r and participating object p, namely

p→Creation Timestamp ≤ r→ Creation Timestamp[31]

In addition, the core object property must be valid for every object o, namely

o→Creation Timestamp ≤ o→Destruction Timestamp

- A ‘**present participation**’**,** as presented in Figure 5.5.a, obliges that the participant object p is alive during the whole lifetime of the refined object r.

  p→Creation Timestamp ≤ r→Creation Timestamp ≤ r→Destruction Timestamp ≤ p→Destruction Timestamp

- A ‘**past participation**’**,** as presented in Figure 5.5.b, obliges that the participant object p that is related to the refined object r, is dead.

  p→Creation Timestamp ≤ p→Destruction Timestamp ≤ r→Creation Timestamp ≤ r→Destruction Timestamp

[31] Notice that this property only applies when no mutation has taken place, as defined in Section 5.1.7.

- A '**total participation**', as presented in Figure 5.5.c, obliges that the participant object p can be either alive or dead during the lifetime of the refined object r.

  (p→Creation Timestamp ≤ r→Creation Timestamp ≤ r→Destruction Timestamp) ∧ (p→Creation Timestamp ≤ p→Destruction Timestamp)

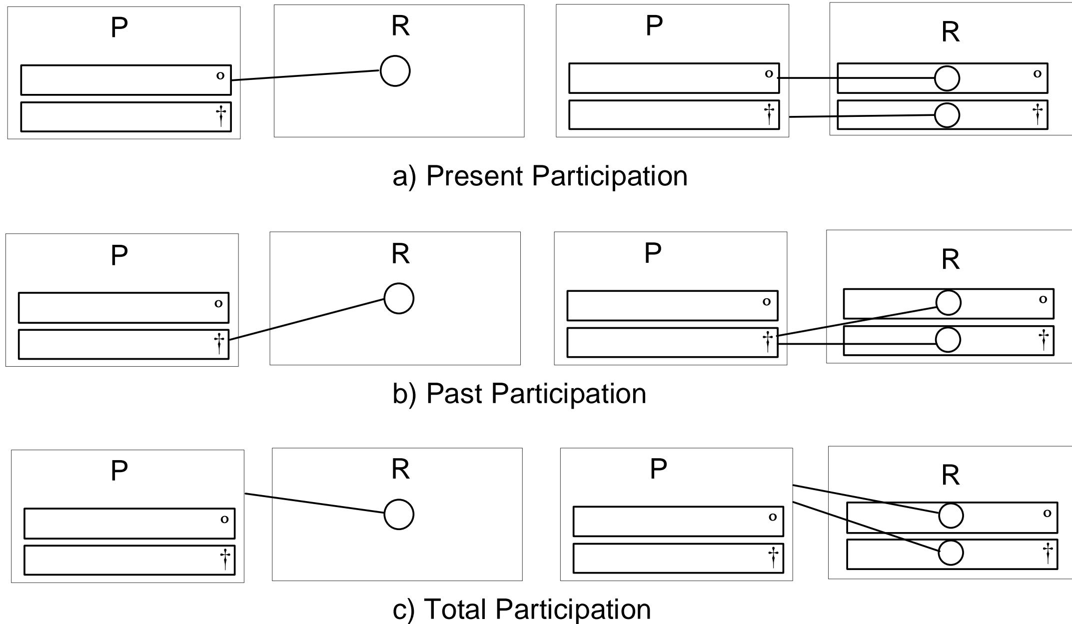


Figure 5.5: Participation Types for an EROOS Class

In addition to the three participation types of a relation, additional integrated constraints can be added to a participation, further restricting the dependency rules between the refined object and the participating object. A number of these restrictions can also be represented in a graphical form, as presented in Figure 5.6.

- A '**not deceased**' participant expresses an additional constraint on the participant object at the time of the creation of a refined object, obliging that the participating object is not yet deceased prior to the moment the refined object is created. This is presented in Figure 5.6.a.

  r→Creation Timestamp ≤ p→Destruction Timestamp

  This constraint cannot be combined with a present participation, since it is already implied by the condition for a present participation. It can only be combined with a total participation or a past participation. For a past participation, the restriction is strengthened to

  p→Creation Timestamp ≤ p→Destruction Timestamp = r→Creation Timestamp ≤ r→Destruction Timestamp

- A ‘**not surviving**’ participant expresses an additional constraint on the participant object at the time of the destruction of the refined object, obliging that the refined object cannot be destroyed before the related participating object is destroyed. This is presented in Figure 5.6.b.

  $$p\rightarrow\text{Destruction Timestamp} \leq r\rightarrow\text{Destruction Timestamp}$$

  This constraint cannot be combined with a past participation, since it is already implied by the condition for a past participation. It can only be combined with a total participation or a present participation. For a present participation, the restriction is strengthened to

  $$p\rightarrow\text{Creation Timestamp} \leq r\rightarrow\text{Creation Timestamp} \leq r\rightarrow\text{Destruction Timestamp} = p\rightarrow\text{Destruction Timestamp}$$

  Notice that ‘*not surviving*’ can be combined with ‘*not deceased*’ for a total participation. This is presented in Figure 5.6.c.

  $$p\rightarrow\text{Creation Timestamp} \leq r\rightarrow\text{Creation Timestamp} \leq p\rightarrow\text{Destruction Timestamp} \leq r\rightarrow\text{Destruction Timestamp}$$

- A ‘**significantly**’ restriction expresses a strict time ordering, namely ‘$<$’, excluding border conditions in which objects can be created or destroyed simultaneously, namely ‘$\leq$’. A ‘*significantly*’ indication can be combined with all of the previous introduced participation types.

  – A ‘**significantly not deceased**’ total or past participant obliges that the participating object will remain living after the refined object is created.

    $$r\rightarrow\text{Creation Timestamp} < p\rightarrow\text{Destruction Timestamp}$$

    Notice that this restriction can be combined with the ‘*not surviving*’ restriction for a total participation.

    $$p\rightarrow\text{Creation Timestamp} \leq r\rightarrow\text{Creation Timestamp} < p\rightarrow\text{Destruction Timestamp} \leq r\rightarrow\text{Destruction Timestamp}.$$

  – A ‘**significantly not surviving**’ present or total participant obliges that that the refined object will still remain living after the destruction of the participating object.

    $$p\rightarrow\text{Destruction Timestamp} < r\rightarrow\text{ Destruction Timestamp}$$

    Notice that this restriction can be combined with the ‘*not deceased*’ restriction for a total participation.

    $$p\rightarrow\text{Creation Timestamp} \leq r\rightarrow\text{Creation Timestamp} \leq p\rightarrow\text{Destruction Timestamp} < r\rightarrow\text{Destruction Timestamp}$$

    It can even be combined with the ‘*significantly not deceased*’ restriction for a total participation.

    $$p\rightarrow\text{Creation Timestamp} \leq r\rightarrow\text{Creation Timestamp} < p\rightarrow\text{Destruction Timestamp} < r\rightarrow\text{Destruction Timestamp}$$

- A ‘**significantly deceased**’ past participation obliges that the lifetime of the past participant object must clearly be ended before the creation of the refined object can take place.

  $$p \rightarrow \text{Creation Timestamp} \leq p \rightarrow \text{Destruction Timestamp} < r \rightarrow \text{Creation Timestamp} \leq r \rightarrow \text{Destruction Timestamp}$$

- A ‘**significantly surviving**’ present participant obliges that that the participating object will still remain living after the destruction of the refined object.

  $$r \rightarrow \text{Destruction Timestamp} < p \rightarrow \text{Destruction Timestamp}$$

- A ‘**significantly lived**’ present, past or total participation obliges that the participant object has clearly be created before the creation of the refined object can take place.

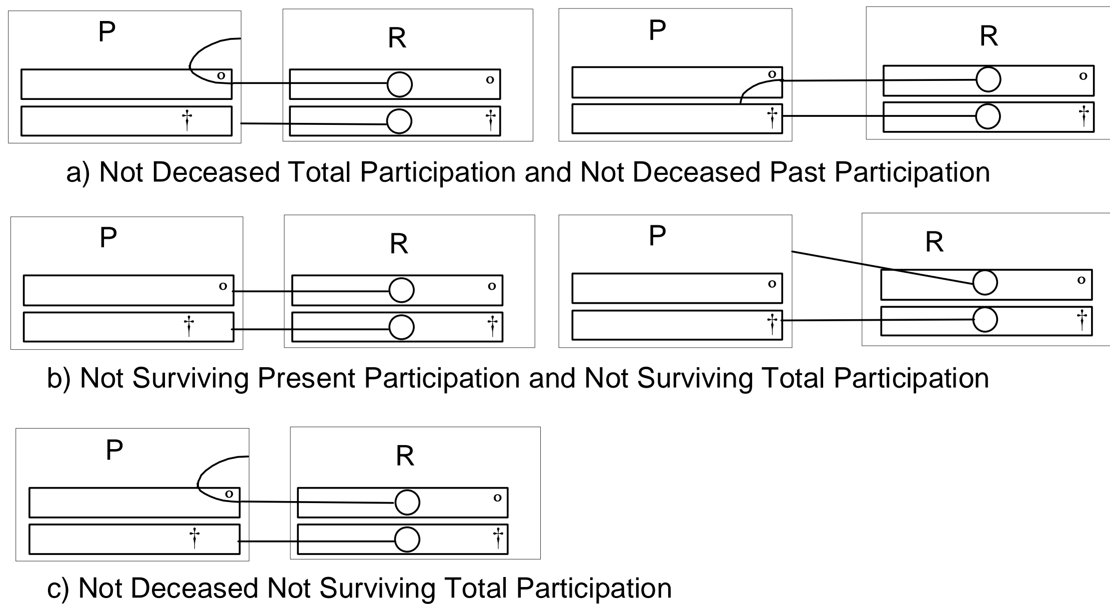
  $$p \rightarrow \text{Creation Timestamp} < r \rightarrow \text{Creation Timestamp}$$

  This restriction can be combined with all previous introduced participant restriction, except for the ‘*significantly deceased*’ participant restriction since it is already implied by this restriction.

- A ‘**significantly not instantaneous**’ participation obliges that the participant object has clearly be created before the destruction of the refined object can take place.

  $$p \rightarrow \text{Creation Timestamp} < r \rightarrow \text{Destruction Timestamp}$$

  This restriction can be combined with a number of participant restrictions in order to create eleven additional combinations of meaningful participant restrictions.



a) Not Deceased Total Participation and Not Deceased Past Participation

b) Not Surviving Present Participation and Not Surviving Total Participation

c) Not Deceased Not Surviving Total Participation

Figure 5.6: EROOS Archive Participation Constraints

Name of Participation Restriction	Restriction between participant p and refinement r				Number of cases
	p.CTS…r.CTS	p.CTS…r.DTS	p.DTS…r.CTS	p.DTS…r.DTS	
Sign.Deceased Past	<	<	<	<	1
Sign.Living Sign.Not Surviving Past	<	<	<=	<	2
Sign.Living Past	<	<	<=	<=	3
Sign.Living Not Deceased Sign.Not Surviving Past	<	<	=	<	1
Sign.Living Not Deceased Past	<	<	=	<=	2
Sign.Living Not Deceased Sign.Not Surviving Total	<	<	>=	<	2
Sign.Living Not Deceased Not Surviving Total	<	<	>=	<=	4
Sign.Living Not Surviving Present	<	<	>=	=	2
Sign.Living Present	<	<	>=	>=	3
Sign.Living Not Deceased Total	<	<	>=	<=>	5
Sign.Living Sign.Not Deceased Sign.Not Surviving Total	<	<	>	<	1
Sign.Living Sign.Not Deceased Not Surviving Total	<	<	>	<=	2
Sign.Living Sign.Not Deceased Not Surviving Present	<	<	>	=	1
Sign.Living Sign.Not Deceased Present	<	<	>	>=	2
Sign.Living Sign.Surviving Present	<	<	>	>	1
Sign.Living Sign.Not Deceased Total	<	<	>	<=>	3
Sign.Living Sign.Not Surviving Total	<	<	<=>	<	3
Sign.Living Not Surviving Total	<	<	<=>	<=	5
Sign.Living Total	<	<	<=>	<=>	6
Sign.Not Surviving Past	<=	<	<=	<	3
Sign.Not Instantaneous Past	<=	<	<=	<=	4
Not Deceased Sign.Not Surviving Past	<=	<	=	<	2
Sign.Not Instantaneous Not Deceased Past	<=	<	=	<=	3
Not Deceased Sign.Not Surviving Total	<=	<	>=	<	4
Sign.Not Instantaneous Not Deceased Not Surviving Total	<=	<	>=	<=	7
Sign.Not Instantaneous Not Surviving Present	<=	<	>=	=	3
Sign.Not Instantaneous Present	<=	<	>=	>=	5
Sign.Not Instantaneous Not Deceased Total	<=	<	>=	<=>	9
Sign.Not Deceased Sign.Not Surviving Total	<=	<	>	<	2
Sign.Not Deceased Not Surviving Total	<=	<	>	<=	4
Sign.Not Deceased Not Surviving Present	<=	<	>	=	2
Sign.Not Instantaneous Sign.Not Deceased Present	<=	<	>	>=	4
Sign.Not Instantaneous Sign.Surviving Present	<=	<	>	>	2
Sign.Not Instantaneous Sign.Not Deceased Total	<=	<	>	<=>	6
Sign.Not Surviving Total	<=	<	<=>	<	5
Sign.Not Instantaneous Not Surviving Total	<=	<	<=>	<=	8
Sign.Not Instantaneous Total	<=	<	<=>	<=>	10
Past	<=	<=	<=	<=	5
Not Deceased Past	<=	<=	=	<=	4
Not Deceased Not Surviving Total	<=	<=	>=	<=	8
Not Surviving Present	<=	<=	>=	=	4
Present	<=	<=	>=	>=	6
Not Deceased Total	<=	<=	>=	<=>	10
Sign.Not Deceased Present	<=	<=	>	>=	5
Sign.Surviving Present	<=	<=	>	>	3
Sign.Not Deceased Total	<=	<=	>	<=>	7
Not Surviving Total	<=	<=	<=>	<=	9
Total	<=	<=	<=>	<=>	11

Table 5.6: Alternatives for a Relation with Participation Restriction

The specification of an EROOS universe relation with participation constraints can be found in Table 5.7. As presented in Table 5.6, the offered integrated constraints cover all potential order restrictions between the Creation and Destruction Timestamps of a refined and a participating object, except those cases for which

- an object is obliged to have a life span of zero length. This must rather be modelled as an event instead of an object (see also Table 5.14 on page 170).
- a participant object is obliged to be created at the same time as a refined object, which violates the existential dependency of the refined object on the participant object. Such obligation creates a mutual dependency between the refined and participating object. Mutual dependencies cannot be expressed using EROOS relations. The concept of an EROOS compound, presented in Section 5.2.5, is introduced for modelling mutual dependencies (see also Table 5.13 on page 169).

```
<EROOS universe relation script> =
"class" <CLASS NAME>
  "definition"
  ( "refined with binary relation"
    ( <positive number> | "unlimited" | "∞")
      <participant description> ","
    ( <positive number> | "unlimited" | "∞")
      <participant description>
  | "refined with unary relation" <participant description> )
    ( ( "unlimited" | "∞" ) "occurrences" | "one occurrence"
    | <positive number larger than 1> "occurrences" )
  "creation event"
    <creation event name>
      "(" <parameter name> ":" <CLASS NAME>
        [ "," <parameter name> ":" <CLASS NAME> ] ")"
    "effect"
      "new self↓"<Participant Name> "=" <parameter name>
      ["new self↓"<Participant Name> "=" <parameter name> ]
"end class" <CLASS NAME>

<participant description> = ( "mutable" | "immutable" ) 32
[["significantly"]"not deceased" | "significantly deceased" ]
[["significantly"]"not surviving" |"significantly surviving"]
["significantly lived" | "significantly not instantaneous"]
["present" | "past" | "total" ] <CLASS NAME> [ ° | † | °† ]
["as" <ROLE NAME> ]
```

Table 5.7: EROOS Universe Relation Script

[32] See Section 5.2.3 for the definition of a mutable participant.

5.1.6 EROOS Relations for the Library Example Revisited

Given the example of the library system that was presented in Section 2.3, and the relation hierarchy that was defined in Section 4.4.7, we can now revisit this model using EROOS universe classes that have class archives. As already indicated, this will result in a strong reduction of the number of classes, as shown in Figure 5.7. In addition, specific connectivity constraints can now be added to the relations, since these connectivity constraints only deal with restrictions on present objects. The following observations can be made:

- Most participants have a *present participation*, obliging that every refined object has a living participant object. An example is the registration relation between a person and a library
- A number of participants have a *not deceased total participation*, obliging that the participating object must be living when the refined object is created. However, this participating object may afterwards die. Examples are the publisher and book for a copy, and the allowance object for a borrowing. When the library changes its policy on the maximum number of allowed lent items, the ongoing borrowings should be untouched, since they cannot be recalled.
- A selection has a *not deceased past participation* in a borrowing, obliging that the selection has ended when the borrowing is made. In fact, the selection object is transformed into a borrowing object.
- A fine has a *not deceased not surviving total participation*, obliging that the borrowing has not yet ended when the fine is created, but that borrowing must have ended when the fine is paid, represented by the destruction of the fine object.
- A secondary author has *a total participation* in person, meaning that the person attached as an author to a book, can be living or dead. This expresses that it is possible that a certain author has died before the book is finished, expressed by the creation event of book.
- A secondary author has *a not surviving present participation* in author, obliging that the author must exist as long as the secondary author exists, and that the author must be destroyed at the moment the secondary author is destroyed. The only thing that cannot be expressed is the fact that all authors must be created at the same time. The relation that is currently defined, allows new secondary authors to be added to an existing book. However, once a secondary author has been added, it can only be removed by removing all authors, including the book object. In order to model the full dependency between authors, which is a mutual dependency, we need the concept of compounds that is presented in Section 5.2.7.

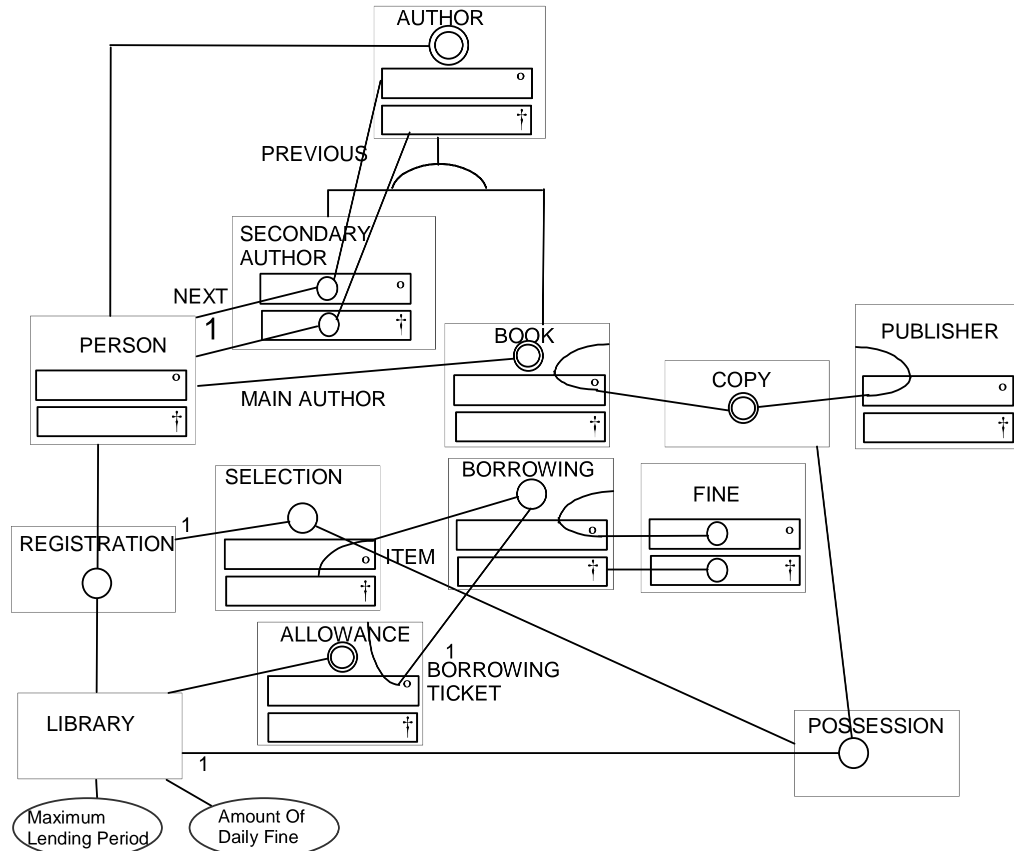


Figure 5.7: EROOS Relations for the Library System Revisited

5.1.7 Contributions, Related Work, and Reflections

Our contributions concerning the class archive concept are the following:

- Our approach concerning **class archives** is a novel and original contribution to conceptual modelling. Other analysis methods do not provide destructors, or consider destroyed objects as useless for the model. The introduction of class archives, and their usage in existential dependency relationships, provides a powerful and high-level modelling concept, in which important dependency constraints can be implied directly by the model structure. All kind of restrictions between the lifetime of a refined object and its participant object, can directly be specified in the relation definition.
- The **default attribute** ***Destruction Timestamp*** for all objects of every class, enables the modeller to reason about the moment at which an object has ceased to exist. This attribute does not have to be modelled explicitly, but it is automatically available for every object in EROOS. The default *Creation* and *Destruction Timestamp* attributes permit reasoning about the moments objects

are created and destroyed, for instance, by formulating queries that calculate the average life span. The modeller does no longer have to decide whether such attributes are needed in the model, since the EROOS methodology automatically exposes this kind of information for all objects.

- Objects that cease to exist are not vanished from the conceptual model, but still can be addressed to gather **historical information** regarding past events, former attribute values and the old relation links. The destruction of an object only reflects that the fact it is representing, has ceased to exist in the universe of discourse. Issues regarding the fact whether the object is still needed for obtaining certain information, or for performing certain tasks, are not under discussion during conceptual modelling.

Most object-oriented analysis methods do not pay much attention to the end of the lifetime of an object. Like many recent programming languages, some methods consider objects to remain living as long as they are relevant for the system being modelled. When objects are no longer needed, a garbage collector can automatically collect them and remove them from the system. A number of analysis methods and programming languages, such as C++ [144] and Ada [9], explicitly deal with the destruction of objects. However, they also consider an object to be deleted when it is no longer needed within the program. The EROOS universe decouples the destruction of an object from the removal of an object from the model. Objects in an EROOS conceptual model are never deleted from the model, but remain always available for querying, and even for participating in relations having past participants. The destruction of an object in the EROOS universe expresses that the original fact from the universe of discourse, which is represented by the object, is no longer valid.

An observation that can be made is that it is rather difficult to uncover the participation restrictions that are present in the universe of discourse. The modeller often tries to deduce them from a number of examples, but one must certain that the set of examples cover the full range of possibilities in the universe of discourse.

5.2 Mutability of Attribute Values and Relation Participants

This section introduces the EROOS kernel analysis pattern that has identified the necessity of introducing mutability of attribute values and relation participants. The definition of a mutable attribute and relation participant, the specification of mutation events, and the revised definition of implicit queries, are described thereafter. Last, mutability is applied on the running example of the library system.

5.2.1 EROOS Kernel Analysis Pattern for Mutability

Attribute values and relation participants do often not remain constant in the universe of discourse, but tend to change over time. After fixating their initial value at creation time, additional events can lead to a situation where the initial value becomes outdated, irrelevant, and useless. Such situation results in a specific EROOS kernel

analysis pattern that is presented in Figure 5.8. In the left part of the figure, attribute update objects are attached to the object that contains the initial attribute value. The provided query returns the attribute value of the latest update. In the right part, a generalisation class clusters the objects representing the initial attribute values, with the objects representing the updated values in order to obtain a single modelling of the attribute value on the generalisation level. In addition, the updates are sequentially ordered by refining the update with the generalisation class as the participant, using a '*not deceased past participation*' restriction for the participant. This expresses that only the last attribute update object is alive. Objects modelling previous values that have become irrelevant, must be destroyed in the model. In order to simplify the specification of this recurring EROOS kernel analysis pattern, the EROOS universe provides notational support by means of mutation events to model attribute value updates and relation participant updates.

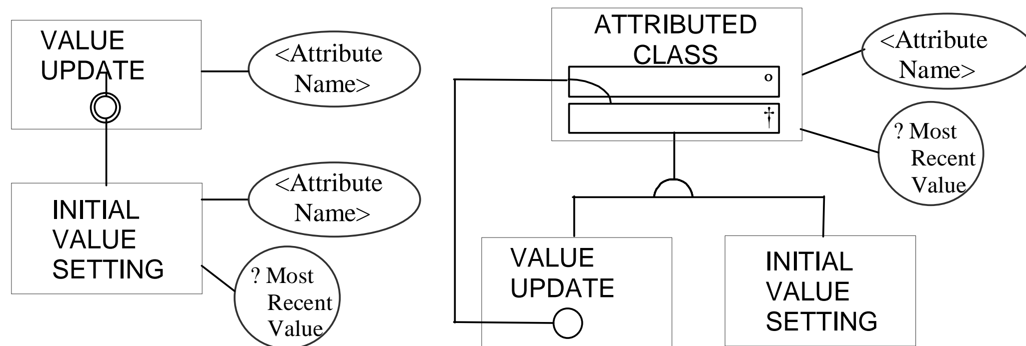


Figure 5.8: EROOS Analysis Pattern for a Mutable Attribute

5.2.2 Specification of a Mutable EROOS Attribute

Attribute mutability is introduced in the EROOS universe as an extension of the attribute concept. A mutable attribute allows changing the binding of an object with its attribute value over time, under the condition that a specific domain value is defined at each moment in time. In fact, a mutable attribute is a contraction of the EROOS kernel analysis pattern, as presented in Figure 5.8, into a single class. The syntax of a decoration script is given in Table 5.8, while Definition 5.3 presents the definition of an attribute in the EROOS universe. As presented in Figure 5.9, a mutable attribute is graphically represented by a small wave interrupting the line that connects the attribute to its decorated class.

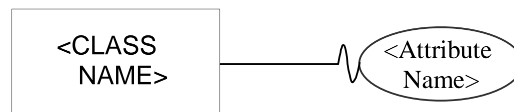


Figure 5.9: Graphical Representation of a Mutable EROOS Attribute

```
<EROOS universe attribute script> =
"class" <CLASS NAME>
  "definition"
    "decorated by"
      ( "mutable" | "immutable" ) [ "unique" ] "attribute"
        <Attribute Name> ":" <DOMAIN NAME>
        [ "constrained by" [ <lower bound> ( "<" | "≤" ) ]
          <Attribute Name> [ ( "<" | "≤" ) <higher bound> ] ]
  "creation event"
    <creation event name>
      [ "(" <parameter name> ":" <DOMAIN NAME> ")" ]
    "effect"
      ( "new self→"<Attribute Name> "=" <parameter name>
      | "new self→"<Attribute Name> "=" <domain expression> )
"end class" <CLASS NAME>
```

Table 5.8: EROOS Universe Attribute Script

An **attribute** is a model entity defining a property for a class for which, at each moment in time, every object of the class, called a decorated object, must be associated with a specific value of the domain defined for the attribute.

A **domain** is a collection of values that refer to static and unchangeable properties in the universe of discourse. A domain can be a magnitude, reference, denomination, or a composed domain.

An **immutable attribute** is an attribute for which the domain value associated to the object, is fixed during the entire lifetime of the object.

A **mutable attribute** is an attribute for which the domain value associated to the object, can change during the lifetime of the object.

Given

Model M; Class $C \in M_{cl}$; Domain $D \in M_d$;

Immutable Attribute $CIA \in M_a$; Mutable Attribute $CMA \in M_a$;

$CIA: TIME \rightarrow (C_t \rightarrow D) \mid$ (permanent binding)

$\forall\ t \in TIME: CIA_t \subseteq CIA_{t+1}$ (immutability)

$CMA: TIME \rightarrow (C_t \rightarrow D)$ (permanent binding)

Definition 5.3: EROOS Universe Attribute

Notice that archive attributes, as well as the default attributes *Creation Timestamp* and *Destruction Timestamp*, are immutable by nature. The destruction event moves

the object to the class archive and puts the object in a kind of final state. As such, the information that is encapsulated in the object, is frozen and cannot be changed anymore. Since archive attributes are only defined for objects in the class archive, their values can only be defined in the destruction event. Concerning the *Creation Timestamp*, it is obvious that, since an object can only be created once, the creation time is fixated at the moment the creation event occurs, and cannot be revised or changed anymore. This does not mean that the knowledge inside the software system about creation events that have occurred in universe of discourse, can never be revised. Only the fact that an event has happened and, therefore, an object has been created, cannot be rectified. Issues regarding the uncertain of knowledge about creation times, which can occur for an actual system, are design issues and should be dealt with during the design phase.

5.2.3 Specification of a Mutable EROOS Relation Participant

In the same manner as attributes can be changed, it is possible to change the participant of a refined object by changing the relation link that is encapsulated in a refined object. The EROOS universe allows the definition of a mutable relation participant that can change over time, allowing a refined object to change its participant object into another object of the participating class. However, the refined object has to adhere to the existential dependency constraint, stating that, at each moment in time, a specific object of the participating class must be connected to the refined object. A mutable participant removes the immutability constraint implied by the relation concept, while keeping the existential dependency constraint of the refined class on the participating class. The syntax of a refinement script was given in Table 5.7 on page 148, while the definition of a relation in the EROOS universe is given in Definition 5.4. As presented in Figure 5.10, a mutable participant is graphically represented by a small wave interrupting the line that connects the refined class to the participating class.

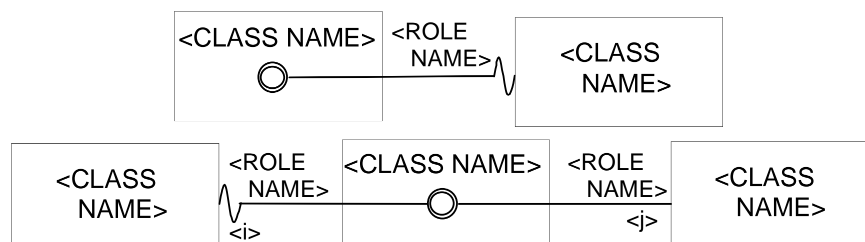


Figure 5.10: Graphical Representation of a Mutable EROOS Participant

A **relation** is a model entity defining a property for a class for which, at each moment in time, every object of the class, called a refined object, must be associated to a specific object, called a participant object, of the participating class defined for the relation.

A relation can either be a **binary relation,** defining exactly 2 participating classes for the refined class, or a **unary relation,** defining exactly 1 participating class.

An **immutable participant** is a participant for which the participant object associated to the refined object, is fixed during the entire lifetime of the refined object.

A **mutable participant** is a participant for which the participant object associated to the refined object, can change during the lifetime of the refined object.

Given

Model M; Class C,D,E,F,G $\in M_{cl}$;

Binary Immutable/Immutable Relation CIIB $\in M_{br}$;

Binary Immutable/Mutable Relation CIMB $\in M_{br}$;

Binary Mutable/Immutable Relation CMIB $\in M_{br}$;

Binary Mutable/Mutable Relation CMMB $\in M_{br}$;

Unary Immutable Relation CIU $\in M_{ur}$;

Unary Mutable Relation CMU $\in M_{ur}$;

CIIB: $TIME \rightarrow (C_t \rightarrow (D_t \times E_t)) \mid$ (existential dependency)

$\forall t \in TIME: CB_t \subseteq CB_{t+1}$ (immutability of D and E)

CIMB: $TIME \rightarrow (C_t \rightarrow (D_t \times E_t)) \mid$ (existential dependency)

$\forall t \in TIME: D(CB_t) \subseteq D(CB_{t+1})$ (immutability of D)

CMIB: $TIME \rightarrow (C_t \rightarrow (D_t \times E_t)) \mid$ (existential dependency)

$\forall t \in TIME: E(CB_t) \subseteq E(CB_{t+1})$ (immutability of E)

CMMB: $TIME \rightarrow (C_t \rightarrow (D_t \times E_t)) \mid$ (existential dependency)

CIU: $TIME \rightarrow (F_t \rightarrow G_t) \mid$ (existential dependency)

$\forall t \in TIME: CU_t \subseteq CU_{t+1}$ (immutability)

CMU: $TIME \rightarrow (F_t \rightarrow G_t)$ (existential dependency)

Definition 5.4: EROOS Universe Relation

5.2.4 Attribute and Relation Mutation Events

When attributes and relation participants are declared mutable, a mutation event can be defined. The mutation event will reflect a change that occurs in the universe of discourse, by changing the object properties in the model. In addition to creation and destruction events, mutation events will change the properties of the object on which it is applied. After the moment the mutation event has occurred, the implicit decoration query '→', refinement query '↓', and participation query '↑' will return the new value, respectively the new objects, that has been defined in the mutation

event. In addition to the current value of an attribute, a participant or a refinement object, the old information remains accessible by using a time indication '@t' for the query. This allows retrieving historical information concerning past attribute values and relation links. Figure 5.11 shows the graphical representation of a mutation event. The mutation script is given in Table 5.9, while the definition of a mutation event is presented in Definition 5.5 for participants, and Definition 5.6 for attributes.

```
<EROOS mutation event script> =
"class" <CLASS NAME>
  "context" <context clause>
  "mutation event"
    <mutation event name>
      [ "(" ( <parameter name> ":" <DOMAIN NAME> |
              <parameter name> ":" <CLASS NAME>
        [ "," <parameter name> ":" <CLASS NAME> ]) ")" ]
    "effect"
      ( "new self→"<Attribute Name> "=" <parameter name>
      | "new self→"<Attribute Name> "=" <domain expression>
      | "new self→"<Attribute Name> "="
         <domain function> ("self→"<Attribute Name>
           [ "," <parameter name> ] )
      | "new self↓"<Participant Name> "=" <parameter name>
      [ "new self↓"<Participant Name> "=" <parameter name> ])
"end class" <CLASS NAME>
```

Table 5.9: EROOS Mutation Event Script

A **relation mutation event** for a class refinement, is an event of a class that, if applied on a model instance at a certain time t, define a new link to objects of the participating classes as the relation link of the object starting from t.

Given

Model M; Event Universe EU; Event Set Instance E;

Class B,B1,B2,U,U1 $\in M_{cl}$; Object o $\in$ B; Object p1,p2 $\in$ B1;

Object q1,q2 $\in$ B2; Object r $\in$ U; Object s $\in$ U1;

Binary Relation CB $\in M_{br}$; Unary Relation CU $\in M_{ur}$;

Relation Mutation Event m1,m2,m3,m4 $\in$ EU;

$m1(o,p1,q1) \in E_{t+1} \Rightarrow CB_{t+1}(o) = (p1,q1)$

$m2(o,p2) \in E_{t+1} \Rightarrow CB_{t+1}(o) = (p,Q(CB_t(o)))$

$m3(o,q2) \in E_{t+1} \Rightarrow CB_{t+1}(o) = (P(CB_t(o)),q)$

$m4(r,s) \in E_{t+1} \Rightarrow CU_{t+1}(r) = s$

Definition 5.5: EROOS Relation Mutation Event

```
An attribute mutation event for a class decoration, is an
event of a class that, if applied on a model instance at a
certain time t, will define a new value of the appropriate
domain as the attribute value for the object starting from
moment t.
Given
Model M; Event Universe EU; Event Set Instance E;
Class C ∈ M_cl; Object o ∈ C; Attribute CA ∈ M_a;
Domain D ∈ M_d; Attribute Value a ∈ D;
Attribute Mutation Event m ∈ EU;
m(o,a) ∈ E_t+1 ⇒ CA_t+1(o) = a
```

Definition 5.6: EROOS Attribute Mutation Event

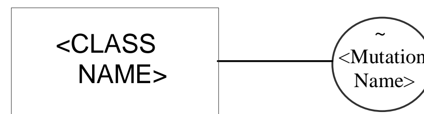


Figure 5.11: Graphical Representation of a Mutation Event

A mutation event is a special case of an EROOS event, as presented in Section 4.7.3. Instead of clustering a number of events in a new event, a mutation event is a core building block that can be used in the definition of other events. It defines the possibility of changing the value of an attribute or participant object. Table 5.10 presents an extension of the specification formalism for events that was given earlier in Table 4.16 on page 124, allowing the usage of destruction and mutation events in addition to creation and general events.

```
< event expression> =
  ( ["let" <mnemonic> "=" ] <CLASS NAME>"."
     <creation event name>"("<parameter expression>")"
  | <object expression>"." ( <destruction event name>
    | mutation event name> | <general event name> )
      "(" <parameter expression> ")" )+
```

Table 5.10: Event Expression in an EROOS Event Script

5.2.5 Implicit Attribute, Refinement, and Participation Queries

In order to retrieve old values of attributes, participants and refinements, the implicit decoration query '→', refinement query '↓' and participation query '↑' can be extended with a time indication '@t'. When using a time indication, the implicit

query does not return the latest defined value or object, but will return the value or object that was defined in the model instance at the moment t. In order to be applicable, the query cannot be specified for moments that are later than '**now**', nor for moments t that are sooner than the *Creation Timestamp* of the object on which it is applied. The definition of the implicit attribute query '→A@t' can be found in Definition 5.7. Since the definition of the refinement query '↓P@t' and participation query '↑R@t' are analogous, they have been omitted.

An **implicit query →A**@t or **→C/A**@t for an attribute A of a class C is a query that can be applied on an object of class C at any moment t′ ≥ t, and that returns the attribute value that was associated to the object on moment t (with t ≥ *Creation Timestamp* of the object on which it is applied).

Given

Model M; Class C $\in M_{cl}$; Attribute CA $\in M_a$; Domain D $\in M_d$;

Query →C/A $\in M_q$;

$\rightarrow C/A: TIME \rightarrow (\ ]-\infty, t'] \rightarrow (C_t \rightarrow D)\ )$

$\forall\ t,t' \in TIME: \forall\ o \in C_t: (t' \geq t) \Rightarrow (\rightarrow C/A_{t'}(t,o) = CA_t(o))$

Definition 5.7: Implicit EROOS Universe Attribute Query

5.2.6 EROOS Mutability for the Library Example

Given the example of the library system that was presented in Section 2.3, and the relation hierarchy that was defined in Section 5.1.6, we can identify the attribute '*Amount Of Daily Fine*' as being mutable, as presented in Figure 5.12. We could also specify the attribute '*Maximum Lending Period*' as mutable, but this creates a potential problem for the running borrowings and the applied fines. The analyst must clearly identify the rules that are applicable in such situation. It is possible that running borrowings are considered to keep their old deadline, but it could also be the case that the new shorter or longer deadline is retroactively imposed on these running borrowings, removing earlier applied fines or creating additional fines.[33] When the new rules apply to all running and new borrowings, the constraint that checks the deadline must refer to the most recent attribute value. However, when the old rules still apply for the running borrowings, the constraint that checks the deadline must refer to the attribute value at the moment the borrowing object has been created.

[33] Regardless of the fact whether it is a good policy of a library to retroactively charge fines when the maximum lending period is unilaterally reduced.

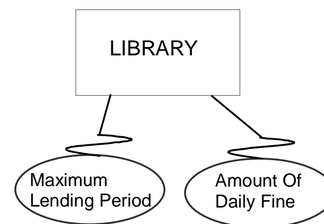


Figure 5.12: EROOS Mutability for the Library System

5.2.7 Contributions, Related Work, and Reflections

The EROOS mutability concept is largely comparable with the *{readOnly}* property modifier in UML. Our approach concerning **availability of past attribute and relation information** is a novel and original contribution to conceptual modelling. This is achievable due to the fact that the mutability concept is defined on top of the constructional model of the EROOS kernel. While other analysis methods consider attributes as instance variables that are overwritten when a new value is defined, EROOS allows the modeller to reason about any past model instance state using a time indication for an attribute, a refinement or a participation query.

An observation that can be made is that **mutability support** contradicts the principle of *Uniqueness*. However, it has a large impact on controlling the size of conceptual model. Instead of having to model all changes of attribute values or relation participants as separate objects, the attribute value of the original object can be adjusted. Mutability is commonly used in software engineering, and, therefore, the EROOS universe offers the mutability concept for conceptual modelling. In contrast to other analysis notations and programming languages, all information concerning the previous values and attached objects remain reachable in the model. However, one must be aware that mutability raises an important question of how certain changes must be represented in a conceptual model, either as a class or as a mutation. This can often not clearly be decided and is therefore left to the judgement of the modeller. The EROOS universe guides the modeller to use mutations when (1) no additional information is needed concerning the actual update, and (2) no specific constraints are imposed on the update. In all other cases, the explicit creation of mutation objects is obliged, since information and constraints can only be attached to objects and not to events.

5.3 Compounds and Mutual Dependency

The EROOS kernel offers a number of concepts to build a conceptual model of the universe of discourse. Existential dependency between objects, expressed through relations, is used as the main criterion for building a model of the relevant

information, facts and knowledge of the universe of discourse, and their interrelations and dependencies. However, no explicit support is provided for modelling mutually dependent elements in the universe of discourse. Mutual dependency expresses the fact that one element cannot exist without another element, and vice versa. EROOS relations cannot be used to express mutual dependency between objects. Although a refined object is existentially dependent on its participant objects, a participant object cannot depend directly or indirectly on the object in which it participates. Mutually dependent elements must be merged into a single object. However, such approach introduces a disruption between the universe of discourse and the conceptual model, since the traceability of the information from the universe of discourse into objects of the conceptual model is no longer evident. Therefore, the EROOS universe introduces the concept of *compound* to model mutual dependency between objects. Additional constraints, such as connectivity and mutability constraints, are integrated in the definition of a compound.

5.3.1 EROOS Compounds and Object Compound Links

An EROOS compound involves two classes, namely (1) a class expressing enclosure objects, having the '*whole*' role in the compound, and (2) a class expressing enclosed objects, having the '*part*' role. A compound is a special kind of association that expresses mutual dependency between a single compound-whole and number of compound-part objects. An additional restriction is placed on the objects of the *part* class, which states that they can only be connected to exactly one object of the *whole* class. As such, each *part* object must at all times be attached to exactly one *whole* object, while each *whole* object must at all times be attached to at least one *part* object. In analogy with EROOS relations and relation links, the connection between a *part* object and a *whole* object is called compound link, which expresses that the *part* object is attached to the *whole* object. A compound link can be seen as encapsulated in both the *whole* object and the *part* object.

The object structure that can be associated with a compound relationship at a certain moment t, is illustrated in Figure 5.13. As presented in Figure 5.14.a, a compound is graphically represented in the form of a line between the *whole* class and the *part* class that forks at the side of the *part* class. The fork at the side of the *part* class expresses the fact that one or more *part* objects can be attached to a single *whole* object. This achieves the principle of uniqueness as defined in Section 3.1, forcing the analyst to introduce additional model entities, or use more appropriate modelling concepts in specific cases.

- In order to model compounds between two *whole* objects containing each other, the analyst is forced to decompose it into a number of basic whole-part compounds between the *whole* objects, revealing hidden *part* objects linking the two *whole* objects, as presented in Figure 5.14.b.
- It is impossible to model duplicate compound links in EROOS. If a *part* object participates more than one time in a *whole* object, the participation of the *part* object in the *whole* object must be made explicit using an intermediate class as shown in Figure 5.14.c.

- Compounds between a *whole* object and zero or more *part* objects, must be transformed into an ordinary relation, refining the *part* class and having the *whole* class as an ordinary participant, as presented in Figure 5.14.d. As such, objects of the transformed *whole* class can exist without being connected to a transformed *part* object, which can be added at the moment of creation of a transformed *whole* object or in its later lifetime.

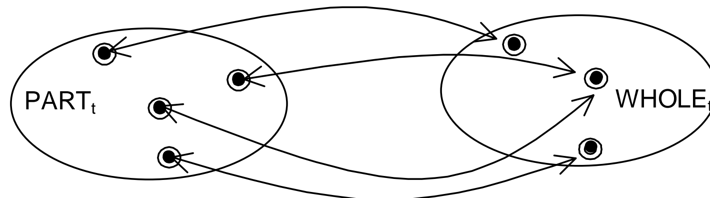


Figure 5.13: Objects involved in an EROOS Compound at Moment t

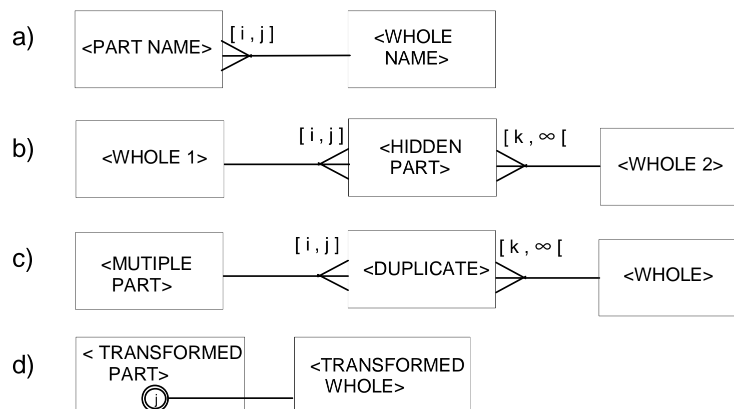


Figure 5.14: EROOS Compound and Alternative Constructs

Compounds can thus be seen as a restricted form of a part-whole structure, in which (1) a *whole* must contain 1 or more *parts* of the same kind, and (2) a *part* cannot exist without being attached to a single *whole*. Notice that some dependencies seem to be part-whole dependencies, such as the connection between a car and its wheels, but are of a different nature, since a wheel can exist without being connected to the car, and a car can exist without having four wheels attached. However, there is a dependency between a driving car and its mounted wheels.

5.3.2 Model Constraints implied by the Compound Concept

EROOS incorporates important model constraints directly in the methodological concepts. The following constraints are directly implied by the compound concept:

- **Mutual existential dependency**: A compound between two classes implies the existential dependency of (1) the *part* object on exactly one *whole* object, and (2)

of the *whole* object on one or more *part* objects. This means that each object of the *part* class must at all times be associated with exactly one object of the *whole* class, and each object of the *whole* class must at all times be associated with at least one object of the *part* class.

- **Immutability**: When a *whole* and a *part* class are declared immutable, the compound association between a *whole* and a *part* object is considered to be static. Moreover, the total set of compound links for a *whole* object is also considered to be static and not expandable. In particular, at the moment a *whole* object is created, it must be associated with all its *part* objects, and it will keep that association for its entire lifetime, i.e., *part* objects cannot switch from one *part* object to another during their lifetime. Notice that the immutability constraint can be relaxed for a *whole* object, a *part* object, or both.

5.3.3 Integrated Compound Constraints on Connectivity

The definition of a compound can be complemented with constraints restricting the amount of *part* objects that can be connected to a single *whole* object. An EROOS compound definition allows the possibility of specifying a lower and upper bound on the number of *part* objects that may exist for a specific *whole* object. The lower bound must be at least one, while the upper bound can be defined as a value, or as **unlimited** ('∞'). The specification of a compound connectivity constraint is integrated in the *part* clause of the compound specification, as presented in Section 5.3.5. A compound connectivity constraint is graphically represented by noting the lower and upper connectivity values at the *part* class, as presented in Figure 5.14.

5.3.4 Integrated Compound Constraints on Mutability

EROOS provides the possibility of defining a compound whole and a compound part as mutable, which allows changing the links between a *whole* and a *part* object. These mutations are only allowed under the condition that, at each moment in time, a specific *whole* object must be connected to at least one *part* object, and a specific *part* object must be connected to exactly one *whole* object. Notice that mutability for a compound-whole and a compound-part object is different in nature:

- Defining mutability for a *whole*, allows a *whole* to change the set of *parts* to which it is connected. As such, it is possible to change existing compound links, by adding *parts* to a *whole*, or removing *parts* from a *whole*. This does not necessarily mean that the *parts* are mutable too. New *parts* can be added to a *whole* when they are created, and removed when they are destroyed. As such, *parts* can stay connected all the time to the same *whole*.
- Defining mutability for a *part*, allows a *part* to change the *whole* to which it is attached. This does not necessarily mean that the *whole* is mutable too. It is for instance possible that the *part* can only be reconnected to a new *whole* at the moment an old *whole* is destroyed and a new *whole* is created. As such, a *whole* can stay connected all the time to a fixed set of *parts*.

As presented in Figure 5.15, a mutable *whole* is graphically represented by a small wave interrupting the line connecting the *whole* class, while a mutable *part* is represented by a small wave interrupting the line connecting the *part* class.

Figure 5.15: Graphical Representation of a Mutable EROOS Compound

5.3.5 Specification of an EROOS Compound

The definition of a compound between a *whole* class and a *part* class, is represented in a compound script. It can be specified from the viewpoint of the *part* or the *whole* class. The property of mutual dependency influences the creation of *whole* and *part* objects. Each time a *whole* object is created, the binding with its *part* objects must be established as well. In the same manner, a binding with a *whole* object must be established each time a *part* object is created. Functionality acting on the compound and its compound links will be placed at the *part* object, extending the creation event for the *part* class that defines its binding with a *whole* object. A class can be involved in many compounds, as a compound-whole as well as a compound-part. The *whole* class of the compound must always be different from the *part* class of the compound. The syntax of a compound-part script is given in Table 5.11. The definition of a compound can be found in Definition 5.8, while the definition of the extended creation event for a compound can be found in Definition 5.9.

```
<compound-part script> =
"class" <CLASS NAME>
  "definition"
    "involved as compound-part"
      "min" <positive number>
      "max" ( <positive number> | "unlimited" | "∞" )
      <part description> [34]
    "having compound-whole" <whole description> [34]
  "creation event"
    <creation event name>
      "(" <parameter name> ":" <CLASS NAME> ")"
    "effect"
      "new self ➢"<Compound-Whole Name> "=" <parameter name>
"end class" <CLASS NAME>
```

Table 5.11: EROOS Compound-Part Script

[34] See Table 5.15 on page 170 for the definition of the part and whole descriptions.

A **compound** is a model entity defining a property for two classes, namely a *part* class and a *whole* class, for which, at each moment in time, every object of the *part* class must be associated to exactly one object of the *whole* class, and every object of the *whole* class must be associated to at least one object of the *part* class.

An **immutable *part*** is a *part* class for which the *whole* object associated to each *part* object, is fixed during the entire lifetime of the *part* object.

A **mutable *part*** is a *part* class for which the *whole* object can change during the lifetime of the *part* object.

An **immutable *whole*** is a *whole* class for which the set of *part* objects associated to each *whole* object, is fixed during the entire lifetime of the *whole* object.

A **mutable *whole*** is a *whole* class for which the set of *part* objects can change during the lifetime of the *whole* object.

Given

Model M; Class P,W $\in M_{cl}$;

Mutable Whole/Mutable Part Compound CMM $\in M_{co}$;

Mutable Whole/Immutable Part Compound CMI $\in M_{co}$;

Immutable Whole/Mutable Part Compound CIM $\in M_{co}$;

Immutable Whole/Immutable Part Compound CII $\in M_{co}$;

CII: TIME $\rightarrow$ ($P_t \rightarrow W_t$) | (dependency P to W

$\forall$ t $\in$ TIME: $\forall$ w $\in W_t$: $\exists$ p $\in P_t$: $CII_t(p) = w$ and W to P)

$\forall$ t $\in$ TIME: $CII_t \subseteq CII_{t+1}$ (immutability of *part*

$\forall$ t $\in$ TIME: $\forall$ w $\in W_t$: $CII^{-1}_t(w) = CII^{-1}_{t+1}(w)$ and *whole*)

CMI: TIME $\rightarrow$ ($P_t \rightarrow W_t$) | (dependency P to W

$\forall$ t $\in$ TIME: $\forall$ w $\in W_t$: $\exists$ p $\in P_t$: $CMI_t(p) = w$ and W to P)

$\forall$ t $\in$ TIME: $CMI_t \subseteq CMI_{t+1}$ (immutability of *part*)

CIM: TIME $\rightarrow$ ($P_t \rightarrow W_t$) | (dependency P to W

$\forall$ t $\in$ TIME: $\forall$ w $\in W_t$: $\exists$ p $\in P_t$: $CIM_t(p) = w$ and W to P)

$\forall$ t $\in$ TIME: $\forall$ w $\in W_t$: $CIM^{-1}_t(w) = CIM^{-1}_{t+1}(w)$ (immut. of *whole*)

CMM TIME $\rightarrow$ ($P_t \rightarrow W_t$) | (dependency P to W

$\forall$ t $\in$ TIME: $\forall$ w $\in W_t$: $\exists$ p $\in P_t$: CMM(p) = w and W to P)

Definition 5.8: EROOS Compound

The different components in the specification of a compound script are:

- The specification of a compound essentially identifies the *whole* and *part* class involved in the compound. In addition, constraints of multiplicity and mutability can be integrated into the compound definition. A role name for the *whole* and the *part* class can be specified as a reference to it.

- The specification of the creation event for the *part* class specifies the binding of the *part* object with an object of the *whole* class. For that purpose, a formal argument is provided for the creation event of a compound-part class in order to establish the binding of the newly created *part* object with a compound-whole object. The assertion states that, if the given implicit query ➣ <Compound-Whole Name>, defined in section 5.3.6, will be applied to the newly created object, referred to as **self**, at the moment the creation has occurred, referred to as **new**, the object on the right-hand side must be returned as a result.

An **extended creation event** for a compound, is an event of a *part* class that, if applied on a model instance at a certain time, in addition to adding a new object to the object population set for that class, will define a compound link to the object of the appropriate *whole* class.

Given

Model M; Object Universe OU; Event Universe EU;

Event Set Instance E; Class P $\in M_{cl}$; Object w $\in$ C;

Compound CO $\in M_{co}$; Creation event c $\in$ EU;

$c(w) \in E_{t+1} \Rightarrow \exists\ p \in OU$:

$(p \notin P_t) \wedge (p \in P_{t+1}) \wedge (CO_{t+1}(p) = w)$

Definition 5.9: Extended Creation Event for an EROOS Compound-Part

5.3.6 Implicit Compound Queries

The definition of an EROOS compound is automatically complemented with two implicit compound queries, offering the ability to inspect the compound links for a *part* and *whole* object. Given a compound for a *whole* class W and a *part* class P,

- the implicit compound query '➣W', or '➣WR' in case that WR is a role name given to the compound-whole class W, applicable to each object p of the compound-part class P, returns the object of the *whole* class W incorporated in the compound link for p.
- the implicit compound query '⤙ P', or '⤙ PR' in case that PR is a role name given to the compound-part class P, applicable to each object w of the compound-whole class W, returns the set of objects of the *part* class P that are connected to the object w.

The relation between '➣W' and '⤙P' can be defined as follows:

$$\forall\ p \in P: \forall\ w \in W : p➣W = w \Leftrightarrow p \in w⤙P$$

The notation supports the view of navigating between the *whole* and the *part* class according to its graphical notation. These implicit queries are mainly used in (1) specifying the semantics of creation events and mutation events, (2) specifying

queries in order to retrieve information from a model instance, and (3) specifying model navigation expressions between a *whole* and a *part* class. Notice that, in addition to the current value of a compound-whole and compound-part, old information remains accessible by using a time indication '@t' for the implicit compound queries. This allows the analyst to retrieve historical information concerning past compound links. The definition of the implicit compound queries '➣' and '⤛' can be found in Definition 5.10.

An **implicit compound query** ➣**W**@t or ➣**P/W**@t for a *part* class P, or ➣**WR**@t, ➣**P/WR**@t**,** ➣**WR/W**@t, or ➣**P/WR/W**@t for a *whole* with role name WR, is a query that can be applied on an object of the *part* class at a moment t' ≥ t, and that returns the *whole* object contained in the compound link for that *part* object on moment t (t ≥ *Creation Timestamp* of the object)

An **implicit compound query** ⤛**P**@t or ⤛**W/P**@t for a *whole* class W, or ⤛**PR@**t**,** ⤛**W/PR@**t**,** ⤛**PR/P@**t or ⤛**W/PR/P@**t for a *part* with role name PR, is a query that can be applied on an object of the *whole* class at a moment t' ≥ t, and that returns the set of all *part* objects contained in the compound links for that *whole* object on moment t (t ≥ *Creation Timestamp* of the object).

Given

Model M; Compound CO $\in M_{co}$; Class P,W $\in M_{cl}$;

Query ➣P/W, ⤛W/P $\in M_q$;

$$➣P/W: TIME \rightarrow (\ ]-\infty,t'] \rightarrow (P_t \rightarrow W_t)\ ) \ |$$

$$\forall\ t,t' \in TIME: \forall\ p \in P_t :$$

$$(t' \geq t) \Rightarrow (\ ➣P/W_{t'}(t,p) = CO_t(p)\ )$$

$$⤛W/P: TIME \rightarrow (\ ]-\infty,t'] \rightarrow (W_t \rightarrow \mathcal{P}(P_t))\ ) \ |$$

$$\forall\ t,\ t' \in TIME: \forall\ w \in W_t : \forall\ p \in P_t :$$

$$(t' \geq t) \Rightarrow (\ P \in ⤛W/P_{t'}(t,w) \Leftrightarrow CO_t(p) = w\ )$$

Definition 5.10: Implicit EROOS Compound Query

5.3.7 Compound Mutation Events

In order to change compound links for a *whole* and a *part* object, a mutation event must be defined. Such event reflects a change within the universe of discourse into the conceptual model. After the moment the mutation event has occurred, the implicit compound queries will return the new objects that have been defined in the mutation event. The *part* and *whole* mutation script is given in Table 5.12, while the definition of a compound mutation event is presented in Definition 5.11.

```
<compound mutation event script> =
"class" <CLASS NAME>
  "context"
    <context clause>
  "mutation event"
    <mutation event name>
      "(" <parameter name> ":"
          ( <CLASS NAME> | <object set name> ) ")"
    "effect"
     ( "new self≻ "<Compound-Whole Name> "=" <parameter name>
     | "new self≺ "<Compound-Part Name> "="
         <object set expression> )
"end class" <CLASS NAME>
```

Table 5.12: EROOS Compound Mutation Event Script

A **compound *part* mutation event** for a compound, is an event of a *part* class that, if applied on a model instance at a certain time t, will define a new compound link to the *whole* object as the compound link of the *part* object.

A **compound *whole* mutation event** for a compound, is an event of a *whole* class that, if applied on a model instance at a certain time t, will define a new set of compound links to *part* objects as the compound link of the *whole* object.

Given

Model M; Event Universe EU; Event Set Instance E;

Class $P,W \in M_{cl}$; Object $p \in P$; Object $w,x \in W$;

Object Set $Q \in \mathcal{P}(P)$; Compound $CO \in M_{co}$;

Compound Mutation Event event $m1,m2 \in EU$;

$m1(p,w) \in E_{t+1} \Rightarrow CO_{t+1}(p) = w$

$m2(x,Q) \in E_{t+1} \Rightarrow \forall\, q \in Q: CO_{t+1}(q) = x$

Definition 5.11: EROOS Compound Mutation Event

5.3.8 Class Archive as a Compound Participant

In analogy with relations, EROOS compounds offer the possibility of using the class archive of the *whole* class and the *part* class in the definition of the compound. This allows the formulation of additional restrictions between the lifetimes of the *whole* object and the *part* objects. Given the fact that a compound expresses a mutual dependency between the *whole* object and the *part* objects, the past population cannot be used in the definition of a compound. This leads to four types of class involvement in a compound. All four types include the core mutual existential dependency

property that must be valid for every *whole* object w and *part* objects object p, namely

w→Creation Timestamp = p→ Creation Timestamp[35]

- A '**present *part*-present *whole* participation**', as presented in Figure 5.16.a, obliges that the *part* object p is alive during the entire lifetime of the *whole* object w, and that the *whole* object w is alive during the entire lifetime of the *part* object p. This can be expressed as:

  p→Creation Timestamp = w→Creation Timestamp ≤ w→Destruction Timestamp = p→Destruction Timestamp

- A '**present *part*-total *whole* participation**', as presented in Figure 5.16.b, obliges that the *part* object p is alive during the entire lifetime of the *whole* object w, but that the *whole* object w can die during the lifetime of the *part* object p. This can be expressed as:

  p→Creation Timestamp = w→Creation Timestamp ≤ w→Destruction Timestamp ≤ p→Destruction Timestamp

- A '**total *part*-present *whole* participation**', as presented in Figure 5.16.c, obliges that the *whole* object w is alive during the entire lifetime of the *part* object p, but that the *part* object p can die during the lifetime of the *whole* object w. This can be expressed as:

  p→Creation Timestamp = w→Creation Timestamp ≤ p→Destruction Timestamp ≤ w→Destruction Timestamp

- A '**total *part*-total *whole* participation**', as presented in Figure 5.16.d, only obliges that the *part* object p and *whole* object w are created together. The *part* object p can die during the lifetime of the *whole* object w, and vice versa.

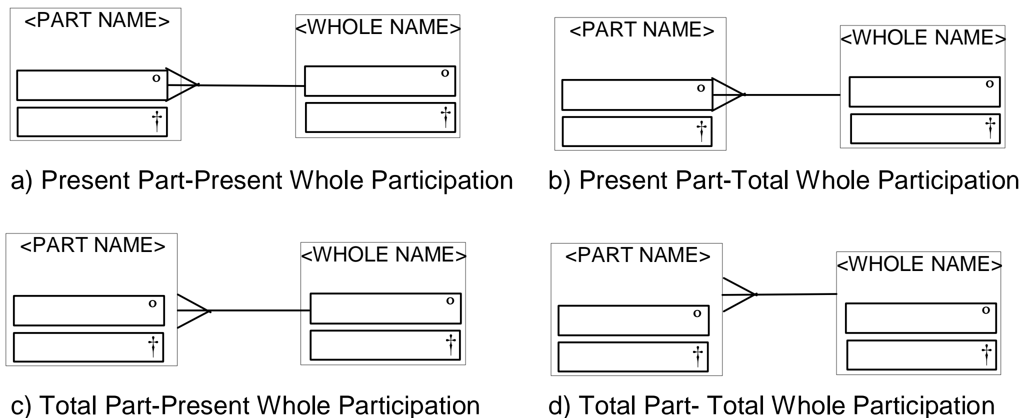


Figure 5.16: EROOS Compound Participation Types

[35] Notice that this property only applies when no mutation of the *part* or the *whole* object has taken place.

In addition to the four compound participation types, additional integrated constraints can be added to a compound participation, further restricting the dependency rules between the *whole* object and the *part* objects. These restrictions can be combined in order to create additional combinations of meaningful participant restrictions.

- A '**significantly not deceased *whole* participation**' obliges that the *whole* object will remain living after the *whole* and the *part* objects have been created.

  w→Creation Timestamp = p→Creation Timestamp < w→Destruction Timestamp

- A '**significantly not deceased *part* participation**' obliges that the *part* object will remain living after the *whole* and the *part* objects have been created.

  p→Creation Timestamp = w→Creation Timestamp < p→Destruction Timestamp

- A '**significantly surviving present *part*/significantly not surviving total *whole* participation**' obliges that a *part* object will still remain living after the destruction of the *whole* object.

  w→Destruction Timestamp < p→ Destruction Timestamp

- A '**significantly not surviving total *part*/significantly surviving present *whole* participation**' obliges that the *whole* object will still remain living after the destruction of the *part* objects.

  p→Destruction Timestamp < w→ Destruction Timestamp

Name of Compound Restriction	Restrictions between *whole* object w and *part* object p				Number of cases
	w.CTS/p.CTS	**w.CTS/p.DTS**	**w.DTS/p.CTS**	**w.DTS/p.DTS**	
Sign.Surviving Present *Part*-Sign.Not Deceased Sign.Not Surviving Total *Whole*	=	<	>=	<	2
Present *Part*-Sign.Not Deceased Total *Whole*	=	<	>=	<=	3
Total *Part*-Sign.Not Deceased Total *Whole*	=	<	>=	<=>	4
Sign.Not Deceased Sign.Surviving Present *Part*-Sign.Not Deceased Sign.Not Surviving Total *Whole*	=	<	>	<	1
Sign.Not Deceased Present *Part*-Sign.Not Deceased Total *Whole*	=	<	>	<=	2
Sign.Not Deceased Present *Part*-Sign.Not Deceased Present *Whole*	=	<	>	=	1
Sign.Not Deceased Total *Part*-Sign.Not Deceased Present *Whole*	=	<	>	>=	2
Sign.Not Deceased Sign.Not Surviving Total *Part*-Sign.Not Deceased Sign.Surviving Present *Whole*	=	<	>	>	1
Sign.Not Deceased Total *Part*-Sign.Not Deceased Total *Whole*	=	<	>	<=>	3
Present *Part*-Total *Whole*	=	<=	>=	<=	4
Present *Part*-Present *Whole*	=	<=	>=	=	2
Total *Part*-Present *Whole*	=	<=	>=	>=	4
Total *Part*-Total *Whole*	=	<=	>=	<=>	6
Sign.Not Deceased Total *Part*-Present *Whole*	=	<=	>	>=	3
Sign.Not Deceased Sign.Not Surviving Total *Part*-Sign.Surviving Present *Whole*	=	<=	>	>	2
Sign.Not Deceased Total *Part*-Total *Whole*	=	<=	>	<=>	4

Table 5.13: Possibilities for a Compound with *Part* and *Whole* Restrictions

The offered integrated constraints cover all potential restrictions between the *Creation* and *Destruction Timestamps* of a *part* object and a *whole* object, as presented in Table 5.13, except those cases presented in Table 5.14. In these cases, an object is obliged to have a life span of zero length, which models an event rather than an object. The specification of integrated compound constraints, as a further detailing of the compound script in Table 5.11 on page 163, can be found in Table 5.15.

Name of Participant Restriction	Restriction between participant p and refined object r				Number of possibilities
	p.CTS/r.CTS	p.CTS/r.DTS	p.DTS/r.CTS	p.DTS/r.DTS	
Zero lifespan for object r	<	<	=	=	1
Zero lifespan for object r	<=	<=	=	=	2

Name of Compound Restriction	Restrictions between *whole* object w and *part* object p				Number of possibilities
	w.CTS/p.CTS	w.CTS/p.DTS	w.DTS/p.CTS	w.DTS/p.DTS	
Zero lifespan for object w	=	<	=	<	1
Zero lifespan for object w	=	<=	=	<=	2
Zero lifespan for objects p and w	=	=	=	=	1
Zero lifespan for object p	=	=	>=	>=	2
Zero lifespan for object p	=	=	>	>	1

Table 5.14: Unsupported Restriction Cases in EROOS

```
<whole description> =
( "mutable" | "immutable" )
[ "significantly not deceased" ]
[ "significantly not surviving" | "significantly surviving"]
["present" | "total" ] <CLASS NAME> [ ° | °† ]
[ "as" <ROLE NAME> ]

<part description> =
( "mutable" | "immutable" )
[ "significantly not deceased" ]
[ "significantly not surviving" | "significantly surviving"]
["present" | "total" ] <CLASS NAME> [ ° | °† ]
[ "as" <ROLE NAME> ]
```

Table 5.15: EROOS Compound Script Usage of Class Archive

5.3.9 EROOS Compounds for the Library Example

Given the example of the library system that was presented in Section 2.3, and the revisited relation hierarchy that was defined in Section 5.1.6, we can identify a mutual dependency between a book and its authors. As stated during the discussion of the revisited relation hierarchy, it was not yet possible to express the fact that all authors

must be created at the same moment in time. Moreover, the book and the main author had to be modelled as a single object. By introducing an EROOS compound between a book as a *whole* object and its authors as *part* objects, as presented in Figure 5.17, we can (1) segregate the book object from its main author object, and (2) define that there is a mutual dependency between a book and its authors, expressing that that all authors must be created at the same moment as the book is created. An observation that can be made is that it is not necessary to introduce ordered relation participant sets or ordered compound-part sets, since the ordering can be made explicit using specialisation hierarchies.

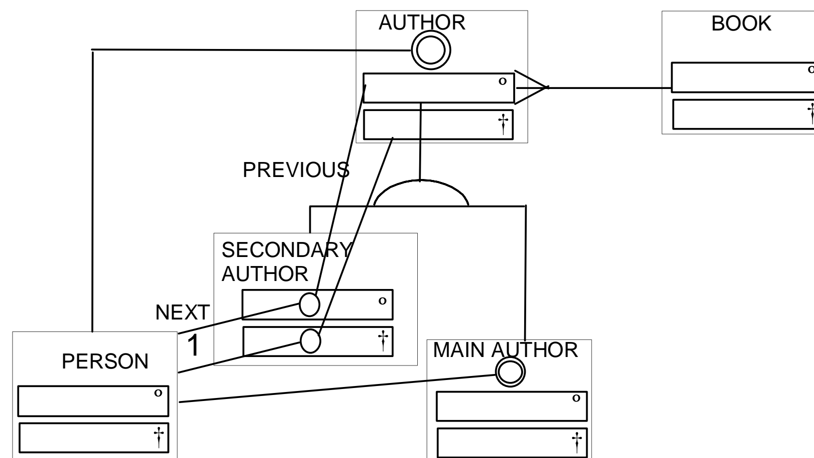


Figure 5.17: EROOS Compounds for the Library System

5.3.10 Contributions, Related Work, and Reflections

Our contributions concerning the compound concept are the following:

- The introduction of **compounds** offers the modeller a clear and well-defined concept for modelling mutual dependency and part-whole structures, consisting of a non-empty *whole* and a number of dependent *parts*. While UML offers an ambiguous definition for aggregates and composition, which (1) do not imply the obligation of mutual dependency, and (2) do not clearly indicate the differences between associations, aggregates and compositions, EROOS explicit defines the distinction between relations, expressing a unilateral existential dependency, and compounds, expressing a mutual dependency.
- A consequent application of the **mutability, class archive, and integrated constraints approach for the compound concept**, offers a coherent methodological approach for conceptual modelling.

The EROOS compound concept is somewhat comparable with the aggregation and composition concept in UML. UML offers (1) aggregation, which expresses whole-part relationships, and (2) composition, which expresses a strong ownership of parts

by the composite and coincident lifetime. However, these concepts do not incorporate an obliged dependency, since it is possible to express that an aggregation and a composite can have optional parts, or that a part possibly does not belong to any aggregation or composite. The only restriction in UML is that a part object can only belong to no more than one composite object, although a part object can also belong to other composite objects through different compositions. In addition, although there is coincident lifetime of parts with the composite, a part can even be removed from a composite before the death of a composite. UML does not make a clear semantic distinction between association, aggregation, and composite. Rumbaugh [128] even explicitly states that the distinction in UML between aggregation and association is a matter of taste, rather than a difference in semantics. EROOS only offers relations, expressing existential dependency, and compounds, expressing mutual dependency. The difference between these two relational concepts is clearly defined:

- When two objects are not dependent on each other, they should both be participants in an additional relation between them, captured in a refined class.
- When one object is dependent on the other object, the class of the first object is refined by a relation, having the class of the second object as participant.
- When the two objects are mutually dependent, a compound between the two classes must be defined.

An observation that can be made is that compounds offer to possibility for modelling object slicing, using a *whole* having a *part* with connectivity [1,1]. It is currently still unclear how to make a distinction between desired object slicing, e.g., when the *whole* object specifies a continuing property of a membership whereas the *part* objects specify yearly renewals, and unwanted object slicing, e.g., when the lifetimes of the *whole* and the *part* fully coincides.

5.4 EROOS Constraint Triggers

The EROOS methodology introduces events to specify functionality in the conceptual model, and uses implied, integrated, and first-class EROOS constraints as a means to control the validity of the event occurrences and the resulting model instance. During the specification of events, an analyst often has to detect and avoid constraints violations, since the analyst must take care that all model constraints remain valid when an event is executed. Constraint triggers[36] are introduced in the EROOS universe in order to specify constraint exception handling mechanisms. Constraint triggers can resolve constraint violations in an active manner, by injecting additional functionality at the moment that possible constraint violations occur.

---

[36] A part of the work presented in this chapter has been published in [148].

5.4.1 Semantics of Functionality in EROOS

A conceptual model is a mapping of facts and knowledge from the universe of discourse. At each moment in time, a specific situation in the universe of discourse is represented in a model instance, containing specific objects that have properties and interrelationships. Events that occur in the universe of discourse are represented by EROOS events, which can be creation events, mutation events, destruction events, and general events, and give rise to a transition from an existing model instance into a new model instance that will be valid from that moment on.

Model constraints, represented in EROOS by means of implied, integrated, and first-class EROOS constraints, serve as validators for the allowed model instances. A constraint restricts the set of possible instances of a model by defining rules that must be valid for each model instance at each moment in time. Whenever a set of events lead to a model instance that is in contradiction with the defined model constraints, the events will be refused and the new model is rejected. In such case, the old model instance that was valid at the moment when the set of events occurred, will be preserved. This *all or nothing* property is fundamental for the specification of functionality in EROOS. The conditions under which an event can occur, are not fully specified for each event, but must be deduced from the specification of the event, and the specification of all properties and constraints described in the model. A model transition due to a set of events will only succeed when the new model instance complies with all model constraints, or will be rejected if it fails to do so.

In addition to the '*all or nothing*' property, a second property is of utmost importance in the understanding of the basic semantics of EROOS functionality. The '*Frame Axiom*', also called '*Inertia Axiom*', states that each element in the model instance that has not explicitly been changed in the specification of an event, must remain unaltered. This enables proper reasoning about a model transition since, due to the fact that an event only changes a few model items, it would be impossible to make any statement on the expected values of the unspecified model entities.

A drawback in the specification of functionality is that the effect it has on the model instance, must be fully specified at the level of the event. Since functionality transforms a model instance into a new model instance, the functionality has to take into account all model constraints that the model instance must comply with. This often means that specification of an event is heavily dependent on the existing model constraints, having to consider them thoroughly, and formulate a number of conditional expressions in order to comply with them in every situation. Otherwise, the event will violate a model constraint and will be refused. This leads to a recurring pattern of (1) describing the standard behaviour of an event, (2) checking whether the state of the new model instance remains valid, and (3) providing an constraint exception handling mechanism that tries to resolve the constraint violation. Therefore, a model contains a lot of duplication of constraint checking and resolving specifications. In addition, the definition of a new model constraint often has a direct impact on the existing functionality that acts upon the properties involved in the constraint. As such, the specification of functionality using events has a very

centralised approach. Each event has to take into account that its specification satisfies every constraint in the model, often branching off certain cases that need to be dealt with in a specific manner.

5.4.2 The Trigger Concept

To overcome the drawbacks of a centralised functionality description, and to avoid the repetition of constraint checking and resolving specifications, constraint triggers have been introduced in the EROOS universe. In order to describe specific functionality that deals with constraint preservation, integrated, implied, and EROOS constraints can be extended with a trigger specification clause. A constraint trigger is a kind of exception handling mechanism for the constraint that specifies a general constraint solver, which can be used in order to resolve occurring constraint violations. The trigger describes a number of actions that must be performed when a violation of the constraint occur in order to try to solve the constraint violation.

5.4.2.1 Triggers and Model Validity

Constraint triggers only come into action when the newly obtained model instance does not comply with the defined model constraints. Based on the invalid model instance, constraint triggers inject functionality that tries to resolve the constraint violations, and restore the validity of the obtained model instance. As such, constraint triggers can be seen as a kind of firing rules that are only triggered when the constraint to which they are attached, becomes invalid.

The model normally refuses an event that would violate a model constraint, causing the model instance to remain in the state it was at the moment that the event has occurred. However, when an event trigger is specified, events that violate certain constraints can be tolerated in those cases where the trigger rule can resolve the constraint violation. In such cases, the event is tolerated and the trigger rule will be added to the global effect of the event in order to fulfil the specified constraints. As such, a trigger rule serves as a constraint violation solver for the constraint to which the trigger is attached. When an event occurs, there are four possible situations:

- The event does not violate any constraint. In this case, the event is allowed, and its effect is realised. The presence of an event trigger is irrelevant in this situation.
- The event violates a specific constraint that has no event trigger specified. In this case, the event is refused, and its effect is not realised. The state of the model instance will not be changed.
- The event violates a specific constraint, but the trigger for that constraint can resolve the violation. In this case, the event is allowed, and its effect is realised. Moreover, the effect of the trigger is added to the effect of the event.
- The event violates a specific constraint that has an event trigger, but the trigger cannot solve the constraint violation. In this case, the event is refused, and its effect is not realised. The state of the model instance will not be changed.

Constraint triggers allow a constraint centred description of functionality, introducing both a specific event part and a number of constraint specific trigger parts. As such, the functionality dealing with constraint preservation can be defined at the level of the constraint to which it belongs. However, it is not compulsory to specify an event trigger for a constraint. The modeller is free to define an event trigger for constraint solving, or to refrain from defining a trigger. In the last case, the event will only be accepted when no constraint violations have occurred.

Notice that constraint triggers do not question or weaken the validity of the model constraints. The fact that the model instance must comply with all constraints at each moment in time, remains an intrinsic principle. Also, the '*all or nothing*' property for functionality remains valid, meaning that functionality is only accepted when all constraints are preserved, and refused when at least one constraint is violated. Constraint triggers allow the extension of the net effect of an event, in order to preserve the validity of the constraint. The invalid model instance that is obtained before the constraint trigger fires, is only an intermediate state that is used in the calculation of the ultimate state, and is comparable with the semicolon (';') operator in Z [140]. This intermediate state will not be visible in the overall model instance transition. There exists no single moment in time on which the invalid model instance is reached.

5.4.2.2 Addition Triggers versus Adaptation Triggers

In order to solve constraint violations, different type of actions can be taken to resolve the situation. Typical actions that can solve a constraint violation, are (1) the destruction of objects that violate the constraint, (2) the creation of violation registration objects that record the violation, (3) the application of additional mutation events to adjust certain attribute values, (4) the replacement of certain participant objects in order to obtain a better fit, or (5) the refining of the event parameters. In general, constraint triggers can be classified in two categories:

- In addition to the functionality of the event that caused the violation, a constraint trigger can superadd functionality in order to preserve the constraint validity. Such triggers are called ***addition triggers*****,** since they add functionality to the original event. In order to preserve the validity of the constraint, addition triggers will extend the original functionality by, e.g., creating new objects, destroying existing objects, or changing object properties. Thus, addition triggers can only extend the original functionality by changing object properties that were not yet the subject of change by the original functionality, and cannot contradict the original functionality that was the cause of the invalid model instance. Therefore, constraint triggers therefore relax the boundaries of the frame axiom, since they add additional functionality to the event functionality before the frame axiom is applied. Whenever an addition trigger would revoke event behaviour, the trigger is considered invalid and the event will be refused.
- Instead of using the functionality description that was defined by the event, the trigger can adjust or revoke certain functionality in order to obtain a valid state of the model instance, for instance, by change certain parameters of the event. Such

triggers are called ***adaptation triggers,*** since they adapt the functionality that was originally defined by the event. In order to preserve the validity of the constraint, addition triggers will change the stated functionality by, e.g., introducing mutation events to change certain attribute values or participant objects. The effect of an adaptation trigger is comparable with a multi-level effect definition using the full capabilities of the semicolon (';') operator in Z [140]. The original functionality defines a provisional assistance model instance, whereupon the final model instance is defined through adjusting or changing certain values that are not in line with the stated model constraints.

In order to point out the important difference between addition and adaptation trigger, the modeller must explicitly state whether it is allowed to adapt the originally specified event behaviour or not. The specification of the trigger must define whether it is an addition (default) or an adaptation trigger. The EROOS methodology provides both kinds of triggers, since it is sometimes necessary to adjust the functionality that has caused a constraint violation. For instance, consider the decrease of a certain deadline value that results into the definition of the deadline on a moment in the past. This will certainly violate a number of deadline constraints in the model. A modeller could specify that in such cases, the deadline must only be decreased to the next day instead of a moment in the past. But since the functionality description of the deadline already indicated that the deadline must be on that specific moment in the past, an addition trigger could not change the deadline attribute anymore. In such case, an adaptation trigger that adjusts the deadline to the next day, would be most appropriate.

5.4.2.3 Multiple Trigger Violations

It is possible that an event violates more than one constraint at the same time. If a number of constraints have a trigger specification attached, these triggers will be activated simultaneously. The effect clauses of all triggers are added to the effect clause of the original event that violated these constraints. When the total effect of the event and all triggers of the violated constraints solve the constraint violations, so that the newly obtained model instance is compliant with all defined model constraints, the event will be accepted and the unified effect of the event and the triggers will be realised. It is impossible to ignore a specific trigger of a violated constraint, even in the case where another trigger can solve both constraint violations at the same time. In fact, the four situations for an event occurrence can be restated as follows:

- The event does not violate any constraint, in which case the event is allowed.
- The event violates certain constraints without trigger specifications, in which case the event is refused.
- The event violates certain constraints, and the union of all triggers for these violated constraints solves all constraints violations. In this case, the event is allowed, but will be extended with the total functionality defined in the triggers of the violated constraints.

- The event violates certain constraints, but the union of all violated constraints triggers does not solve all violations. In this case, the event is refused.

5.4.2.4 Trigger Chains

Constraint triggers that are activated when a constraint violation occurs, can at their turn cause a following constraint violation. This second constraint violation can at its turn trigger a second set of trigger event, that may eventually lead to a third violation, et cetera. Such trigger chain is allowed when it finally leads to a valid model instance without any contradictions between the functionality introduced by the triggers. In fact, when the model instance obtained after the evaluation of a certain trigger chain is a valid model instance, the original event plus all triggers form the total functionality that will be applied. However, when the chain of triggering (1) causes a contradiction with the original event or with previous triggers, (2) creates a recursive trigger loop, or (3) leads to a constraint violation for which no trigger has been defined, the original functionality and the functionality injected by the triggers will be refused.

5.4.2.5 Event Triggers versus Time Triggers

Constraints cannot only be violated due to an occurrence of an event. It is possible that constraints become violated in a model without any specific event occurrences. Certain constraint involving time, such as the ones having an expression in the form of '**now** ≤ upper limit', can be violated by the progress of time. When the actual time exceeds the stated upper limit, the constraint will become violated without any further event occurrence. It is impossible to refuse the *event* that caused this constraint violation, since the source of the violation is the mere progress of time. It is unnatural and intolerable to refuse the progress of time in a conceptual model. This leads to a *time freeze*, which is a temporarily freezing of the time in order to preserve the specified constraints until another event resolves the erroneous situation. One can never prevent the progress of time in the universe of discourse. Therefore, a constraint that can give rise to a time freeze, is incorrect and, thus, forbidden in the EROOS methodology.

Constraints that pose an upper limit restriction on the actual time, must always be extended with a trigger that can solve situations in which a time freeze can occur. It is not only obliged to add a so-called time trigger to these kind of constraints, but the specified trigger must be defined in such a manner that it can resolve possible constraint violations due to progress of time in all circumstances. When a time trigger is specified, it must be provable that a constraint violation by progress of time can always be resolved by the trigger, thereby preventing a time freeze. Since a time trigger defines an event that will occur when a constraint is violated due to the progress of time, it can be seen as an extension of the effect of the progress of time. Another way to view time triggers, is as a special kind of automatic event occurrence at a certain moment in time, since the time trigger will fire at the moment the constraint is going to be violated by the progress of time.

When specifying a trigger for a constraint, one can make a distinction, if needed, between a constraint violation by an event occurrence or by the progress of time. In fact, two kinds of triggers can be specified for a constraint.

- An **event trigger** is only triggered when the constraint violation is caused by an event or another trigger. It will not be applied to solve a constraint violation due to the progress of time.
- A **time trigger** is only triggered when the constraint violation is caused by the progress of time. It will not be applied to solve a constraint violation due to a direct or indirect event occurrence.

It is possible to specify a single trigger that acts both as a **time and event trigger,** and can be used for each violation of any source whatsoever. It is also possible to merely specify a time trigger, thereby refusing changes caused by an event and that violate the constraint. Notice that it is impossible to merely specify an event trigger for a constraint that can lead to a time freeze. Given the example of the upper time limit, one can specify a different reaction on a constraint violation when it is caused

- by changing the upper limit to a value less than the current time, for instance, refusing such change,
- or by the progress of time that exceeds the upper limit, for instance, extending the upper limit with a certain period or destroying the object.

When a time trigger causes a following constraint violation, this violation will be considered as an event triggered violation, since the source of the violation is not directly the progress of time, but the event that was triggered by the time trigger. Even in such cases, it must be provable that the number of consecutive violations ultimately resolves into a valid situation and does not lead to a time freeze.

5.4.2.6 Time Triggers and Object Creation

A typical analysis pattern using time triggers, is the creation of a specific object in order to objectify the occurrence of a constraint violation. Such cases occur often in the universe of discourse, in which violations must be recorded, fines must be given at certain moments, interventions must be started at a specific moment in time, et cetera. Such situations lead to an EROOS solution pattern as described in Figure 5.18. The activity, for which the duration is restricted for a certain limited period, will be given a deadline attribute indicating the expiry time for the activity. A constraint for each activity is specified, indicating that a violation object must exist when an activity exceeds its deadline. A time trigger attached to this constraint, creates a violation object at the moment the deadline is reached, expressing the fact that the validity period of the activity has expired. The constraint and trigger attached to the activity class are specified in Table 5.16. Often, an additional constraint will be attached to the violation class stating that a violation object may only exist when the validity period of the underlying activity actually has expired.

```
constraint in violation when deadline exceeded
  top class ACTIVITY
  context
    ACTIVITY having attribute Deadline
    being participant of VIOLATION
  definition
    for all activity in ACTIVITY
      not participating in VIOLATION:
        now ≤ activity→Deadline
  time trigger
    VIOLATION.create (activity)
end constraint in violation when deadline exceeded
```

Table 5.16: EROOS Constraint for a Time Trigger Creating Objects

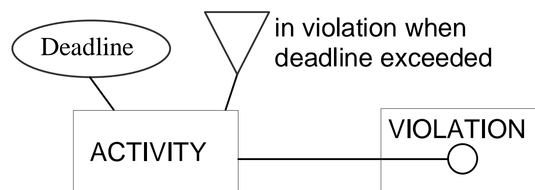


Figure 5.18: EROOS Analysis Pattern for a Time Trigger Creating Objects

5.4.3 Extending EROOS Model Concepts with Trigger Specifications

In EROOS, model constraints can be extended with a trigger specification clause in order to specify additional behaviour that can solve possible constraint violations. Not only first-class EROOS constraints can be extended with a trigger specification, but also integrated and implicit model constraints, since they also incorporate a restriction on the allowed model instances. The following sections presents how triggers are specified in the EROOS methodology.

5.4.3.1 Trigger Specification for EROOS Constraints

EROOS constraints can be extended with a trigger clause, specifying events that are added to the original event in case that the constraint is violated. Table 5.17 presents the specification of a trigger clause for a constraint, while Definition 5.12 provides its definition.

An **EROOS constraint trigger** is an event that is applied on a model instance at a certain moment in time whenever its associated EROOS constraint is not valid in the intermediate model instance. The ultimite model instance results from applying the triggered event on the invalid intermediate model instance, where all constraints are valid for the new model instance obtained by applying the constraint trigger. Whenever the newly obtained model instance do not comply with certain model constraints, the events that occurred are refused, and the original model instance that held when the event occurrence occurred, remains preserved.

Given

Model M; Event Universe EU; Model Instance MI;

Constraint CT $\in M_{ct}$; Constraint Trigger ctt $\in$ EU;

$MI_{t+1}{}^{1} = t(MI_t, E_{t+1}{}^{1})$

$MI_{t+1}{}^{i+1} = t(MI_{t+1}{}^{i}, E_{t+1}{}^{i+1})$

$ctt \in E_{t+1}{}^{i+1} \Leftrightarrow MI_{t+1}{}^{i} \notin CT_{t+1}$

$MI_{t+1} = MI_{t+1}{}^{i} \mid (\forall CT \in M_{ct} : MI_{t+1}{}^{i} \in CT_{t+1}) \wedge$

$(\forall j \mid 1 \leq j < i : \exists CT2 \in M_{ct} : MI_{t+1}{}^{j} \notin CE2_{t+1})$

Definition 5.12: EROOS Constraint with Trigger

```
<constraint script> =
"constraint" <constraint name>
  ( "top class" <TOP CLASS NAME> |
  | "top classes" <TOP CLASS NAME> ("," <TOP CLASS NAME> )* )
  "context" ( <TOP CLASS NAME> <context clause> )+
  "definition"
    ( "for all" <identifier> ( "," <identifier> )*
      "in" <TOP CLASS NAME>
      ["not participating in" <CLASS NAME> ("↑"<CLASS NAME>)*
        ( "," < CLASS NAME> ( "↑"<CLASS NAME> )* )* ] ":" )+
      <logical clause>
      <trigger specificaton>
"end constraint" <constraint name>

<trigger specificaton> =
( [ "addition" | "adaption" ] "event trigger"
    [<logical clause> "⇒"|"default ⇒"] <event expression> )*
[ [ "addition" | "adaption" ] "time trigger"
    <event expression> ]
[["addition"|"adaption"]"time &" ["default"] "event trigger"
    <event expression> ]
```

Table 5.17: Trigger Specification for EROOS Constraints

A trigger can be identified as an addition trigger or an adaptation trigger. In addition, the different types of triggers that can be specified are the following:

- A **time trigger** is triggered when the constraint violation is caused due to the progress of time. A time trigger can only be specified when no *time & event trigger* has been specified. As stated higher, the specification of a time trigger is obliged whenever the progress of time can violate a constraint, since a time freeze must at all times be avoided.
- An **event trigger** is triggered when the constraint violation is caused due to the occurrence of an event or another trigger. An event trigger can be made conditionally, so that it only triggers when invalid model state meets a certain condition. When more than one overlapping condition has been met, the functionality of the triggers will be joined, in the same manner as triggers of multiple constraints violations are joined. It is possible to specify a default trigger that will only be used when none of the conditional triggers are fulfilled.
- When a **time & event trigger** is specified, it will be used as a trigger to solve each violation of the constraint, irrespective of the source of the violation being an event or the progress of time. A *time & event trigger* can also be designated as default trigger, but only when no other default event trigger is been defined.

5.4.3.2 Trigger Specification for Integrated Model Constraints

In the same manner as triggers try to solve constraint violations for EROOS constraints, it is possible to attach triggers to integrated model constraints, in order to specify behaviour that can solve violations of such constraints. Constraints can be attached to the following integrated model constraints in the EROOS methodology:

- *Immutability for attributes, relation participants, and compounds*. Whenever an immutable attribute, participant, *part* object, or *whole* object is being changed, an immutability adaptation trigger can be specified to restore the original situation. This trigger seems to be identically the same as the default behaviour when an immutability constraint is violated, since the events are refused and the model instance remains unchanged. However, such adaptation trigger can be very useful to preserve the effect of other events that occurred in conjunction with the forbidden mutation event. In contrast with the default behaviour that refuses all occurred events, the trigger only revokes the attribute change and allows all other event functionality.
- *Uniqueness for attributes*. Whenever an object is created using a non-unique value for its attribute, the attribute uniqueness trigger can try to resolve the situation by selecting a (eventually default) value that is suitable for the newly created object.
- *Lower and/or upper bound restrictions for attributes*. Whenever an attribute value is changed into a new value that lays outside the range of the allowed values, the attribute lower and/or upper bound trigger tries to resolve the situation by selecting a suitable value that lies inside the allowed range.

- *Connectivity and multiplicity restriction for relations and compounds.* Whenever a relation or compound violates its connectivity and multiplicity restrictions, the connectivity or multiplicity trigger tries to resolve the situation by destroying objects or changing object links.
- *Participation restriction for relations and compounds.* Whenever a relation or compound violates its participation restrictions, the participation trigger tries to resolve the situation by destroying objects or changing object links.

Triggers attached to integrated model constraints are usually event triggers. However, it is possible to specify a time trigger for an integrated constraint, for instance, in case of a lower bound restriction based on the current time. Triggers for integrated model constraints are specified as part of the model entity script, and refer to the integrated constraints for which they are applicable. The specification of integrated constraint triggers is presented in Table 5.18.

```
<EROOS model entity> =
"class" <CLASS NAME>
  <model entity definition>
   ( <trigger type> ":" <event expression> )*
"end class" <CLASS NAME>

<trigger type> =
[ <reference> ] "immutability" | "uniqueness" | "bounds" |
"lower bound" | "upper bound" | "connectivity" |
"multiplicity" | "participation" | "permanent binding" |
"existential dependency" | "mutual dependency" ]
[ "addition" | "adaption" ]
[ "event" | "time" | "time & event" ] "trigger"
```

Table 5.18: Trigger Specification for Integrated and Implicit Constraints

5.4.3.3 Trigger Specification for Implicit Model Constraints

In addition to EROOS constraints and integrated model constraints, triggers can also be attached to implicit model constraints to solve potential violations. Such triggers are always event triggers. They are specified as part of the model entity script, as presented in Table 5.18, and refer to the implicit constraint on which they apply.

- *Permanent binding for attributes.* Whenever an object is created without defining a specific value for one of its attributes, the attribute permanent binding trigger tries to resolve the situation by selecting a (eventually default) value that is suitable for the newly created object. In the same manner, an archive attribute can be set when an object is destroyed without providing a specific value for it.

- *Existential dependency for relations*. Whenever an object is created without determining a specific participant, the existential dependency trigger tries to resolve the situation by selecting a suitable object for the missing participant.
- *Mutual dependency for compounds*. Whenever a *whole* is created without determining a specific *part*, or vice versa, the mutual dependency trigger tries to resolve the situation by selecting a suitable *part*, respectively *whole*.

5.4.4 Techniques for Describing the Overall Model Behaviour

Although an EROOS model fulfils the principle of uniqueness for the structural part of the model, the behavioural part can be described in several manners. There exist a variety in the granularity of the model behaviour descriptions. The model behaviour can be specified using large-scale events as presented in Figure 5.19.a, which create a rather centralised effect description, or using rather small-scale events as presented in Figure 5.19.b, which create a modular fine-grained effect description. The introduction of constraint triggers enables a new technique for describing the model behaviour as presented in Figure 5.19.c, consisting of a distribution of the overall effect of an event into a basic description of the event that is mandatory, and a number of optional trigger specifications attached to model constraints, and that fire whenever the event violates the constraints.

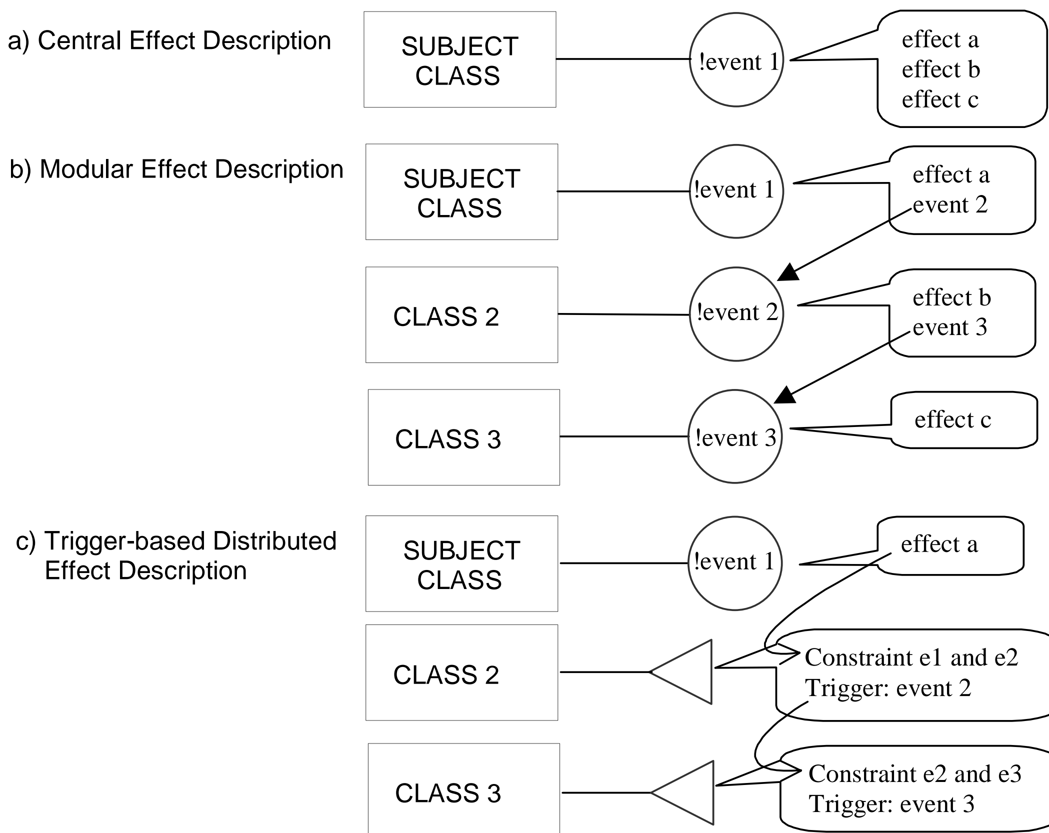


Figure 5.19: Techniques for Describing the Overall Model Behaviour

5.4.4.1 Central Effect Descriptions

The technique of central effect descriptions consists of the introduction of a single event description for each event that can occur in the universe of discourse. This event description will fully describe the overall effect of the event on the model instance. The advantage of a central effect description technique is that the impact of an event onto the model instance is clearly and completely visible at a central place. By a mere examination of the effect description of the event, one can obtain fully insight in the changes that occur in a model instance when the event is executed, and the impact it will have on the existing situation.

```
class BORROWING
  context
    having participant (SELECTION
      having participants (
        POSSESSION being participant of (RESERVED ITEM)
          having participant (COPY
            having participant (BOOK
             being participant of (PRESENCE
               being participant of (RESERVATION
                 being participant of (RESERVED ITEM))))),
        REGISTRATION being participant of (RESERVATION)
          having participant (LIBRARY)))
  general event
    return book
      effect
        self.return
        let possession = self↓SELECTION↓POSSESSION
        let library = self↓SELECTION↓REGISTRATION↓LIBRARY
        let reservations= library↑REGISTRATION↑RESERVATION
           ∩ possession↓COPY↓BOOK↑PRESENCE↑RESERVATION
        if reservations ≠ empty set
         then
           let oldest = random one of {r in reservations |
             not exists r2 in reservations:
             r2→Creation Timestamp < r→Creation Timestamp}
        RESERVED ITEM.create(oldest, possession)
        oldest.destroy
end class BORROWING
```

Table 5.19: Central Effect Description of *return book*

As an example, consider the returning of a borrowed book for the example of the library system that was presented in Section 5.1.6 on page 149. Suppose that the library has set up a reservation system for its clients. When a book is returned, the event description must provide a full specification of all tasks that must be performed at the moment of returning the book. For instance, if there exists a reservation for that book, the book copy must be labelled as reserved. The central description of the return event can be found in Table 5.19.

A drawback of central effect descriptions is that it is very complicated to develop them, since one must have a complete view on the model, including all its constraints. By specifying the effect description of the event, one must consider the consequences of each slightest change in the model, since a small change can violate a number of model constraints. The technique of central effect descriptions forces the modeller to provide a full description of the impact of the event on the model, describing the complete transition of the old into the new model instance, while preserving the validity of all implied, integrated, and EROOS constraints.

A second drawback is that commonalities in the effect descriptions of events lead to duplication inside the model. When a number of events have identical parts within their effect descriptions, this functionality must be duplicated for each event. Such duplication creates overhead (1) in the understanding of the events, forcing the model reader to detect such commonalities on its own, and (2) in changing or correcting errors in the model descriptions, since the errors will multiply and must be adjusted inside each duplicated description.

5.4.4.2 Modular Effect Descriptions

The technique of modular effect descriptions resembles the way of specifying and decomposing methods in object-oriented programming. Each event that can occur in the universe of discourse, leads to the introduction of a representative event in the conceptual model. Instead of having a single description for the specification of the event, the specification is decomposed into additional events, creating a modular effect description for the original event. These additional events can consist of fully contained effect descriptions of their own, or can again be decomposed into a number of supportive events.

We illustrate modular effect descriptions using the example of the book reservation for the library system, as described in the previous section. The specification of the return event, which uses an auxiliary event for the reservation creation, can be found in Table 5.19. Notice that the event ‘check reservations’ can be reused, e.g., when the library acquires a new printed copy of a book.

The advantage of the modular effect description technique is that common functionality for events can be factored out into a separate event. When a good decomposition of an event description is made, split up in logical parts with suitable names, the complexity of effect descriptions can be largely reduced. Complex event descriptions can be decomposed into a number of more simple events that are easier

to describe and understand, while these event descriptions can at their turn be reused in order to compose more complex events.

```
class BORROWING
  context
    having participant (SELECTION
      having participant (POSSESSION
          having event check reservations))
  general event
    return book
      effect
        self.return
        let possession = self↓SELECTION↓POSSESSION
        possession.check reservations
end class BORROWING

class POSSESSION
  context
    being participant of (RESERVED ITEM)
    having participants (
      COPY having participant (BOOK
          being participant of (PRESENCE
            being participant of (RESERVATION
              being participant of (RESERVED ITEM)))),
      LIBRARY being participant of (REGISTRATION
        being participant of (RESERVATION)))
  general event
    check reservations
      effect
        let reservations=
          self↓LIBRARY↑REGISTRATION↑RESERVATION
          ∩ self↓COPY↓BOOK↑PRESENCE↑RESERVATION
        if reservations ≠ empty set
         then
           let oldest = random one of {r in reservations |
             not exists r2 in reservations:
             r2→Creation Timestamp < r→Creation Timestamp}
        RESERVED ITEM.create(oldest, self)
        oldest.destroy
end class POSSESSION
```

Table 5.20: Modular Effect Description of *return book*

A second advantage is that change and error correction is facilitated, since analogous effect descriptions only occur at a single place in the model. This reduces the amount of corrections that must be made to a model, since the adjustments must only be made at a single point in the model.

A drawback is that the impact of an event on the model instance is not easily assessed, since it is not contained in an all-embracing description. A model reader must compose the overall effect of an event on her or his own, by following an event trace from the top event to a number of auxiliary events. In order to comprehend the total impact of the top event onto a model instance, all auxiliary events must be comprehended. As already stated higher, this process can be facilitated by choosing appropriate names for the auxiliary events.

A second and more important drawback is that, as in the case of central effect descriptions, it is quite complicated to develop modular effect descriptions due to the presence of model constraints that have to be preserved. On the one hand, the modeller must provide a full description of the impact of the event on the model through the specification of a number of auxiliary events. On the other hand, one must take care that the obtained new model instance complies with all implied, integrated, and EROOS constraints that are present in the model. As argued in the previous section, the modeller must have a complete view on the whole model in order to preserve the validity for all model constraints.

5.4.4.3 Distributed Effect Descriptions using Triggers

The introduction of constraint triggers in the EROOS methodology allows a modeller to include triggered functionality in the overall specification of events. This results in a distributed effect description for events, in which the basic event description can be augmented with small pieces of functionality that are specified in constraint triggers. These trigger specifications will be added when needed, according to the constraints that are violated by the basic event behaviour. Instead of having a full effect description in the event, at a single point as for central descriptions, or through the composition of multiple events as for modular descriptions, only the basic model change is defined directly in the event. As such, the overall effect on the model instance is obtained by combining the basic effect of the event with the effect of all constraint triggers that are activated.

We illustrate distributed effect descriptions using the example of the book reservation for the library system, as described in the previous sections. The specification of the basic return event and the constraint trigger can be found in Table 5.19.

The main advantage of the distributed effect description technique is the simplicity to develop effect descriptions, since one must no longer take care that the event description fully complies with all implied, integrated, and EROOS constraints that are present in the model. Every constraint violation can be dealt with in its attached constraint trigger. This allows that the basic event description only contains the core

```
class BORROWING
  general event
    return book
      effect
        self.return
end class BORROWING

constraint no free copy for a reservation
  top classes RESERVATION, POSSESSION
  context
    RESERVATION being participant of (RESERVED ITEM)
      having participants (
        PRESENCE having participant (BOOK),
        REGISTRATION having participant (LIBRARY)),
    POSSESSION being participant of
      (RESERVED ITEM , BORROWING)
      having participant (COPY
        having participant (BOOK))
  definition
    for all r in RESERVATION:
    for all p in POSSESSION not participating in
        RESERVED ITEM, BORROWING:
      r↓PRESENCE↓BOOK ≠ p↓COPY↓BOOK
  addition event trigger
    let reservations = { r2 in RESERVATION :
      (r2↓REGISTRATION↓LIBRARY = r↓REGISTRATION↓LIBRARY) and
      (r2↓PRESENCE↓BOOK = r↓PRESENCE↓BOOK) }
    let older reservations = { r2 in reservations :
      (r2→Creation Timestamp < r→Creation Timestamp) }
    if older reservations = empty set
       then let rr = random one of {r2 in reservations |
             (r2→Creation Timestamp = r→Creation Timestamp) }
            RESERVED ITEM.create(rr,p)
            rr.destroy
end constraint no free copy for a reservation
```

Table 5.21: Distributed Effect Description of *return book*

changes that have to occur in the model instance. The preservation of the model constraints can be delegated to the constraint triggers, which have more knowledge about the context and are better focussed to solve the constraint violation. The

modeller no longer needs a complete view on every detail of the entire model, since the preservation of the constraint validity does not solely belong to the event.

The distributed effect description technique has the same advantages as the modular effect description technique, that is (1) offering the possibility of factoring out common functionality description parts into separate events, which allows decomposition and reuse of event descriptions, and (2) facilitating changes and error correction by reducing duplication inside event descriptions.

The drawback of distributed effect descriptions is that, even more than in the case of the modular effect description technique, the impact of an event on the model instance is not easily assessed. The totality of the change caused by an event on the overall model instance, is not directly visible at a central place, but has to be composed using the basic event description and the constraint triggers of all relevant constraints. However, this process can be facilitated by offering tool support to (1) identify constraints that can be violated by the event, and (2) stating the impact of the associated triggers relevant for this event.

5.4.5 Using Nondeterminism in Functionality Specifications

During the specification of events, an analyst often has to interpret and to circumvent the model constraints in order to preserve their validity. For example, objects and values often have to be chosen based on the compliance of their properties with certain model constraints. This leads to a recurring pattern of (1) selecting the set of all potential objects, (2) restricting this set to those objects that can satisfy the stated model constraints when they are selected, and (3) making a random selection between these potential objects using a specific EROOS selection operator, namely '**random one of**'. This pattern can be observed in the specifications that are presented in Table 5.19, Table 5.20, and Table 5.21. Such specification causes a duplication of constraint checking and resolving descriptions, since the constraints already express the required properties that the objects in the model must fulfil. Since the EROOS kernel only offers support for erratic nondeterminism, by which an arbitrary and random choice is made between all available elements, the modeller oneself must first create the set of eligible elements on which then a nondeterministic choice can be made.

The EROOS universe offers a global angelic nondeterministic operator [139][157] '**selective one of**' that does not make a random choice between all possible elements, but restrict its choice to those elements that fulfil certain boundary condition, more precisely, those objects that comply with the desired properties as stated by the model constraints. This means that only those objects are selected that can lead to a model instance in which all model constraints are fulfilled. A random selection between all remaining candidate objects is only made at the final stage, when it is clear which valid model instances can be obtained. The elements that are taken into account while making the selective angelic nondeterministic choice are the following:

- The model instance that was valid before the event occurred is the starting point for making a selective nondeterministic choice.

- The effects of the deterministic events that occur at the same time as the nondeterministic event are all taken into consideration.
- In case that a nondeterministic choice can be made that leads to a possible model instance fulfilling all model constraints, taken into account all other nondeterministic choices that have to be made simultaneously, one of the solutions will be chosen at random.
- If no nondeterministic choice can be made that leads to a possible model instance, every case in which a nondeterministic choice causes a violation of a constraint with attached trigger clause, is taken further into consideration. The functionality defined in the trigger clause is added to the basic event functionality and the specific choice that gave rise to this possible solution. Since different choices can possibly trigger different constraints, a number of possible model instances can be obtained that ultimately lead to a valid model instance. In such case, one of these valid model instances will be chosen at random. As explained in Section 5.4.2.4, it is possible that a further trigger chain is caused that ultimately leads to a valid model instance.
- In case that no nondeterministic choice can be made that leads directly or indirectly to a valid model instance, the event is refused.

```
constraint no free copy for a reservation
  top classes RESERVATION, POSSESSION
  context
    RESERVATION being participant of (RESERVED ITEM)
      having participant PRESENCE
        having participant (BOOK),
    POSSESSION being participant of
      (RESERVED ITEM , BORROWING)
      having participant (COPY
        having participant (BOOK))
  definition
    for all r in RESERVATION:
    for all p in POSSESSION not participating in
        RESERVED ITEM, BORROWING:
      r↓PRESENCE↓BOOK ≠ p↓COPY↓BOOK
  addition event trigger
    let reservation = selective one of(RESERVATION)
    RESERVED ITEM.create(reservation,
        selective one of(POSSESSION))
    reservation.destroy
end constraint no free copy for a reservation
```

Table 5.22: Nondeterministic Distributed Effect Description of *return book*

The global angelic nondeterministic operator reduces a large number of overspecification for events and triggers, for which otherwise constraint avoidance specifications must be made by explicitly calculating the set of eligible elements. We illustrate non-deterministic distributed effect descriptions using the book reservation example for the library system, as described in the previous section. The specification of the constraint trigger can be found in Table 5.22. Notice that this constraint trigger supports on the presence of certain constraints, such as (1) the fact that the registration and possession for a reserved item must contain the same library and book, and (2) the fact that only the oldest reservation for a book can be involved in a reserved item. Bekaert [13] provides an elaboration of nondeterminism in EROOS.

5.4.6 EROOS Constraint Triggers for the Library Example

Given the example of the library system that was presented in Section 2.3, and the relation hierarchy that was defined in Section 5.1.6, we can specify a constraint trigger for the class of borrowing that automatically creates a fine object when the borrowing exceeds its deadline, as presented in Table 5.23. Notice that the deadline is defined by the value of the attribute Maximum Lending Period at the moment the borrowing was created. Since this constraint can be violated due to the progress of time, the trigger is formulated as a time trigger.

```
constraint borrowing not exceeded
  top class BORROWING
  context
    BORROWING being participant of (FINE)
      having participant (ALLOWANCE
        having participant (LIBRARY
          having attribute Maximum Lending Period))
  definition
    for all b in BORROWING not participating in FINE:
      now - b→Creation Timestamp <=
      self↓ALLOWANCE↓LIBRARY
        →Maximum Lending Period@(b→Creation Timestamp)
  addition time trigger
    FINE.create(b)
end class BORROWING
```

Table 5.23: Time and Event Trigger for the Library Example

5.4.7 Contributions, Related Work, and Reflections

Our approach concerning **constraint triggers** is a novel and original contribution to conceptual modelling. The introduction of **constraint triggers** provides an elegant

description of the universe of discourse, in which a generic constraint solver can be attached to a constraint. The goal of a constraint trigger is to resolve constraint violations by injecting error handling behaviour into an event, or by firing an event due to the progress of time. This enables the specification of distributed effect descriptions for events, in which only the basic effect is specified for an event. Small additional pieces of functionality are specified in constraint triggers that are added to the basic event description according to the constraint violations caused by the event. This approach creates a separation between the description of the *normal* event handling and the *exceptional* event handling. The normal event handling is specified in the event, whereas the exceptional event handling is specified in a number of constraint triggers. Without constraint triggers, event descriptions contain a lot of duplicated constraint checking and avoidance specifications, in which possible constraint violations must be captured and resolved. Such approach leads to a lot of duplication overhead inside a model. Constraint triggers support separation of concerns, by clustering all functionality regarding the constraint handling in a single place. It can be used to introduce specific constraint related crosscutting behaviour into a model, through the extension of all events that can violate the constraint. Therefore, it can be considered as a kind of Aspect-Oriented Software Development (AOSD) [46] technique.

The EROOS constraint trigger concept is somewhat comparable with ECAA rules in active databases, as presented in Section 2.2.4. It can be seen as a kind of adaptation and extension of ECAA rules for constraints. Other analysis methods, such as OOIE [96][97], BON [156], MOSES [64], OBA [125], and SOMA [58], have incorporated ECAA rules, while implementation extensions have been proposed to integrate ECAA rules in programming languages [85][86]. The main difference in our approach is that a constraint trigger must solve the constraint violation that triggered it. Since the goal of constraint triggers is the provision of a generic specification for solving constraint violations, the execution of a constraint trigger must ultimately lead to the preservation of the constraint validity. ECAA rules are merely injecting additional functionality based on certain events that occur, or additional conditions that are valid at a certain moment in time. They are not concerned about preserving certain conditions in a model, although the action could be specified in such manner that the firing condition will become invalid.

Rumbaugh [127] proposes the technique of operation propagation, in which destruction propagation is used in order to automatically delete a number of associated objects from a model whenever an object is removed. The technique arose due to practical issues, since otherwise an explicit specification of the complete destruction event must be made, including the removal of a set of associated objects. However, it can be considered as a specific kind of constraint triggering mechanism. Especially, when the multiplicity of an association at the side of the removed object is larger than zero, there is an existential dependency from the connected object on the removed object. In this case, the destruction propagation can be seen as a trigger for the existential dependency constraint, which deletes all objects that are existentially dependent on the removed object.

An observation that can be made is that the approach of distributed effect descriptions using the constraint trigger concept conceals the impact of an event on the model instance. One has to compose the overall effect description oneself, by combining the basic event description with all trigger descriptions of those constraints that can be violated by the event. Therefore, tool support is appropriate in order to help identifying the constraints that can be violated by an event, and stating the impact of the associated triggers that are relevant for the event.

5.5 Derived Groups and Dynamic Subdivision

This section introduces the EROOS kernel analysis pattern that has identified the desirability of introducing derived groups. We present how dynamic specialisation can be simulated using constraint triggers and object slices, which perform automatic creation and destruction of derived groups. Hereafter, we present the concept of EROOS groups in order to obtain a better suitable modelling of dynamic specialisation and computable groups in the EROOS universe. Last, derived groups are applied on the running example of the library system.

5.5.1 EROOS Analysis Pattern for Dynamic Specialisation

As presented in Chapter 3, the EROOS kernel was founded on a number of key principles for conceptual modelling. Although the principle of *No Redundancy* states that every single item of information inside a model must have an added value of its own, it is often convenient to be able to extract derived information from a model. Derived information is information inside a model that can be deduced from other elements that are already contained in the model. Derived information can be modelled to a certain extent in the EROOS kernel using the concept of a query. Based on information that is contained inside a model instance, a query can calculate a result or a property for an object, or select a number of objects based on certain criteria. Derived attributes and derived specialisation hierarchies are forbidden in the EROOS kernel, but must instead be modelled using the query concept.

Due to the principle of *No Redundancy,* the EROOS methodology does not support dynamic specialisation. Dynamic specialisation is the ability of making dynamic changing 'is-a' specialisation hierarchies, in which objects can dynamically move from one specialised class to another based on certain criteria. We refer to such dynamic object sets as *groups* in order to make a distinction with classes, which are defined in Section 4.6 as static object sets incorporating the constraint of immutability. Dynamic specialisation can be distinguished in two cases:

- When the transfer of an object from one group to another is caused by an event noticeable in the universe of discourse, it should be modelled as a distinct class. These kinds of dynamic properties and object roles are captured in EROOS as first-class elements, instead of making them subgroups of a generalised class. For instance, a student cannot be modelled as a specialised class of person in EROOS, since it is a dynamically changing group. However, the fact that a

person becomes a student, can be modelled as a class on its own, incorporating a relationship between the person doing the enrolment and the educational institute accepting the person as a student. The fact that a person is a student or not, can be derived using a query for the person, evaluating whether the person is involved in any enrolments at a certain moment in time. Figure 5.20.a illustrates an EROOS kernel analysis pattern in which a dynamic property is represented as a refinement. The participant class can dynamically obtain and lose the property.

- When the transfer of the object from one group to another is based on information that is already contained in the model, it should be modelled as a query instead. These kinds of derived properties are captured in EROOS as queries, since the information is already contained in the model. When information about the grouping is needed, it can be derived using a query that returns the fact whether the object has the property or not. Figure 5.20.b illustrates an EROOS kernel analysis pattern in which a class can only be refined using specific participant objects that fulfil a certain property, e.g., based on an attribute value. The group of potential participants is (1) expressed using the query ‘has property’, and (2) enforced using the constraint ‘property true’.

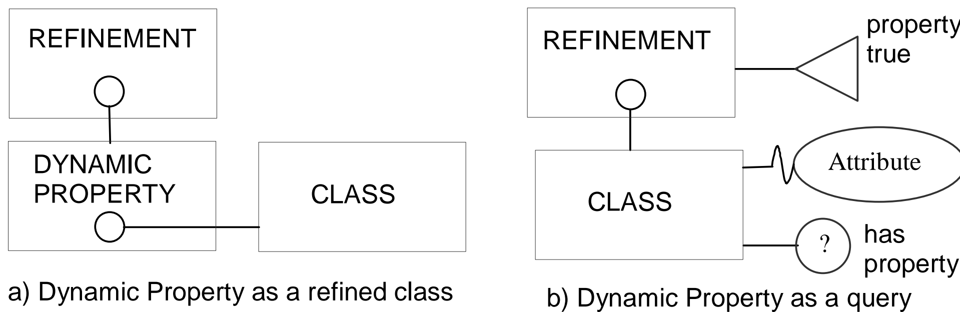


Figure 5.20: Modelling Dynamic Specialisation using EROOS Queries

Due to the introduction of constraint triggers in the EROOS universe, it is possible to realise a simulation of dynamic specialisation by using triggers for the automatic creation and destruction of derived groups. The dynamic specialisation is not performed on the core object, but on a changeable object slice that is connected to it. This object slice is necessary, since it is not possible to destroy and reconstruct the core object, e.g., due to relationship links in which the object already can be involved, and due to its *Creation Timestamp* that would be changed. Constraint triggers can be specified that automatically create the dynamic object slice when a certain condition is valid, and change this dynamic slice when the condition no longer is valid. This specific analysis pattern to realise dynamic specialisation in the EROOS universe, is presented in Figure 5.21 and Table 5.24. Although it is possible to specify dynamic specialisation using derived groups, it is not an appropriate specification since, it duplicates the derivation condition into a positive and negative expression, and introduces additional explicit object creation and destruction triggers. Therefore, EROOS offers an additional concept for modelling dynamic specialisation.

```
constraint property true
  top class SLICE HAVING PROPERTY
  context
    SLICE HAVING PROPERTY
      specialisation of (DYNAMIC PROPERTY SLICE
       having specialisation (SLICE NOT HAVING PROPERTY)
       having compound-whole (CLASS))
  definition
    for all p in SLICE HAVING PROPERTY :
       p➤CLASS.has property
  addition event trigger
     p.destroy
     SLICE NOT HAVING PROPERTY.create(p➤CLASS)
end constraint property true
constraint property false
  top class SLICE NOT HAVING PROPERTY
  context
    SLICE NOT HAVING PROPERTY
      specialisation of (DYNAMIC PROPERTY SLICE
        having specialisation (SLICE HAVING PROPERTY)
        having compound-whole (CLASS))
  definition
    for all n in SLICE NOT HAVING PROPERTY :
       not(n➤CLASS.has property)
  addition event trigger
     n.destroy
     SLICE HAVING PROPERTY.create (n➤CLASS)
end constraint property false
```

Table 5.24: EROOS Constraints for Simulating Dynamic Specialisation

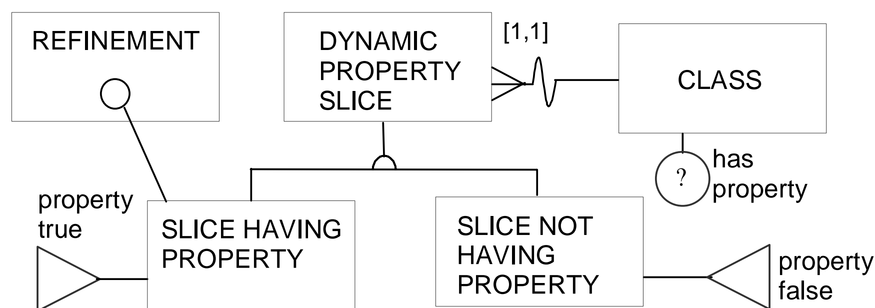


Figure 5.21: EROOS Analysis Pattern for Dynamic Specialisation

5.5.2 EROOS Groups and Dynamic Specialisation

The concept of an EROOS group offers the possibility to model dynamic specialisation in a more convenient manner. Dynamic subdivisions of a class can be defined, which is automatically composed using a condition that is used as a selection criterion for the group. As such, objects remain statically attached to their native class, while they can dynamically (1) become part of a group, or (2) leave the group. The conditional rule that forms the group, serves as a filter on the class for selecting objects that must belong to the group. An EROOS group script is a dynamic specialisation of a single class. However, a group can also be defined as a subgroup of an already existing group, which enables to model a further dynamic subdivision within an existing subdivision.

The syntax of an EROOS group script is given in Table 5.25. A group script is defined for a base class or group, and specifies a rule that dynamically and continuously selects objects from the base class or group in order to form the new group. As presented in Figure 5.22, an EROOS group is graphically represented in the form of a double-bordered rectangle that is connected to its base class or group, using an arrow from the base class or group to the new group. The definition of a group can be found in Definition 5.13.

```
<group script> =
"group" <GROUP NAME>
  "base" [ <CLASS NAME> | <GROUP NAME> ]
  "context" <context clause>
  "definition"
    "composed by"
      <logical clause>
   ( <trigger type> ":" <event expression> )*
"end group" <GROUP NAME>
```

Table 5.25: EROOS Group Script

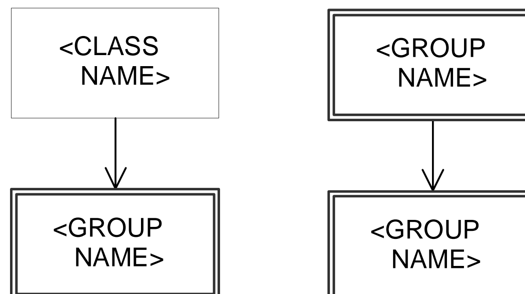


Figure 5.22: Graphical Representation of an EROOS Group

```
A group is a model entity defining, at each moment in time,
an object population set. This set is a subset of the
population set of the base class or the base group from which
the group has been derived. The subset is dynamically,
automatically and continuously created by selecting all
objects of the base class or the base group that fulfil the
selection rule defined by the group.
```

Given

Model M; Object Universe OU; Class C $\in M_{cl}$; Group G $\in M_g$

Direct Group DG $\in M_g$, Indirect Group IG $\in M_g$

DG: TIME $\rightarrow \mathcal{P}(OU) \mid \forall t \in$ TIME: $DG_t \subseteq C_t$

IG: TIME $\rightarrow \mathcal{P}(OU) \mid \forall t \in$ TIME: $IG_t \subseteq G_t$

Definition 5.13: EROOS Group

The following considerations must be made regarding EROOS groups:

- Although a group can be defined as a subgroup of an already existing group, it is forbidden to define mutually dependent groups. Two groups cannot be direct or indirect dependent on each other.
- The archive that is associated to a group complies with the archive of the class of which the group is directly or indirectly a subset. The group archive does not contain objects that ceased to fulfil the group composition rule, but only contains dead objects that fulfil the group composition rule. However, since historical information concerning past model instances remains available in EROOS, and can be obtained using the time indication '@t', it is possible to define a query that select all objects that once fulfilled the group condition but stopped doing so.
- As presented in Figure 5.23, a group can be used as a participant in a relation definition. In fact, it is a representation of an underlying structure that was shown in Figure 5.20.b, in which the relation is directed to the base class of the group, having an additional constraint for the refined class that expresses the same condition as the one used to form the group. However, notice that objects are not statically connected to the group, but can dynamically become part of the group or can cease to be part of it. Therefore, when a model contains a relation to a group participant, the modeller must consider how the model should react when a participant object ceases to belong to the participant group. In fact, the same consideration has to be made in the underlying model that is presented in Figure 5.20.b, since this model also raises a validity issue when the property of the participating object stops to be valid. Although not strictly necessary, it is advisable to define an existential dependency trigger for such relation in order to properly deal with property changes.

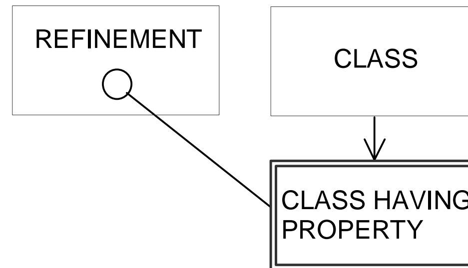


Figure 5.23: Using EROOS Groups for Dynamic Specialisation

5.5.3 EROOS Groups for the Library Example

Given the example of the library system that was presented in Section 2.3, and the relation hierarchy that was defined in Section 5.1.6, we can specify a dynamic group of overdue borrowings. An overdue borrowing can automatically be selected from the class of borrowings, based on the Creation Timestamp of the borrowing and the Maximum Lending Period of a library. Overdue borrowings must be connected to a fine, and a fine must also be connected to an overdue borrowing. Therefore a mutual dependency can be defined between an overdue borrowing and a fine, as presented in Figure 5.24. Since objects can automatically become part of the overdue borrowing group due to the progress of time, the time trigger presented in Table 5.26 must be formulated in order to preventing a time freeze.

```
group OVERDUE BORROWING
  base BORROWING
  context
      having participant (SELECTION
       having participant (LIBRARY
         having attribute Maximum Lending Period))
  definition
    composed by
      (if self in BORROWING† then self→Destruction Timestamp
       else now) - self→Creation Timestamp
      > self↓SELECTION↓LIBRARY
        →Maximum Lending Period@(b→Creation Timestamp)
    involved as compound-whole
      immutable total OVERDUE BORROWING
    having compound-part min 1 max 1 immutable FINE
  mutual dependency addition time trigger
    FINE.create(self)
end group OVERDUE BORROWING
```

Table 5.26: Time and Event Trigger for the Library Example

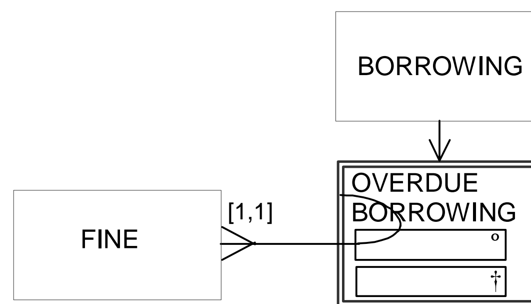


Figure 5.24: EROOS Groups for the Library System

5.5.4 Contributions, Related Work, and Reflections

Our approach concerning **derived groups** is a novel and original contribution to conceptual modelling. Derived groups capture more constraints directly in the model structure. Instead of specifying an explicit EROOS constraint for a relation, the relation can be directed to a specific group, which identifies the set of objects that can be a valid participant in the relation. Hereby, the EROOS constraint expression is transformed into a group composition rule. In addition, derived groups give a deeper insight in the potentials of a class, since it explicitly highlights in the model that the fact of belonging to a certain group, enables the participation in a number of relations.

The EROOS group concept is somewhat comparable with dynamic inheritance and role modelling techniques. Dynamic inheritance refers to the ability to add, delete, or change parents from objects (or classes) at run-time. Dynamic inheritance has been introduced in different forms:

- Nierstrasz [104] defines the concept of dynamic inheritance as a mechanism that permit objects to alter their behaviour in the course of normal interactions between objects. He distinguishes three forms of dynamic inheritance: (1) *part inheritance*, in which an object explicitly changes its behaviour by accepting new parts from other objects, (2) *scope inheritance*, in which the changes occur indirectly through changes inherited from the environment, and (3) *dynamic subclassing*, in which an object moves from one class to another at run-time. Certain programming languages, such as Smalltalk [56] and CLOS [80], provide *scope inheritance* in some form or another. Other languages, such as Self [147], provide dynamic *part inheritance* in the form of delegation. Mohindra [101] uses a *dynamic subclassing* mechanism to dynamically create new classes at run-time. These approaches differ from the dynamic specialisation that we propose. Instead of determining at run-time which inheritance structures we have to adapt, we want to statically determine the groups to which an object can belong at run-time.
- Another approach related to dynamic inheritance is dynamic role modelling [121][166][165][145]. Role modelling allows an object to dynamically change its role in a model, thereby obtaining new functionality that is assigned to the role, and removing functionality that is no longer needed. We take a different

approach to modelling roles. Since roles are optional properties that an object can obtain, we use the base object as a participant in a role relationship. For instance, instead of modelling a student role for a person, we define a class of enrolments, refined with a person and an institute as its participants. Roles can be compared to EROOS groups. Based on the enrolments relation, we could define a student's group for the class of persons, thereby selecting only those persons that are participating in an active enrolment. Although this seems to correspond to the student role of a person, the main difference is that EROOS groups are automatically composed based on the grouping rule, whereas roles must be given explicitly to and taken from an object. The allocation of roles is similar to the creation and destruction of the enrolment object, rather than to the formation of the EROOS group.

- KISS [84] introduces role specialisations that can be acquired by an object based on associations in which the object participates. As such, a person that participates in a study association will automatically be part of the subgroup defined by the student role, while at the same time the person can also be part of other subgroup according to its other participations. However, dynamic specialisation in KISS is restricted to the participation property that can be derived from an existing association, and cannot be defined using a general composition rule as in EROOS.

An observation that can be made is that derived groups offer an alternative manner for modelling constraints in an EROOS model. Modellers can choose between, e.g., using a constraint restricting a participant, and using a derived group as a participant. When constraints are transformed into group composition rules, the number of structural model elements, which is already high in comparison with other analysis methods, will even further increase. Therefore, the right balance must be found between structural elements, such as groups, and EROOS constraint specifications.

5.6 Evaluation of the EROOS Universe

Based on the core concepts offered by the EROOS kernel, we have presented a number of advanced concepts for performing conceptual modelling. These advanced concepts offer methodological support for recurring EROOS kernel analysis patterns, and provides a more practical methodology to the analyst. The EROOS universe offers a methodological approach of integrating model constraints in the modelling concepts and the model structure. Functionality is modelled in a distributed manner, by separating the basic specification of the event from exceptional constraint violation handling. Explicit attention is paid to the lifetime of an object without deciding when the object must be removed from the model, since historical information remains available in the model.

Since the concepts of the EROOS universe were defined from the viewpoint of the analyst, they sometimes contradict to a certain extent with some of the key principles for conceptual modelling as defined in Chapter 3. We again evaluate each key

principle for conceptual modelling and describe the impact of the proposed extensions on achieving the principles. Since there is no direct impact of the proposed additional concepts on the *Principles of Unambiguity, Completeness, Minimalism, Preciseness, No History*, and *Abstraction*, we refer to their evaluation in Section 4.10. Therefore, we only discuss the impact of the EROOS universe on the *Principles of Uniqueness, No Redundancy,* and *Model-Implied Constraints*.

5.6.1 Achieving Uniqueness

The *Principle of Uniqueness* is maintained in the EROOS universe to a reasonably large extent. The impact of the proposed extensions is as follows:

- Class archives offer an alternative way of modelling information that has a restricted lifetime. By introducing class archives, we explicitly force the modeller to reason about the duration of the validity of the modelled elements. Since class archives particularly reduce the number of classes contained in the model, as well as the number of constraints present to express dependencies, we consider it as an adequate concept for conceptual modelling. In order to comply with the *Principle of Uniqueness*, class archives are introduced for every EROOS class.
- Mutability has a certain impact on achieving the *Principle of Uniqueness*. However, it is a concept that is well established in computer science, since variables are a common concept in programming. In an object-oriented approach, objects are considered to be entities that encapsulate an amount of changeable information. Although mutability is a natural concept for a modeller, it raises the question of how certain changes must be represented in a model, e.g., either as a distinct class or as an attribute mutation. This can often not be decided in an unambiguous manner. For example, consider the balance of a bank account. An analyst could model deposits and withdrawals as mutations of the balance attribute, or as first-class transaction objects. A modeller often base decisions on personal preferences or on a personal viewpoint of the universe of discourse.

  Mutability in EROOS is not concerned about optimisation of information, nor on deciding which information is needed for future retrieval. Since EROOS enables the retrieval of old information from past model instances, all information that once was present in a model can be accessed. Mutability in EROOS offers a dense view on a model, by hiding objects that represent information changes. By introducing mutability, the EROOS universe guides the modeller to use mutations when (1) no additional information is needed concerning the actual update, and (2) no specific constraints are imposed on the update. The EROOS universe impose the modeller to use object creation when (1) additional information is needed, such as the person who performed the update, or the actual moment on which the update occurred, or (2) specific constraints must be imposed on the update, such as an explicit authority for performing the update, or a strict increase in attribute values.
- Compounds introduce a limited amount of variability. Whereas the EROOS kernel forces the analyst to model mutually dependent objects as a single object, the EROOS universe offers the ability to separate them into two distinct objects.

When the usage of compounds is restricted to the modelling of mutually dependent whole-part structures, it can clearly be distinguished from the approach in which the objects are merged into one. In fact, a mutual dependency is closer related to the information in the universe of discourse, since the *whole* object can be clearly distinguished from its first *part*. When compounds are used to model object slicing, splitting an object into several parts that are mutually dependent on each other, it is less obvious to provide clear guidelines for obtaining a unique model. Individual object-to-object mutual dependencies should be avoided as much as possible, and only used in cases when the lifecycle of the one object is different from the other object, e.g., an overall membership that is mutually dependent on yearly subscriptions.

- Constraint triggers offer an alternative specification of reactive behaviour for constraint violations. As such, specifications in which all conditions regarding a constraint are explicitly checked in order to preserve its validity, can be avoided. Constraint triggers should be given preference over alternatives that duplicate constraint-checking specifications. Since EROOS propagates the usage of constraint triggers over explicit constraint checking specifications, their introduction does not have an impact on the *Principle of Uniqueness*
- Derived groups offer an alternative manner for modelling constraints in an EROOS model. Since relations can use derived groups as a participant, a constraint restricting the participant in a certain manner can be transformed into a composition rule for the participating group. One could argue that an approach using derived groups is even preferable to constraints, since the constraint is implicitly captured in the model structure.[37] In addition, derived groups give a deeper insight in the potentials of a class, since it explicitly highlights that the fact of belonging to a certain group, enables the participation in a number of relations. Due to these benefits, we relax the *Principle of Uniqueness* to a certain extent, and offer the modeller the choice between using constraints and derived groups. The modeller must consider whether it is beneficial to highlight the derived group in an explicit manner or not.

5.6.2 Achieving No Redundancy

The *Principle of No Redundancy* is maintained in the EROOS universe to a large extent. Only compounds and derived groups have a certain impact on it.

- Compounds offer the ability of modelling mutually dependent objects. As stated in the evaluation of the *Principle of No Redundancy* for the EROOS there is a danger of duplication using mutual dependency. In the previous section, we already argued that individual object-to-object mutual dependencies should be restricted to the cases where the lifetimes of the two objects differ. Using this methodological rule, it is impossible to model duplicate objects, since a different lifetime reflects a difference in knowledge captured inside the model.

---

[37] Actually, the constraint is not really implicitly captured, since it is moved to the group, and acts as a composition rule for that group.

- Derived groups offer a manner to model derived information. Therefore, the usage of derived groups in a model violates the principle of *No Redundancy*. Due to its benefits for the modeller, we relax the *Principle of Uniqueness* to a certain extent and offer the modeller the choice whether to use derived groups or not.

5.6.3 Achieving Model-Implied Constraints

The principle of *Model-Implied Constraints* is enforced in the EROOS universe through the introduction of class archives, compounds, and derived groups.

- Class archives offer the possibility of implying archive related constraints directly by the relation definition. Existential dependency hierarchies can contain living as well as dead objects. As such, existential dependency can be used to model that a certain situation must have occurred before a specific can be created, e.g., a borrowing must have been returned before a fine can be calculated. In addition, constraints concerning lifetime dependencies do no longer have to be formulated explicitly, but can be expressed implicitly in the model structure using class archives as participants. This enlarges the set of constraints that are implied by the model structure.
- EROOS compounds offer the ability to express mutual dependency directly in the model structure. The EROOS kernel only supports the expression of unidirectional existential dependency, so that mutually dependent objects must be merged into a single fused object. The EROOS universe enables to capture mutual dependency constraint in the model structure.
- Derived groups enable to transform a large part of the explicit model constraints into constraints implied by the model structure. Instead of having to specify an explicit EROOS model constraint for a refined class, the relation can directly be targeted to the group of participant objects that fulfil the necessary conditions for participating in the relation. The derived group must of course be defined, thereby transforming the constraint into a group composition rule. Nevertheless, the dependency of the refined class on a specific subgroup of the participating class is implicitly captured in the model structure, and can be visualised in the model. As stated higher, this visualisation of the constraint gives a deeper insight in the potentials of a class, since it explicitly highlights that the fact of belonging to a certain group, enables the participation in a number of relations.

5.6.4 Final Reflections

The EROOS universe tries to obtain the right balance between adhering to the principles for conceptual modelling as defined in Chapter 3, and achieving a more practical approach for conceptual modelling, through the offering of advanced conceptual modelling concepts for recurring EROOS kernel analysis patterns. Although the EROOS universe concepts are adhering less to the stated key principles for conceptual modelling, in comparison with the EROOS kernel, the EROOS universe still addresses the conceptual modelling principles much better than UML and most analysis methods [114]. However, a number of issues could be raised.

- The concepts that are proposed by the EROOS universe could be too complex for customers, end users, and practitioners to understand. According to our experience in teaching EROOS through industrial workshops, a large number of people can learn rather quickly to comprehend the knowledge represented inside an EROOS model, and were able to judge the correctness of a representation of the universe of discourse. Higher educated people and people with a mathematical background, can easily learn to assess the EROOS methodology in an active manner. But it is likely that a number of persons involved in the software engineering process, will not easily be able to read and review EROOS conceptual models. Model-Driven Development (MDD) [50][83] techniques can help to transform an EROOS model into a form that is more easily understood by people not familiar with the EROOS methodology. It is possible to transform a model element into a number of statements in natural language that tries to formulate the precise meaning of the element in a more understandable manner. For example, the HOORA Analysis Tool (HAT) [70][71] is a modelling tool that uses model transformation techniques to generate textual documents from a UML model. The same approach could be applied to transform EROOS models.
- The EROOS methodology forces the analyst to highlight certain information explicitly as objects. For instance, integer, Boolean and multi-valued attributes must be transformed into objects, which specifically model the individual elements that are concealed behind these attribute. As an example, it is impossible to model the fact that hundred seats are available in a concert hall without modelling every single seat as an object. This leads to the presence of a large number of objects in a model instance. We claim that it is necessary to highlight and make explicit all concealed information in a conceptual model in order to alert the analyst about these facts. Although an analyst would prefer to model certain elements in a different manner, one must be aware that conceptual modelling is not a form of art, in which the analyst tries to make a personal impression of the universe of discourse. Conceptual modelling is a discipline, in which a rigorous process must be followed to capture all knowledge from the universe of discourse into a conceptual model. Note that the conceptual model does not dictate the implementation structures. Based on an EROOS conceptual model, the architectural and design phase will determine the most optimal implementation structure for the software system. In the same manner, the design phase will determine which information concerning archived objects must be stored, and which default EROOS attributes must be implemented.
- The concepts that are proposed by the EROOS universe, force the analyst to integrate statechart and activity diagrams into the class diagram, thereby transforming states into first-class objects. One could argue that statechart diagrams can be useful to model the possible states of an element. We claim that a single model approach for conceptual modelling (1) improves the consistency of the model, avoiding inconsistencies between information in the class model and the statechart model, and (2) highlights concealed information in a conceptual model by reifying object states into explicit state objects.

Chapter 6

Conclusions

In this chapter, we summarise the main contributions of this dissertation in the area of object-oriented conceptual modelling, and indicate possible directions for further research.

6.1 Summary and Contributions

In this dissertation, a constraint-centric approach towards object-oriented conceptual modelling is proposed. This is achieved by the usage of high-level constraint specifications as the core model structure for conceptual modelling. Our approach has converged into the EROOS methodology for conceptual modelling, of which two versions are proposed. A core version, the EROOS kernel, uses a constructional modelling approach in which information can only be added to a conceptual model instance. An extended version of the methodology, the EROOS universe, provides additional support for recurrent EROOS kernel analysis patterns through advanced and more practical concepts using the core version as the underlying base.

The contributions of this dissertation can be situated on three levels: (1) advanced methodological concepts for achieving the key principles for conceptual modelling, (2) the definition of new structural concepts to express model constraints implicitly in the model structure, and (3) the introduction of constraints with supporting resolution mechanisms as a first-class model concept.

6.1.1 Advanced Methodological Concepts

Concerning methodological concepts for conceptual modelling, we have made following contributions:

- *Taxonomy for model constraint formalisms in object-oriented analysis.* We have developed a taxonomy for model constraints in object-oriented analysis, based on an evaluation and comparison of model constraint specification formalisms and notations. We defined 5 types of constraint specification types, and have argued the advantages and disadvantages of each approach. We have shown that formulating constraints as informal text or as operational restrictions is not a suitable approach for conceptual modelling. We propose to (1) enrich the conceptual model structure using existential dependency, thereby implying constraints by the model structure, (2) integrate constraints in specific model entities if they are closely related to each other, and (3) specify constraints that can spread out over several model entities as a first-class model concept.
- *Principles for conceptual modelling.* We have defined the key principles for conceptual modelling that are of utmost importance during analysis for making suitable conceptual models. We have argued why these principles are important for conceptual modelling, and used them as evaluation and validation criteria for our own work. Although a number of principles and quality criteria for modelling can be found in literature, we claim to have a more elaborate set of principles, and a more precise definition of the principles. Based on our taxonomy for model constraint formalisms in object-oriented analysis, we have proposed the Principle of Model-Implied Constraint as a key principle for conceptual modelling.
- *Constructional conceptual model approach.* In order to comply with the principle of uniqueness, we have developed a constructional model approach, in which model instances can only grow and information can only be added to a model. Our approach allows modellers to focus on *which* information from the universe of discourse to model, instead of *how* to model the information when it could become outdated. A modeller does no longer have to decide about whether the information must be kept inside a model or can be overridden, since the set of knowledge and facts inside an EROOS model can only be enlarged.
- *Availability of historical information.* We have defined the concept of a class archive to express the fact that some items do not longer exist, or that some information has ceased to be valid. Although objects can be destroyed in an EROOS model, they do not vanish from the conceptual model. They still can be addressed to gather historical information regarding former attribute values and relation links. So the destruction of an object only reflects the fact that information represented by the object has ceased to exist in the universe of discourse. Issues regarding the need for an object in order to obtain certain information, or for performing certain tasks, should not be considered during conceptual modelling. Our approach allows modellers to focus on which information to model, instead of how to preserve the information. Since a

conceptual model does not only need to express the facts that occurred in the universe of discourse, but also when they occurred, we have provided a default creation and destruction timestamp for each object. The presence of these default attributes for all objects of every class enables the modeller to reason about the moments at which an object has come into existence and has ceased to exist. Since a modeller often has to reason about the time a certain event occurred, and about the life span of objects, the EROOS methodology automatically offers this kind of information for all objects. In our approach, the modeller does no longer have to decide whether such attributes are needed within a model, since they are always available. The decision on whether they have to be realised in the actual software system can be deferred to the design phase.

- *Formal notation for the semantics of queries and events.* We have developed a formal specification of model events and queries in order to obtain a complete and precise description of the behaviour part of a model. As such, the conceptual model can be used for simulation, which leads to a better validation of the model by the customers, as well as for model transformation to more software focussed models at a lower abstraction level. Our work predates and is largely comparable with the Object Constraint Language (OCL).

6.1.2 Model Constraints implied by the EROOS Model Structure

Concerning the definition of new structural concepts, we proposed well-defined modelling concepts with a dedicated applicability context in order to express model constraints implicitly in the model structure. We have made following contributions:

- *The incorporation of model constraints in each methodological concept by definition.* Contrary with UML that provides a large set of loosely defined concepts that are applicable in many situations, we have defined a small set of well-defined methodological concepts that incorporate important constraints by definition. The EROOS concepts incorporate a number of model-implied constraints, such as disjunctness, immutability, finiteness, uniqueness, permanent binding, existential dependency, abstractness, and partition disjunctness. This deliberately limits their usage to specific usage contexts, and forces the analyst to use adequate concepts in each situation. Our approach guides the analyst to the most optimal conceptual model, and reifies certain concealed elements from the universe of discourse into explicit objects in the conceptual model.
- *The usage of existential dependency as the key modelling criterion for constructing the conceptual model structure*. A key contribution of the EROOS methodology is its hierarchical relational model structure. We have developed a model structure that is solely determined by existential dependency of information in the universe of discourse. Our approach leads to a hierarchical object dependency structure that gives a clear insight in which information is dependent on certain other information. It leads to a powerful model that implies a large number of model constraints directly in its model structure. Whereas UML offers the choice of using different kinds of associations, our approach always encapsulates a relation into a refined class. We propose three kinds of

relational concepts: (1) A unary relation captures a dependency on a single object, (2) a binary relation captures a dependency on two objects, which can be seen as the reification of a relation between two classes into a class of its own, and (3) a compound captures a mutual dependency between a non-empty *whole* and a number of dependent *parts*.

- *Explicit class archives.* We have defined the concept of a class archive to express the fact that some items do not longer exist, or that some information has ceased to be valid. We have extended the relation concept in order to use class archives in existential dependency relationships. Our approach results in a powerful and high-level modelling concept, in which important dependency constraints can directly be implied by the model structure. All kinds of restrictions between the lifetime of a refined object and its participant object can be specified directly in the relation definition.

6.1.3 Model Constraints as a First-Class Model Concept

Concerning the introduction of constraints as a first-class model concept, we have made following contributions:

- *Model constraints as a first-class model concept.* In addition to a large number of constraints that are directly implied by the EROOS model structure, we have developed a mechanism for specifying model constraints as a first-class model concept. Using a formal notation based on many-sorted first order logic, model constraints can be superimposed on a model in order to express rules and regulations from the universe of discourse. Our work predates and is largely comparable with the Object Constraint Language (OCL) for UML, which originated in 1995 within IBM. EROOS constraints enforce logical rules on a certain part of the conceptual model. Every event that occurs in a conceptual model, due to an occurrence in the universe of discourse, may only change the model instance in such a manner that it satisfies all constraints. The formalisation of model constraints in a conceptual model is a key necessity for a further usage of the conceptual model in the software engineering lifecycle. For instance, model transformations from a conceptual model into lower-level design models can only succeed when the conceptual model is fully formalised. In addition, we developed in EROOS a single and unique viewpoint from which a constraint must be formulated, namely the top classes in the relation hierarchy. This leads to standardised and uniform constraint specifications in a conceptual model.
- *Constraint trigger concept.* We have proposed the constraint trigger concept in order to attach a generic constraint solver to a constraint. The goal of a constraint trigger is to resolve constraint violations by injecting error handling behaviour into an event, or by firing an event due to progress of time. This enables the specification of distributed effect descriptions for events, in which only the basic effect is specified for an event. Small additional pieces of functionality are specified in constraint triggers that will be added to the basic event description according to the constraint violations caused by the event. This approach creates a separation between the description of the *normal* event handling and the

exceptional event handling, since the normal event handling is specified in the event, whereas the exceptional event handling is specified in a number of constraint triggers. Without constraint triggers, event descriptions contain a lot of duplicated constraint checking and avoidance specifications, in which possible constraint violations must be captured and resolved. Constraint triggers support separation of concerns, by clustering all functionality regarding the constraint handling in a single place. It can be used to introduce specific constraint related crosscutting behaviour into a model, through the extension of all events that can violate the constraint. Therefore, it can be considered as a kind of Aspect-Oriented Software Development (AOSD) technique.

6.1.4 Value Added for Model-Driven Development

EROOS brings added value to a Model-Driven Development (MDD) approach by the formalisation of conceptual modelling. This enables to advance the start of the MDD process towards the analysis phase, starting with a conceptual model of the universe of discourse.

Model-Driven Development (MDD) is a framework for software development, which uses a rigorous development by translation approach to construct lower-level Platform-specific Models (PSM) based on higher-level Platform-Independent Models (PIM). The goal is to separate architectural and design-oriented issues from technology and implementation-oriented decisions using a layered model transformation structure. This allows to gradually introduce more detail and platform dependency into the lower-level development models. Such approach can ultimately result in a (semi-) automatic generation of the implementation code for the software system. MDD supports on formalised models that (1) can be used as an input for a model transformer, and (2) are produced as the outcome of a transformation step.

Since most analysis methods produce models containing informal descriptions, these models are not usable in an MDD approach. Informal descriptions cannot be used as an input for model transformation, since it is extremely difficult to extract structured information from an informal model element. Models can only be used within an MDD approach when they contain their information in a formal notation that can be investigated, evaluated, and transformed into a different format. Since EROOS provides a full formalisation of all structural and behavioural elements of a conceptual model, it is highly suited as an input model notation for an MDD transformation.

The transformation of conceptual models in a (semi-) automatic manner is beneficial to the areas of conceptual modelling as well as MDD.

- Concerning conceptual modelling, it can help to capitalise the analysis results during consecutive software engineering phases. A conceptual model is not a mere description of the universe of discourse and the functional requirements, but can be uses as a profitable asset that serves as a base for the overall system development. In addition, MDD transformations can enable rapid prototyping

and produce model simulations in order to verify and validate the conceptual model. Moreover, abstract views on a conceptual model could be generated techniques to improve the communication with the clients and end users. As such, a detailed conceptual model can be translated into a suitable customer interaction model using MDD model transformation techniques.

- Concerning MDD, a formal conceptual model enables to start the MDD process from the formalised conceptual model of the universe of discourse, instead of having to start from a platform-independent software model. This very first platform-independent software model could be generated by transforming the conceptual EROOS model.

6.2 Validation

We have validated the EROOS methodology on three levels:

- In order to assess the capabilities of EROOS for conceptual modelling, we have performed a large number of case studies. In cooperation with other members of SOM research group, and a number of industrial partners, we have applied the EROOS methodology on a large number of case studies from different application domains. These case studies have been performed as research projects in cooperation with industrial partners, as Master of Science theses, often in cooperation with industry, and as student projects part of a Master's course on object-oriented analysis (OGA). The studied domains include:
  - *Workflow and administrative systems*: management information system (thesis with E2S), trouble ticketing system (thesis with LUDIT-KULeuvenNet), high school administration system (thesis with NVKSO), library system (thesis), sale by auction system (thesis), departmental database (thesis), boat rental system (thesis), car rental system (OGA), airline reservation and check-in system (OGA), and a telephone decree (OGA).
  - *Planning and scheduling systems:* electronic agenda system (thesis).
  - *Process steering and control systems*: air conditioning system (thesis with E2S and Daikin), cooking simulation and expert system (thesis with Alma), railway infrastructure control system (thesis with 'De Leuvense modeltrein-club'), elevator control system (thesis), and traffic light system (thesis).
  - *Electronic and mechanical systems:* flexible multiplexer (research project with Alcatel), Network Management System (thesis with Siemens), audio set usage specification (thesis with Philips), Internet Telephony System (thesis with Philips), radiological workstation (thesis with 'UZ Gasthuisberg'), steel factory material flow (thesis with Sidmar), PABX telephone system (thesis), and physics measurement environment (thesis).

  - *Entertainment and visualisation systems*: augmented reality man machine interface (thesis), graphical user interface capability modelling (thesis), adventure game (thesis), and 'Magic: The Gathering' card game (thesis).
  - *Software systems*: syntax oriented editor generator (thesis with E2S), software modelling case tool (thesis), programming environment for Logo (thesis), load balancing for multiprocessor systems (thesis), E-mail system (thesis), specification of EROOS in EROOS (thesis), and electronic agenda (OGA).

  The large variety of domains that have been modelled using EROOS, support our claim that the EROOS methodology is not only adequate for the description of information systems, but is actually suitable for the conceptual modelling of many domain types.

  Our findings concerning these case studies, are that (1) EROOS is suited to describe a wide variety of domain types, (2) the EROOS methodology helps to reveal hidden domain knowledge, (3) EROOS is a good vehicle to teach object-orientation, in general, and conceptual modelling, in particular, (4) it requires a rather large effort and a precise approach and attitude to construct EROOS conceptual models, (5) MDD tool support is needed to capitalise on the conceptual modelling activity, and (6) people with a M.Sc. degree can be trained rather easily to acquire active EROOS modelling skills, while people with a B.Sc. degree often only manage to acquire passive EROOS modelling skills, which means that they succeed to understand, assess and review EROOS models but have difficulties in constructing them.

- In order to assess the capabilities of EROOS in achieving the principle of uniqueness, we have compared and evaluated the use of EROOS in a Master's course on object-oriented analysis (OGA - 'objectgerichte analyse', formerly OGO- 'objectgericht ontwerp'). Our findings were that besides naming diversity, the three main causes of model differences are (1) the level of detail of the performed modelling, in which students had a different opinion on the relevance of certain facts from the universe of discourse, (2) personal knowledge of the universe of discourse, in which certain errors were introduced due to the inability to obtain a proper insight in the universe of discourse, and (3) errors that were made against the EROOS methodology, in which the students did not use the modelling concepts in a valid manner.
- We have developed tool support for the EROOS methodology concerning modelling, simulation, and transformation, consisting of
  - a modelling tool that allows a modeller to construct EROOS models, and generate script specifications and model diagrams. This EROOS tool is actually developed by Bart Swennen of the SOM research group.
  - a generator for model simulations that automatically generates a C++ or a Java application with an accompanied generic user interface for an EROOS model, in order to support rapid prototyping and early model validation. The application contains automatically generated constraint checking code,

which enforces the model constraints by checking the resulting model instance after an event occurrence, and performing a rollback whenever a constraint is violated. This generator is developed in a number of successive Master's theses.

– a transformer of EROOS models to UML models, in which the EROOS hierarchical model structure is flattened into a UML model, comprising a number of classes and plain associations. This transformer is also developed in a Master's thesis.

6.3 Directions for Future Work

The search for the perfect conceptual modelling methodology is far from over. To conclude this dissertation, we point out some possible directions for further research:

- *EROOS methodological improvements.* Concerning the EROOS methodology, some issues could be further elaborated:
  - *Support for distributed effect descriptions.* The introduction of the constraint trigger concept enables the specification of a distributed effect description, in which the basic event functionality is separated from the constraint error handling functionality. However, a drawback of such distributed effect descriptions is that the impact of an event on the model instance is not easily assessed. The analyst has to compose the overall effect description oneself, by combining the basic event description with all constraint trigger descriptions of those constraints that can be violated by the event. Therefore, tool support is appropriate in order to help identifying the constraints that can be violated by an event, and stating the impact of the associated triggers that are relevant for the event.
  - *Soft Constraints.* EROOS constraints impose restrictions on a conceptual model that must be satisfied at all times. Soft constraints are constraints that have an attached level of preference or importance. As such, not all constraints must be satisfied at all times, but the goal is to reach the highest available satisfaction level. Further research could integrate the notion of soft constraint in the EROOS methodology.
  - *Extensions to EROOS.* Bekaert [13] suggests a number of improvements that could be made to EROOS, such as (1) triggered-only events, which are events that can only occur as a constraint trigger, (2) creation and destruction time constraints, which are constraints that only must be satisfied at the moment on which an object is created or destroyed, and (3) the introduction of temporal logic and artificial intelligence techniques.
  - *An ontological mapping to the Bunge-Wand-Weber (BWW) reference model.* The semantics of an EROOS conceptual model could be mapped to the Bunge-Wand-Weber (BWW) model [21][22][158][160][159], which is a

reference model for constructs in information systems modelling. Opdahl [114] presented such mapping for the Unified Modeling Language (UML).

- *Model-Driven Development Support from EROOS to UML.* In contrast with the standardisation of UML by the Object Management Group (OMG), which forced most analysis methods to the usage of UML as their modelling notation, the Model Driven Development (MDD) approach supports the usage of models in several kinds of notations different from UML. By introducing model transformation techniques, MDD propagates to use the right model and notation at each level of abstraction. For instance, EROOS could be used as a conceptual modelling notation, whereupon the EROOS model can be transformed into a UML model that serves as a design notation. Such approach capitalises the analysis results obtained by the EROOS methodology into practical software-oriented UML models. Although we have developed a prototype transformer from EROOS to UML, a suitable MDD transformation infrastructure demands additional advanced support, namely
  - providing suitable tool support for EROOS, including multi-user facilities and versioning support,
  - providing a MOF (Meta Object Facility) description of EROOS models,
  - studying and developing transformation patterns from EROOS to UML constructs,
  - realising a transformation environment in which software engineers can easily select a number of transformation patterns to generate a suitable UML model starting from a conceptual EROOS model.
- *Model transformations to abstract customer views.* EROOS models can become very complex, which makes them difficult for customers and end users to comprehend. More abstract representations, either textual or graphical, could be made starting from the information contained in an EROOS model. In fact, it is possible to transform a model element into a number of statements in natural language, which could try to formulate the precise meaning of the EROOS model element in a more understandable manner. Such abstractions could be better suited as a customer interaction format during the model validation activities. MDD techniques could be developed to transform EROOS models into more abstract representations, and vice versa.
- *Realisation of a constraint-centric approach in UML.* This dissertation focuses on the integration of constraints in the EROOS methodology. Although UML offers the Object Constraint Language (OCL) as a constraint specification formalism, it does not offer proper notational support for expressing integrated and applied constraints. A number of techniques that have been developed in EROOS could be transposed to UML in order to make UML better suitable for conceptual modelling, e.g., by defining an EROOS profile for UML. Since UML is the de facto common language for object-oriented modelling, through its standardisation by the OMG, the impact on bringing our results into practice will become much larger when they can be applied in a UML context.

Bibliography

[1] Abrial, J.-R., The B-Book: Assigning Programs to Meanings. Cambridge University Press, 1996.

[2] Aho, A.V., and Ullman, J.D., The Theory of Parsing, Translation and Compiling, Volume I: Parsing. Prentice-Hall, 1972.

[3] Ambler, S.W., Agile Modeling: Effective Practices for eXtreme Programming and the Unified Process. Wiley, 2002.

[4] Ambler, S.W., The Object Primer: Agile Model-Driven Development with UML2.0, Third Edition. Cambridge University Press, 2004.

[5] Apt, K.R., Principles of Constraint Programming. University Press, 2003.

[6] Arnold, P., Bodoff, S., Coleman, D., Gilchrist, H., and Hayes, F., An Evaluation of Five Object-oriented Development Methods. In: Wiener, R.S., editor, Journal of Object-Oriented Programming (JOOP) Special Issue on Analysis and Design, pages 107-121, 1991.

[7] Atkinson, C., Bayer, J., Bunse, C., Kamsties, E., Laitenberger, O., Laqua, R., Muthig, D., Paech, B., Wüst, J., and Zettel, J., Component-based Product Line Engineering with UML. Addison-Wesley, 2002.

[8] Bar-David, T., Formal Methods: The Elevator, A Rigorous and Friendly Introduction to Object Modeling. In: Report on Object Analysis and Design (ROAD), 1(1):10-16, 1994.

[9] Barnes, J.G.P., Programming in Ada Plus an Overview of Ada 9X, Fourth Edition. Addison-Wesley, 1993.

[10] Bass, L., Clements, P., and Kazman, R., Software Architecture in Practice, Second Edition. Addison-Wesley, 2003.

[11] Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R.C., Mellor, S., Schwaber, K., Sutherland, J., and Thomas, D.: Manifesto for Agile Software Development. The Agile Alliance, The Lodge at Snowbird Ski Resort, Utah, USA, 2001.

[12] Beck, K., Test-Driven Development by Example. Addison-Wesley, 2003.

[13] Bekaert, P., Behavioral Semantics for EROOS Conceptual Modeling: Separation of Concerns Through Nondeterminism. Ph.D. Dissertation, K.U.Leuven, Department of Computer Science, Leuven, Belgium, 2006.

[14] Bistarelli, S., Frühwirth, T., Marte, M., and Rossi, F., Soft Concurrent Propagation and Solving in Constraint Handling Rules. In: Computational Intelligence, 20(2):287-307, 2004.

[15] Booch, G., Object Oriented Analysis and Design with Applications, Second Edition. Benjamin-Cummings, 1994.

[16] Booch, G., Object Oriented Design with Applications. Benjamin-Cummings, 1991.

[17] Booch, G., Object-Oriented Design. In: ACM SIGAda Ada Letters, 1(3):64-76, 1982.

[18] Borger, M., Baier, T., Wienberg, F., and Lamersdorf, W., Extreme Modeling. In: Succi, G., and Marchesi, M., editors, Extreme Programming Examined, Addison Wesley, pages 175-189, 2001.

[19] Born, G., Process Management to Quality Improvement: The Way to Design, Document and Re-engineer Business Systems. Wiley, 1994.

[20] Brooks Jr., F.P., No Silver Bullet: Essence and Accidents of Software Engineering. In: Computer 20(4):10-19, 1987.

[21] Bunge, M., Treatise on Basic Philosophy, Volume 3: Ontology I: The Furniture of the World. Reidel, 1977.

[22] Bunge, M., Treatise on Basic Philosophy, Volume 4: Ontology II: A World of Systems. Reidel, 1979.

[23] Carmichael, A., editor, Object Development Methods. SIGS Books, 1994.

[24] Chen, P.P., The Entity-Relationship Model: Toward a Unified View of Data. In: ACM Transactions on Database Systems, 1(1):9-36, 1976.

[25] Chidamber, S.R., and Kemerer, C.F., A Metrics Suite for Object-Oriented Design. In: IEEE Transactions on Software Engineering, 18(11):943-956, 1994.

[26] Chidamber, S.R., and Kemerer, C.F., Towards a Metrics Suite for Object-Oriented Design. In: Paepcke, A., editor, ACM Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA '91), ACM SIGPLAN Notices, 26(11):197-211, 1991.

[27] Chrissis, M.B., Konrad. M., and Shrum, S., CMMI: Guidelines for Process Integration and Product Improvement. Addison-Wesley, 2003.

[28] Coad, P., and Yourdon, E., Object-Oriented Analysis, Second Edition. Yourdon Press, 1991.

[29] Coad, P., and Yourdon, E., Object-Oriented Design. Yourdon Press, 1991.

[30] Codd, E.F., A Relational Model of Data for Large Shared Data Banks. In: Communications of the ACM, 13(6):377-387, 1970.

[31] Coleman, D., Arnold, P., Bodoff, S., Dollin, C., Gilchrist, H., Hayes, F., and Jeremaes, P., Object-Oriented Development: The FUSION Method. Prentice-Hall, 1994.

[32] Cribbs, J., Moon, S., and Roe, C., An Evaluation of Object-Oriented and Design Methodologies. SIGS Books, 1992.

[33] Dahl, O.-J., and Nygaard, K., SIMULA: An ALGOL-based Simulation Language. In: Communications of the ACM, 9(9):671-678, 1966.

[34] de Champeaux, D., America, P., Coleman, D., Duke, R., Lea, D., and Leavens, G., Formal Techniques for OO Software Development (PANEL). In: Paepcke, A., editor, ACM Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA '91), ACM SIGPLAN Notices, 26(11):166-170, 1991.

[35] de Champeaux, D., and Faure, P., A Comparative Study of Object-Oriented Analysis Methods. In: Journal of Object-Oriented Programming (JOOP), 5(1):21-33, 1992.

[36] Dechter, R., Constraint Processing. Morgan Kaufmann, 2003.

[37] Demarco, T., Structured Analysis and System Specification. Prentice-Hall, 1979.

[38] Devos, F., Patterns and Anti-Patterns in Object-Oriented Analysis. Ph.D. Dissertation, K.U.Leuven, Department of Computer Science, Leuven, Belgium, 2004.

[39] Dodani, M., Semantically Rich Object-Oriented Software Engineering Methodologies. In: Report on Object Analysis and Design (ROAD), 1(1):17-21, 1994.

[40] Downs, E., Clare, P., and Coe, I., Structured Systems Analysis and Design Method: Application and Context. Prentice Hall, 1987.

[41] D'Souza, D, Rationalizing Object Models and Design, Part 1: Models versus Designs. In: Report on Object Analysis and Design (ROAD), 1(1):22-27, 1994.

[42] Elmasri, R., Weeldreyer, J., and Hevner, A., The Category Concept: An Extension to the Entity-Relationship Model. In: International Journal on Data and Knowledge Engineering, 1(1):75-116, 1985.

[43] Embley, D.W., Kurtz, B.D., and Woodfield, S.N., Object-Oriented Systems Analysis. Yourdon Press, 1992.

[44] Fenton, N.E., and Pfleeger, S.L., Software Metrics: A Rigorous & Practical Approach, Second Edition. PWS Publishing Company, 1997.

[45] Fichman, R.G., and Kemerer, C.F., Object-Oriented and Conventional Analysis and Design Methods: Comparison and Critique. In: IEEE Computer, 25(10):22-39, 1992.

[46] Filman, R.E., Elrad, T., Clarke, S., and Aksit, M., Aspect-Oriented Software Development. Addison-Wesley, 2005.

[47] Firesmith, D.G., Object-Oriented Software Requirements Analysis and Logical Design: A Software Engineering Approach. Wiley, 1993.

[48] Fowler, M., Analysis Patterns: Reusable Object Models. Addison-Wesley, 1996.

[49] Fowler, M., Cairns, T., and Thursz, M., Observations and Measurements. In: Report on Analysis and Design (ROAD), 2(3):20-37 1995.

[50] Frankel, D.S., Model Driven Architecture: Applying MDA to Enterprise Computing. Wiley, 2003.

[51] Freuder, E.C., and Wallace, R.J., Partial Constraint Satisfaction. In: Artificial Intelligence, 58(1-3):21-70, 1992.

[52] Frühwirth, T., and Abdennadher, S., Essentials of Constraint Programming. Springer-Verlag, 2003.

[53] Furey, T.R., Garlitz, J.L., and Kelleher, M.L., Applying Information Technology to Reengineering. In: Planning Review, 21(6):22-25, 1993.

[54] Gane, C., and Sarson, T., Structured Systems Analysis: Tools and Techniques. Prentice-Hall, 1979.

[55] Gilb, T., Software Metrics. Little, Brown, and Co., 1976.

[56] Goldberg, A., and Robson, D., Smalltalk-80: The Language and its Implementation. Addison-Wesley, 1983.

[57] Gosling, J., Joy, B., Steele, G., and Bracha, G., The Java Language Specification, Third Edition. Prentice Hall, 2005.

[58] Graham, I, Migrating to Object Technology. Addison-Wesley, 1995.

[59] Harel, D., and Politi, M., Modeling Reactive Systems with Statecharts: The Statemate Approach, McGraw-Hill, 1998.

[60] Harel, D., On Visual Formalisms. In: Communications of the ACM, 31(5):514-530, 1988.

[61] Harel, D., StateCharts: A Visual Formalism for Complex Systems. In: Science of Computer Programming, 8(3): 231-274, 1987.

[62] Hayes, F., and Coleman, D., Coherent Models for Object-Oriented Analysis. In: Paepcke, A., editor, ACM Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA '91), ACM SIGPLAN Notices, 26(11):171-183, 1991.

[63] Hayes, I, editor, Specification Case Studies. Prentice-Hall, 1987.

[64] Henderson-Sellers, B., and Edwards, J.M., Book Two of Object-Oriented Knowledge: The Working Object. Prentice-Hall, 1994.

[65] Henderson-Sellers, B., Fung, M, and Yap, L.M., The Role of Business Rules and Quality in Methodologies. In: Report on Object Analysis and Design (ROAD), 2(4):10-17, 1995.

[66] Hoeydalsvik, G.M., Object Analysis and Design: Description of Methods. Wiley, 1994.

[67] Hunt, V.D., Process Mapping: How to Reengineer Your Business Processes. Wiley, 1996.

[68] Hutt, A.T.F., editor, Object-Oriented Analysis and Design: Comparison of Methods. Wiley, 1994.

[69] Hutt, A.T.F., editor, Object-Oriented Analysis and Design: Description of Methods. Wiley, 1994.

[70] Huybrechts, M., and Pauwels, G., Agile MDA. Internal ITEA-AGILE Project Report, 2005.

[71] Huybrechts, M., Rammeloo, S., and Van Baelen, S., Realizing Agility through Model Driven Architecture. In: AGILE Newsletter 2/2006, ITEA-AGILE consortium, 2006.

[72] Jackson, M.A., Principles of Program Design. Academic Press, 1975.

[73] Jackson, M.A., System Development. Prentice-Hall, 1983.

[74] Jacobson, I., Booch, G., and Rumbaugh, J., The Unified Software Development Process. Addison-Wesley, 1999.

[75] Jacobson, I., Christerson, M., Jonsson, P., and Övergaard, G., Object-Oriented Software Engineering: A Use Case Driven Approach. Addison-Wesley, 1992.

[76] Jacobson, I., Object Oriented Development in an Industrial Environment. In: Meyrowitz, N., editor, ACM Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA '87), ACM SIGPLAN Notices, 22(12):183-191, 1987.

[77] Jacobson, I., The Object Advantage: Business Process Re-engineering with Object Technology. Addison-Wesley, 1995.

[78] Johansson, H.J., McHugh, P., Pendlebury, A.J., and Wheeler, W.A., Business Process Reengineering: Break Point Strategies for Market Dominance. Wiley, 1993.

[79] Jones, C.B., Systematic Software Development using VDM. Prentice-Hall, 1986.

[80] Keene, S., Object-oriented Programming in Common Lisp: A Programmer's Guide to CLOS. Addison-Wesley, 1988.

[81] Kendall, K.E., and Kendall, J.E., Systems Analysis & Design. Prentice Hall, 1988.

[82] Khoshafian, S.N., and Copeland, G.P., Object Identity. In: Meyrowitz, N., editor, ACM Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA '86), ACM SIGPLAN Notices, 21(11):406-416, 1986.

[83] Kleppe, A., Warmer, J., and Bast, W., MDA Explained: The Model Driven Architecture: Practice and Promise. Addison-Wesley, 2003.

[84] Kristen, G.J.H.M., Object Orientation: The KISS Method, From Information Architecture to Information System. Addison-Wesley, 1994.

[85] Laffra, C., and van den Bos, J., Constraints in Concurrent Object-Oriented Environments. In: Agha, G., Hewitt, C., Wegner, P., and Yonezawa, A., Proceedings of the ECOOP-OOPSLA Workshop on Object-Based Concurrent Programming, ACM SIGPLAN OOPS Messenger, 2(2):64-67, 1991.

[86] Laffra, C., and van den Bos, J., Propagators and Concurrent Constraints. In: Agha, G., Hewitt, C., Wegner, P., and Yonezawa, A., editors, Proceedings of the ECOOP-OOPSLA Workshop on Object-Based Concurrent Programming, ACM SIGPLAN OOPS Messenger, 2(2):64-67, 1991.

[87] LaLonde, W., and Pugh, J., Smalltalk: Subclassing ≠ Subtyping ≠ Is-A. In: Journal of Object-Oriented Programming (JOOP), 3(5):57-62, 1991.

[88] Larman, C., and Basili, V.R., Iterative and Incremental Development: A Brief History. In: IEEE Computer, 36(6):47-56, 2003.

[89] Larman, C., Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process, Second Edition. Prentice-Hall, 2002.

[90] Lewi, J., Steegmans, E., and Van Baelen, S., EROOS: Entity-Relationship Object-Oriented Specifications. K.U.Leuven, Department of Computer Science, CW Report 111, Leuven, Belgium, 1990.

[91] Lewi, J., Steegmans, E., Dockx, J., Swennen, B., Van Baelen, S., and Van Riel, H., Object Oriented Software Development with EROOS: The Analysis Phase. K.U.Leuven, Department of Computer Science, CW Report 169, Leuven, Belgium, 1993.

[92] Liskov, B., Data Abstraction and Hierarchy. In: ACM SIGPLAN Notices 23(5):17-34, 1988.

[93] Loomis, M., Shah, A., and Rumbaugh, J., An Object Modeling Technique for Conceptual Design. In: Bézivin, J., Hullot, J.-M., Cointe, P., and Lieberman, H., editors, ECOOP '87 - European Conference on Object-Oriented Programming, Lecture Notes in Computer Science (LNCS), Vol. 276, Springer-Verlag, pages 192-202, 1987.

[94] Ludewig, J., Models in Software Engineering. In: Software and Systems Modeling, 2(1):5-14, 2003.

[95] Manzano, M., Extensions of First Order Logic. Cambridge University Press, 1996.

[96] Martin, J., and Odell, J.J., Object-Oriented Analysis and Design. Prentice-Hall, 1992.

[97] Martin, J., and Odell, J.J., Object-Oriented Methods: A Foundation. Prentice-Hall, 1994.

[98] Martin, R.C., Agile Software Development: Principles, Patterns, and Practices. Prentice-Hall, 2003.

[99] Mellor, S.J., and Balcer, M.J., Executable UML: A Foundation for Model-Driven Architecture. Addison-Wesley, 2002.

[100] Meyer, B., Object-Oriented Software Construction. Prentice-Hall, 1988.

[101] Mohindra, A, and Devarakonda, M.V., Dynamic Insertion of Object Services. In: Proceedings of the USENIX Conference on Object-Oriented Technologies (COOTS), USENIX, 1995.

[102] Monarchi, D.E., and Puhr, G.I., A Research Typology for Object-Oriented Analysis and Design. In: Communications of the ACM, 35(9):35-47, 1992.

[103] Nerson, J.-M., Applying Object-Oriented Analysis and Design. In: Communications of the ACM, 35(9):63-74, 1992.

[104] Nierstrasz, O., A Survey of Object-Oriented Concepts, Object-Oriented Concepts. In: Kim, W. and Lochovsky, F., editors, Databases and Applications, pages 3-21, ACM Press and Addison-Wesley, 1989.

[105] Nijssen, G.M., A Gross Architecture for the Next Generation Database Management Systems. In: Nijssen, G.M., editor, Proceeding of the 1976 IFIP Working Conference on Modelling in Data Base Management Systems, North-Holland Publishing, pages. 1-24, 1976.

[106] Nijssen, G.M., Current Issues in Conceptual Schema Concepts. In: Nijssen, G.M., editor, Proceeding of the 1977 IFIP Working Conference on Modelling in Data Base Management Systems, North-Holland Publishing, pages 31-66, 1977.

[107] Object Management Group, OMG Unified Modeling Language Specification, Version 1.3. OMG, 1999.

[108] Object Management Group, UML 2.0 OCL Specification. OMG, 2003.

[109] Object Management Group, Unified Modeling Language: Superstructure, Version 2.0. OMG, 2005.

[110] Odell, J., and Fowler, M., Analysis and Design: From Analysis to Design Using Templates, Part I. In: Report on Analysis and Design (ROAD), 1(6):19-23, 1995.

[111] Odell, J., and Fowler, M., Analysis and Design: From Analysis to Design Using Templates, Part II. In: Report on Analysis and Design (ROAD), 2(1):10-14, 1995.

[112] Odell, J., and Fowler, M., Analysis and Design: From Analysis to Design Using Templates, Part III. In: Report on Analysis and Design (ROAD), 2(3):7-10, 1995.

[113] Odell, J.J., Specifying Requirements using Rules. In: Journal of Object-Oriented Programming (JOOP), 6(2):20-24, 1993.

[114] Opdahl, A.L., and Henderson-Sellers, B., Ontological Evaluation of the UML Using the Bung-Wand-Weber Model. In: Software and Systems Modeling, 1(1):43-67, 2002.

[115] Page-Jones, M., Constantine, L., and Weiss, S., Modeling Object-Oriented Systems: The Uniform Object Notation. In: Computer Language, 7(10):69-87, 1990.

[116] Page-Jones, M., The Practical Guide to Structured System Design. Prentice-Hall, 1988.

[117] Paton, N.W., Diaz, O., Williams. M.H., Campin, J., Dinn, A., and Jaime, A., Dimensions of Active Behavior. In Paton, N.W., and Williams. M.H., editors, Proceedings of the 1st International Workshop on Rules in Database Systems, Springer-Verlag, pages 40-57, 1994.

[118] Pfleeger, S.L., Software Engineering: The Production of Quality Software, Second Edition. Macmillan, 1991.

[119] Rational Software Corporation, UML Semantics, Version 1.0. Rational, 1997.

[120] Rational Software Corporation, Unified Modeling Language Notation Guide, Version 1.0. Rational, 1997.

[121] Reenskaug, T., Wold, P., and Lehne, O.A., Working with Objects: The OORAM Software Engineering Method. Prentice-Hall, 1996.

[122] Robertson, S., and Robertson, J., Mastering the Requirements Process, Second Edition. Addison-Wesley, 2006.

[123] Robinson, K., and Berrisford, G., Object Oriented SSADM. Prentice-Hall, 1994.

[124] Rubin, K., and Goldberg, A., Object Behavior Analysis. In: Communications of the ACM, 35(9):48-62, 1992.

[125] Rubin, K.S., McClaughry, P., and Pelligrini, D., Modeling Rules using Object Behavior Analysis and Design. In: Object Magazine 4(3):63-67, 1994.

[126] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorenson, W., Object-Oriented Modeling and Design. Prentice-Hall, 1991.

[127] Rumbaugh, J., Controlling Propagation of Operations using Attributes on Relations. In: Meyrowitz, N., editor, ACM Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA ’88), ACM SIGPLAN Notices, 23(11):285-296, 1988.

[128] Rumbaugh, J., Jacobson, I., and Booch, G., The Unified Modeling Language Reference Manual, Second Edition. Addison-Wesley, 2005.

[129] Rumbaugh, J., Relations as Semantic Constructs in an Object-Oriented Language. In: Meyrowitz, N., editor, ACM Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA ’87), ACM SIGPLAN Notices, 22(12):466-481, 1987.

[130] Said, J., Pattern-Based Approach for Object-Oriented Software Design. Ph.D. Dissertation, K.U.Leuven, Department of Computer Science, Leuven, Belgium, 2003.

[131] Schlaer, S., and Lang, N., Dependence between Attributes. In: Report on Analysis and Design (ROAD), 2(4):18-23, 1995.

[132] Shlaer, S., and Mellor, S., A Deeper Look at the Transition from Analysis to Design. In: Journal of Object-Oriented Programming (JOOP), 5(9):16-21, 1993.

[133] Shlaer, S., and Mellor, S.J., Object Lifecycles: Modeling the World in States. Prentice-Hall, 1992.

[134] Shlaer, S., and Mellor, S.J., Object-Oriented Systems Analysis: Modeling the World in Data. Prentice-Hall, 1988.

[135] Shlaer, S., Methods and Architectures: Modeling Dynamic Behavior. In: Report on Object Analysis and Design (ROAD), 1(1):6-9, 1994.

[136] Snoeck, M., and Dedene, G., Existence Dependency: The Key to Semantic Integrity between Structural and Behavioural Aspects of Object Types. In: IEEE Transactions on Software Engineering, 24(24):233-251, 1998.

[137] Snoeck, M., Dedene, G., Verhelst, M., and Depuydt, A.-M., Object-Oriented Enterprise Modelling with MERODE. Leuven University Press, 1999.

[138] Snoeck, M., On A Process Algebra Approach for the Construction and Analysis of M.E.R.O.DE.-based Conceptual Models. Ph.D. Dissertation, K.U.Leuven, Department of Computer Science, Leuven, Belgium, 1995.

[139] Søndergaard, H., and Sestoft, P., Non-Determinism in Functional Languages. In: The Computer Journal, 35(5):514-523, 1992.

[140] Spivey, J.M., The Z Notation: A Reference Manual. Prentice-Hall, 1989.

[141] Starr, L., Executable UML: How to Build Class Models. Prentice-Hall, 2002.

[142] Steegmans, E., Lewi, J., D'Haese, M., Dockx, J., Jehoul, D., Swennen, B., Van Baelen, S., and Van Hirtum, P., EROOS Reference Manual. Version 1.0. K.U.Leuven, Department of Computer Science, CW Report 208, Leuven, Belgium, 1995.

[143] Steegmans, E., Lewi, J., D'Haese, M., Dockx, J., Jehoul, D., Swennen, B., Van Baelen, S., and Van Hirtum, P., EROOS Reference Manual. Version 1.1. K.U.Leuven, Department of Computer Science, Leuven, Belgium, 1996.

[144] Stroustrup, B., The C++ Programming Language. Addison-Wesley, 1985.

[145] Subieta, K., Jodlowski, A., Habela, P., and Plodzień, J., Conceptual Modeling of Business Applications with Dynamic Object Roles. In: Corchuelo, R., Ruiz-Cortés, A. and Wrembel, R., editors, Technologies Supporting Business Solutions, Advances in Computation: Theory and Practice (ACTP), pages 49-71, Nova Science Publishers, 2003.

[146] Tasker, D., Object Facts: Sources, Derived, or a Combination of Both. . In: Report on Analysis and Design (ROAD), 2(1):41-45, 1995.

[147] Ungar, D., Smith, R., Chambers, C., and Hölzle, U., Object, Message, and Performance: How They Coexist in Self. In: IEEE Computer, 25(10):53-64, 1992.

[148] Van Baelen, S., Enriching Constraints and Business Rules in Object-Oriented Analysis with Models Trigger Specifications. In: Demeyer, S., and Bosch, J., editors, Proceedings of European Conference on Object Oriented Programming Workshop Reader (ECOOP'98), Lecture Notes in Computer Science (LNCS), Vol. 1543, Springer-Verlag, pages 197-199, 1998.

[149] Van Baelen, S., Gorinsek, J., and Wils, A., editors, The DESS Methodology. ITEA-DESS Project Report D.1, ITEA-DESS Consortium, 2001.

[150] Van Baelen, S., Lewi, J., and Steegmans, E., Abstraction Stratification in an Object-Oriented Development Method like EROOS. In: de Champeaux, D., editor, Workshop on Object-Oriented Software Development Process, Sixth European Conference on Object-Oriented Programming (ECOOP 1992), Utrecht, The Netherlands, 1992.

[151] Van Baelen, S., Lewi, J., and Steegmans, E., Constraints in Object-Oriented Analysis and Design. In: Magnusson, B., Meyer, B., Nerson, J.M., and Perrot, J.F., editors, Technology of Object-Oriented Languages and Systems TOOLS 13 (TOOLS Europe 1994), Prentice-Hall, pages 185-199, 1994.

[152] Van Baelen, S., Lewi, J., Steegmans, E., and Swennen, B., Constraints in Object-Oriented Analysis. In: Nishio, S., and Yonezawa, A., editors, Object Technologies for Advanced Software (ISOTAS), Lecture Notes in Computer Science (LNCS), Vol. 742, Springer-Verlag, pages 393-407, 1993.

[153] Van Baelen, S., Lewi, J., Steegmans, E., and Van Riel, H., EROOS: An Entity-Relationship based Object-Oriented Specification Method. In: Heeg, G., Magnusson, B., and Meyer, B., editors, Technology of Object-Oriented Languages and Systems TOOLS 7 (TOOLS Europe 1992), Prentice-Hall, pages 103-117, 1992.

[154] Van Baelen, S., Lewi, J., Steegmans, E., and Van Riel, H., EROOS: An Entity-Relationship based Object-Oriented Development Method. In: DECUS BELUX 1992 Symposium Proceedings, DECUS, pages 42-64, 1992.

[155] Van Gestel, E., MOOSE: A Framework uniting Data Base Modelling, Object-Orientation an Formal Specifications, Engineering Style. Ph.D. Dissertation, K.U.Leuven, Department of Computer Science, Leuven, Belgium, 1994.

[156] Walden, K., and Nerson, J.-M., Seamless Object-Oriented Architecture. Prentice-Hall, 1994.

[157] Walicki, M., and Meldal, S., Singular and Plural Nondeterministic Parameters. In: SIAM Journal on Computing, 26(4):991-1005, 1997.

[158] Wand, Y., and Weber, R., An Ontological Model of an Information System. In: IEEE Transactions on Software Engineering, 16(11):1082-1092, 1990.

[159] Wand, Y., and Weber, R., On the Deep Structure of Information Systems. In: Information Systems Journal, 5(3):203-223, 1995.

[160] Wand, Y., and Weber, R., On the Ontological Expressiveness of Information Systems Analysis and Design Grammars. In: Journal of Information Systems, 3(4):217-237, 1993.

[161] Warmer, J., and Kleppe, A., The Object Constraint Language, Second Edition: Getting Your Models Ready For MDA. Addison-Wesley, 2003.

[162] Wegner, P., Concepts and Paradigms of Object-Oriented Programming. In: ACM SIGPLAN OOPS Messenger, 1(1):7-87, 1990.

[163] Widom, J., and Ceri, S., Active Database Systems. Morgan Kaufmann, 1996.

[164] Wieringa, R., A Survey of Structured and Object-Oriented Software Specification Methods and Techniques. In: ACM Computing Surveys (CSUR), 30(4):459-527, 1998.

[165] Wieringa, R., de Jonge, W., and Spruit, P., Using Dynamic Classes and Role Classes to Model Object Migration. In: Theory and Practice of Object Systems (TAPOS), 1(1):61-83, 1995.

[166] Wirfs-Brock, R., Stereotyping: A Technique for Characterizing Object and Their Interactions. In: Object Magazine, 3(4):50-53, 1993.

[167] Wirfs-Brock, R., Wilkerson, B., and Wiener, L., Designing Object-Oriented Software. Prentice Hall, 1990.

[168] Wu, S.-I., Integrating Logic and Object-Oriented Programming. In: ACM SIGPLAN OOPS Messenger, 2(1):28-37, 1991.

[169] Yourdon, E., and Constantine, L., Structured Design. Prentice-Hall, 1979.

[170] Yourdon, E., Modern Structured Analysis. Yourdon Press, 1989.

[171] Yuan, G., A Depth-First Process Model for Object-oriented Development with Improved OOA/OOD Notations. In: Report on Analysis and Design (ROAD), 2(1):23-37, 1995.

[172] Yue, K., What Does It Mean to Say that a Specification is Complete?. In: Proceedings of the Fourth International Workshop on Software Specification and Design, 1987.

[173] Zurcher, F.W., and Randell, B., Iterative Multi-Level Modeling: A Methodology for Computer System Design. In: Proceedings of the IFIP Congress, IEEE, pages 867-871, 1968.

List of Publications

Edited Volumes

1. Pikkarainen, M., Bozheva, T., and Van Baelen, S., editors, International Workshop on Agile: Experience, Standardization and Application in the Embedded Domain. Nemetschek, 141 pages, 2006.

Book Chapters

1. Berbers, Y., Rigole, P., Vandewoude, Y., and Van Baelen, S., CoConES: An Approach for Components and Contracts in Embedded Systems. In: Atkinson, C., Bunse, C., Gross, H.-G., and Peper, C., editors, Component-Based Software Development for Embedded Systems: An Overview of Current Research Trends, Lecture Notes in Computer Science (LNCS), Vol. 3778, Springer-Verlag, pages 209-231, 2005.

Contributions at International Conferences, Published in Proceedings

1. Van Beirendonck, H., Beaufays, J., Van Baelen, S., and De Vlaminck, K., Petri Nets for Modeling Dynamic Characteristics in HOOD. In: The Management of Large Software Projects in the Space Industry, Cépaduès-Editions, pages 121-129, 1991.

2. Van Baelen, S., Lewi, J., Steegmans, E., and Van Riel, H., EROOS: An Entity-Relationship based Object-Oriented Development Method. In: DECUS BELUX 1992 Symposium Proceedings, DECUS, pages 42-64, 1992.

3. Van Baelen, S., Lewi, J., Steegmans, E., and Van Riel, H., EROOS: An Entity-Relationship based Object-Oriented Specification Method. In: Heeg, G., Magnusson, B., and Meyer, B., editors, Technology of Object-Oriented Languages and Systems TOOLS 7 (TOOLS Europe 1992), Prentice-Hall, pages 103-117, 1992.

4. Van Baelen, S., Lewi, J., Steegmans, E., and Swennen, B., Constraints in Object-Oriented Analysis. In: Nishio, S., and Yonezawa, A., editors, Object Technologies for Advanced Software (ISOTAS), Lecture Notes in Computer Science (LNCS), Vol. 742, Springer-Verlag, pages 393-407, 1993.

5. Van Baelen, S., Lewi, J., and Steegmans, E., Constraints in Object-Oriented Analysis and Design. In: Magnusson, B., Meyer, B., Nerson, J.-M., and Perrot, J.F., editors, Technology of Object-Oriented Languages and Systems TOOLS 13 (TOOLS Europe 1994), Prentice-Hall, pages 185-199, 1994.

6. Van Baelen, S., Enriching Constraints and Business Rules in Object-Oriented Analysis with Models Trigger Specifications. In: Demeyer, S., and Bosch, J., editors, Proceedings of European Conference on Object Oriented Programming Workshop Reader (ECOOP'98), Lecture Notes in Computer Science (LNCS), Vol. 1543, Springer-Verlag, pages 197-199, 1998.

7. Urting, D., Van Baelen, S., and Berbers, Y., Embedded Software using Components and Contracts. In: Gerard, S., Terrier, F., Selic, B., Damm, G., Yi, W., and Petterson, P., editors, ECOOP 2001 Workshop on Specification, Implementation and Validation of Object-Oriented Embedded Systems (SIVOES'2001), Budapest, Hungary, 2001.

8. Barbaix, Y., Van Baelen, S., and De Vlaminck, K., Handling Time Constraints with Virtual Timers. In: Gerard, S., Terrier, F., Selic, B., Damm, G., Yi, W., and Petterson, P., editors, ECOOP 2001 Workshop on Specification, Implementation and Validation of Object-Oriented Embedded Systems (SIVOES'2001), Budapest, Hungary, 2001.

9. Urting, D., Van Baelen, S., Holvoet, T., and Berbers, Y., Embedded Software Development: Components and Contracts. In: Gonzalez, T., editor, Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems, ACTA Press, pages 685-690, 2001.

10. Urting, D., Berbers, Y., Van Baelen, S., Holvoet, T., Vandewoude, Y., and Rigole, P., A Tool for Component Based Design of Embedded Software. In: Noble, J., and Potter, J., editors, Technology of Object-Oriented Languages and Systems TOOLS 40 (TOOLS Pacific 2002): Objects for Internet, Mobile and Embedded Applications, Conferences in Research and Practice in Information Technology, Vol. 10, Australian Computer Society, pages 159-168, 2002.

11. Gorinsek, J., Van Baelen, S., Berbers, Y., and De Vlaminck, K., EMPRESS: Component based Evolution for Embedded Systems. In: Kniesel, G., Costanza, P., and Dimitriev, M., editors, Workshop on Unanticipated Software Evolution (USE 2002), European Conference on Object-Oriented Programming (ECOOP 2002), Malaga, Spain, 2002.

12. Van Baelen, S., Urting, D., and Berbers, Y., The SEESCOA Composer Tool: Using Contracts for Component Composition and Run-Time Monitoring. In: Gerard, S., Ober, I., Papadopoulos, G., Plouzeau, N., Rioux, L., Selic, B., and Weis, T., editors, UML 2002 Workshop on Component Based Software

Engineering and Modeling Non-functional Aspects (SIVOES-MONA 2002), Dresden, Germany, 2002.

13. Gorinsek, J., Van Baelen, S., Berbers, Y., and De Vlaminck, K., Managing Quality of Service during Evolution using Component Contracts. In: Kniesel, G., Costanza, P., and Fiadeiro, J.L., editors, ETAPS 2003 Second International Workshop on Unanticipated Software Evolution (USE 2003), Warsaw, Poland, pages 57-62, 2003.

14. Wils, A., Gorinsek, J., Van Baelen, S., Berbers, Y., and De Vlaminck, K., Flexible Component Contracts for Local Resource Awareness. In: Bryce, C., and Czajkowski, G., editors, 9th Workshop on Mobile Object Systems: Resource-Aware Computation (MOS 2003), European Conference on Object-Oriented Programming (ECOOP 2003), Darmstadt, Germany, 2003.

15. Berbers, Y., Rigole, P., Van Baelen, S., and Vandewoude, Y., Components and Contracts in Software Development for Embedded Systems. In: De Backer, L., editor, Proceedings of the First European Conference on the Use of Modern Information and Communication Technologies, pages 219-226, 2004.

16. Pauty, J., Van Baelen, S., and Berbers, Y., Adapting Model-Driven Architecture to Ubiquitous Computing. In: Kortuem, G., editor, Workshop on Software Engineering Challenges for Ubiquitous Computing, Lancaster University, pages 42-43, 2006.

17. Wils, A., and Van Baelen, S., Agility in the Avionics World. In: Pikkarainen, M., Bozheva, T., and Van Baelen, S., editors, International Workshop on Agile: Experience, Standardization and Application in the Embedded Domain, Nemetschek, 2006.

18. Wils, A., and Van Baelen, S., Agility and Component-based Development. In: Pikkarainen, M., Bozheva, T., and Van Baelen, S., editors, International Workshop on Agile: Experience, Standardization and Application in the Embedded Domain, Nemetschek, 2006.

19. Wils, A., Van Baelen, S., Holvoet, T., and De Vlaminck, K., Agility in the Avionics Software World. In: Abrahamsson, P., Marchesi, M., and Succi, G., editors, Extreme Programming and Agile Processes in Software Engineering (XP 2006), Lecture Notes in Computer Science (LNCS), Vol. 4044, pages 123-132, 2006.

20. Hovsepyan, A., Van Baelen, S., Vanhooff, B., Joosen, W., and Berbers, Y., Key Research Challenges for Successfully Applying MDD within Real-Time Embedded Software Development. In: Vassiliadis, S., Wong, S., and Hämäläinen, T., editors, Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS VI), Lecture Notes in Computer Science (LNCS), Vol. 4017, pages 49-58, 2006.

21. Vanhooff, B., Van Baelen, S., Hovsepyan, A., Joosen, W., and Berbers, Y., Towards a Transformation Chain Modeling Language. In: Vassiliadis, S., Wong, S., and Hämäläinen, T., editors, Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS VI), Lecture Notes in Computer Science (LNCS), Vol. 4017, pages 39-48, 2006.

Contributions at International Conferences, not Published or only as Abstract

1. Van Baelen, S., Lewi, J., and Steegmans, E., Abstraction Stratification in an Object-Oriented Development Method like EROOS. In: de Champeaux, D., editor, Workshop on Object-Oriented Software Development Process, Sixth European Conference on Object-Oriented Programming (ECOOP 1992), Utrecht, The Netherlands, 1992.

2. Van Baelen, S., Urting, D., Van Belle, W., Jockers, V., Holvoet, T., Berbers, Y., and De Vlaminck, K., Toward a Unified Terminology for Component-based Development. In: Bosch, J., Szyperski, C., Weck, W., Fifth International Workshop on Component-Oriented Programming (WCOP 2000), Fourteenth European Conference on Object-Oriented Programming (ECOOP 2000), Cannes, France, 2000.

3. Van Baelen, S., and Van Genechten, H., DESS: Project and Methodology Overview. In: ITEA Software Engineering Session, Third ITEA Symposium, Amsterdam, The Netherlands, 2002.

Contributions at National Conferences, Published in Proceedings

1. Steegmans, E., Dockx, J., Swennen, B., and Van Baelen, S., Object Gericht Programmeren: Revolutie of Evolutie. In: Proceedings Object Gericht Programmeren, KVIV, pages 1-25, 1994.

2. Steegmans, E., Dockx, J., Swennen, B., and Van Baelen, S., Het Object-Gerichte Ontwikkelingsproces: Object-Gerichte Analyse. In: Proceedings Object-Gerichte Technologie, BIRA, pages 1-14, 1995.

3. Steegmans, E., Dockx, J., Swennen, B., and Van Baelen, S., Het Object-Gerichte Ontwikkelingsproces: Object-Gericht Ontwerp. In: Proceedings Object-gerichte Technologie, BIRA, pages 15-20, 1995.

4. De Backer, S., De Vlaminck, K., Steegmans, E., and Van Baelen, S., Unified Modeling Language. In: Noë, C., and Baute, W., editors, Proceedings Unified Modeling Language (UML), KVIV, 1999.

5. Van Baelen, S., Agile Development: What and How? In: Peeters, B., and Smets, S., editors, Proceedings Agile Development and Testing, KVIV, 2005.

6. Van Baelen, S., Introduction to Agile Development. In: Peeters, B., editor, Proceedings Agile Development en Testing: Een Vernieuwde Kijk op Software Ontwikkeling, KVIV, pages 1-8, 2006.

Technical Reports

1. Lewi, J., Steegmans, E., and Van Baelen, S., EROOS: Entity-Relationship Object-Oriented Specifications. K.U.Leuven, Department of Computer Science, CW Report 111, Leuven, Belgium, 1990.

2. Lewi, J., Steegmans, E., Dockx, J., Swennen, B., Van Baelen, S., and Van Riel, H., Object Oriented Software Development with EROOS: The Analysis Phase. K.U.Leuven, Department of Computer Science, CW Report 169, Leuven, Belgium, 1993.

3. Steegmans, E., Lewi, J., D'Haese, M., Dockx, J., Jehoul, D., Swennen, B., Van Baelen, S., and Van Hirtum, P., EROOS Reference Manual, Version 1.0. K.U.Leuven, Department of Computer Science, CW Report 208, Leuven, Belgium, 1995.

4. Steegmans, E., Lewi, J., D'Haese, M., Dockx, J., Jehoul, D., Swennen, B., Van Baelen, S., and Van Hirtum, P., EROOS Reference Manual, Version 1.1. K.U.Leuven, Department of Computer Science, Leuven, Belgium, 1996.

5. Barbaix, Y., and Van Baelen S., editors, Methodology Document for Addressing Resource Constraints Problems in Embedded Systems. ITEA-DESS Project Report D.1.3.1, ITEA-DESS Consortium, 2000.

6. Van Baelen S., editor, Guidelines for Component-based Development. ITEA-DESS Project Report D.1.4.3, ITEA-DESS Consortium, 2001.

7. Barbaix, Y., Van Baelen S., and Wils, A., editors, Timing, Memory and other Resource Constraints. ITEA-DESS Project Report D.1.3.2, ITEA-DESS Consortium, 2001.

8. Van Baelen S., editor, Definition of Components and Notation for Components. ITEA-DESS Project Report D.1.4.4, ITEA-DESS Consortium, 2001.

9. Van Baelen, S., Gorinsek, J., and Wils, A., editors, The DESS Methodology. ITEA-DESS Project Report D.1, ITEA-DESS Consortium, 2001.

10. Van Baelen S., editor, Essentials and Requisites for the Management of Evolution: Evolution of Component Systems. ITEA-EMPRESS Project Report D1.2 part 2, ITEA-EMPRESS Consortium, 2003.

11. Gross, H.-G., and Van Baelen, S., editors, Modeling and System Support for Design-Time Evolution. ITEA-EMPRESS Project Report D2.1-2.2, ITEA-EMPRESS Consortium, 2003.

12. Gerlach, J., and Van Baelen, S., editors, Run-time Evolution and Dynamic (Re)Configuration of Components: Model, Notation, Process and System Support. ITEA-EMPRESS Project Report D2.4-2.5, ITEA-EMPRESS Consortium, 2003.

13. Bozheva, T., Hulkko, H., Ihme, T., Jartti, J., Salo, O., Van Baelen, S., and Wils, A., Agile in Embedded Software Development: State-of-the-Art Review in Literature and Practice. ITEA-AGILE Project Report D1, ITEA-AGILE Consortium, 2005.

14. Van Baelen S., An Agile Approach on Model-Driven Architecture (MDA). In: Abrahamsson, P., and Dooms. K., editors, AGILE Newsletter 2/2005, ITEA-AGILE Consortium, 2005.

15. Wils, A., Van Baelen, S., and Rammeloo, S., Introducing Agility in the Avionics Software World. In: Abrahamsson, P., and Dooms. K., editors, AGILE Newsletter 1/2006, ITEA-AGILE Consortium, 2006.

16. Wils, A., and Van Baelen, S., Architecture Centric Agility. In: Abrahamsson, P., and Dooms. K., editors, AGILE Newsletter 1/2006, ITEA-AGILE Consortium, 2006.

17. Huybrechts, M., Rammeloo, S., and Van Baelen, S., Realizing Agility through Model Driven Architecture. In: Abrahamsson, P., and Dooms. K., editors, AGILE Newsletter 2/2006, ITEA-AGILE Consortium, 2006.

18. Wils, A., and Van Baelen, S., Towards An Agile Avionics Process. ITEA-AGILE Project Report D2.12, ITEA-AGILE Consortium, 2007.

19. Wils, A., and Van Baelen, S., Agile Practices for Embedded Systems. ITEA-AGILE Project Report D2.13, ITEA-AGILE Consortium, 2007.

20. Wils, A., and Van Baelen, S., Software Architecture and eXtreme Programming. ITEA-AGILE Project Report D2.14, ITEA-AGILE Consortium, 2007.

Biography

Stefan Van Baelen was born in Mol, Belgium, on August 7, 1967. He received a Bachelor of Science degree (‘Kandidaat Informatica’) in 1987, and a Master of Science degree (‘Licentiaat Informatica’) in Informatics in 1989 from the K.U.Leuven. He graduated magna cum lauda with the thesis ‘Development of an Environment for the Design of Systems using State Machines’ under the supervision of Prof. Dr. ir. Johan Lewi, and in cooperation with Alcatel Bell Antwerp, Belgium. The same year, he started to perform his civil service at the Department of Computer Science and became a member of the Software Development Methodology (SOM - ‘Software Ontwikkelingsmethodologie’) research group. From 1991 until 1995, he obtained a doctoral research grant (‘aspirant NFWO’) from the Research Foundation of Flanders (‘FWO Vlaanderen’).

In 1996, Stefan Van Baelen participated in a research project with Acunia (formerly SmartMove and Take Five) on defining a software architecture for a mobile computing platform (EWACS), in cooperation with the DistriNet (Distributed Systems and Computer Networks) research group. Afterwards, he joined the DistriNet research group, and participated in a large number of national and international research projects, including the IWT-STWW project on Software Engineering for Embedded Systems using a Component Oriented Approach (SEESCOA), in cooperation with the Free University of Brussels (VUB), the University of Ghent (UGent), and the University of Hasselt (UHasselt), the ITEA project on software on Development process for real-time Embedded Software Systems (DESS), the ITEA project on Evolution Management and Process for Real-Time Embedded Software Systems (EMPRESS), the ITEA project on Agile Software Development of Embedded Systems (AGILE), the ongoing ITEA project on Model-based Approach to Real-Time Embedded Systems development (MARTES), and the ongoing ITEA project on Support for Predictable Integration of Mission Critical Embedded Systems (SPICES). The ITEA-EMPRESS project was in cooperation with Jabil Circuit (formerly Philips Hasselt), while all mentioned ITEA projects were in cooperation with Barco and a large number of international partners including, amongst others, Airbus, Nokia, Philips, NXP, DaimlerChrysler, Siemens, Bosch, Bull, Thales, Telelogic, Telefónica, and France Télécom. For many of these projects, Stefan Van Baelen performed the tasks of local DistriNet research coordinator, European Task Leader, and European Work Package Leader.

Stefan Van Baelen has served as an external referee for Communications of the ACM (Special Section on Flexible and Distributed Software Development Processes), and for the Dutch National Science Foundation.

Een beperkingscentrale benadering voor objectgeoriënteerde conceptuele modellering

Samenvatting

Objectgeoriënteerde analyse, en meer bepaald conceptuele modellering, is een softwareontwikkelingsactiviteit die streeft naar het bestuderen, analyseren en vastleggen van het probleemdomein voor een systeem in ontwikkeling. Dit moet resulteren in een specificatie van een consistent en ondubbelzinnig model dat alle domeinkennis, feiten en regels beschrijft. Hierbij heeft elk element van het probleemdomein een transparante een-op-een overeenkomst met een entiteit uit het conceptueel model.

In dit doctoraat stellen we een beperkingscentrale benadering voor objectgeoriënteerde conceptuele modellering voor, gebruik makende van hoogniveau beperkingsspecificaties als kernstructuur van het conceptueel model. Deze aanpak verrijkt de conceptuele modelstructuur op twee vlakken: enerzijds door de definitie van nieuwe structurele concepten om modelbeperkingen impliciet in de modelstructuur zelf uit te drukken, en anderzijds door de introductie van beperkingen met bijbehorende oplossingsmechanismen als een eersteklas modelconcept.

Betreffende de definitie van structurele concepten, ontwikkelden we nieuwe concepten met een bijbehorende duidelijke toepasbaarheidscontext, om zo modelbeperkingen impliciet in de modelstructuur te kunnen uitdrukken. De integratie van modelbeperkingen in elk methodologisch concept, het gebruik van existentiële afhankelijkheid als het kerncriterium voor modellering, de introductie van expliciete klassenarchieven en de formele specificatie van modelgebeurtenissen (‘*event*s’) en modelquery’s verrijken de expressieve kracht van een conceptuele modelstructuur.

Betreffende de introductie van beperkingen als een eersteklas modelconcept, ontwikkelden we een mechanisme om modelbeperkingen te specificeren met behulp

van meersoortige eerste orde logica. Het concept beperkingsreactie (‘*constraint trigger*’) verbonden met een beperking, definieert een algemene beperkingsoplosser die beperkingsschendingen kan oplossen door bijkomend gedrag in een gebeurtenis te injecteren of door gebeurtenissen te laten starten door de vooruitgang van de tijd.

Onze aanpak convergeerde in de EROOS methodiek waarvan twee versies worden voorgesteld. De basisversie, namelijk de EROOS kern, gebruikt een constructieve modelleeraanpak waarbij informatie enkel kan toegevoegd worden aan een conceptuele modelinstantiatie. De uitgebreide versie, namelijk het EROOS universum, biedt bijkomende ondersteuning aan voor terugkerende analysepatronen voor de EROOS kern door middel van geavanceerde en meer praktische concepten. Hierbij wordt de EROOS kern als onderliggende basis gebruikt.

1 Inleiding

Een van de belangrijkste uitdagingen voor softwareontwikkeling is om enerzijds een goed inzicht te krijgen in de noden en vereisten van het softwaresysteem dat gebouwd dient te worden, en anderzijds te kunnen omgaan met veranderende omstandigheden en vereisten tijdens het ontwikkelingsproces. Om een softwaresysteem te kunnen construeren dat voldoet aan de noden van de klanten en eindgebruikers moet een ontwikkelaar een duidelijk inzicht krijgen in alle kwesties betreffende het systeem en de omgeving waarin het moet opereren. Daarenboven zijn moderne softwaresystemen veel te complex om op een ad hoc wijze te construeren. Een duidelijke en ondubbelzinnig methodiek en notatie zijn noodzakelijk om kwalitatieve systemen te ontwikkelen die aan de noden en verwachtingen van de klanten voldoen. Om zowel de continue veranderende vereisten als de complexiteit van softwaresystemen te kunnen beheersen, zijn rigoureuze methoden, technieken en notaties nodig voor softwareontwikkeling om zo de vereisten op een optimale manier te modelleren en structureren.

Objectgeoriënteerde analyse, en meer bepaald conceptuele modellering, is een sleutelelement om veranderende vereisten te beheersen, aangezien het ervoor zorgt dat de specificatie van deze vereisten op een consistente manier in de context van het probleemdomein kan gebeuren. Zodoende schept dit de mogelijkheid om een duidelijk zicht te krijgen op de impact van een veranderende vereiste op het softwaresysteem en de omgeving waarin het opereert.

1.1 Problemen en vraagstukken betreffende objectgeoriënteerde analyse

Een aantal problemen en vraagstukken betreffende objectgeoriënteerde analyse kunnen worden geïdentificeerd.

- **Modelleringsnotatie:** Objectgeoriënteerde analyse heeft nood aan een beperkte set van krachtige concepten die specifiek gericht zijn naar het uitdrukken van

kennis en informatie uit het probleemdomein, en een bijbehorende notatie die deze kennis in een conceptueel model kan beschrijven. Hoewel een beperkte set van UML (*Unified Modeling Language*) concepten geschikt kunnen zijn voor analyse, zal het gebruik van UML de analist eerder naar een computationele dan naar een conceptuele zienswijze sturen. Hoewel UML mogelijkheden biedt om de notatie uit te breiden, is dit niet voldoende om UML in een geschikte analysenotatie te transformeren.

- **Modelconsistentie:** Hoewel de meeste objectgeoriënteerde analysemethodieken een aantal regels bevatten om consistentie in een model te bekomen, wordt er maar beperkt aandacht besteed aan consistentie tussen modellen. Modelelementen worden gedefinieerd in een welbepaald model, waarna ze kunnen gebruikt worden in andere modellen. Dit creëert echter een specificatievolgorde tussen de modellen, en kan zelfs leiden tot wederzijdse afhankelijkheid. Daarenboven moeten modelveranderingen worden gepropageerd naar alle modellen die gebruik maken van de veranderde elementen. Een andere aanpak om modelconsistentie te bekomen kan bestaan uit het gebruik van een uniek model dat alle informatie van de verschillende modellen bevat. Een dergelijke aanpak geeft aanleiding tot grotere klassenmodellen, omdat alle informatie hierin moet vervat zijn. Maar aangezien consistentie enkel binnen één model moet worden verwezenlijkt, zullen de nadelen van consistentie over meerdere modellen vermeden worden.

- **Modelinformaliteit:** Informele modellen geven aanleiding tot een groot aantal problemen, zoals modelfouten, onvolledigheden, tegenstrijdigheden en ambiguïteiten. Objectgeoriënteerde analyse heeft enerzijds nood aan een formele definitie van de concepten die gebruikt worden voor modellering, en anderzijds aan een formele beschrijving van de kennis die bevat zit in een conceptueel model.

- **Methodologische ondersteuning:** Het is ontoereikend om de analist enkel een modelleringsnotatie aan te bieden zoals UML. Een analist heeft nood aan een methodiek, richtlijnen en een concrete leidraad voor het construeren van conceptuele modellen, het gebruik van de methodologische concepten, en de transformatie van de kennis in het probleemdomein naar analyse-entiteiten. Bij voorkeur moet dit resulteren in een ondubbelzinnig en uniek analysemodel, waarin geen ontwerp- en implementatieaspecten aan bod komen.

- **Analyseafbakening en verdere transitie:** In veel objectgeoriënteerde methodieken is de grens tussen analyse, architectuur en ontwerp heel vaag. Vanaf de analysefase sluipen aspecten betreffende de softwarerealisatie in het model en veroorzaken zo een softwarematige vooringenomenheid ten opzichte van het probleemdomein. Daarenboven propageren een aantal methodieken een geleidelijke transitie van analyse naar ontwerp, zodat er geen breuk ontstaat tussen deze fasen. De analysefase moet echter gericht zijn op het probleemdomein, en moet daarom duidelijk gescheiden worden van latere softwaregerichte fasen. De transitie van een conceptueel model naar een softwarearchitectuur is bovendien vrij complex en niet evident, waardoor het niet kan beschouwd worden als een loutere modelverfijningsactiviteit.

Modelgedreven ontwikkelingstechnieken (MDD) kunnen hierbij wel op een nuttig manier helpen om de analyseresultaten op een adequate wijze te kapitaliseren door een (semi-) automatische transformatie van analysemodellen naar kleine of grote delen van een ontwerpmodel.

1.2 Doelstellingen

De doelstellingen van dit doctoraat zijn drieledig:

- *Definitie van de kernprincipes voor conceptuele modellering.* De huidige objectgeoriënteerde analysemethodieken hebben een aantal gebreken aangaande de modelleringsnotatie, modelconsistentie, modelinformaliteit, methodologische ondersteuning en analyseafbakening. Op basis van deze identificatie is de eerste doelstelling van dit doctoraat om een aantal kernprincipes voor conceptuele modellering op te stellen die nodig zijn om een degelijke ondersteuning te bieden voor het modelleren van de kennis uit het probleemdomein.
- *Evaluatie en vergelijking van specificatieformalismen en notaties voor modelbeperkingen.* Modelbeperkingen spelen een belangrijke rol in objectgeoriënteerde analyse. Er bestaan verscheidene specificatieformalismen om modelbeperkingen uit te drukken. Het gebruik van een bepaald formalisme kan een verschillende impact veroorzaken van de modelbeperking op het resulterende conceptueel model. Er worden zelfs verschillende alternatieve modelleerconcepten voor modelbeperkingen aangeboden binnen eenzelfde analysemethodiek. De tweede doelstelling van dit doctoraat is om specificatieformalismen voor modelbeperkingen te vergelijken, te evalueren, een taxonomie ervoor te ontwikkelen, en hun geschiktheid voor de representatie van kennis uit het probleemdomein te onderzoeken.
- *Ontwikkeling van een geschikte objectgeoriënteerde analysemethodiek en bijbehorende notatie voor conceptuele modellering.* De huidige analysemethodieken en notaties, met inbegrip van UML, zijn niet geschikt om conceptuele modellen op een adequate manier te beschrijven. De derde doelstelling van dit doctoraat is een objectgeoriënteerde analysemethodiek en bijbehorende notatie voor conceptuele modellering te ontwikkelen die voldoet aan de geïdentificeerde kernprincipes voor conceptuele modellering. De methodiek moet bovendien een geschikt specificatieformalisme voor beperkingen aanbieden. Een dergelijke analysemethodiek is essentieel om kennis, eigenschappen en structuren van het probleemdomein in een geschikt formaat vast te leggen, en het voorziene softwaresysteem te positioneren in zijn reële omgeving.

1.3 Bijdragen

De belangrijkste bijdragen van dit doctoraat zijn:

- **Geavanceerde methodologische concepten om de kernprincipes van conceptuele modellering te bereiken.** De geleverde bijdragen zijn (1) de

definitie van de kernprincipes voor conceptueel modelleren die nodig zijn om een adequaat model van het probleemdomein te bekomen, (2) een taxonomie voor modelbeperkingsformalismen in objectgeoriënteerde analyse, (3) een constructionele aanpak voor conceptueel modelleren waarbij informatie enkel kan toegevoegd worden aan een modelinstantiatie, (4) een querymechanisme om historische informatie betreffende oude attribuutwaarden, objectverbanden en tijdstippen van objectcreatie en -destructie te bekomen, en (5) een formele notatie voor de semantiek van query's en gebeurtenissen die voorafgaat aan en grotendeels vergelijkbaar is met de *Object Constraint Language (*OCL).

- **De definitie van nieuwe structurele concepten om modelbeperkingen impliciet in de modelstructuur zelf uit te drukken.** De geleverde bijdragen zijn (1) de integratie van modelbeperkingen in de definitie van elk methodologisch concept, (2) het gebruik van existentiële afhankelijkheid als het kerncriterium voor het bepalen van het conceptueel model, wat resulteert in een hiërarchische relationele modelstructuur, en (3) de introductie van expliciete klassenarchieven die beperkingen op de kunnen uitdrukken. Deze concepten verrijken de expressieve kracht van de conceptuele modelstructuur.
- **De introductie van beperkingen met bijbehorende resolutiemechanismen als een eersteklas modelconcept.** We stellen een mechanisme voor om modelbeperkingen te specificeren als eersteklas modelconcept, gebruik makende van een formele notatie die gebaseerd is op meersoortige eerste orde logica. Het mechanisme voor beperkingen gaat vooraf aan en is grotendeels vergelijkbaar met de *Object Constraint Language (*OCL). Daarenboven stellen we het concept van beperkingsreacties voor, welke een algemene oplosser voor beperkingsschendingen kan specificeren. Dit gebeurt door specifiek foutenbehandelingsgedrag te injecteren in een gebeurtenis, of door gebeurtenis op te starten op basis van de vooruitgang van de tijd.

2 Een taxonomie voor modelbeperkingsformalismen in objectgeoriënteerde analyse

Modelbeperkingen spelen een sleutelrol in objectgeoriënteerde analyse. Door middel van modelbeperkingen kunnen intrinsieke eigenschappen van het te modelleren systeem elegant worden beschreven. We categoriseerden de verschillende specificatieformalismen voor modelbeperkingen in een taxonomie, bestudeerden en vergeleken ze, en beschreven hun geschiktheid voor conceptuele modellering. Na vergelijking van verschillende benaderingen voor de specificatie van modelbeperkingen, zijn onze conclusies de volgende:

- Modelbeperkingen kunnen worden gespecificeerd als informele tekst, waarbij de beperking in een natuurlijke taal als een informeel addendum bij de modelspecificatie wordt uitgedrukt. Dit is echter te informeel als resultaat van de

analysefase en geeft aanleiding tot menselijke interpretatiefouten gedurende latere ontwikkelingsfasen.

- Modelbeperkingen kunnen worden gespecificeerd als operationele restricties, waarbij de modelbeperking wordt gerealiseerd door uitvoeringscontroles op de operaties. Dit is nuttig tijdens de ontwerpfase maar van een te laag niveau tijdens de analysefase. Een dergelijke aanpak is niet wenselijk, omdat het een grote kloof introduceert tussen het probleemdomein en het analysemodel. In plaats van te beschrijven welke regels er gelden in het probleemdomein, beschrijft het analysemodel hoe deze regels worden afgedwongen. Bovendien moeten beperkingen steeds worden geconverteerd vanuit hun conceptuele betekenis naar hun operationele implementatie, en vice versa.
- Modelbeperkingen kunnen worden gespecificeerd als een eersteklas modelconcept, waarbij modelbeperkingen worden behandeld als bouwblokken van een analysemodel. Hierbij worden modelbeperkingen als onafhankelijke modelentiteiten behandeld, wat hun belangrijkheid op een gepaste manier benadrukt. In bepaalde gevallen zijn echter andere constructies geschikter. Allereerst worden beperkingen die nauw gerelateerd zijn aan bestaande modelentiteiten beter direct hierin geïntegreerd. Zodoende is er een duidelijke focus op dergelijke beperkingen tijdens de analyse. Daarnaast worden existentiële afhankelijkheid en andere structurele modelbeperkingen beter direct uitgedrukt in de modelstructuur in plaats van te worden gemodelleerd als afzonderlijke beperkingen. In plaats van de basisstructuur van het model te benadrukken, wordt de structuur verwaarloosd en verborgen in de gespecificeerde beperkingen.
- Modelbeperkingen kunnen worden geïntegreerd in bestaande modelconcepten, waarbij een modelbeperking wordt gespecificeerd in de modelentiteit waarop de beperking betrekking heeft. Dit is mogelijk voor beperkingen die nauw verbonden zijn met een modelentiteit, zoals multipliciteit voor attributen en associatieuiteinden, beperkingen op het attribuutbereik, en algemene veranderingseigenschappen van attributen en associaties. Als beperkingen zich echter uitstrekken over verschillende modelentiteiten, is het niet aangewezen om ze te integreren in een bepaalde entiteit omdat dit leidt tot asymmetrie en willekeur in de specificatie van beperkingen.
- Modelbeperkingen kunnen impliciet uitgedrukt worden in de modelstructuur, waarbij existentiële afhankelijkheid, verplichte attribuutwaarden en gereïficeerde objecttoestanden worden gebruikt om de modelstructuur te verrijken. Een hiërarchische associatiestructuur kan existentiële afhankelijkheidsbeperkingen impliciet in de modelstructuur uitdrukken. Dit limiteert het aantal bijkomende beperkingen, en benadrukt en incorporeert de logische structuur van het probleemdomein direct in het overeenkomstige analysemodel.

UML biedt geen geschikte ondersteuning voor een gepaste specificatie van beperkingen in een analysemodel. De expressieve kracht van de UML modelstructuur moet worden verrijkt om zo geschikte conceptuele modellen te bekomen die de structuren van het probleemdomein rechtstreeks in de modelstructuur uitdrukken.

3 Kernprincipes voor conceptuele modellering

Alvorens we de EROOS methodiek voorstellen, presenteren we eerst de kernprincipes voor conceptuele modellering die geleid hebben tot bepaalde methodologische beslissingen in EROOS. We geven in de volledige tekst argumenten waarom deze principes van het allergrootste belang zijn voor conceptuele modellering om zo de meest geschikte modellen te bekomen.

- Het **principe van uniciteit** stelt dat elk feit van het probleemdomein moet resulteren in een uniek modelelement in het overeenkomstige conceptueel model. Er mag geen alternatief bestaan om feiten van het probleemdomein te modelleren, zodat we vermijden om verschillende conceptuele modellen te bekomen die enigszins equivalent zijn. In plaats daarvan moeten de door de methodiek aangeboden modelconcepten de analist leiden van het te modelleren probleemdomein naar het meest geschikte conceptueel model dat deze feiten representeert.
- Het **principe van geen overtolligheid** stelt dat elk individueel informatie-element, voorgesteld in een conceptueel model, een waarde op zich moet hebben. Het mag niet afleidbaar zijn van andere elementen in het conceptueel model. Elk feit van het probleemdomein moet direct gereflecteerd worden in het conceptueel model door een bepaalde modelentiteit, dat op zijn beurt terug getraceerd kan worden naar dit probleemdomein.
- Het **principe van geen ambiguïteit** stelt dat elk element in het conceptueel model moet voortvloeien uit een feit van het probleemdomein. Twee verschillende situaties in een probleemdomein mogen niet resulteren in een éénzelfde element binnen een conceptueel model.
- Het **principe van volledigheid** stelt dat alle relevante informatie van het probleemdomein moet gereflecteerd worden in het conceptueel model. Dit wil zeggen dat een conceptueel model onvolledig is als een aantal feiten niet expliciet beschreven zijn, maar enkel aanwezig zijn in het hoofd van de analist of domeinexpert. Dit kan leiden tot fouten, misverstanden, verwarring en arbitraire beslissingen tijdens latere fasen van het ontwikkelingsproces. Alhoewel het aanvaardbaar is dat bepaalde technische aspecten uit het oplossingsdomein niet voorgesteld worden in een conceptueel model, moet het probleemdomein in volle omvang worden gemodelleerd.
- Het **principe van minimalisme** stelt dat enkel relevante informatie uit het probleemdomein mag voorgesteld worden in het conceptueel model. Het model mag geen irrelevante informatie bevatten die niet gerelateerd kan worden met het probleemdomein of de vereisten voor het softwaresysteem. Als modelinformatie niet afgeleid kan worden uit een relevant kennisfeit uit het probleemdomein, is het overbodig en moet het worden weggelaten. Een analist moet bewust zijn van de grenzen van het probleemdomein en mag niet proberen om onbelangrijke of ongerelateerde feiten te modelleren.

- Het **principe van nauwkeurigheid** stelt dat elk feit uit het probleemdomein op een formele wijze moet gemodelleerd worden met behulp van de concepten uit de ondersteunende analysemethodiek. Tekstuele elementen of aantekeningen in een natuurlijke taal mogen geen deel uitmaken van het conceptueel model zonder een overeenkomstige formele representatie in het model.
- Het **principe van geen historiek** stelt dat het conceptueel model onafhankelijk moet zijn van de volgorde waarin de feiten uit het probleemdomein werden gemodelleerd. Het conceptueel model mag enkel afhankelijk zijn van de totale verzameling van informatie uit het probleemdomein dat gemodelleerd moet worden, en niet van de volgorde waarin deze informatie-elementen toegevoegd werden aan het model. Een conceptueel model moet een representatie zijn van een probleemdomein, en mag dus geen informatie bevatten betreffende de historiek van de constructie van het model.
- Het **principe van modelgeïmpliceerde beperkingen** stelt dat beperkingen die resulteren uit wetten en reguleringen van het probleemdomein ook weerspiegeld moeten worden in de structuur van het conceptueel model. Dit betekent dat de concepten uit een analysemethodiek in staat moeten zijn om deze belangrijke beperkingen direct in de modelstructuur uit te drukken. Daarenboven moet informatie die existentieel afhankelijk is van andere basisinformatie, eveneens gereflecteerd worden in het conceptueel model. Een modelentiteit die een feit beschrijft dat afhangt van een ander basisfeit, moet ook in het model afhankelijk zijn van de representatie van dit basisfeit.
- Het **principe van abstractie** stelt dat complexe informatie, voortvloeiend uit de intrinsieke complexiteit van het probleemdomein, gedetailleerd moet worden voorgesteld in het overeenkomstige conceptueel model. Een conceptueel model kan echter modeloverzichten geven in een meer abstracte vorm voor het welbehagen van de personen die het model moeten bestuderen. Het opstellen van abstracte modeloverzichten mag echter geen kernpunt zijn bij het conceptueel modelleren, aangezien het conceptueel model het probleemdomein gedetailleerd moet voorstellen. Maar ter bevordering van de interactie met de klanten en eindgebruikers, kan het wel nuttig zijn om modeloverzichten op te maken die een gecomprimeerde visie bieden op een mogelijkerwijs complex conceptueel model.

4 Een methodologische kern voor conceptuele modellering

De EROOS methodiek wenst de analist te begeleiden naar een uniek conceptueel model voor een bepaald probleemdomein. In het modelleringsproces spelen beperkingen een cruciale rol. Ten eerste introduceert EROOS het gebruik van existentiële afhankelijkheid als het hoofdcriterium om de modelstructuur te bepalen, waarbij modelbeperkingen impliciet in deze structuur worden uitgedrukt. Ten tweede werd de impact van modelbeperkingen op elk modelelement grondig bestudeerd, waarbij modelbeperkingen geïntegreerd worden in modelconcepten indien

aangewezen. Ten derde worden modelbeperkingen als een eersteklas modelconcept behandeld en verbonden met de betrokken modelelementen.

We presenteren twee versies van de EROOS methodiek: een kernversie ('*EROOS kernel*'), waarbij informatie enkel kan worden toegevoegd aan een conceptueel modelinstantiatie, en een uitgebreide versie ('*EROOS universe*'), waarbij er bijkomende ondersteuning wordt geboden voor EROOS analysepatronen door middel van geavanceerde en meer praktische concepten, met de EROOS *kernel* als onderliggende basis.

4.1 Klassen, objecten en statische classificatie

Het EROOS klassenconcept is grotendeels vergelijkbaar met het klassenconcept in UML. Onze bijdragen in dit verband zijn:

- De **constructionele modelaanpak**, waarbij modelinstantiaties enkel kunnen groeien door informatie toe te voegen aan een modelinstantiatie, is een cruciale eigenschap van de EROOS *kernel* om het principe van uniciteit te bereiken. Objecten kunnen niet vernietigd worden, maar in plaats daarvan moet de vernietiging van een object gereïficeerd worden in de creatie van een afzonderlijk object dat de vernietigingsgebeurtenis voorstelt.
- De methodologische aanpak met **ogenblikkelijke gebeurtenissen** verplicht de analist om een gebeurtenis met een relevante duur te splitsen in twee modelgebeurtenissen. Deze aanpak stuurt de analist naar een uniek conceptueel model voor het probleemdomein.

4.2 Attributen, domeinen, waarden en decoratie

Het EROOS attribuutconcept is grotendeels vergelijkbaar met het attribuutconcept in UML. Onze bijdragen in dit verband zijn:

- De **constructionele modelaanpak** die reeds hoger werd toegelicht. Attribuutwaarden kunnen niet veranderen, maar de verandering moet gereïficeerd worden in de creatie van een afzonderlijk object dat de veranderingsgebeurtenis voorstelt. Hierdoor kunnen analisten focussen op de informatie uit het te modelleren probleemdomein. Een analist moet niet beslissen over welke informatie in het model beschikbaar moet blijven en welke mag overschreven worden. De hoeveelheid kennis en feiten in een modelinstantiatie kan namelijk enkel vergroot worden.
- Het **standaardattribuut *Creation Timestamp*** voor elk object van iedere klasse laat de analist toe om te redeneren over het moment waarop een object ontstaan is. Dit attribuut moet niet expliciet gemodelleerd worden, maar is automatisch beschikbaar voor elk object in EROOS. Een analist moet vaak redeneren over het tijdstip waarop een bepaalde gebeurtenis heeft plaatsgevonden, bijvoorbeeld om de volgorde van bepaalde gebeurtenissen te reconstrueren, om de ouderdom van een object te bepalen of om de duur van een bepaalde activiteit te berekenen. De analist moet dergelijke attributen niet langer modelleren, en hoeft zich ook niet af

te vragen of dergelijke attributen nodig zijn in het model. De EROOS methodiek voorziet deze informatie automatisch voor alle objecten.

- Het **verbod om Booleaanse en getalattributen** te gebruiken in EROOS, het feit dat **attribuutwaarden niet ongedefinieerd** kunnen zijn en het **verbod om afgeleide attributen** te modelleren. Dit verplicht de analist om een aantal feiten in het model expliciet te modelleren met behulp van klassen, specialisatiehiërarchieën of query's, in plaats van deze informatie compact als een attribuut voor te stellen. Een dergelijke integratie van impliciete modelbeperkingen in elk methodologisch concept geeft een welbepaalde semantiek aan elk modelconcept. Hierdoor het gebruik ervan wordt gelimiteerd tot een specifieke context en de analist gedwongen wordt om het meest geschikte concept in elke situatie te gebruiken.

4.3 Relaties, verbanden en verfijning

Het EROOS relatieconcept is enigszins vergelijkbaar met het associatieconcept in UML. Onze bijdragen in dit verband zijn:

- Het systematische gebruik van **existentiële afhankelijkheid** als basiscriterium om de modelstructuur te bepalen, is een kernbijdrage van dit werk. Een dergelijke aanpak leidt tot een hiërarchische afhankelijkheidsstructuur voor objecten. Deze geeft een duidelijk inzicht in de afhankelijkheden tussen de informatie-elementen. Dit leidt tot een krachtig model dat een groot aantal modelbeperkingen direct in de modelstructuur impliceert. Relaties in EROOS zijn expliciet en op unieke wijze gemodelleerd, aangezien ze steeds ingekapseld worden in een verfijnde klasse. UML daarentegen biedt een aantal mogelijkheden aan om relaties te modelleren, zoals *associations*, *association classes*, *qualified associations*, *aggregates*, *compositions*, en een associatie gereïficeerd in een klasse.
- De **constructionele modelaanpak** die reeds hoger werd toegelicht. Relatieparticipanten kunnen niet veranderen, maar de verandering van een relatieparticipant moet gereïficeerd worden in de creatie van een afzonderlijk object dat de veranderingsgebeurtenis voorstelt.

4.4 EROOS beperkingen en restrictie

Het EROOS beperkingsconcept is grotendeels vergelijkbaar met het invariantconcept in OCL. Onze bijdragen in dit verband zijn:

- In aanvulling van een groot aantal beperkingen die geïmpliceerd worden door de EROOS modelstructuur, biedt EROOS de mogelijkheid aan om **beperkingen als eersteklas modelconcept** voor te stellen. Gebruik makende van een formele notatie, kunnen modelbeperkingen opgelegd worden om regels en regulaties van het probleemdomein uit te drukken. Ons werk dat voor het eerst gepubliceerd werd in 1993, is vergelijkbaar met OCL dat in 1995 binnen IBM werd ontwikkeld.

- In tegenstelling met OCL, legt EROOS een **unieke manier voor de specificatie van EROOS beperkingen** op. Dit wordt bereikt door (1) de verplichting om beperkingen te formuleren vanuit de top klasse(n) van de relatiehiërarchie, en (2) door de introductie van de '*not participating'* clausule. Een unieke specificatiemanier voor beperkingen heeft als voordeel dat het voor duidelijke criteria zorgt bij de ontwikkeling van conceptuele modellen. Dit leidt tot een uniek gemeenschappelijk model voor alle analisten die betrokken zijn bij de ontwikkeling van een conceptueel model. EROOS verbiedt ook expliciet de specificatie van een EROOS beperking die logisch kan afgeleid worden van andere beperkingen die reeds aanwezig zijn in het conceptueel model. Zo kan de specificatie van beperkingen in een EROOS model begrensd worden tot de verzameling van relevante beperkingen en zal het weinig afgeleide beperkingen bevatten.
- In tegenstelling met OCL, legt EROOS de analisten de **verplichting op om indien mogelijk geïmpliceerde beperkingen te gebruiken**. Het formalisme voor EROOS beperkingen is zo ontwikkeld dat het niet mogelijk is om geïmpliceerde beperkingen uit te drukken met behulp van het EROOS beperkingsconcept. Dit wordt bereikt door het verbod op het gebruik van de participatiequery ($\uparrow$) in de formulering van een EROOS beperking.

4.5 *Is-A* specialisaties en statische onderverdeling

Het EROOS specialisatieconcept is grotendeels vergelijkbaar met het generalisatieconcept in UML. Onze bijdragen in dit verband zijn:

- Het mechanisme om **beperkingen binnen een specialisatie te verstrengen**, is een kernbijdrage van dit werk. Een participantklasse van een relatie, die een existentiële afhankelijkheid uitdrukt van een verfijnd object op een participerend object, kan verstrengd worden. Bij een dergelijke verstrenging kan een participantklasse vervangen worden door een klasse die deze participantklasse via een aantal relaties en specialisaties direct of indirect bevat. Dit laat de analist toe om strengere afhankelijkheden voor een bepaald deel van de verfijnde klasse op te leggen.
- De systematische aanpak voor een specialisatie, die **partitiedisjunctie** voor elke specialisatiehiërarchie, een strikte scheiding tussen abstracte generalisatieklassen en concrete eindklassen ('*leaf classes*') en een verbod op causale afhankelijkheid oplegt. Dit stuurt de analist naar een model met zuivere specialisatiestructuren en overzichtelijke meervoudige overervingsbomen.

4.6 Query's en ornamentatie

Het EROOS queryconcept is grotendeels vergelijkbaar met *query operations* in OCL. Onze bijdrage in dit verband is de **formele notatie om de semantiek van query's uit te drukken**. Dit ondersteunt een complete en precieze beschrijving van het gedragsgedeelte van een model. Zo kan een conceptueel model gebruikt worden voor simulatie, wat tot een betere validatie van het model door de klant leidt, alsook voor

modeltransformatie naar een meer softwaregeoriënteerd model op een lager abstractieniveau. Ons werk is vergelijkbaar met OCL, dat op een later tijdstip werd ontwikkeld.

4.7 Gebeurtenissen en verrijking

Het EROOS gebeurtenisconcept is grotendeels vergelijkbaar met operaties in OCL. Onze bijdrage in dit verband is de **formele notatie om de semantiek van gebeurtenissen uit te drukken** laat een complete en precieze beschrijving van het gedragsgedeelte van een model toe en is vergelijkbaar met OCL.

5 Geavanceerde concepten voor conceptuele modellering

Alhoewel de concepten van de EROOS kern toereikend zijn om een model te construeren dat voldoet aan de kernprincipes voor conceptuele modellering, is het nuttig om geschiktere concepten ter beschikking te stellen voor de specificatie van veel voorkomende analysepatronen. Op basis van de identificatie van dergelijke analysepatronen, biedt het EROOS universum geavanceerde en praktischere concepten voor het modelleren van het probleemdomein.

5.1 Klassenarchief en objectvernietiging

Onze bijdragen in verband met het klassenarchiefconcept zijn:

- Het concept klassenarchief is een origineel en vernieuwende bijdrage tot het domein van conceptuele modellering. Andere analysemethodieken bieden geen destructoren aan of beschouwen vernietigde objecten als nutteloos voor een model. De introductie van klassenarchieven en hun gebruik in existentiële afhankelijkheidsrelaties biedt een krachtig en hoogniveau modelleringsconcept waarbij belangrijke afhankelijkheidsbeperkingen impliciet in de modelstructuur worden uitgedrukt. Verschillende soorten beperkingen tussen de levensduur van een verfijnd object en zijn participantobject kunnen zo rechtstreeks in de relatiedefinitie worden gespecificeerd.
- Het **standaardattribuut *Destruction Timestamp*** voor elk object van iedere klasse laat de analist toe om te redeneren over het moment waarop een object vernietigd is. Dit attribuut moet niet expliciet gemodelleerd worden, maar is automatisch beschikbaar voor elk object in EROOS. Hierdoor kunnen bijvoorbeeld query’s worden gedefinieerd die de gemiddelde levensduur van een object berekenen. De analist hoeft niet meer te beslissen of dergelijke attributen nodig zijn in het model aangezien de EROOS methodiek deze informatie automatisch voor alle objecten aanbiedt.
- Objecten die vernietigd worden, verdwijnen niet uit het model maar zijn nog steeds raadpleegbaar om historische informatie te verkrijgen over voorbije

gebeurtenissen, vroegere attribuutwaarden en oude relatieverbanden. De vernietiging van een object houdt enkel in dat het feit uit het probleemdomein opgehouden heeft te bestaan. Tijdens conceptuele modellering zijn kwesties betreffende de relevantie van informatie om bepaalde taken uit te voeren, niet aan de orde

5.2 Mutabiliteit van attribuutwaarden en relatieparticipanten

Het EROOS mutabiliteitsconcept is grotendeels vergelijkbaar met de *{readOnly} property modifier* in UML. Onze bijdrage in dit verband is de beschikbaarheid van **oude informatie van attributen en relaties**. Dit wordt mogelijk gemaakt door het feit dat het mutabiliteitsconcept gedefinieerd wordt bovenop het constructionele model van de EROOS kern. In tegenstelling tot andere analysemethodieken die attributen als variabelen beschouwen welke overschreven worden als een nieuwe waarde wordt gedefinieerd, biedt EROOS de mogelijkheid om te redeneren over vroegere modelinstantiaties. Dit kan door gebruik te maken van een tijdsindicatie voor attribuut-, verfijnings- en particatiequery's.

5.3 Composieten en wederzijdse afhankelijkheid

Het EROOS composietconcept is enigszins vergelijkbaar met aggregatie en compositie in UML. Onze bijdragen in dit verband zijn:

- De invoering van **composieten** geeft de analist een duidelijk omlijnd concept voor het modelleren van wederzijdse afhankelijkheid en geheel-deelstructuren, bestaande uit een niet leeg geheel en een aantal van afhankelijke delen. UML geeft een ambigue definitie voor aggregaten en composities die (1) geen wederzijdse afhankelijkheid impliceert en (2) niet duidelijk de verschillen aangeeft tussen associaties, aggregaten en composities. EROOS geeft een expliciete definitie van de verschillen tussen relaties die een unilaterale existentiële afhankelijkheid uitdrukken en composieten die een wederzijdse afhankelijkheid uitdrukken.
- Een consequente toepassing van **mutabiliteit, klassenarchieven en geïntegreerde beperkingen voor het composietconcept** biedt een coherente methodologische aanpak voor conceptuele modellering.

5.4 EROOS beperkingsreacties

Onze aanpak betreffende beperkingsreacties is een origineel en vernieuwende bijdrage tot het domein van conceptuele modellering. De introductie van beperkingsreacties zorgt voor een elegante beschrijving van het probleemdomein, waarbij een algemene beperkingsoplosser aan een beperking kan gerelateerd worden. De beperkingsreacties kunnen beperkingsschendingen oplossen door specifiek foutenbehandelingsgedrag in een gebeurtenis te injecteren. Beperkingsreacties kunnen ook gebeurtenissen opstarten op basis van de vooruitgang van de tijd. Dit laat toe om gedistribueerde effectbeschrijvingen voor gebeurtenissen op te stellen, waarbij

allereerst het basiseffect van een gebeurtenis wordt gespecificeerd. De beperkingsreacties specificeren kleine bijkomende functionaliteitsbeschrijvingen die aan het effect van de gebeurtenis worden toegevoegd op basis van de beperkingen die geschonden worden door de gebeurtenis. Een dergelijke aanpak zorgt voor een scheiding van de specificatie van de normale functionaliteit voor de gebeurtenis en de exceptionele functionaliteit voor de foutenbehandeling. De normale functionaliteit wordt in de gebeurtenis zelf beschreven, terwijl de functionaliteit van de foutenbehandeling in een aantal beperkingsreacties wordt beschreven. Zonder het gebruik van beperkingsreacties bevat de specificatie van een gebeurtenis een groot deel gedupliceerde beschrijvingen. Deze gedupliceerde specificaties komen vooral van de foutenbehandeling en van de functionaliteit voor het bewaren van de geldigheid van beperkingen. Dit veroorzaakt een grote hoeveelheid aan duplicatie in een model. Beperkingsreacties ondersteunen een aanpak van scheiding van belangen (*separation of concerns*), door de centrale groepering van alle functionaliteit betreffende foutenbehandeling. Beperkingsreacties kunnen gebruikt worden om *crosscutting behaviour* betreffende beperkingen in een model te specificeren, waardoor alle gebeurtenissen worden uitgebreid met de functionaliteit om de geldigheid van de beperking te behouden. Het kan daarom aanzien worden als een techniek voor aspectgeoriënteerde softwareontwikkeling (AOSD).

5.5 Afleidbare groepen en dynamische onderverdeling

Onze aanpak betreffende afleidbare groepen is een originele en vernieuwende bijdrage tot het domein van conceptuele modellering. Afleidbare groepen kunnen meer beperkingen direct in de modelstructuur uitdrukken. In plaats van een expliciete EROOS beperking voor een relatie te specificeren, kan de relatie een specifieke objectengroep als participant bepalen. Deze groep identificeert de verzameling van objecten die als een geldige participant in de relatie kunnen optreden. Hierbij wordt de EROOS beperking getransformeerd naar een compositieregel voor een afleidbare groep. Daarenboven geven afleidbare groepen ook een dieper inzicht in het potentieel van een klasse. In het model wordt namelijk expliciet aangegeven dat een object in een aantal relaties kan participeren als het object tot een specifieke groep behoort.

6 Conclusies

In dit doctoraat hebben we een beperkingscentrale benadering voor objectgeoriënteerde conceptuele modellering voorgesteld, gebruik makende van hoogniveau beperkingsspecificaties als kernstructuur van het conceptueel model. Onze aanpak convergeerde in de EROOS methodiek waarvan twee versies werden voorgesteld. De basisversie, namelijk de EROOS kern, gebruikt een constructieve modelleeraanpak waarbij informatie enkel kan toegevoegd worden aan een conceptuele modelinstantiatie. De uitgebreide versie, namelijk het EROOS universum, biedt bijkomende ondersteuning aan voor terugkerende analysepatronen

van de EROOS kern door middel van geavanceerde en meer praktische concepten. Hierbij wordt de EROOS kern als onderliggende basis gebruikt.

De bijdragen van dit doctoraat, die hierboven gedetailleerd werden beschreven, situeren zich op drie vlakken, namelijk (1) geavanceerde methodologische concepten om de kernprincipes van conceptuele modellering te bereiken, (2) de definitie van nieuwe structurele concepten om modelbeperkingen impliciet in de modelstructuur zelf uit te drukken, en (3) de introductie van beperkingen met bijbehorende resolutiemechanismen als een eersteklas modelconcept.

6.1 Toegevoegde waarde voor modelgedreven ontwikkeling

EROOS verleent toegevoegde waarde aan modelgedreven ontwikkeling (*Model-Driven Development - MDD*) door de formalisering van conceptuele modellering. Dit laat toe het MDD proces te starten vanaf de analysefase, vertrekkende van het conceptueel model van het probleemdomein.

MDD is een raamwerk voor softwareontwikkeling, waarbij een rigoureuze aanpak wordt gevolgd die steunt op ontwikkeling door transformatie. Op basis van hoogniveau platformonafhankelijke modellen (*Platform-Independent Models - PIM*) worden laagniveau platformspecifieke modellen (Platform-Specific Models - PSM) geconstrueerd. Het doel is om architecturale en ontwerpgeoriënteerde zaken te scheiden van technologische en implementatiegeoriënteerde beslissingen door middel van een gelaagde structuur van modeltransformaties. Dit laat toe om gradueel meer detail en platformafhankelijkheid in de laagniveau modellen te introduceren. Een dergelijke aanpak kan uiteindelijk leiden tot (semi-) automatische codegeneratie voor het softwaresysteem. MDD steunt op geformaliseerde modellen die (1) als invoermodel kunnen gebruikt worden voor een modeltransformator en (2) geproduceerd worden als het resultaat van een transformatiestap.

Omdat uit de meeste analysemethodieken modellen voortvloeien die informele beschrijvingen bevatten, zijn deze modellen niet geschikt voor een MDD aanpak. Informele beschrijvingen kunnen niet gebruikt worden als invoer voor een modeltransformatie omdat het zeer moeilijk is om gestructureerde informatie uit een informeel element te extraheren. Modellen kunnen enkel gebruikt worden voor een MDD aanpak als de informatie in een formele notatie wordt uitgedrukt die onderzocht, geëvalueerd en getransformeerd kan worden naar een ander formaat. Omdat EROOS een volledige formalisering van de structurele en gedragselementen van een conceptueel model aanbiedt, is het geschikt als notatie voor de invoermodellen van een MDD transformatie.

De transformatie van conceptuele modellen op een (semi-) automatische manier is zowel interessant voor het domein van conceptuele modellering als voor MDD.

- Betreffende conceptuele modellering kan het helpen om de analyseresultaten in de volgende softwareontwikkelingsfasen te laten renderen. Een conceptueel model is niet louter een beschrijving van het probleemdomein en de functionele

vereisten, maar wordt ook als een waardevol bezit beschouwd dat als basis kan dienen voor de verdere systeemontwikkeling. Daarenboven maken MDD transformaties *rapid prototyping* en modelsimulaties mogelijk om zo de conceptuele modellen te verifiëren en valideren. Daarnaast kunnen abstracte visies op het conceptueel model gegenereerd worden om zo de communicatie met de klant en eindgebruiker te bevorderen.

- Betreffende MDD laat een formeel conceptueel model toe om het MDD proces vanaf het probleemdomein te starten in plaats van vanaf een platformonafhankelijk softwaremodel. Dit softwaremodel kan dan door een transformatie van het conceptueel model worden bekomen.

6.2 Validatie

We hebben de EROOS methodiek op drie vlakken gevalideerd:

- Om de mogelijkheden en de geschiktheid van EROOS voor conceptuele modellering te kunnen evalueren, hebben we een groot aantal gevalstudies uitgevoerd. In samenwerking met andere leden van de SOM onderzoeksgroep en verscheidene industriële partners, hebben we de EROOS methodiek toegepast op een aantal gevalstudies uit diverse applicatiedomeinen. Deze gevalstudies werden uitgevoerd in het kader van onderzoeksprojecten in samenwerking met de industrie, licentiaatsthesissen die meestal door een industriële partner werden begeleid en als studentenprojecten in een licentiaatscursus over objectgeoriënteerde analyse (OGA, vroeger OGO). De grote variëteit aan applicatiedomeinen toont aan dat EROOS een algemene methodiek is die kan toegepast worden op een groot aantal domeinen, en dus niet enkel geschikt is voor het modelleren van informatiesystemen.
- Onze conclusies betreffende de gevalstudies zijn dat (1) EROOS geschikt is om een grote variëteit aan applicatiedomeinen te modelleren zoals hoger reeds vermeld, (2) de EROOS methodiek helpt om verborgen domeinkennis aan het licht te brengen, (3) EROOS een goed middel is om objectoriëntatie en conceptuele modellering aan te leren, (4) het een vrij grote inspanning en een nauwkeurige aanpak en attitude vraagt om conceptuele modellen in EROOS te construeren, (5) MDD toolondersteuning nodig is om het conceptueel modelleren ten volle te laten renderen en (6) personen met een opleiding van academisch niveau redelijk eenvoudig in staat zijn om de EROOS methodiek aan te leren. Personen met een opleiding van professioneel niveau daarentegen zijn vaak enkel in staat om passieve modelleringsvaardigheden verwerven. Dit betekent dat ze meestal wel in staat zijn om EROOS modellen te begrijpen en te evalueren, maar moeilijkheden ondervinden om dergelijke modellen zelf te construeren.
- Om het principe van uniciteit te evalueren, hebben we het gebruik van EROOS in een licentiaatscursus over objectgeoriënteerde analyse (OGA, vroeger OGO) vergeleken en geëvalueerd. Onze bevindingen zijn dat er drie grote oorzaken van modelverschillen zijn, namelijk (1) het niveau van detail bij het modelleren, waarbij studenten een verschillende inschatting hadden van de relevantie van

bepaalde feiten uit het probleemdomein, (2) de persoonlijke kennis van het probleemdomein, waardoor fouten werden geïntroduceerd door het gebrek aan inzicht in het probleemdomein en (3) fouten die gemaakt werden tegen de regels van de EROOS methodiek, waardoor studenten bepaalde concepten op een incorrecte manier gebruikten.

- We hebben toolondersteuning voor de EROOS methodiek ontwikkeld voor modellering, simulatie en transformatie, bestaande uit:
  - Een modelleertool dat toelaat om EROOS modellen te construeren en hieruit specificaties en modeldiagrammen te genereren. Deze EROOS tool is ontwikkeld door Bart Swennen van de SOM onderzoeksgroep.
  - Een generator voor modelsimulaties die automatisch een C++ of Java applicatie genereert met een bijbehorende generische gebruikersinterface voor een EROOS model. Dit ondersteunt *rapid prototyping* en de mogelijkheid tot vroege modelvalidatie. De applicatie bevat automatische gegenereerde code voor beperkingscontrole die de modelbeperkingen afdwingt door de resulterende modelinstantiatie te controleren na elke gebeurtenis. Indien de modelinstantiatie bepaalde beperkingen schendt, zal er deze geweigerd worden en de toestand die bestond voor de uitvoering van de gebeurtenis hersteld worden (*rollback)*. Deze generator werd ontwikkeld in een aantal opeenvolgende licentiaatsthesissen.
  - Een transformator van EROOS naar UML modellen waarbij de hiërarchische EROOS modelstructuur afgevlakt wordt in het UML model door gebruik te maken van klassen en simpele associaties. Deze transformator werd ook ontwikkeld in een licentiaatsthesis.

6.3 Verder onderzoek

De zoektocht naar de perfecte conceptuele modelleringsmethodiek is verre van beëindigd. Enkele mogelijke richtingen voor verder onderzoek zijn (1) methodogische verbeteringen voor EROOS, zoals ondersteuning voor gedistribueerde effectbeschrijvingen, zachte beperkingen en EROOS uitbreidingen, (2) ondersteuning voor modelgedreven softwareontwikkeling van EROOS naar UML, (3) modeltransformaties naar abstracte visies op het conceptueel model om de communicatie met de klant en eindgebruiker te bevorderen, en (4) de realisatie van een beperkingscentrale benadering in UML.