Aspect-Oriented Requirements Modeling

DISSERTATION

DER WIRSCHAFTWISSENSCHAFTLICHEN
FAKULTÄT
DER UNIVERSTITÄT ZÜRICH

zur Erlangung der Würde
eines Doktors der Informatik

vorgelegt von
SILVIO MEIER
von
Muri AG, Schweiz

genehmigt auf Antrag von

Prof. Dr. Martin Glinz
Prof. Dr. Harald Gall

Dezember 2009

Die Wirtschaftswissenschaftliche Fakultät der Universität Zürich, Lehrbereich Informatik, gestattet hierdurch die Drucklegung der vorliegenden Dissertation, ohne damit zu den darin ausgesprochenen Anschauungen Stellung zu nehmen.

Zürich, den 9. Dezember 2009[1]

Der Lehrbereichsvorsteher: Prof. Dr. Harald Gall

[1] Datum der Promotionsfeier

The supreme goal of all theory is to make the irreducible basic elements as simple and as few as possible without having to surrender the adequate representation of a single datum of experience.

— Albert Einstein

Acknowledgments

There are many people to whom I would like to express my gratitude. Many students contributed work to the aspect-oriented ADORA approach. My most profound thanks to Adrian Gygax, Ivo Vigan, Jürg Schlä i, Pascal Schöni, and Marc Vontobel, who contributed as student workers the major part of the proof-of-concept implementation of the present work. They also gave invaluable inputs for improvements. Without their help, the present implementation would not have been possible and probably many weaknesses in the concepts would not have been discovered and left unadjusted. I would also like to thank the students who contributed with their diploma work to the project, especially Irene Bonomo who delivered a great diploma thesis about the state-of-the-art of aspect-oriented software development.

I am much obliged to my colleagues Tobias Reinhard, Reinhard Stoiber, Christina Cramer, Christian Seybold, Nancy Merlo-Schett, Samuel Fricker, and Yong Xia who gave helpful input for the present work and discussed several issues with me. I am also indebted to Norbert Fuchs, who was ever ready to lend me a sympathetic ear to my problems. My thanks also go to my supervisor Martin Glinz, who critically examined many pages of draft.

Gérard Milmeister and Jody Weissmann reviewed parts of the present work, too. Gérard also critically examined the concepts, partially revised the English, and gave hints for further improvements. John Melaugh discerningly proofread the English in the work.

Last but not least I want to thank my fiancée Jeannette for her love and patience, which was a strong support in finishing the present work.

Many thanks to all of you.

Zurich, August 2009

Abstract

Over, the last few years, a research field called Aspect-Oriented Software Development has been emerging. It aims at the proper separation of concerns in a piece of software. By keeping the so-called crosscutting concerns separate from other concerns, the description of a software system can be simplified, which increases the understandability of the artifacts. In turn, this results in easier maintainability of the software and, therefore, in lower costs.

Aspect-oriented techniques can also be useful at the early phases of the software development process. Requirements engineering is the most important early phase because its result is the foundation for all following stages. Errors made at this stage have far-reaching consequences and a strong impact on the success of a software project.

Crosscutting concerns may become manifest in various kinds of requirements artifacts, such as the description of non-functional requirements which are inherently crosscutting. Documenting crosscutting concerns with conventional means leads to unnecessary, problem-exogenous complexity in the software requirements, which can be an unnecessary source of errors. Aspect-oriented techniques are a promising means to eliminate this problem. It also entails other advantages, such as a better traceability of crosscutting concerns. However, aspect-oriented techniques may also sometimes negatively affect the understandability because additional complexity may be introduced under certain circumstances.

Requirements must be recorded on an adequate level of formality and detail so that they are easy to communicate to the stakeholders in a project. Moreover, they must be easily tangible and understandable. A good means which has all these desiderata are graphical modeling techniques.

The work at hand investigates the characteristics that an optimal aspect-oriented requirements modeling approach should have. It demonstrates how existing requirements and architectural modeling languages may be enriched by aspect-oriented language elements. The insights of the present work are exemplified by a new aspect-oriented requirements modeling approach based on the ADORA language. ADORA was chosen because it innately possesses several characteristics which are desirable for an aspect-oriented requirements approach. The proposed approach allows switching between the conventional and the aspect-oriented view, which is useful to mitigate the additional complexity introduced by the aspect-oriented constructs themselves. Moreover, the language supports a planned evolution of aspect-oriented requirements from a very early point in the software process.

Zusammenfassung

In den letzen Jahren ist ein neues Forschungsgebiet namens *Aspektorientierte Software-Entwicklung* entstanden. Aspektorientierte Software-Entwicklung zielt darauf ab, die Belange (engl. Concerns) einer Software sauber voneinander zu trennen. Indem die sogenannten *querschneidenden Belange* (engl. Crosscutting Concerns) von den anderen Belangen getrennt werden, wird die Beschreibung einer Software vereinfacht, was deren Verständlichkeit erhöht. Dies wiederum führt zu einer besseren Wartbarkeit der Software und zu weniger Kosten bei der Entwicklung.

Der Einsatz von aspektorientierten Techniken ist ebenfalls in den frühen Phasen der Software-Entwicklung sinnvoll. Das Erarbeiten der *Software-Anforderungen* ist die wichtigste frühe Phase, da deren Resultate die Grundlage für alle folgenden Aktivitäten bildet. Fehler, die zu diesem Zeitpunkt gemacht werden, haben weitreichende Konsequenzen und eine grosse Auswirkung auf den Erfolg eines Software-Projekts.

Querschneidende Belange manifestieren sich in verschiedenen Artefakten der Anforderungsphase, wie z.B. in nicht funktionale Anforderungen, die inhärent querschneidend sind. Die Beschreibung von querschneidenden Belangen mit konventionellen Mitteln führt zu unnötiger, nicht problem-inhärenter Komplexität in den Software-Anforderungen, was eine unnötige Fehlerquelle darstellt. Aspektorientierte Techniken sind eine vielversprechende Möglichkeit, welche die beschriebenen Probleme beheben kann. Zudem ergeben sich weitere Vorteile, wie z.B. eine bessere Rückverfolgbarkeit der Anforderungen. Nichtsdestotrotz kann die Verwendung von aspektorientierten Techniken auch Nachteile mit sich bringen, weil unter bestimmten Umständen zusätzliche Komplexität eingeführt wird.

Anforderungen müssen so adäquat dargestellt und formalisiert sein, damit diese möglichst einfach an alle Projektbeteiligten kommuniziert werden können. Zudem sollten diese einfach fassbar und verständlich sein. Dafür eignen sich grafische Modellierungstechniken besonders gut.

Die vorliegende Arbeit identifiziert die Eigenschaften, die ein optimaler aspektorientierter grafischer Modellierungsansatz für Anforderungen besitzen sollte. Sie demonstriert, wie existierende Anforderungsansätze mit den nötigen Sprachelementen angereichert werden können. Die Erkenntnisse der Arbeit werden anhand einer aspektorientierten Erweiterung für die grafische Modellierungssprache ADORA gezeigt. ADORA wurde ausgewählt, weil sie von Grund auf verschiedene erwünschte Eigenschaften einer aspektorientierten Anforderungssprache besitzt. Der gezeigte Ansatz erlaubt es, zwischen der konventionellen und der aspektorientierten Modellierungssicht zu wechseln, was hilft, die zusätzliche Komplexität, welche durch die aspektorientierten Elemente eingeführt wurde, abzuschwächen. Weiter unterstützt die Spracheerweiterung von ADORA die kontrollierte Evolution von aspektorientierten Elementen von einem frühen Zeitpunkt im Software-Entwicklungsprozess an.

Contents

List of Figures xix

List of Tables xxiii

Listings xxiv

I Basics and Motivation of Aspect-Oriented Requirements Engineering 1

1 Introduction 3
1.1 Motivation 3
1.2 Gaps in Existing Aspect-Oriented Requirements Approaches 5
1.3 Goals of the Present Work 5
1.4 Contribution 6
1.5 Structure of this Thesis 7

2 Basics of Requirements Engineering and Modeling 9
2.1 Requirements Engineering 10
2.1.1 Functional and Non-Functional Requirements 10
2.1.2 Requirements Process 11
2.2 Requirements Document 15
2.2.1 General Quality Characteristics 16
2.2.2 Degree of Formality and Detailedness 17
2.2.3 Constructive vs. Descriptive Specification 18
2.3 Modeling 19
2.3.1 General Model Theory 19
2.3.2 Languages for the Visual Modeling of Software Systems 19

3 Aspect-Oriented Software Development 23
3.1 Fundamental Terms and Concepts of AOSD 23
3.1.1 Concerns and Separation of Concerns 24

3.1.2 Crosscutting Concerns vs. Core Concerns 25
3.1.3 Documenting Concerns in Software 26
3.1.4 Tangling, Scattering, and the Resulting Problems 30
3.1.5 Decoupling Crosscutting Concerns 32
3.1.6 Separation vs. Composition 34
3.1.7 Complexity Caused by the Use of Aspect-Oriented Artifacts 36
3.1.8 Characteristics of Aspects 37
3.1.9 Connection between Concerns and Requirements 38
3.2 State of the Art Approaches 40
3.2.1 Categorizing Aspect-Oriented Approaches 40
3.2.2 Approaches to Aspect-Oriented Programming 41
3.2.3 Approach to Software Design 44
3.2.4 Approaches to Software Architecture 45
3.2.5 Approaches to Requirements Engineering 46
3.3 Criticism of AOSD 48
3.3.1 Criticism of AOSD at the Requirements Stage 48
3.3.2 Problems with the Understandability of Aspect-Oriented Constructs . . . 50
3.3.3 Breaking the Principle of Information Hiding 51
3.3.4 Fragile Join Points 51
3.3.5 Further Criticism 52
3.4 Discussion 52

4 Motivating a Novel Aspect-Oriented Requirements Engineering Approach 53
4.1 Gaps in the Existing Approaches 53
4.2 Proposal for a New Aspect-Oriented Requirements Engineering Approach 58

5 Basics of the ADORA Approach 61
5.1 Language Concepts of ADORA 62
5.1.1 Modeling with Abstract Objects 63
5.1.2 Hierarchical Decomposition 64
5.1.3 Integrated Modeling Language Concepts 64
5.1.4 Visual Abstraction Mechanisms 66
5.1.5 Variable Degree of Formality 67
5.1.6 Requirements Evolution Support 68
5.2 Overview of the ADORA Language 68
5.2.1 Base View 68
5.2.2 Structural View 70
5.2.3 Behavioral View 73
5.2.4 User View 76
5.2.5 Context View 77
5.2.6 Functional View 78
5.2.7 Additional Structures 84

6 Analyzing and De ning the ADORA Language 87
6.1 ADORA Grammar 88
6.1.1 Grammar Definition 88
6.1.2 Applying the Grammar to Textual ADORA Models 90
6.1.3 Representing Informal Elements 91
6.1.4 Expressing Model Relationships by Nesting Textual Models 92
6.1.5 Identifying ADORA Model Elements 93
6.1.6 The Representation of Connections 95
6.1.7 Non-Graphical Language Elements 95
6.2 Formalizing the Model Data Structure and Operations 96
6.2.1 A Data Structure for the Textual Representation of ADORA Models . . . 96
6.2.2 Functions on Syntax Trees 98
6.3 Language Constraints 99
6.3.1 Time-Dependant Enforcement of Constraints 100
6.3.2 Example of a Constraint Description 101
6.4 Graphical Mapping 102

II The Aspect-Oriented ADORA Modeling Approach 105

7 Aspect-Oriented Language Extension for ADORA 107
7.1 Motivation 107
7.2 Overview of the new Aspect-Oriented Approach 111
7.3 Aspect Module 115
7.3.1 Grammar Production Rules 115
7.3.2 Language Constraints 116
7.4 Crosscutting Behavior 116
7.4.1 Production Rules 118
7.4.2 Language Constraints 119
7.5 Crosscutting User View 121
7.5.1 Production Rules 122
7.5.2 Language Constraints 122
7.6 Join Relationships 125
7.6.1 Production Rules 128
7.6.2 Language Constraints 129
7.7 Crosscutting Environment Objects 132
7.7.1 Grammar Production Rules 133
7.7.2 Language Constraints 134
7.8 Crosscutting Functional Specification 135
7.8.1 Grammar Production Rules 136
7.8.2 Language Constraints 138
7.9 Aspect Decomposition 139
7.9.1 Grammar Production Rules 140

7.9.2 Language Constraints . . . 141
7.10 Summary and Discussion . . . 142

8 Visualization of Aspect-Oriented Model Elements 145
8.1 Applying Abstractions to Aspects . . . 145
8.1.1 View Transition Semantics for Aspect Modules . . . 146
8.1.2 Join Relationships . . . 147
8.2 Extending the View Concept . . . 149
8.3 Discussion . . . 150

9 Composing Aspect-Oriented ADORA Models 153
9.1 Weaving Process Overview . . . 154
9.1.1 Weaving Preparation . . . 156
9.1.2 Weaving Transformation . . . 159
9.2 Weaving Semantics of Non-Partial Aspect-Oriented Model Elements . . . 160
9.2.1 Weaving Semantics of Behavior Chunks . . . 160
9.2.2 Weaving Semantics of Scenario Chunks . . . 166
9.2.3 Weaving of Crosscutting Statecharts . . . 171
9.2.4 Crosscutting Scenariocharts . . . 172
9.2.5 Weaving of Embedded Components . . . 176
9.2.6 Weaving Environment Objects . . . 177
9.2.7 Weaving the Functional View of an Aspect . . . 179
9.2.8 Weaving Server Components Connected to Aspects . . . 182
9.2.9 Solving Naming Con icts, Handling Context Mappings, and Adjusting Scope . . . 186
9.2.10 Post processing . . . 188
9.3 Weaving Semantics Involving Partial Aspect-Oriented Elements . . . 188
9.3.1 Partial Join Relationship Connecting Scenario/Behavior Chunks with a Scenario/Transition . . . 189
9.3.2 Partial Join Relationship Connecting Other Elements . . . 190
9.3.3 Weaving Semantics of Partial Join Relationships Between Environment Objects . . . 191
9.3.4 Weaving Partial Aspects . . . 193
9.4 Formal Weaving Semantics . . . 194
9.4.1 A Description Schema for the ADORA Weaving Semantics . . . 195
9.4.2 Description Schema of the ADORA Weaving Semantics . . . 197
9.4.3 Illustration of the Formal Weaving Semantics . . . 199
9.5 Weaving the Layout Information . . . 202
9.6 Summary and Discussion . . . 205

10 Applying the Aspect-Oriented ADORA Approach 207
10.1 ADORA Modeling Process Overview . . . 207
10.2 Functional Requirements Model Increment . . . 208

10.3 Detection of Functional Crosscutting Requirements . . . 210
10.4 Eliciting/Refining Non-functional Requirements . . . 212
10.5 Discussion . . . 215

III ADORA Tool Implementation and Validation 217

11 Tool Implementation 219
11.1 Tool Overview . . . 220
11.1.1 Features of the ADORA Tool . . . 220
11.1.2 Architecture of the ADORA Tool . . . 221
11.2 Meta-Model Implementation . . . 224
11.2.1 Choosing an Appropriate Design for the Meta-model Implementation . . . 224
11.2.2 Grammar Mapping . . . 225
11.2.3 Tool Support for the Mapping of the Meta-model . . . 229
11.2.4 Discussion of the ADORA Tool Implementation . . . 229
11.3 Constraints Checking . . . 230
11.4 Model Transformations . . . 231

12 Experimental Validation of the Aspect-Oriented Modeling Approach 233
12.1 Experiment . . . 236
12.1.1 Planning and Preparation of the Experiment . . . 237
12.1.2 Case Studies . . . 237
12.1.3 Realization of the Experiment . . . 240
12.2 Analysis of the Results . . . 242
12.2.1 Validity of the experiment . . . 246
12.3 Summary . . . 246

IV Conclusions 249

13 Conclusions 251
13.1 Discussion and Contribution of the Present Work . . . 252
13.1.1 Summary, Discussion, and Contribution . . . 252
13.2 Outlook . . . 255
13.3 Conclusion . . . 257

V Appendix 259

A Discussion of Aspect-Oriented Requirements Approaches 261
A.1 Evaluation Criteria . . . 261
A.2 Conventional Approaches . . . 262
A.2.1 PREView . . . 262

A.2.2 NFR Framework . . . 264
A.2.3 KAOS Approach . . . 265
A.2.4 I* Approach . . . 267
A.2.5 Use Case Method . . . 268
A.3 Aspect-Oriented Approaches . . . 270
A.3.1 AORE with Arcade . . . 270
A.3.2 ARGM . . . 271
A.3.3 AOSD/UC . . . 273
A.3.4 SMA . . . 274
A.3.5 AUCDA . . . 275
A.3.6 Cosmos . . . 277
A.3.7 CORE . . . 278
A.3.8 AOREC . . . 279
A.3.9 Theme/Doc . . . 281

B EBNF of the Aspect-Oriented ADORA language 283
B.1 Extended Backus Naur Form . . . 283
B.2 Regular Expressions in the Grammar . . . 284
B.3 EBNF Grammar of the ADORA Language . . . 285
B.4 Production Rules of Identifiers and References . . . 291

C Mapping of Graphical ADORA Model Elements 295

D Textual ADORA Example Model 299
D.1 Example of a Conventional ADORA Text Model . . . 299
D.2 Example of Textual Description of an Aspect Module . . . 302

E Functions on Syntax Trees 305
E.1 Formalized Data Structure for Syntax Tree . . . 305
E.2 Alphabetical Catalog of Functions . . . 306
E.3 Primitive Functions . . . 309
E.4 Basic Functions . . . 312
E.5 Aspect Specific Functions . . . 343
E.6 Transformation Functions . . . 352

F Formal Language Constraints of Aspect-Oriented Constructs 365
F.1 Aspect Module . . . 365
F.2 Behavior Description . . . 365
F.2.1 State Groups Must Be Well-Formed . . . 365
F.2.2 No Out-Going Join Relationships from Crosscutting Statecharts . . . 366
F.2.3 No Crossing of the Aspect Border by Transitions . . . 366
F.2.4 Transitions May Connect to Exit Points . . . 366
F.3 User View . . . 367

F.3.1 No Crossing of Aspect Border by a Scenario Connection 367
F.3.2 Well-Formed Scenario Chunks 367
F.3.3 Well-Formed Crosscutting Scenariocharts 368
F.3.4 Disallowed Embedding of A Scenario Node in a Component Belonging to a Statechart 368
F.4 Join Relationships 369
F.4.1 Constituents of Non-Partial Join Relationships 369
F.4.2 Constituents of Partial Join Relationships 369
F.4.3 Join Relationships Connecting to Scenariochart Root Nodes 370
F.4.4 No Cycles in Join Relationships 370
F.4.5 Border Crossing of the Join Relationship 371
F.4.6 Priority within the Range of 1–10 371
F.5 Crosscutting Environment Objects 371
F.5.1 Only One Join Relationship between Two Environment Objects 371
F.5.2 Crosscutting Environment Objects Must Be Connected To a Scenariochart 371
F.5.3 No Association of a Crosscutting Environment to More than One Aspect 372
F.6 Aspect Decomposition 372
F.6.1 Aspect-Refined Aspect Modules May Contain only Aspect Modules . . . 372
F.6.2 Associations Originating within an Aspect May not Cross the Border of the Aspect 373
F.6.3 Components and Aspects Must Be Connectable 373

G Formal Weaving Semantics 375
G.1 Formal Weaving Semantics of Non-Partial Elements 375
G.1.1 Weaving Semantics of Behavior Chunks 375
G.1.2 Formal Weaving Semantics of Scenario Chunks 380
G.1.3 Formal Weaving Semantics of Crosscutting Statecharts 386
G.1.4 Formal Weaving Semantics of Crosscutting Scenariocharts 388
G.1.5 Formal Weaving Semantics of Embedded Components 391
G.1.6 Formal Weaving Semantics of Environment Objects 395
G.1.7 Formal Weaving Semantics of the Functional Specification 399
G.1.8 Formal Weaving Semantics of Server Components 403
G.1.9 Formal Semantics of the Postprocessing 412
G.2 Formal Weaving Semantics for Partial Aspect-Oriented Elements 413
G.2.1 Formal Weaving Semantics of Partial Join Relationships Connecting Scenario/Behavior Chunks with a Scenario/Transition 413
G.2.2 Formal Weaving Semantics of Partial Join Relationships Connecting other Elements 414
G.2.3 Formal Weaving Semantics of Partial Join Relationships between Crosscutting Environment Objects 416
G.2.4 Formal Weaving Semantics of Partial Aspect Modules 417

H More Details on the Tool Implementation **419**
H.1 Relating the Visual Representation to the Model Elements 419
H.2 Mapping between EBNF Rules and Classes in the Object-Oriented Meta-model . 420
H.3 Using the Constraints Checking Plug-in and ICL 423

Bibliography **427**

List of Figures

2.1 An example for the refinement of a high-level functional requirement to executable code . . . 12
2.2 Example for the operationalization of non-functional requirements . . . 14
2.3 Example model facets of a UML model . . . 20

3.1 Illustration of the crosscutting concerns and the crosscutting relationship between concerns . . . 26
3.2 Documenting crosscutting concerns with conventional modular artifacts . . . 27
3.3 Example for crosscutting concerns in architectural artifacts . . . 31
3.4 Excerpt from user documentation containing crosscutting concerns . . . 32
3.5 Illustration of the scattering and tangling of crosscutting concerns . . . 33
3.6 Illustration of the separation and integration process of crosscutting concerns . . . 35

4.1 Illustration of the number of integration operations needed to compose different facets described by an aspect-oriented UML approach . . . 57

5.1 An object composition for a library system vs. the class representation of the same system . . . 63
5.2 Parts of a library system modeled in ADORA . . . 66
5.3 The library system with a different focus on the model . . . 70
5.4 Examples of associations, abstracted associations and interrelationships . . . 72
5.5 Illustration of the behavioral description . . . 74
5.6 Illustration of the life cycle modeling of objects . . . 75
5.7 An example of the scenario syntax and all syntax elements of scenarios . . . 78

6.1 A concrete syntax tree of a textual model . . . 97
6.2 An example of a syntax tree represented as nested tuple structure . . . 98
6.3 An example tuple structure containing an informal comment . . . 98
6.4 Illustration of a constraint violation . . . 101

7.1 A partial conventional view of the library system example. . . . 109
7.2 An example ADORA model that contains aspect-oriented elements . . . 114
7.3 Examples violating and satisfying the constraints of an aspect's behavior description . . . 120

7.4 Examples violating and satisfying the user view constraints. 124
7.5 Example motivating the usage of a context map. 127
7.6 Allowed use of partial and concrete join relationships 130
7.7 Concrete join relationships connected to the root node of a scenariochart. 131
7.8 Model which exemplifies cyclic that join relationships are not allowed 131
7.9 A join relationship may not cross the border of the aspect's parent. 132
7.10 Illustration of the constraints for crosscutting environment objects 134
7.11 Illustrations of the language constraints for the aspect refinement 141

8.1 Several situations which illustrate the hiding of a node in an aspect 147
8.2 View transition when hiding an aspect module. 147
8.3 Examples of calculated abstract join relationships 148
8.4 Further examples of calculated abstract join relationships 149
8.5 Illustration of hiding the behavioral view, the user view, the aspect view and the join relationship view 151

9.1 Illustration of the effects on a model/syntax tree when executing a weaving operation 155
9.2 Illustration of a complex network of join relationships 157
9.3 A partial view of the aspect-oriented library system model 161
9.4 The woven model of the library system 162
9.5 Illustration of the weaving semantics for behavior chunks 164
9.6 Illustration of the weaving semantics for multiple behavior chunks impacting the same target transition 165
9.7 Illustration of the weaving semantics for multiple behavior chunks impacting the same target transition 166
9.8 Weaving semantics for scenario chunks crosscutting a target scenario node . . . 169
9.9 Weaving semantics for more than one join relationship targeting at the same scenario node of type *sequence* 170
9.10 Further cases with more than one join relationship targeting at the same scenario node 171
9.11 Weaving semantics for crosscutting statecharts 173
9.12 Example for the weaving semantics of crosscutting scenariocharts 174
9.13 Illustration of the weaving process for crosscutting scenariocharts 175
9.14 Example of the weaving semantics for embedded components 177
9.15 Illustration of the weaving process for embedded components 178
9.16 Example of the weaving semantics for a crosscutting environment object 179
9.17 Illustration of the weaving steps for crosscutting environment objects 180
9.18 Model illustrating the weaving semantics of the functional specification 180
9.19 Illustration of the weaving semantics for server components 184
9.20 Example for the handling of naming con icts, the weaving semantics of a context map, and the scope extension of message arguments 187

9.21 Illustration of the weaving semantics for an abstract join relationship connecting a behavior chunk with a transition and a scenario chunk with a scenario node, respectively . . . 189
9.22 Illustration of the weaving semantics for a join relationship between an aspect and a component . . . 191
9.23 Illustration of the weaving semantics for an abstract join relationship between an aspect and an association . . . 192
9.24 Illustration of the weaving semantics for the abstract join relationship between an entry state of a behavior chunk and a component . . . 192
9.25 Illustration of the weaving semantics for an abstract join relationship connecting two environment objects . . . 193
9.26 Weaving semantics for a partial aspect . . . 194
9.27 Illustration of the weaving process. . . . 198
9.28 Model illustrating the visual weaving of behavior and scenario chunks . . . 203
9.29 Model illustrating the visual weaving of crosscutting statecharts, crosscutting scenariocharts, and embedded components . . . 204
9.30 Illustration of the open issues in the visual weaving of ADORA models . . . 205

10.1 Overview of the requirements process . . . 208
10.2 Elicitation and refinement process for functional requirements . . . 209
10.3 Example of the first few steps in an evolutionary process creating the model of a library system . . . 211
10.4 Elicitation and refinement process for non-functional requirements . . . 213
10.5 Some further steps in the evolution of the example . . . 215

11.1 A screen shot of the ADORA tool . . . 221
11.2 The current architecture of the ADORA tool . . . 223
11.3 A coarse class diagram of the meta-model implementation of the ADORA tool . 226

12.1 Example illustrating the isolated and the interrelated focus on a crosscutting concern . . . 234
12.2 Screen shot of the empirical testing environment of the ADORA tool . . . 239
12.3 Modeling skills of the test persons . . . 241
12.4 Process of the validation experiment . . . 241
12.5 Average performance results per question for the objective questions with an isolated focus on the crosscutting concerns . . . 243
12.6 Average performance results per question for the objective questions having an interrelated focus on the crosscutting concerns . . . 244
12.7 Subjective answers about the usefulness of the aspect-oriented modeling view . 245

E.1 A simple example model for the illustration of functions working on syntax trees 306
E.2 A concrete syntax tree example . . . 310
E.3 A tuple representation of an ADORA model . . . 311

H.1 An overview of the relationships between the specification, the ADORA model nodes, and its representation 420
H.2 An overview of the representation of the meta-model elements 421
H.3 A screen shot of the constraint checker view in the ADORA tool 425

List of Tables

4.1 Criteria for the evaluation of aspect-oriented requirements approaches. 53

6.1 Excerpt of the ADORA EBNF grammar showing the most important production rules 89
6.2 Points in time for evaluating language constraints 100
6.3 Excerpt of the graphical mapping in Appendix C. 104

7.1 The EBNF grammar for aspect modules 116
7.2 Syntax rules describing the embedding of aspects in components and the root of an ADORA model 116
7.3 EBNF syntax rules describing the behavioral view of an aspect 119
7.4 EBNF syntax rules for the user view of an aspect. 123
7.5 EBNF grammar defining join relationships. 129
7.6 EBNF grammar rules of the join relationships of crosscutting environment objects. 133
7.7 EBNF grammar rules for the functional specification of an aspect. 137

11.1 Excerpt of the ADORA EBNF grammar rules defining a component 229

12.1 Impact on the performance of tasks resulting from different combinations of focus type and the type of view. 235
12.2 Some examples of tasks with an isolated or an interrelated focus on a crosscutting concern. 235

A.1 Criteria for the evaluation of aspect-oriented requirements approaches. 261
A.2 Evaluation of the PREView approach. 263
A.3 Evaluation of the NFR approach. 265
A.4 Evaluation of the KAOS approach. 266
A.5 Evaluation of I* 267
A.6 Evaluation of the Use Cases Method approach. 269
A.7 Evaluation of the AORE with Arcade. 270
A.8 Evaluation of ARGM. 272
A.9 Evaluation of AOSD/UC. 273
A.10 Evaluation of SMA. 275
A.11 Evaluation of the AUCDA approach. 276

A.12 Evaluation of the Cosmos approach. 277
A.13 Evaluation of the CORE approach. 279
A.14 Evaluation of AOREC. 280
A.15 Evaluation of the Theme/Doc. 281

B.1 Informal Description of the EBNF grammar. 283
B.2 Regular Expressions in the ADORA grammar. 284
B.3 EBNF grammar of the aspect-oriented ADORA language. 285
B.4 Grammar rules of identifiers and references 292

C.1 Graphical Mapping for the aspect-oriented ADORA Language 295

E.1 Alphabetical list of syntax tree function. 306
E.2 Table defining the valid part-of relationships of ADORA elements. 319

Listings

3.1 Example of a piece of code with two tangled concerns. A business concern managing the transfer of money is tangled with a logging concern. 28
5.1 An example of provides/requires section. 79
5.2 A example of an invariant which may be part of the library system. 80
5.3 Examples of data types, standardized properties and attributes. 80
5.4 An example for the definition of a synchronous operation. 83
6.1 Excerpt of a textual model example based on the model in Fig. 5.2. 90
6.2 Example of a informal description: The object set is described by an informal comment. 92
6.3 The model from Fig. 6.4 as textual description. 102
7.1 The *Authentication* aspect from Fig. 7.2 as textual specification. 112
9.1 The functional specification of the component *C* of model t_0 in Fig 9.18. 181
9.2 The functional specification of the aspect *A* of model t_0 in Fig 9.18. 181
9.3 The functional specification of the woven component *C* of model t_n in Fig 9.18. . 182
D.1 An example of a textual ADORA model which shows a partial version of the library system. 299
D.2 The *Authentication* aspect from the model of Fig. 7.2 as textual specification. . . 302
E.1 The function $actionPart$. 313
E.2 The function $attributes$. 313
E.3 The function $calcdistance$. 314
E.4 The function $childOfType$. 315
E.5 The function $childrenSet$. 315
E.6 The function $childrenSetOfType$. 315
E.7 The function $conditionPart$. 316
E.8 The function $connections$. 317
E.9 The function $containsElement$. 317
E.10 The function $createElementReferenceTree$. 318
E.11 The function $createUniqueElementIdentTree$. 318
E.12 The function $dataTypeDeclarations$. 319
E.13 The function $decompositionParent$. 320
E.14 The function *descendants*. 320
E.15 The function $distance$. 321

E.16 The function *elementReference*. . . . 321
E.17 The function *equals*. . . . 322
E.18 The function *filterFsElement*. . . . 322
E.19 The function *filterOperationsOrProperties*. . . . 323
E.20 The function *filterProvidesRequires*. . . . 324
E.21 The function *filterSet*. . . . 324
E.22 The function *find*. . . . 325
E.23 The auxiliary function *findParentAndChildrenScenarios*. . . . 326
E.24 The auxiliary function *findScenarioGroupMembers*. . . . 326
E.25 The auxiliary function *findScenarioSubtreeMembers*. . . . 327
E.26 The function *findStateGroupMembers*. . . . 328
E.27 The function *flatenTuple*. . . . 328
E.28 The function *functionalSpec*. . . . 329
E.29 The function *invariants*. . . . 329
E.30 The function *operations*. . . . 330
E.31 The function *orderedChildrenOfType*. . . . 330
E.32 The function *partial*. . . . 330
E.33 The function *parts*. . . . 331
E.34 The function *provides*. . . . 331
E.35 The function *requries*. . . . 332
E.36 The function *roleChannelNames*. . . . 333
E.37 The function *roleName*. . . . 333
E.38 The function *rootScenarios*. . . . 334
E.39 The function *scenarioGroups*. . . . 334
E.40 The function *scenarioParent*. . . . 335
E.41 The function *scenarioSiblings*. . . . 336
E.42 The function *scenarioType*. . . . 336
E.43 The function *seekTargetConnections*. . . . 337
E.44 The function *seqNo*. . . . 338
E.45 The function *source*. . . . 339
E.46 The function *sourceRole*. . . . 339
E.47 The function *specialIdentifier*. . . . 340
E.48 The function *standardizedProperties*. . . . 340
E.49 The function *startStates*. . . . 340
E.50 The function *stateGroups*. . . . 341
E.51 The function *target*. . . . 342
E.52 The function *targetConnections*. . . . 342
E.53 The function *targetRole*. . . . 343
E.54 The function *treeAncestor*. . . . 344
E.55 The function *uniqueElementIdentifier*. . . . 344
E.56 The function *containedAspects*. . . . 345
E.57 The function *enclosingAspects*. . . . 346
E.58 The function *endTargetModules*. . . . 347

E.59 The function *exitPoints*. 347
E.60 The function *findEoJrs*. 348
E.61 The auxiliary function *findJoinRelationshipCycle*. 349
E.62 The function *gatherAspects*. 350
E.63 The function *hasJoinRelationshipCycles*. 350
E.64 The function *jrHostingAspect*. 350
E.65 The function *ordering*. 351
E.66 The function *priority*. 351
E.67 The function *serverComponentAssociations*. 352
E.68 The function *targetModule*. 353
E.69 The function *adaptCloneReferences*. 353
E.70 The function *cloneElement*. 354
E.71 The function *createAdditionalExitStateClone*. 355
E.72 The function *createCloneMap*. 356
E.73 The function *findClone*. 357
E.74 The function *findOrderingGroups*. 358
E.75 The function *firstJr*. 358
E.76 The function *gatherJrs*. 359
E.77 The function *identicalElement*. 360
E.78 The function *isPredecessorGroup*. 361
E.79 The function *mappedReference*. 361
E.80 The function *prioritySort*. 362
E.81 The function *removeScenarios*. 363
E.82 The function *sortTargetGroups*. 363
E.83 The function *topologicJrSort*. 364
H.1 Constraint C_3 formulated for the ADORA tool in ICL. 424

Part I

Basics and Motivation of Aspect-Oriented Requirements Engineering

Chapter 1

Introduction

1.1 Motivation

The size and complexity of software systems has been steadily increasing since the beginning of computing. On the one hand, the currently developed systems are orders of magnitude larger and fulfill broader and more complex tasks than decades ago. On the other hand, errors are more likely to occur in large-scale software systems than in a small piece of software, which is additionally amplified by the higher complexity of contemporary computer systems. These errors have been leading to delayed or even abandoned projects, software failures, maintenance problems, and, hence, higher costs than planned.

As a consequence of these problems, the demand for more deliberate development of software began to grow decades ago [Buxt69, Dijk72, Rand96], which finally resulted in systematic and better software engineering methods. Errors in software systems which originate from the requirements phase are especially problematic. This is due to the fact that errors are the more expensive, the later they are discovered in the software process [Boeh81]. This insight has led to the development of various systematic requirements engineering methods which aim at the systematic gathering, managing and documenting of requirements (cf. for instance [Robe99, Davi92, Sawy96, Koto98, Rupp04]). Such a method usually consists firstly of a language that is used to document and communicate requirements between the stakeholders[1] in the software project and secondly of a process which is employed for eliciting, managing and validating the requirements.

The document emanating from the requirements phase is called software requirements specification (SRS) and describes the desired properties of the system to be built. The SRS is crucial, as all following phases in the software process depend on this document. The SRS should satisfy a set of desiderata, such as being adequate, complete, understandable, etc. [Joos99, Davi92]. Obviously, errors in the requirements document should be avoided because they compromise these properties. Nevertheless, there should be an optimal balance between the costs saved by removing errors from the SRS and the costs generated by the creation of the SRS [Boeh81, Glin05].

[1] In this work, role names like *reader of the model, user, stakeholder, customer, etc.* stand for female, as well as for male persons. However, where it is necessary to use pronouns, the male form is employed for the sake of simplicity.

According to [Joos99, p. 3], errors in the SRS are caused by two different problems. First, a *volatile environment* may result in an inappropriate capturing of the requirements, and/or inconsistencies due to changes in the requirements. This problem can barely be avoided and must be accepted by the stakeholders in a project because it is inherently given. The second problem leading to errors in the SRS is the *communication gap* between the various stakeholders in a software project. It is strongly in uenced, amongst other factors, by poor intelligibility of the communicated requirements.[2] In turn, the understandability may be in uenced by the complexity of the requirements.

There are two types of complexity in requirements: problem-endogenous complexity and problem-exogenous complexity. *Problem-endogenous complexity* in requirements results from the problem domain: the requirements and the resulting system are at least as complex as the problem which is solved. Thus, this kind of complexity can obviously not be eliminated [Glin07b]. In contrast, *problem-exogenous* complexity is not caused by the problem itself and can therefore be mitigated or reduced to a minimum with adequate means.

Aspect-oriented software development (AOSD) is a newer research field which has gained a lot of attention in the past few years. It deals with the reduction of the problem-exogenous complexity in software artifacts. The idea of AOSD originates in the field of aspect-oriented programming (AOP), which addresses the proper *separation of concerns* [Dijk82] at the programming level. The code of so-called crosscutting concerns scatters and tangles with the code of other concerns in the implemented system. The scattering and tangling problem results from the lack of adequate means to modularize software [Tarr99], and results in complex and badly understandable code, maintainability problems, and higher costs. Crosscutting concerns cannot be handled adequately by conventional modularization techniques. AOP tries to overcome these problems by introducing additional modularization dimensions, which allows the separation of crosscutting concerns, aiming at resulting systems that are easier to understand and simpler to handle.

The advantages of AOP techniques are manifold [Ladd03, Kicz01b]: the responsibility of code artifacts is defined more clearly due to their better modularization, and therefore, they are easier to reuse and to maintain. The resulting design is clearer and therefore leads to more stable and better maintainable systems. Overall, this lowers the costs of the development and the operation of a system.

However, the problem of crosscutting concerns is not only a phenomenon at the implementation stage. Tangling and scattering effects occur also in the artifacts describing crosscutting concerns at the design, architectural and requirements phases. Hence, the idea of AOP can be transferred to the other phases of the software engineering process, which results in a clearer separation of concerns in those phases, too. AOSD propagates the application of aspect-oriented techniques to artifacts of software engineering stages other than the implementation phase. In particular, the use of aspect-oriented techniques in the requirements phase may result in benefits, as the separation of concerns is achieved very early and the concerns are separated in the following phases. Moreover, it facilitates the traceability of crosscutting concerns. Furthermore,

[2] There are also other factors which contribute to the communication gap, such as the use of a differing semantics in the vocabulary of the stakeholders.

when aspect-oriented paradigms are used at the implementation phase of a software system, aspect-oriented techniques used at the earlier stages reduce or eliminate the clash of different paradigms. However, apart from the advantages, aspect-oriented techniques may also increase complexity due to the strong modularization achieved.

The present work deals mainly with the documentation of software requirements by employing aspect-oriented techniques to decrease the problem-exogenous complexity of the requirements. It also partially sketches the process employed in the requirements phase.

1.2 Gaps in Existing Aspect-Oriented Requirements Approaches

There are many aspect-oriented requirements approaches. However, not all of them solve the problem of the separation of concerns satisfactorily. Moreover, they have deficiencies when used for communicating between the various stakeholders in a software project.[3]

The existing aspect-oriented approaches neglect the simple understandability of requirements. Some of them, such as goal-oriented approaches, use rather abstract concepts which are remote from the problem domain they describe. This may be difficult to comprehend for non-expert stakeholders. Some of the approaches do not use an adequate representation for the artifacts and their relationships.

Other approaches use XML [WWWC06] files for structuring natural language requirements statements. However, an XML representation is not very suitable for communicating with stakeholders.

Some of the approaches use the modeling language UML [OMaG03b] which facilitates the understandability of the represented requirements, because the artifacts are modeled graphically and are easier for non-expert stakeholders to comprehend than abstract concepts, such as goal-oriented approaches. However, the major drawback of UML is that it uses a set of sublanguages which are visualized separately, thus imposing greater demands on the user because a model needs to be integrated piecemeal in the mind to achieve understanding of the whole model. Moreover, in combination with aspect-oriented language constructs, the difficulties in the understandability of a model may be amplified.

Furthermore, some of the examined approaches do not have a readily scalable representation of the artifacts. Some of the approaches also lack support for other characteristics which are desirable for requirements approaches, such as support for easy adjustment of the artifacts.

1.3 Goals of the Present Work

The present work investigates the question of how aspect-oriented techniques can help to close the communication gap by reducing the exogenous complexity and therefore improving the understandability of an SRS. It aims at the development of a comprehensive requirements approach

[3] A detailed evaluation and discussion of the approaches is given in Section 3.2 and Appendix A.

to overcome the problems identified in the existing approaches with a special focus on:

- *Understandability:* The approach should facilitate a good understandability of the communicated requirements for all stakeholders.
- *Concern handling:* It should be able to handle all types of crosscutting concerns adequately.
- *Scalability:* The representation, as well as the corresponding process must be scalable.
- *Other characteristics:* The usual desired characteristics of a requirements specification, such as traceability, consistency, evolution support, should be facilitated as well.

For this purpose, the concept for a language is developed that supports the identification, the handling and the modular description of crosscutting concerns at the requirements and architectural level. The focus of the present work is on the description of crosscutting concerns during the requirements and architectural phase of the software process. To some extent, it also deals with the requirements process needed to identify and separate the crosscutting concerns.

1.4 Contribution

This work demonstrates how an existing requirements modeling language can be extended to be capable of handling crosscutting concerns adequately. It also discusses the pitfalls and problems that are covered by the approach as well as open issues.

The graphical modeling language ADORA [Joos99, Glin02b] innately possesses several of the desired characteristics for an extension and is therefore chosen as a basis for a new aspect-oriented requirements approach. In order to extend the ADORA language with aspect-oriented language constructs, it is first analyzed and the existing language concepts are extended. The language definition created by previous work [Joos99, Glin02b, Xia04, Seyb06a] is unified and integrated in one meta-model framework based on a textual syntax. The corresponding aspect-oriented language constructs are introduced.

Moreover, a weaving semantics for the aspect-oriented elements is developed that concentrates on the transformation from aspect-oriented syntax to the conventional syntax. It allows one to switch between the conventional and the aspect-oriented model. Apart from the model transformations, the work also sketches how the visual representation of the model can be transformed for an optimal understanding. The presented language extensions are implemented in a modeling tool to demonstrate their feasibility. This implementation can also be used as an experimental platform for future research on the aspect-oriented ADORA approach.

Furthermore, a generic evolutionary requirements process is sketched which exemplifies how to employ the introduced modeling language in a software development project.

Apart from the demonstration of how a conventional modeling language can be extended by aspect-oriented constructs, several other contributions are made by the present work.

There is a discussion and a clarification of the terminology used in the field of AOSD. Several important terms in the field have no clear-cut definition which makes it difficult to capture a

common meaning of the terminology in the field. Apart from the terminology, the desirable characteristics of an aspect-oriented requirements approach are derived. State of the art approaches, their strengths and weaknesses are identified and discussed.

The work also investigates the question of how aspect-oriented techniques can improve intelligibility and whether the introduced language elements can simplify the handling of requirements documents. An empirical study demonstrates that, under some circumstances, there are improvements when aspect-oriented artifacts are used.

1.5 Structure of this Thesis

The remainder of this thesis is structured in several parts. The first part introduces the prerequisites for understanding the content of the work. It sketches the field of requirements engineering and motivates the use of modeling techniques for documenting requirements (Chapter 2). The field of aspect-oriented software engineering is introduced (Chapter 3), and then motivation for a new requirements modeling approach is given (Chapter 4). Moreover, the ADORA modeling language, which is used as a basis of the aspect-oriented approach, is introduced (Chapter 5), analyzed and defined in detail (Chapter 6).

The second part of this thesis presents the proposed aspect-oriented approach, which consists of a graphical modeling language definition (Chapter 7), a weaving semantics (Chapter 9), and a process (Chapter 10). Furthermore, the visual abstraction mechanisms which can be applied to the newly introduced syntax elements are outlined (Chapter 8).

The third part deals with the feasibility of the approach and its validity. To exploit the advantages of ADORA, the approach strongly relies on tool support. The features of the tool prototype, the meta-model implementation, the realization of the constraint checking mechanism as well as the weaving engine are outlined (Chapter 11) in the third part. The subsequent chapter presents the results of an experiment (Chapter 12) which tested the approach for an improvement of the efficiency and effectiveness of the understandability of aspect-oriented models compared to conventional models.

The fourth part (Chapter 13) summarizes the present work and compares the presented approach with other existing aspect-oriented requirements approaches. The scientific contribution is discussed, open issues are identified and a preview of future work is given.

The appendix contains various additional materials which complements the main text. A detailed discussion of the current state-of-the-art requirements approaches can be found in Appendix A. In Appendix B, the context-free syntax of the ADORA language is provided, and in Appendix C, the mapping rules between the graphical language and the textual grammar are given. A full example of a textual model, which illustrates the syntax, is given in Appendix D. Moreover, the appendix also contains a catalog of functions that are used for expressing properties of the ADORA model syntax trees (Appendix E). Appendix F contains the formal language constraints for the introduced aspect-oriented elements. In Appendix G, the weaving semantics of the aspect-oriented extension is discussed formally, and, finally, Appendix H discusses some facets of the tool implementation in more detail.

Chapter 2

Basics of Requirements Engineering and Modeling

This chapter gives a brief introduction to the field of *modeling* as well as to *requirements engineering*.

The development of systematic requirements engineering dates back to the late 1960s. At this time, which is also known as an era of software crisis [Buxt69, Dijk72, Rand96], software projects encountered more and more problems because the systems developed in those days were becoming bigger and more complex than the previously. As a consequence, the demand for more systematic and deliberate approaches to software development increased, which finally led to numerous approaches and process models. Examples of such process models are the waterfall model [Boeh81, Royc70], the spiral model [Boeh88], incremental process models [Basi75] and agile software development [Cock02]. All software process models have the following phases in common:

1. **Requirements Phase.** In the requirements phase, the software requirements are systematically elicited, gathered and managed. Typically, the requirements phase usually does not deal with technical problems but with the problems of the application domain.

2. **Architectural Phase.** The architectural phase aims at creating a coarse organizational structure of the software. Moreover, first decisions about its technical realization are made during this phase.

3. **Design.** The design phase results in a detailed description of the software.

4. **Implementation.** The implementation concerns the coding, i.e., the realization of the system's executable description.

5. **Integration and Deployment.** In the integration and deployment phase, the software parts are assembled into an executable system and deployed within the test or production environment.

6. **Maintenance.** During the maintenence phase, the software is adapted to new requirements arising from the a volatile environment of the software system [Lehm97].

Depending on the type of software process, the phases described above may be performed sequentially or iteratively. Furthermore, there are several tasks which can be found throughout all of the phases.

i. **Testing.** Software tests [Bind99, Myer04] have mainly two purposes. First they are used to verify the correctness of a software implementation against a set of test cases derived from the software specification. Second, tests are employed to validate the software against the requirements of the software stakeholders.

   Verification tests can be classified into Unit tests, integration tests, system tests, and regression tests [Glin05]. Apart from the verification tests, validation tests or so-called *acceptance tests* are used to show that all customer requirements are satisfied. They are usually executed at the end of a software project or before a deliverable is deployed into a productive environment.

ii. **Documentation.** The documentation of a piece of software is used to communicate knowledge. Documentation is created in every phase of the software process. The documents produced during the software process include, for example, the requirements specification document, the architectural descriptions, as well as the comments in the code.

The approach presented in this work mainly deals with the specification document created during the requirements phase. Therefore, the remainder of this chapter gives a brief introduction to the field of requirements engineering (Section 2.1), the software requirements specification document (see Section 2.2), and the use of modeling techniques during the software process (Section 2.3) used as a way of represent a software requirements specification.

2.1 Requirements Engineering

Requirements engineering is a natural consequence of the need for systematic software engineering. It concerns the methodical elicitation, gathering and management of *software requirements* and has to be supported by an adequate process. In the following, a definition of software requirements and a brief discussion of the requirements engineering process and management is given. A deeper discussion of the topic can be found in various sources, such as [Koto98, Somm97, Robe99, Davi92, Rupp04].

2.1.1 Functional and Non-Functional Requirements

A software requirement is defined as *a condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, speci cation, or other formally imposed documents* [IEEE90]. There exist *functional requirements* (FR), as well as *non-functional requirements* (NFR). Both terms have been given various definitions in the literature.

A functional requirement is commonly defined as the description of *what the system should do* [Somm97]. Thus, a FR describes the output of a computation that is performed by a system which is in a particular state. The output may be stimulated by an external input, or, if the system is proactive, the system can itself produce internal stimuli.

In contrast to the definition of functional requirements, there is no consensus about the term *non-functional requirement* [Glin05, Glin07a] although there has been an extensive discussion about its definition, e.g., in [Davi92, Koto98, Lams01, IEEE98]. For example, [Somm97] defines NFRs by a specific characteristic common to all NFRs: an NFR *puts restrictions on other requirements* (cf. [Somm97, p. 7] and [Koto98, p.187]). The restriction relationship is unidirectional, i.e., the constrained requirement cannot in uence the way in which it is restricted by the constraining NFR. However, this characteristic is not unique[1] to NFRs, and therefore, it is not precise enough to be used as their definition.

This work rather sticks to the definition given in [Glin07a] which overcomes the divergent use of the terms employed in the field. The term *non-functional requirements* is defined according to a consistent taxonomy which provides a set of classification rules. A requirement is simply assigned to a given category in the case it matches the corresponding rule. NFRs belong to the categories *attributes* and *constraints*:

- *Attributes*:
  - *Performance requirements* are requirements concerning the efficiency of a system. Examples are *timing*, *speed*, *memory usage*, and *throughput boundaries* of the software.
  - *Speci c quality requirements* deal with the quality of a system. For instance, the demand for a certain level of *reliability*, *usability*, *security*, *availability*, *portability*, and *maintainability* fits this category.
- A *constraint* is *any other* restriction which is not an attribute or a functional requirement. For example, *physical*, *legal*, *cultural*, *environmental*, *design & implementation*, or *interface* restrictions belong to this category.

2.1.2 Requirements Process

The *requirements engineering process* consists of the four tasks: *elicitation*, *analysis and negotiation*, *documentation* and *validation* [Koto98]. It is usually iterative or evolutionary, or, for the case of small and simple systems, it can be linear. The process needs to be customized for the type of software being developed [Glin05].[2]

Volatile environments may call for changes in already negotiated software requirements, which causes software evolution [Lehm97]. The changes in the requirements for a piece of

[1] There are also functional requirements which can restrict other (functional) requirements, as shown in Section 3.1.9.

[2] For example, off-the-shelf software for the mass market needs another requirements process than custom software.

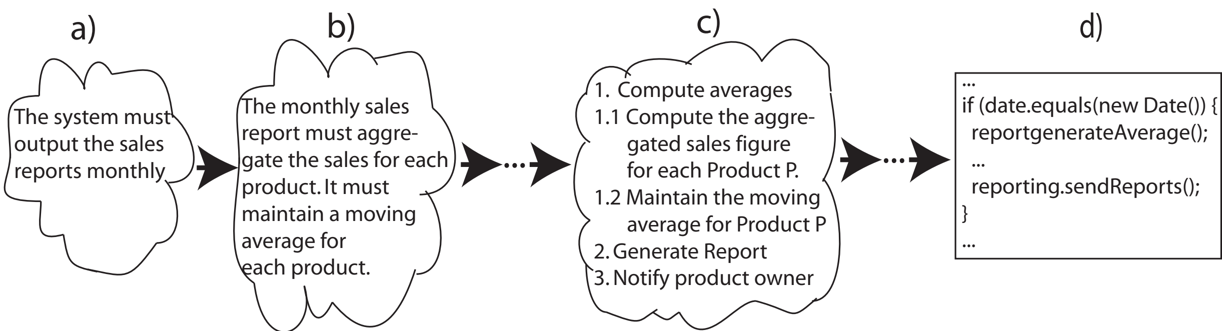


Figure 2.1: An example for the refinement of a high-level functional requirement to executable code. The cloud shapes denote functional requirements on an abstract level and the arrows represent refinement steps. The rectangle indicates a concrete piece of code resulting from the refinement process.

software must be managed with an adequate *requirements management process*. It is performed concurrently to the actual requirements process. It concerns the identification, the change management and the traceability of the requirements acquired during the requirements phase.

Elicitation, Analysis and Negotiation of Requirements

The initial discovery of requirements is called elicitation [Koto98, p. 32] and is mostly intertwined with the analysis and negotiation of software requirements. At a first elicitation round, requirements are on a highly abstract level. During the analysis and negotiation phase, the software requirements are analyzed, refined, and prioritized, and any inherent con icts are resolved. As soon as a requirement is fixed, it can be used in the subsequent phases of the software process, i.e., the architectural phase, the design, and the implementation phase.

There are several different approaches to conducting the requirements process, such as goal-oriented, e.g., [Dari96], [Chun00], or view-point-oriented approaches, e.g., [Sawy96]. However, the following discussion is not focused on a specific approach, but rather on *what* is done during the elicitation phase.

Re ning Functional Requirements. An FR is *re ned* towards a more concrete and detailed description until an adequate level of detail is reached. The level of detail depends mainly on the risk that a person might misconceive the requirements. Thus, parts with a high risk need a more thorough and a more detailed specification than parts with a low risk.

In the phases of the software process which follow on the requirements stage, a FR is gradually refined to a piece of runnable code. Figure 2.1 exemplifies this refinement by means of showing the resulting artifacts for a sales application. The refinement steps between (a)–(c) are performed in the requirements phase. After an initial elicitation, the problem is formulated as high-level functional requirements (a). After a refinement step, it is elaborated as shown in

(b), and some refinement steps later the initial high-level FR is refined to a very detailed requirement statement. The resulting requirement (c) can be expressed for example as *use case* [Jaco92, Cock01]. Subsequent refinement steps in the software process result in an architecture, a design and finally, in the implementation (d) of the corresponding FR.

Operationalizing Non-Functional Requirements. NFRs are *operationalized* during the software process, which means that they are incrementally refined towards more concrete requirements. Initially, NFRs are stated as high-level requirements. In the following steps they are concretized, i.e., additional information describing the NFR in more detail is accumulated. Moreover, a concretization step of a requirement can result in the creation of new sub-NFRs. This is illustrated by Fig. 2.2 (a), the (high-level) NFR "The system must be secure" leads to several sub-NFRs. In turn, the sub-NFRs are refined further, until the operationalization stops. There are three possible types of results from the an NFRs operationalization [Meie07].

A result of type (i) is *a piece of code in the system s implementation*. A high-level NFR which ends up in this type of outcome contributes directly to the functionality of a system. An example of such an NFR is any security requirement which demands an authorization before a particular function of a system is executed. This situation is exemplified in Fig. 2.2 (a), where a high-level security requirement is operationalized to an authorization mechanism. The resulting artifact is a piece of code, denoted by the rectangle shape. The code shown implements an authentication/authorization mechanism.

An artifact of type (ii) is a *decision which in uences the architecture, the design, or the implementation of the software*. Hence, in this case the NFR is not traceable to a specific piece of code, but it in uences *how* the functionality of a system is *organized* and *executed*. Thus, the NFRs which result in these kind of artifacts in uence *how* the system performs. For example, performance requirements, such as memory or throughput boundaries can result in this type of artifact. The operationalization of an NFR to an outcome of type (ii) is illustrated in Fig. 2.2 (b), where a maintainability requirement is operationalized. The resulting outcome is represented as a triangle and denotes an architectural decision.

An outcome of type (iii) is a *decision which does not contribute directly to the functionality of a system or how it performs*. This type rather in uences the course of a software process or the form of the resulting software artifacts. For example, an NFR which demands a high maintainability of the software may result in a decision to use a specific coding guide line for method comments. This is exemplified by the operationalization in Fig. 2.2 (b). The resulting decision is denoted by the circle shape. Another example of a type (iii) decision can be seen in Fig. 2.2 (a), where the security NFR leads, apart from the outcome of type (i), also to one of type (iii).

Documenting Requirements

The requirements found in the elicitation, negotiation, and analysis phase must be documented appropriately, which is necessary in order to allow communication with the stakeholders. The requirements specification document finally results from the process, and its form strongly depends on the type of the developed software system and the risks involved. The following phases

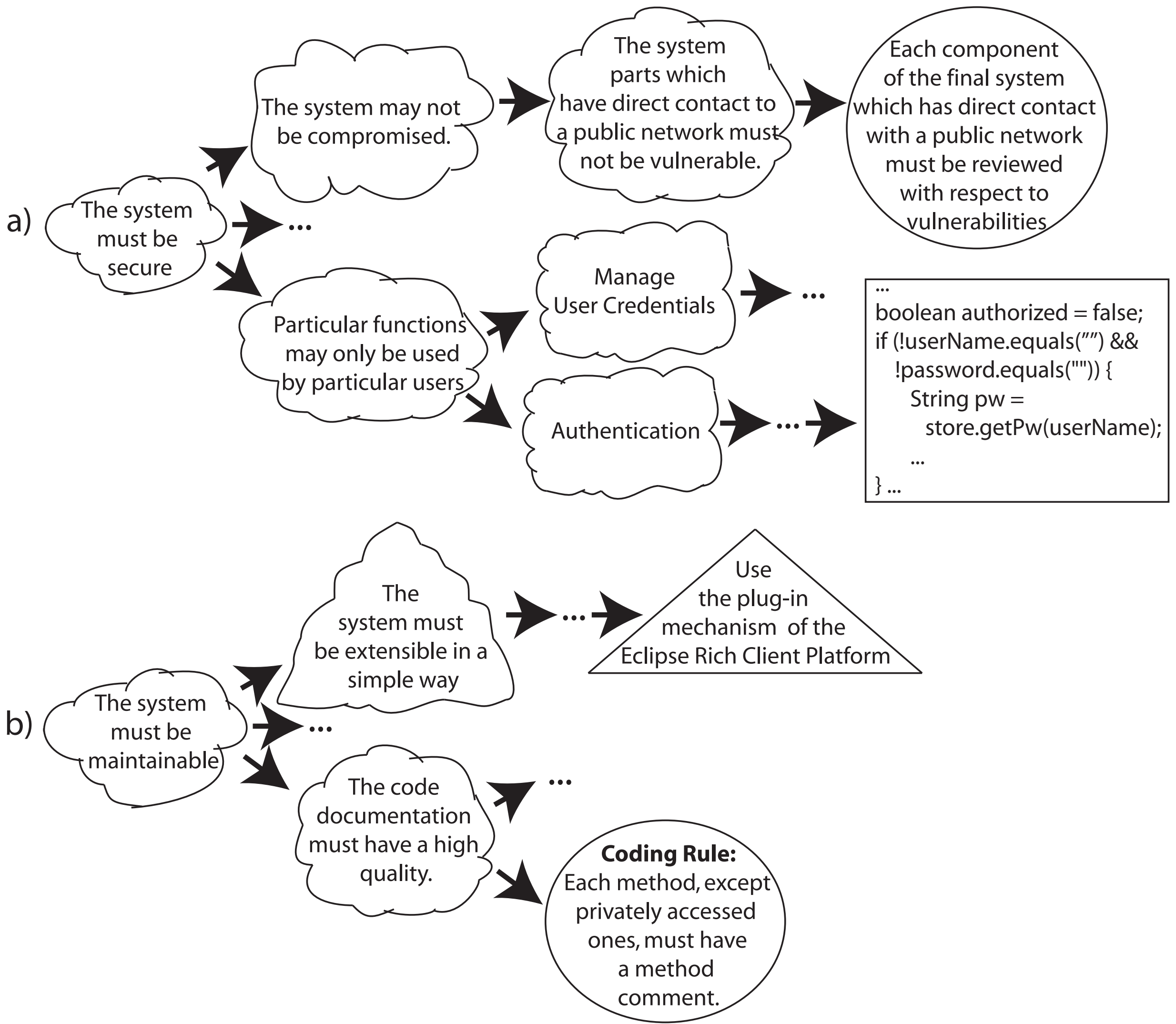


Figure 2.2: Example for the operationalization of non-functional requirements. Cloud shapes denote abstract requirements and the arrows indicate the operationalization steps. Figure (a) shows the operationalization of a security NFR. Several sub-NFRs result from the initially given NFRs, which are then operationalized. Two resulting artifacts are illustrated: the circle shape denotes an outcome of type (iii), the rectangle shape denotes an outcome of type (i), i.e., concrete code in the system. Figure (b) illustrates the operationalization of a maintainability NFR which results, besides an outcome of type (iii), in an outcome of type (ii), i.e., an architectural decision.

in the software process, such as the architecture, design, and the implementation are based on this requirements document, and it is also employed for the creation of unit, integration, system tests, and acceptance tests. The documentation of requirements is a focal point of this work and will therefore be discussed more extensively in Section 2.2.

Validating Requirements

The requirements validation phase is intended to reveal any problems in the requirements document, which may lead, if not discovered, to problems and unnecessary feedback cycles in later stages. Therefore, the software specification is checked for completeness, consistency, and whether the needs of the stakeholders are adequately reflected. Consequently, a validation always involves stakeholders.

There are static and dynamic means for doing a validation. Static means are for example peer reviews [Faga86], or manual desk tests of artifacts. They involve a *manual check and interpretation* of the requirements artifacts. In contrast, a dynamic validation is based on machine-processible artifacts. Means for dynamic requirements validation are simulation and animation techniques (e.g., [Seyb06a]), model checking (e.g., [Chec03]), or the use of prototypes [Floy84]. If a requirement changes (see below), usually a revalidation is done to ensure the completeness and consistency of the other requirements.

Requirements Evolution, Requirements Management and Traceability

The requirements evolution is mostly caused by a volatile context in which the software is built or used and results in a change of the software system [Lehm97].

The term *evolution of requirements* has no clear definition [Feli02, p. 16] in the literature. In this work, the *evolution of requirements* denotes the changes or the extension of software requirements which have already been formally negotiated and fixed during the requirements process. These changes can either occur at any time during the software development process, or during the maintenance phase of the software process when the system has already gone into service.

For an appropriate management of the requirements and their changes, as well as the maintenance of the consistency in a software specification, each requirement must be traceable [Koto98]. Cross-references as well as the origins must be determinable, i.e., each requirement must be uniquely *identified* and stored in an adequate way [Koto98].

2.2 Requirements Document

The requirements specification document records the requirements found and agreed upon during the elicitation, analysis and negotiation phase. There are different characteristics which influence the quality of a software requirements specification document:

- **General Quality Characteristics.** There is a set of different quality characteristics which

must be satisfied by requirements specification documents, such as completeness and consistency.

- **Degree of Formality and Detailedness.** The degree of formality and the level of detail of a requirements specification must be adequately adapted to the risks involved in a software project.

- **Constructive vs. Descriptive Speci cation.** Requirements can either be specified constructively or descriptively. A constructive description partially shows how a system performs a functionality, whereas a descriptive specification considers the system as a black box.

2.2.1 General Quality Characteristics

In general, a requirements specification should satisfy the following characteristics to ensure an adequate quality [Joos99, Davi93, Davi92]:

- **Problem and Risk Adequate.** A requirements specification must document the demands of a customer adequately according to the given problem, the risk of system failure, and the risk of a misconception.

- **Complete.** The specification document must describe *all* requirements as far as they are needed to unambiguously specify the system (see also below in Section 2.2.2).

- **Consistent.** The requirements described in a requirements specification must not contradict each other. For example, contradictions may result from the dissimilar viewpoints of different stakeholders or from changes in the requirements. In the first case, the con icts must be resolved during the requirements analysis and negotiation process. In the latter case, the requirements management process must enforce the corresponding quality measures, such as reviews and consistency checks.

- **Unambiguous.** The requirements of a specification document must not be interpretable in unintended ways.

- **Veri able.** A software system must be verifiable against its specification. Depending on the degree of formality, the verification can either be done manually or automatically. Formal specifications can also be verified automatically.

- **Understandable.** Specification documents should be as easy to understand as possible. Therefore, they should stick to understandable representations. However, understandability depends strongly on psychological and cognitive factors which may vary from individual to individual. Furthermore, the form of a specification cannot reduce problem-endogenous complexity: a specification document is at least as complex as the problem described [Glin05].

- **Traceable.** A requirements document is subject to change. Therefore, its content must be actively managed. The dependencies of a requirement which needs to be changed must be easy to find in a software specification. A requirement's origin should be easy to determine and traceable through the whole software life cycle.
- **Modi able.** For the sake of ease of alteration, a requirements document must be well structured. The redundancy in a document should be minimal, as redundancy compromises the consistency when changing parts of the document.[3]
- **Independent of Design.** A good requirements document must not anticipate the architecture, design or implementation, unless there are explicit requirements concerning those elements.
- **Mappable/Feasible.** A good requirements document should be easily transferable to later stages in the software process. The paradigms used for the specification should match the paradigms in later stages and provide instructions on how to map the requirements artifacts to later stages.

2.2.2 Degree of Formality and Detailedness

There are two different risks involved when using or developing a software system. First, failures may occur during the operation of a system. Failures are costly or even hazardous to the environment of the software system.[4] Second, there is the risk of a misconception during the development of the software system, which may result in a late schedule and higher costs or even in the failure and abandonment of the project. The means for handling both risks are an adequate degree of formality and detailedness of the software specification.

Degree of Formality A software requirements specification document can be informal, formal, or semi-formal. The creation of formal specifications demands a greater effort than the creation of informal specifications. To be economical, the degree of formality should be adequate to the risk of failure involved by a system. For example, a system with a high risk of failure should be based on a more formal specification than a system with a low risk.

An informal specification has no fixed syntax and semantics. It can differ in the way it is interpreted by a reader of the specification. The most often used informal means is natural language which can be written and understood by any stakeholder of a software project, without any training. However, natural language tends to be imprecise and ambiguous.

In contrast, a formal specification language or method, such as algebraic specification of abstract data types [Gutt77], the Vienna Development Method (VDM) [Bjor78], *Z* [Spiv89], or the Object Constraint Language (OCL) [OMaG06] are more precise, as they have a well-defined

[3] However, redundancy can also deliberately be used to detect problems in a requirements specification (cf. [Joos99]).

[4] See the Therac-25 case in the 1980s [Leve93], where several people were killed as a consequence of software failures in the radiation device used.

syntax and semantics. However, formal specifications are costly and more difficult to understand [Joos99]. Therefore, formal methods are usually employed in describing systems or parts of a system which involve a high risk of failure.

Semi-formal specification methods are a compromise between formal and informal methods. According to [Seyb04b], a specification is *semi-formal* if it has at least one language element that has an imprecise semantics, or if it fulfills one of the following conditions: (i) it contains at least one syntactically correct[5], but semantically not well defined construct, or (ii) the specification is intentionally incomplete, or (iii), the specification contains at least one syntactically correct element which is (unintentionally) wrongly used, i.e., which has a wrong semantics.

There are different degrees of semi-formality, there are even languages which support a variable degree of formality, such as ADORA [Joos99, Glin02b]. The variable degree of formality can be computed [Seyb06a, p. 96] and means that the more formal elements are contained in a semi-formal specification, the more formal the specification is. For example, the use of structured natural language is more formal than (informal) unstructured natural language.

Detailedness. Apart from the degree of formality, a software requirements specification document can vary in its detailedness. An economical specification process should not over-specify requirements. Software parts which involve a small risk of misconception or ambiguity during the development of the system do not need to be detailed. These parts of a specification document are also known as *implicit requirements*. Implicit requirements are based on the common sense of the stakeholders.

2.2.3 Constructive vs. Descriptive Speci cation

A specification can define the problem domain either descriptively or constructively. A specification is *descriptive* if it handles the system as a black box that defines only *what* the system does. It specifies the initial state and the input of the system as well as the resulting output. As a descriptive specification does not describe *how* the output is computed, it is independent of a concrete realization of the system and does not therefore anticipate an architecture, design, or implementation.

In contrast, a *constructive* specification not only does describe *what* is done by the functionality of a system, but also *how* this is done. The requirements are structured logically according to the dependencies and the interrelationships between the requirements in the modeled application domain resulting in a high-level logical architecture of the requirements. As the resulting constructive description partially anticipates the realization of a system, it is not solution-neutral. It narrows the solution space and may result in a suboptimal solution.

However, compared to descriptive software requirements documents, especially large specifications can be easier to understand if they are constructive (cf. [Joos99, p. 25] and [Davi92, p. 213ff]).

[5] That means a given sentence in a language is recognized by the grammar of the language.

2.3 Modeling

Models are an important means to describe whole software systems and their parts at any stage and also for the description of a software requirements specification. In software engineering, the term *model* usually implies the visualization of a graph structure.[6] To get an overview of modeling, a brief discussion about the general model theory and the graphical modeling of software is given in the following.

2.3.1 General Model Theory

According to the general model theory of Stachowiak [Stac73], a model is a construct which represents an *original* that can either be a *thing of the real world* or *another model*. A model contains a set of entities consisting of attributes. An attribute is either a property, a relationship or an operation. The purpose of a model is to communicate the original in a simplified way and thereby to accentuate specific properties of the original. Each model has three different characteristics [Stac73]:

1. **Mapping.** The entities and attributes of the original are mapped to the model by specific mapping operations. Moreover, the original can be mapped to many different models.
2. **Abstraction.** Not all of the original entities and attributes of the original need to be mapped to the model. Entities as well as attributes from the original may have no counterpart in the model. Vice versa, models may have entities and attributes which have no counterpart in the original. However, the model should only contain elements which are relevant for the communicated information (cf. pragmatics below).
3. **Pragmatics.** A model is used in a specific context and serves a specific purpose. Therefore, one type of model may be better suited for a specific goal than another type of model.

The process of creating a model is called *modeling*. The person who is creating the model is named *modeler*, and the person who is reading it is called *the reader of the model*.

2.3.2 Languages for the Visual Modeling of Software Systems

Models can be used at every abstraction level and at every phase in the software process.[7] Model-driven software development [OMaG03a, Pool01] even tries to use models as *the* central artifacts throughout the whole software process. Software requirements can be described by models too, either to complement particular parts or to describe the whole requirements specification. Requirements are usually modeled constructively.

[6] However, as given by the model theory definition below, a description of a system in natural language can also be seen as model. In the remainder of the work, the term *model* refers to graph visualizations, unless otherwise stated.

[7] A broader discussion and a comparison of modeling languages and their use can be found in [Joos99].

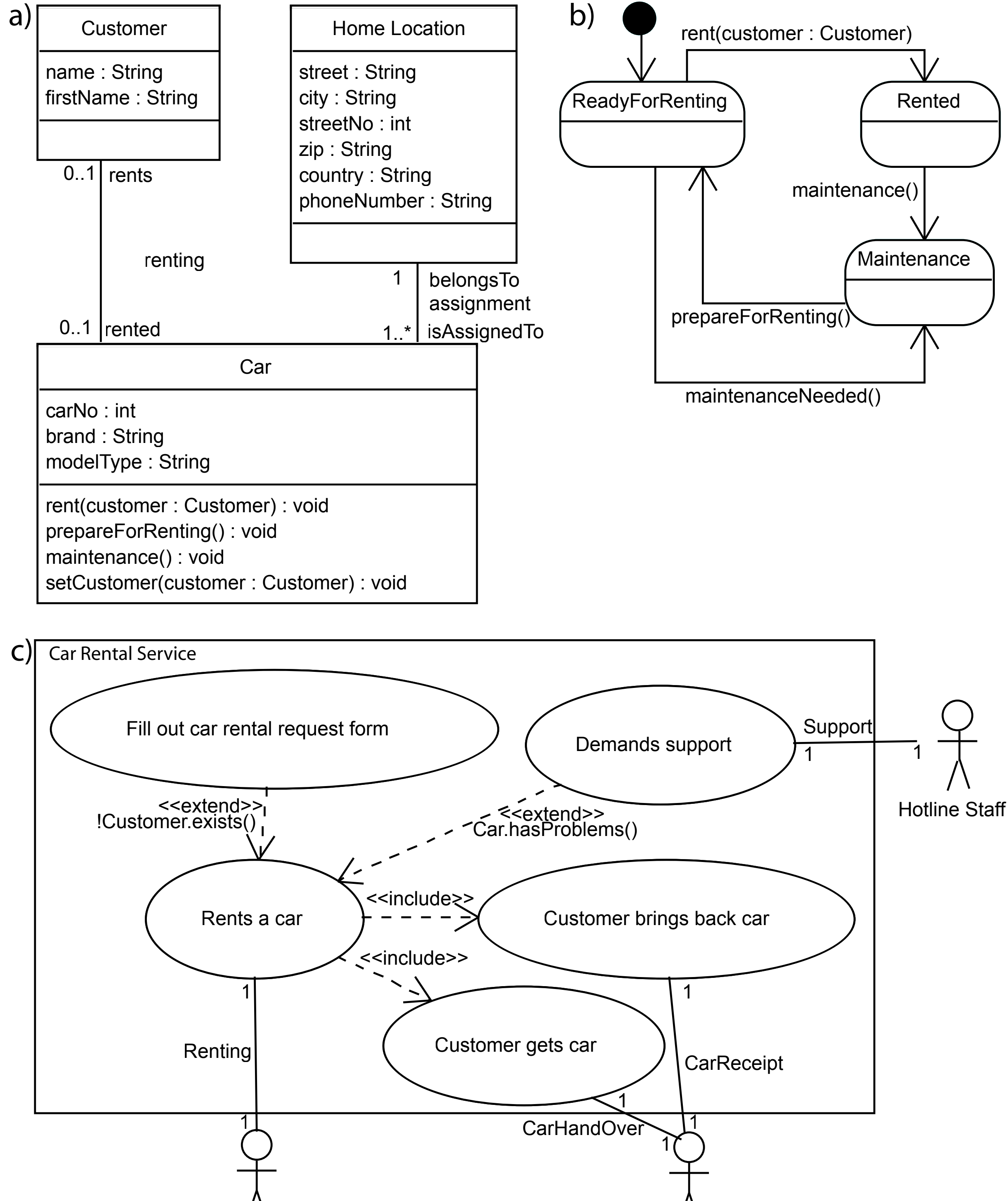


Figure 2.3: Example model facets of a UML model. The figure shows a part of a car rental agency system. Figure (a) contains the domain class model, i.e., the static structure of the system, (b) shows the behavior for a car, and (c) illustrates a use case diagram denoting the interrelationships between use cases.

Depending on the modeling language, one or more facets of a system can be described. A facet is a particular property or characteristic of the system that is a focus of interest. For example, such a facet is the description of the behavior, the relationships between different elements, or the organization of the system.

There are two different types of modeling languages: problem-specific and general-purpose modeling languages. Problem-specific modeling languages allow one either to model just particular facets of a system, or a restricted application domain. For example, finite state machines [Cap93, p. 133ff], statecharts [Hare87], and petri-nets [Petr62] can only be used to model the behavioral facet. The language SysML [OMaG07] is a representative of domain-specific languages. It is mainly used for systems engineering.

In contrast to problem-specific modeling languages, general-purpose modeling languages can be applied in various application domains and allow an extensive description of different system facets. The best known example is the Unified Modeling Language (UML) [OMaG03b, Rumb05] that consists of a set of sublanguages.

Each sublanguage allows one to model a specific facet of the system. For example, state- and activity diagrams allow one to model the behavior, class diagrams are used to specify the organization of the classes, and the use case diagrams give an overview of the dependencies between existing use cases in the system. In the current language version 2.0 [OMaG03b], there is a total of thirteen different sublanguages/facets in UML.

Figure 2.3 illustrates what some parts of a requirements model for a simple software system for a car rental agency can look like in UML. The figure describes a simple car rental agency system. Three different facets are given: Figure 2.3 (a) shows the class structure of the system with the three classes *Customer*, *Car*, and *Home Location*. In Fig. 2.3 (b) the behavior of the class *Car* is given as a state diagram[8]. Finally, Fig. 2.3 (c) shows a use case diagram which describes the system behavior from a user point of view.

Modeling languages can be classified into *non-integrated* and *integrated* modeling languages. A non-integrated modeling language is mostly based on a set of decoupled sublanguages, which is the case for the UML. The reader of a non-integrated model has to integrate in mind the system as a whole, which can be cumbersome even for small models. Furthermore, a non-integrated modeling language may cause unintentional inconsistencies, as it contains more redundancy [Joos99].

In contrast, an integrated modeling language visualizes all system facets in one common model [Joos99, Section 5.2.1]. Therefore, it eases the interpretation for the reader of the model, as the model does not need to be integrated progressively in mind. The data ow diagrams of the Structured Analysis Method [DeMa78] and the ADORA language [Joos99, Glin02b] are examples of integrated languages.

[8]State diagrams are a variant of a statechart [Hare87].

Chapter 3

Aspect-Oriented Software Development

In the last few years a new research field called Aspect-Oriented Software Development (AOSD) has emerged. It had its origin in aspect-oriented programming (AOP) which tackles the problem of so-called *crosscutting concerns* at the programming level. Crosscutting concerns cannot be handled adequately with conventional modularization techniques, which leads to a violation of the separation of concerns principle.

The improper separation leads to a *tangling* of the crosscutting concerns with other concerns and their *scattering* over several modules in the software. The scattering and tangling may result in redundant pieces of code[1] scattered throughout a piece of software. Therefore, the understandability and maintainability of the code may be affected. AOP tries to overcome this deficiency by introducing a new way to modularize the code.

The problem of crosscutting concerns not only manifests itself during the programming phase but rather throughout the whole software life cycle. Artifacts at the design, architectural, or the requirements phase may also be affected by crosscutting concerns. For example at the requirements stage, a use case description belonging to one concern may be tangled with the use case description of another concern. Therefore, it is reasonable to pursue a separation of concerns at all stages of the software process. AOSD deals with the introduction of the corresponding techniques at all phases of the software life cycle.

This chapter provides an introduction to AOSD. Section 3.1 presents the fundamental terminology and concepts. The most important and in uential approaches in the field are brie y discussed in Section 3.2. Section 3.3 provides a critical assessment of AOSD, and finally, Section 3.4 summarizes and discusses the chapter.

3.1 Fundamental Terms and Concepts of AOSD

The knowledge of the terminology of the research field, such as concern, crosscutting, etc., is crucial to the understanding of the topic. In the following the terms as well as an overview of the problems solved by AOSD are treated brie y. A broader overview can be found in [Bono04].

[1] The code pieces are not necessarily equal but similar, i.e., they fulfill the same kind of task.

3.1.1 Concerns and Separation of Concerns

The term concern was first used by Dijkstra in [Dijk82] and [Dijk76]. A concern is an abstract concept in software engineering which describes a specific purpose or interest in a software problem. There are various definitions of the term *concern* in the literature [Bono04]: For example, sometimes a concern is seen as *interest in the system* [IEEE00], or it is defined either as a *requirement* or a *viewpoint* [Ladd03]. However, as the concepts of aspect-orientation can be applied to any stage in the software process (cf. [Gray01]), the concept of the term *concern* needs to be more general.

This work relies on the definition for the term *concern* given by [Stan02, Rose04]: *a concern is any matter of interest in a software system* which deals with the fulfilling of one particular purpose in a software system. Hence, the concept of a concern can be distinguished from the following concepts:

- A concern is not a *detailed requirement*, but it manifests in one or more detailed requirements. Nevertheless, a concern can be seen as a high-level, abstract requirement, i.e., before any refinement of the requirement is done.[2]
- A concern is not a *viewpoint* in the sense of a viewpoint approach [Fink92] because it is not seen from a stakeholder's perspective. A concern is rather the common intersection between two or more viewpoints on the same matter in the system.
- A concern is not a module of a software system, as a module is a concrete artifact. However, a concern can be described by one or more modules (see Section 3.1.3).

The *separation of concerns* is a basic principle in software engineering and the key issue of AOSD. Dijkstra mentions this concept in an essay from 1974[3] [Dijk82]:

> ...We know that a program must be correct and we can study it from that viewpoint only; we also know that it should be efficient and we can study its efficiency on another day. ...But nothing is gained – on the contrary – by tackling these various aspects simultaneously. It is what I sometimes have called "the separation of concerns".

Based on the definition in [Ossh01], the separation of concerns is the ability to keep the concerns of a piece of software and its descriptions separate, i.e., to describe and manage them independently from the other concerns.

[2] See also Section 3.1.9.

[3] The essay was not published until 1982. However, the concept of the separation of concerns was discussed also in [Dijk76].

3.1.2 Crosscutting Concerns vs. Core Concerns

Two concerns are overlapping or crosscutting if some parts of them deal partially or fully with the same matter. Even though the term *crosscutting* is used in every paper about AOSD approaches, attempts to define it are rare. One of the definitions can be found in [Rose04] where the term *crosscutting* is seen as a relationship between two elements where one element in uences or restricts the other. Having a look at different approaches in AOSD regarding the usage of the term crosscutting, e.g., in [Kicz01a, Ossh01, Lieb01, Clar05], reveals that this definition is plausible.

Hence, in this work *crosscutting* is defined as a relationship between two concerns A and B where A in uences or restricts B and B has no control on the way in which A is doing this [Meie07]. The crosscutting concern A is called the *originator* of the crosscutting relationship, whereas the crosscut concern B is called the *target*, which is simply denoted as A *crosscuts* B. In contrast to crosscutting concerns, a so-called *core concern*, which is called sometimes *conventional* or *core concern* in this work, is not the originator of a crosscutting relationship.

A (crosscutting) concern can crosscut any other concern, i.e., the target may be either another crosscutting concern or a core concern. If a concern A crosscuts concern B, and B, in turn, crosscuts concern C, A is said to crosscut C transitively. However, a concern is not allowed to crosscut itself, neither directly, nor transitively, as this is not meaningful.[4] A crosscutting concern is called *strictly crosscutting* if it has at least two different targets which are directly, i.e., not transitively, crosscut. In many approaches, a concern is only seen as crosscutting if it impacts more than one target. Nevertheless, for a proper separation of concerns, even non-strictly crosscutting concerns should be handled as such.

Figure 3.1 exemplifies the concept of *crosscutting concerns* and *core concerns*. The dashed boxes denote concerns, the solid boxes describe artifacts representing one or more concerns, and the dashed arrows are crosscutting relationships. Figure 3.1 (a) shows the concerns A, B, C, D, and E which overlap in some parts of the concerned matter. A overlaps with B and D, as well with C. Moreover, B also overlaps with E. Separating them from each other when describing them with modular artifacts (cf. the definition below) results in minor parts that are common between different artifacts. These parts can be assigned to one artifact which is dominated by one particular concern. This situation is shown in Fig. 3.1 (b).[5] The dominating parts are core concerns, the minor overlappings belong to the crosscutting concerns.

In order not to lose the connection between the concerns that are dispersed over several artifacts, the corresponding crosscutting relationships are introduced instead, as shown in Fig. 3.1 (c). C, D, and B are crosscutting concerns, because they have at least one outgoing crosscutting relationship, whereas A and E are core concerns that are crosscut. B is strictly crosscutting, as it has more than one direct target. In contrast, D and C are not strictly crosscutting. A is crosscut by B, D, and C, whereas E is crosscut by B. The concern C is crosscutting A transitively over B.

[4] Hence, no cycles in a path of crosscutting relationships are allowed.

[5] The assignment of the overlapping parts is done haphazardly in this case.

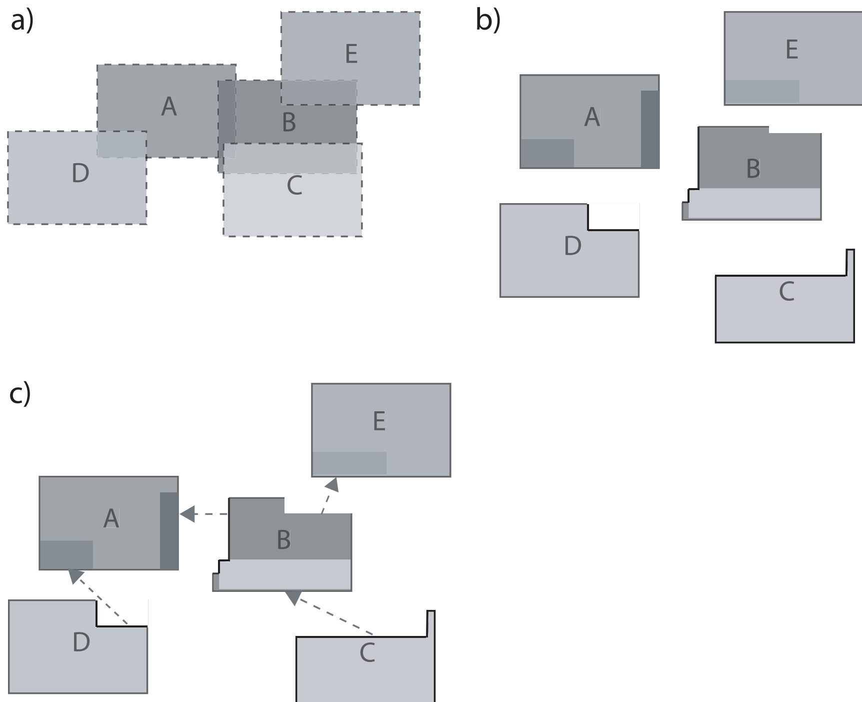


Figure 3.1: Illustration of the crosscutting concerns and the crosscutting relationship between concerns. The dashed rectangles denote concerns, the solid rectangles modular artifacts. In (a), the concerns A, B, C, and D are overlapping. Separating them when building artifacts in (b) results in the assignment of the overlapping area to one artifact of the corresponding concern and to an introduction of a crosscutting relationship in (c) between those artifacts. The relationship indicates that two concerns have a common crosscutting part. The crosscutting relationship is represented by dashed arrows.

3.1.3 Documenting Concerns in Software

The concerns of a software system normally become manifest in various types of software artifacts, such as requirements statements, parts of a software architecture, modules in the implementation[6], or even the user manual. The separation of concerns should be guaranteed in order to ensure an easy understandability of the documented system. This can be achieved by a description of the system that uses *modular artifacts* and also allows the description of *crosscutting relationships*.

A *modular artifact* is a description of a contiguous problem, bounded by boundary elements having an aggregate identifier.[7] A modular artifact should have a high cohesion[8].

[6]The implementation is a special type of document which is an executable description of the problem to be solved.

[7]This definition is based on the one for code modules given by [Scha96].

[8]The concept of cohesion for code modules is well known [Scha96]. However, the concept modular artifacts is more general and therefore the term cohesion can also be applied to a general modular artifacts. A general modular

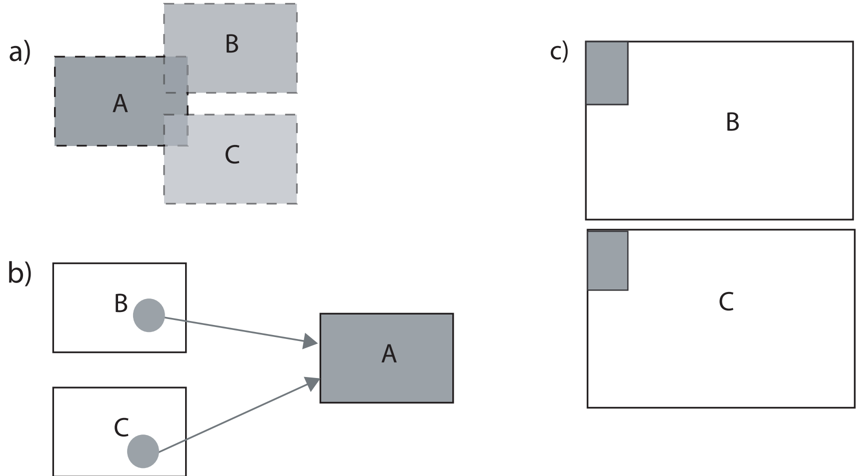


Figure 3.2: Documenting crosscutting concerns with conventional modular artifacts. The solid rectangles denote modules representing the corresponding artifacts and the solid arrows denote references. Dashed rectangles and dashed arrows denote concerns and crosscutting relationships respectively. Figure (a) shows the structure of crosscutting concerns: the crosscutting concern *A* crosscuts *B* and *C*. Figure (b) shows the artifacts documenting *B*, *C*, and *A* using a reference as a decoupling mechanism. Figure (c) illustrates the embedding of the matters of *A* in the artifacts of *B* and *C*. Both mechanisms illustrated by (b) and (c) result in redundancy and a violation of the separation of concerns principle.

Conventional software engineering approaches usually use some kind of *referencing* as a general kind of relationship and do not know the concept of crosscutting relationships. All relationships, are expressed the same way, and crosscutting relationships, which are a restriction of one element on another element, can usually not be properly expressed. Consequently, using conventional means for the modularization ends up in scattered and tangled concerns in the software artifacts, which is known as the tyranny of the dominant decomposition [Tarr99].

Figure 3.2 exemplifies the problems which result from the use of conventional modular artifacts and references for the description of crosscutting concerns. In (a) the crosscutting concern *A* crosscuts *B* and *C*.

There are two possible ways to represent the crosscutting concerns given in Fig. 3.2 (a) by conventional artifacts.

In the first case, a crosscutting concern is represented by one (or more) modular artifacts. One of these artifacts is then referenced by the artifacts of the crosscut concerns. This case is exemplified in Fig. 3.2 (b): the crosscutting concern *A* is represented by one modular artifact (modular artifacts are visualized by solid boxes). To express the crosscutting relationship, the modular artifact of *A* is referred by the artifacts of the concerns *B* and *C* by using a reference. The reference is denoted by a solid gray arrow. References work in the opposite direction to a

artifact with a high cohesion should concern only *one* matter of interest.

crosscutting relationship. As a result, *B* and *C* contain some kind of "glue part", represented as gray dot in the illustration, which describes the reference to the modular artifact of the crosscutting concern *A*. However, there are two reasons why this "glue part" is problematic. First, this glue part is redundant, as it is introduced in each of the crosscut modular artifacts. Second, the "glue part" is *intrusive*, as it properly belongs to the crosscutting concern and, therefore, it violates the separation of concerns.

In the second case, where conventional means are used to represent a crosscutting concern, each *crosscut* concern incorporates the elements forming part of the crosscutting concern. This case is shown in Fig. 3.2 (c). The crosscutting concern *A* is incorporated into each modular target artifact *B* and *C*. Thus, there is no need to express the crosscutting relationship. However, this has grave consequences, as major parts of the crosscutting concern *A* are duplicated in each crosscut artifact. Moreover, this solution also violates the separation of concerns.

Listing 3.1: Example of a piece of code with two tangled concerns. A business concern managing the transfer of money is tangled with a logging concern.

```
1   public void transfer( oat amount, BankAccount account) {
2     Logger.log("transfer() called with" + amount + "f" + account);
3     if (amount <= 0) {
4       RuntimeException ex = new RuntimeException("Error, negative amount!");
5       Logger.log("transfer() exited with exception: " + ex);
6           throw ex;
7     }
8     if (account == null) {
9       RuntimeException ex = new RuntimeException(" Error, account may not be null");
10      Logger.log("Exiting transfer() with Exception " + ex);
11      throw ex;
12    }
13    sum = sum − amount;
14    try {
15      account.deposit(amount);
16      account.checkLastDeposit(amount);
17    } catch (RuntimeException ex) {
18      Logger.log("Exiting transfer() with exception " + ex);
19      throw ex;
20    }
21    Logger.log("Exiting transfer() normally");
22  }
```

Examples. The following four modular artifacts examples, taken from different stages in the software process, illustrate the problems that occur when describing crosscutting concerns by conventional means.

Example (3.1). Listing 3.1 shows a modular code artifact of the implementation of a banking system. It is a Java method of a class implementing the business logic of a *SavingsAccount*

which does not allow a balance to be less than 0. It provides the functionality for transferring money from the current account to a target account. The method contains two different concerns: the *core concern* for transferring money and a *logging concern* which is crosscutting. The lines 4–5, 9–10, 19, and 21 in Listing 3.1 are dealing with the crosscutting concern.

Example (3.2). Suppose that the tourist information bureau of a big city wants to provide a handheld tourist guide device to visitors that is context and location aware.[9] The sketched use cases for this system may look as follows:

- **Suggest City Walk:** The tourist guide suggests a walk through the city. The suggestion is determined by the current location and the configured interests.
  1. The user chooses the “Suggest Walk” function.
  2. *The system checks if the interests have already been entered. If the interest have not yet been con gured, call the use case Enter Interest Con guration.*
  3. ...
- **Show Events:** The tourist guide suggests a set of events taking place at a specified date and time. The events are chosen by the interests profile of the user. After choosing the event, the user has to declare his credit card information if required for a reservation.
  1. The user chooses the “Show Events” function.
  2. *The system checks if the interests con guration has already been entered. If the interests have not yet been con gured, call use case Enter Interest Con guration.*
  3. The system retrieves the current events.
  4. Enter date and time.
  5. *Show the events according to the user s interest.*
  6. Select the event.
  7. Confirm the event.
  8. *If the chosen event requires a reservation with a credit card and the credit card information has not been entered before, then call use case Enter Credit Card Information.*
- **Theater Reservation:** The tourist guide allows the user to choose from a set of theatrical performances. For a reservation, some of the theaters need a credit card number.
  1. The user chooses the “Theater Reservation” function.
  2. The user can choose from a set of theatrical performances and the seat (use case Choose Theatrical Performance).
  3. *If the reservation of the chosen performance requires the credit card information and the information has not been entered before, then call use case Enter Credit Card Information.*
  4. ...
- ***Enter Interest Configuration:*** *The user can choose from a set of interests:*

[9] The system is inspired by the case study used in [Davi01].

   1. ...

- ***Enter Credit Card Information:*** *The user enters credit card information:*

   1. ...

- ...

The decomposition criteria for this example are use cases resulting in the corresponding modular artifacts. The use cases mentioned contain the three core concerns *Suggest City Walk*, *Show Events*, and *Theater Reservation*. Furthermore, there are the two crosscutting concerns "Handle Interest Configuration", and "Handle Credit Card Information". The text lines belonging to the crosscutting concerns are visualized in italics in the use cases. *Handle Interest Con guration* consists of the use case *Enter Interest Con guration*. It crosscuts the use cases *Show Events* and *Suggest City Walk*. The concern *Handle Credit Card Information* comprises the use case *Enter Credit Card Information* and crosscuts the use cases *Show Events* and *Theater Reservation*.

Example (3.3). Figure 3.3 shows an excerpt from an architectural description for a library management system, which describes the process for deleting and editing entries in the register for borrowed books. Both functionalities are only allowed for users with the corresponding privileges. The privileges are accredited by an *authentication*. *Editing a book* and *deleting books* are core concerns, whereas *authentication* is a crosscutting concern with an impact on the concern *editing a book* and *deleting books*. Figure 3.3 (a) shows the behavior description of the *editing books* concern and (b) the *deleting books* concern. The elements of the crosscutting concern are emphasized by a gray background.

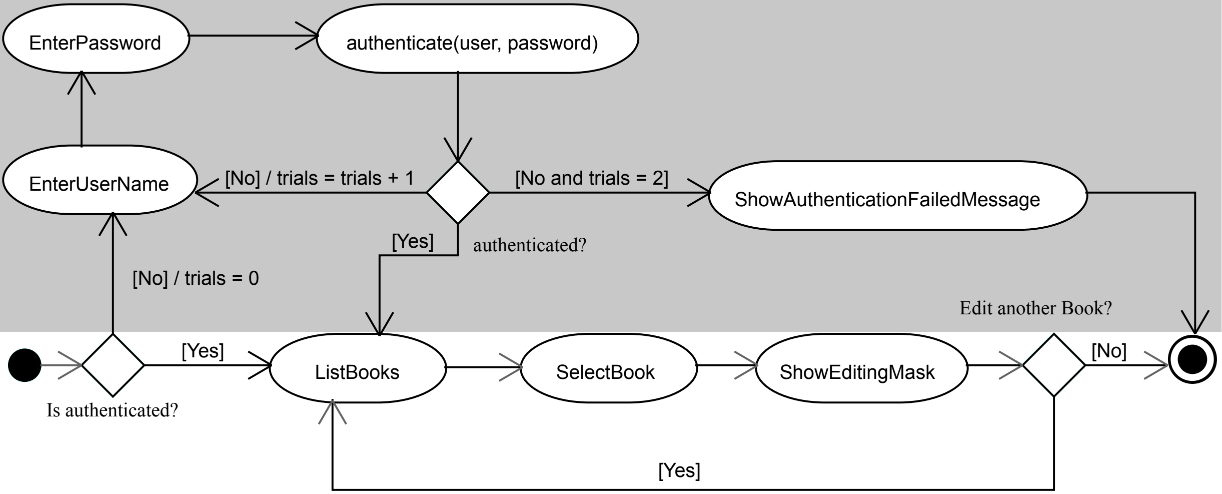
Example (3.4). Crosscutting concerns can even affect auxiliary artifacts of a software system, such as the user documentation. The functionality described by Example (3.3) can be accompanied by the user documentation given in Fig. 3.4, where the text of the crosscutting concern is set in italics.

3.1.4 Tangling, Scattering, and the Resulting Problems

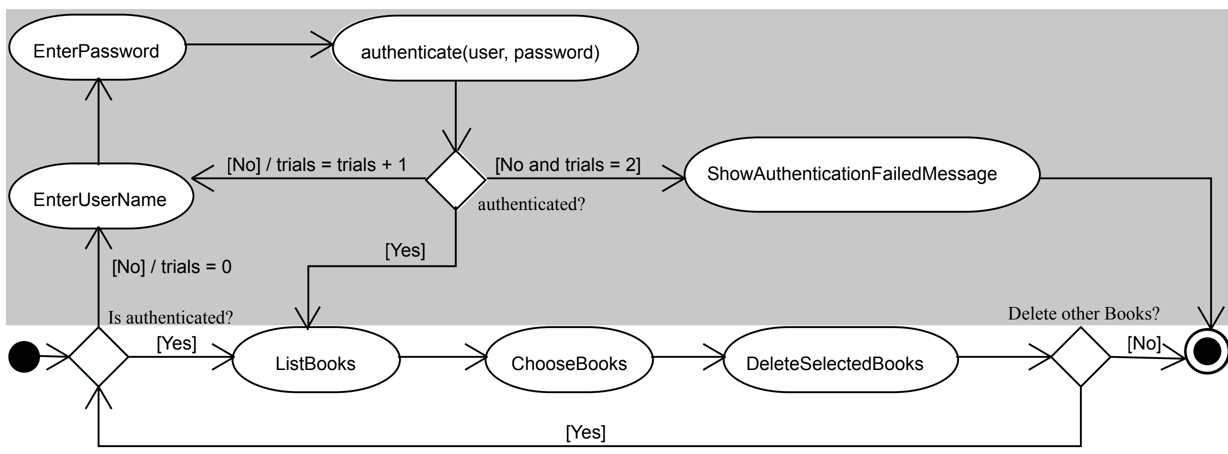
The two ways presented in Section 3.1.3 to describe crosscutting concerns and their crosscutting relationship using conventional modular artifacts are suboptimal as both introduce redundancy and/or violate the principle of the separation of concerns. Both problems manifest in a *scattering* and *tangling* of the crosscutting concerns in the artifacts of the core concern.

Figure 3.5 illustrates the problem of scattering and tangling. It shows a core concern, crosscut by an *Authentication* and *Logging* concern. The mixing of the three concerns is called tangling. The dispersing of one crosscutting concern over several modules is called scattering. Tangling and scattering affect modular artifacts in several ways:

- The *redundancy* in the artifacts, or at least the number of similar elements, is increased.
- An artifact which contains tangled concerns has *bad cohesion*, as concerns with different responsibilities and aims are contained in one modular artifact.



(a) Describes the behavior of *editing books* concern.



(b) Describes the behavior of the *deleting books* concern.

Figure 3.3: Example for crosscutting concerns in architectural artifacts. The figure shows two UML activity diagrams [OMaG03b]. Figure (a) describes the concern for *editing a book*, (b) a concern for *deleting a book*. Both are crosscut by an *authentication* concern.

...
2.1 Editing Book
The function **Editing Book** allows you to edit a book record. *Before performing this operation, you need to authorize it with your* user name *and* password *to be able to execute the editing of book records, and you need to have the corresponding privileges. See also Section 3.1 for additional information about the authentication procedure of the library system.*
After a successful authentication, a search mask for searching for and listing particular books is shown. ...
...
2.2 Deleting Books
The function **Delete Books** allows you to delete a set of book records.*Before performing this operation, you need to authorize it with your* user name *and the* password *to be able to perform this operation, and you need to have the corresponding privileges. See also Section 3.1 for additional information about the authentication procedure of the library system.*
After a successful authentication, a search mask for searching for and listing particular books is shown. ...
...

Figure 3.4: Excerpt from user documentation containing crosscutting concerns. The figure shows the description of the functionality given in Fig. 3.3. The crosscutting concerns of the authentication also manifests in the user documentation.

- The *intelligibility deteriorates* as the redundancy bloats the artifacts unnecessarily.
- The *intelligibility degrades* because particular artifacts cannot easily be assigned to a specific responsibility or concern, respectively.
- The artifacts are *more error-prone* and more difficult to maintain due to the increased complexity of their structure.
- Crosscutting concerns are almost *impossible to trace* as they are distributed over multiple artifacts.
- The artifacts are *less likely to be reusable* since they mix different concerns.

The tangling problem results from the missing separation of concerns. In turn, the scattering of a crosscutting concern over several modules is a consequence of the tangling. Scattering only occurs if the concern is strictly crosscutting, i.e., if there is more than one crosscutting relationship to other modular artifacts.

3.1.5 Decoupling Crosscutting Concerns

To overcome the problems resulting from the tangling and scattering of crosscutting concerns, aspect-oriented approaches provide language elements to keep crosscutting concerns separate. They provide a way to describe modular artifacts for the core concern, separate artifacts for the crosscutting concern, and a way to express the crosscutting relationship.

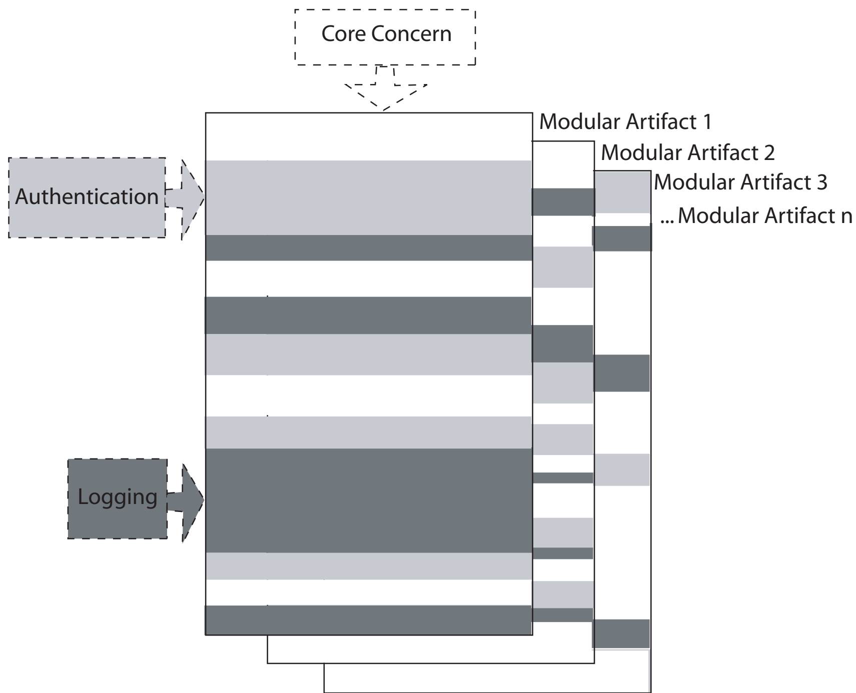


Figure 3.5: Illustration of the scattering and tangling of crosscutting concerns (based on [Ladd03, p. 8] and [Bono04, p. 17f]). The white areas denote the core concern, the light and dark gray ones show the crosscutting concerns *Authentication* and *Logging*, respectively. Both crosscutting concerns are tangled in the modular artifacts. It exemplifies also the scattering of the two crosscutting concerns over several modules.

In aspect-oriented approaches, two types of modular artifacts are given: components and aspects. The superordinate concept of aspects and components is the *modular artifact*. *Components* are modular artifacts that have no outgoing crosscutting relationships. An *aspect* is the originator of at least one crosscutting relationship.

The problem of modularizing crosscutting concerns is not confined to the implementation phase. Moreover, there are crosscutting concerns which do not necessarily end up in the implemented functionality of a system, as discussed below. Therefore, the definition of the term *aspect* given in this work is broader than in most other aspect-oriented approaches, such as [Lieb01] or in [Jaco03], where aspects are defined as a *modular unit of a crosscutting implementation*.[10] Furthermore, an aspect is specified *obliviously* for the target, i.e., the target of an aspect must not be aware that it is crosscut [Film00][11], as otherwise a kind of coupling between aspect and crosscut modular artifact is introduced.

In order that a crosscutting concern can be properly modularized, a crosscutting relationship

[10] It is defined similarly to [Saku04]: *..., an aspect is the unit of modular de nitions of crosscutting concerns.*

[11] In [Film00], the principles discussed deal with aspects in programming. However, the concepts can be transferred to aspect-oriented artifacts in the whole software process.

must be adequately describable, i.e., as unidirectional relationship expressing a restriction or an influence on the target. This is similar to the macro expansion semantics known from the concept of macro assembler programming languages [Gree59] or the preprocessor of the C language [ISO 05] and has therefore similar effects. However, the macro expansion concept demands that the target of a macro defines, i.e., controls, where the macro applies. In contrast, a crosscutting concern and its crosscutting relationship rather define where the impact is located and therefore fully control the impact location.

The crosscutting relationships can either be part of or be separate from the aspect description. Furthermore, sometimes it is necessary that an aspect can access the elements (such as attributes) of the crosscut modular artifact. Therefore, the elements which are accessible in the context of the target must be declared by the crosscutting relationship.

A *quantification* is part of the crosscutting concern that allows the specification of the location where an aspect has an impact on another modular artifact [Film00]. In many aspect-oriented approaches, such as AspectJ [Kicz01a], the quantification is expressed as a condition which specifies *when* and *where* an aspect has an impact on another modular artifact.

3.1.6 Separation vs. Composition

As discussed above, crosscutting concerns can occur in artifacts at every stage in the software process. Therefore, the goal should be throughout to introduce an adequate means for representing them during the whole software-process and to separate them as early as possible from other concerns. Moreover, using an aspect-oriented paradigm in the early phases of the software process simplifies the application of aspect-oriented approaches in later phases, e.g., during the implementation.

It is sometimes necessary to recombine the separated crosscutting and core concerns again. This composition is the reverse process of the identification and separation during the early phases. It results in artifacts with woven crosscutting and core concerns.

Figure 3.6 illustrates the separation and composition of concerns by means of a prism metaphor (cf. [Ladd03, p. 9] and [Wund05, p. 31]). A concern separation mechanism, visualized as a prism, allows to systematically separate the concerns at an early stage of the software process. The separation should take place as early as possible, best when the requirements are elicited. After the identification and separation, the concerns are documented with an aspect-oriented approach supporting a clear separation of concerns. The result of the documentation process is a set of aspects and components that are refined during the subsequent software process. Whenever it is appropriate, an *aspect weaver*, visualized in Fig. 3.6 as an upside down prism, creates the composed artifacts.[12] The composition process is also called *weaving*.

[12] Note that some aspect-oriented approaches, especially in the field of programming, do not integrate the artifacts. Instead the artifacts are used in a strictly separated way. However, when artifacts are interpreted by human beings, it makes sense to have the integrated artifacts, as the specific parts and interrelationships in a system can be better understood in some situations. See also the discussion below.

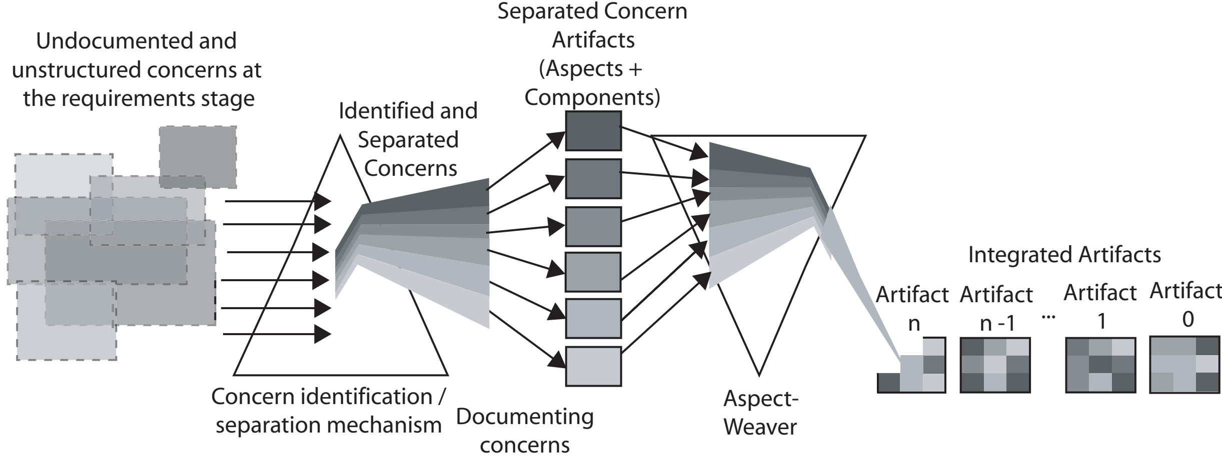


Figure 3.6: Illustration of the separation and integration process of crosscutting concerns.

Discovering and Separating Crosscutting Concerns

In the past, numerous approaches have been proposed for the discovery of crosscutting concerns. A first type of approach deals with mining crosscutting concerns in legacy software artifacts. Reverse engineering and aspect mining approaches in the code of existing systems, such as [Mari04] or [Hann01], belong to this category.

As mentioned above, crosscutting concerns should be separated as early as possible in the software process. Therefore, when building a new software system, crosscutting concerns should be identified and separated at a very early stage in the software life cycle. Most approaches to early identification, such as [Rose04], [Duan07], [Clel06], and [Samp05], deal with the processing of natural language requirements statements. Furthermore, an approach which mixes natural language processing concepts and heuristics is presented in [Bani04a].

Composing Aspects and Components

The weaving of aspect-oriented artifacts is necessary in some situations. A composition may be necessary when the system is executed, or when it must be verified or validated.

Executable and non-executable parts of the system. A software system consists of executable and non-executable artifacts. Executable artifacts are those code parts which can be directly executed by a machine. Non-executable artifacts are any other auxiliary artifacts, such as the user manual, of the software needed by the user or the system to execute.

For executable software parts, a weaving of the aspect-oriented artifacts is required. The weaving either occurs statically, i.e., explicitly, or dynamically, i.e., transparent to the user (cf. Section 3.2.1).

There are non-executable artifacts in a deployed system, such as the user manual, that are

read and interpreted by the users of the software. They are usually composed before they are deployed.

Veri cation and validation. Verification and validation tasks[13] can be done either dynamically or statically (cf. Section 2.1.2). Before aspect-oriented artifacts can be *veri ed or validated dynamically*, they may be woven.

For performing a static verification[14], the aspect-oriented as well as the woven artifacts can be used. Static verification/validation means involving people who assess the quality and the adequateness of the given artifacts by analyzing and interpreting them individually. Therefore, the intelligibility of the document is crucial. In turn, the understandability is in uenced by whether aspect-oriented or conventional artifacts are used. Depending on the situation, aspect-oriented artifacts may increase the understandability or they may not, as argued in the following Section 3.1.7. Therefore, it is useful to provide the capability to switch between aspect-oriented and conventional artifacts when doing a static verification or validation.

3.1.7 Complexity Caused by the Use of Aspect-Oriented Artifacts

An aspect-oriented approach may simplify artifacts, but may also introduce additional complexity [Meie05] due to the strict separation of concerns. In some situations, aspect-oriented artifacts are better suited than their conventional counterparts. However, there can be situations where it is the other way round. The understanding depends on the type of focus that is put on the concerns in the artifacts. There are two different types of focus:

1. An *isolated focus*[15] ignores the interaction between the concerns, and is a separated view of crosscutting and core concerns.

2. In contrast, an *interrelated focus*[16] on concerns deals with the interplay between the crosscutting and other concerns.

Depending on the type of focus, the understandability of the artifacts is affected. An *isolated focus* used on aspect-oriented artifacts improves the understandability. This is due to the fact that modularized crosscutting concerns are strongly decoupled from other concerns. The parts of crosscutting concerns are not scattered in the system but rather concentrated in one modular artifact. The reader of that artifact does not need to separate them individually in mind from the other concerns and the understandability is better.

In contrast, using the *aspect-oriented artifacts* of a system together with an *interrelated focus* does not improve the understandability compared with the use of the conventional artifacts. This is due to the fact that the reader of the model has to *integrate* the crosscutting and the conventional

[13] Validation tasks are for example the simulation of requirements, e.g., [Seyb06a], or the acceptance test [IEEE90]. Verification tasks are the testing (unit tests, integration tests, system tests) of the software.

[14] For example, a static validation is done by performing a code inspection[Faga86] or a manual desk check.

[15] In [Meie05] this type of focus is called *local*. However, in this work it is more adequately called *isolated* focus.

[16] In [Meie05] this type of focus is called *global*. In this work it is called more adequately *interrelated* focus.

concerns in his mind to understand how they interplay. In fact, it may even result in a worse understandability of the described system.

The type of focus which is put on the concerns depends on the operation performed by the person who reads/modifies the artifacts. For example, editing and modifying operations on a particular concern have usually an isolated focus. In contrast, interpreting the relationships between aspects and components demands an interrelated focus.

For this reason, it is useful to be able to choose the most adequate representation when working with the artifacts during the development. As a consequence, an aspect-oriented representation must be *composable* into a integrated (i.e., conventional) representation.

3.1.8 Characteristics of Aspects

Aspects should have a set of specific properties. How well they are satisfied depends on the concrete situation. In the following, they are summarized and discussed brie y (cf. [Bono04, p. 25f]):

- **Modularity:** The separation of concerns and therefore the modular description of all concerns in a software system is the primary aim of an aspect-oriented approach. Aspects should be self-contained modular artifacts. An aspect should declare its dependencies, so that it can be understood in isolation [Elra01][17].

- **Obliviousness:** Obliviousness [Film00] demands that the artifacts of a *crosscut* concern must not be aware of being crosscut. Otherwise, a coupling of the crosscutting concern and the crosscut concern is introduced. Obliviousness has been discussed controversially in the recent past, e.g., in [Sull05].

- **Homogeneity of an aspect:** An aspect is homogeneous if it provides the same content for each target. The aspect is inhomogeneous if the provided content depends on the target and usually has multiple responsibilities. However, such an aspect has low cohesion and is therefore better split up into different aspects handling the different impact situations.

- **Avoidance of side effects:** Aspects should not introduce side-effects in the artifacts of crosscut concerns. This property applies to executable as well as non-executable artifacts. For executable artifacts, such as code in the implementation phase, this means that an aspect must not break the contracts of a target module (cf. the-design-by-contract principle [Meye92]). For non-executable artifacts, such as semi-formal models, this means that an aspect should not introduce content into the target which is ambiguous or misleading to the (human) reader of the artifact.

- **Reusability:** An aspect should be reusable [Elra01]. Reusability is achieved by having a low coupling[18] and a high cohesion.

[17] In [Elra01] this property is called self-containedness.

[18] In the case of crosscutting concerns, coupling results from the usage of conventional relationships, such as references.

- **Composability:** Aspects must be composable. Depending on the situation, certain facts are better understandable to human beings when expressed as conventional artifacts than aspect-oriented artifacts (cf. Section 3.1.7). Vice versa, there may be situations where aspect-oriented artifacts are better suited. Therefore, aspects should be composable with the artifacts of the core concerns [Elra01]. Consequently, there must be a set of well-formed rules that allow transformation from the aspect-oriented to the conventional representation and vice-versa.

- **Additivity:** Aspects should not be invasive[19] [Ladd03] when they are added to or removed from an existing system. A non-invasive aspect is called additive.

- **Orthogonality:** The same target can be impacted by more than one aspect. If the order in which they occur in the final system does not affect the semantics of the software, they are called *orthogonal*. However, orthogonality cannot be always achieved. Therefore, an aspect-oriented approach must provide a mechanism for solving con icts, e.g., by assigning a precedence.

3.1.9 Connection between Concerns and Requirements

A concern categorizes a specific matter of interest in the system and comprises one or more software artifacts. It is expressed by high-level requirements statements, such as *"the system must print out monthly reports"* or *"the system must be secure"*. Therefore, concerns and high-level requirements are seen as equivalent in this work.

Functional and non-functional high-level requirements (or concerns) are refined during the software process towards concrete requirements or other modular artifacts of the software system. Consequently, the artifacts representing a concern can be evolved from the corresponding high-level requirement, and aspects can result from non-functional or functional high-level requirements. The refinement process ends in a set of artifacts which together form the resulting system.

During the refinement process, a concern is identified as crosscutting if at least one of its artifacts is the origin of a crosscutting relationship. Thus, a concern may not be identified as crosscutting until a later stage of the software process. However, this mainly applies to functional concerns. In contrast, non-functional concerns are usually identified from the beginning as crosscutting.

The resulting aspects arising from this refinement process are discussed in the following in more detail.

Non-Functional Aspects

Non-functional requirements (cf. Section 2.1.1) restrict or in uence other requirements, though they have one or more *crosscutting relationships* to other requirements. Therefore, a high-level *non-functional requirement* (NFR) is a *crosscutting concern* and always results in one or more

[19] That means that there should be no need to change an impacted artifact when being impacted by an aspect.

aspects. Consequently, *all* artifacts originating from a high-level non-functional requirement should be represented by aspects. To indicate their origin, aspects resulting from a non-functional high-level requirement are called *non-functional aspects*.

The aspects emanating from a high-level NFR have one of the following three types (cf. Section 2.1.2). Type (i) is an aspect that contributes to the system's functionality. Therefore, it manifests in the system as a piece of code at the end of the development process. Typically, non-functional aspects of type (i) can be handled at the implementation stage through one of the AOP approaches, such as AspectJ [Kicz01a].

Example (3.5). An example of a type (i) non-functional aspect is the authentication functionality given in Example (3.3). It can be modularized in a aspect and is refined from the high-level non-functional requirements *Security or Authorizing/Authenticating, respectively*. In the final system it manifests in a concrete piece of code that is executed.

Non-functional aspects of type (ii) result in a *decision* which in uences *how* the function of the system is performed. They may have an impact on one or more other artifacts of the software. Usually aspects of type (ii) are absorbed by the architecture or the design of the final system. This absorption can be seen as weaving the decision into the system. Sometimes, it is also possible to realize such a decision by an AOP aspect.[20]

Example (3.6). Suppose there is a high-level NFR which demands that a large information system performs within a given time limit. In a further refinement of the NFR, the architectural decision results in the use of a fast performing, transparent cache which already exists as a component-off-the-shelf (COTS) implementation.

Finally, non-functional aspects of type (iii) result in a *decision* which in uences neither *what* nor *how* the functionality of a system is performed. They rather result in a decision which in uences either the software process or the form of other artifacts.

Example (3.7). Suppose there is a high-level NFR which requires a high maintainability of the developed system, because it is embedded in a highly volatile environment. To improve the quality of the source code and therefore to increase its maintainability, a decision is taken that a coding rule must be followed This decision is crosscutting, i.e., it has an effect on the system at various locations.

Functional Aspects

High-level functional requirements can be crosscutting concerns because they may result in more detailed artifacts which "inject" functionality into other concerns. In aspect-oriented approaches, they should be handled as aspects, too. As an indication of their origin, aspects resulting from a refinement process of functional requirements are called *functional* aspects.

[20]An example for the realization of a type (ii) artifact as an AOP aspect is given in [Kicz97], where a non-functional performance requirement for a composable graphics filter library for bitmap graphics is solved by AOP constructs.

Example (3.8). An example for a functional crosscutting concern is given by the tourist guide device of Example (3.2): *Handling the credit card information*, as well as *handling of the personal interest* are crosscutting concerns, as they in uence the behavior of other concerns. They can be modularized and finally implemented as aspects.

3.2 State of the Art Approaches

Originally, aspect-oriented approaches were introduced at the programming stage. Then these concepts were adapted to other phases of the software process. Ever since the first ideas of aspect-oriented programming were proposed [Kicz97], many aspect-oriented approaches have been suggested for supporting each phase in the software process.

In the following, a selection of approaches is presented. The various characteristics of aspect-oriented approaches are discussed in Section 3.2.1. The most in uential programming approaches are presented in more detail in Section 3.2.2. Approaches to the design and the architecture are sketched in Section 3.2.3 and Section 3.2.4, respectively. Existing aspect-oriented requirements approaches are outlined in Section 3.2.5.

3.2.1 Categorizing Aspect-Oriented Approaches

Aspect-oriented approaches can be categorized according to several characteristics, such as their symmetry, and how aspects are composed into the artifacts of the core concerns.

Symmetric vs. asymmetric approaches. In symmetric approaches, such as HyperJ [Ossh01], aspects and components are modularized by the same type of modular artifacts [Harr02]. Thus, core and crosscutting concerns are seen as peers and concerns are said to be overlapping rather than crosscutting. As a consequence, the crosscutting relationship is specified separately from the overlapping artifacts.

In contrast, asymmetric aspect-oriented approaches, such as AspectJ [Kicz01b], provide asymmetric modular artifacts, i.e., there are aspects describing the crosscutting concern as well as components describing core concerns. The artifacts of the core concern augment the artifacts of the crosscutting concern. Moreover, aspects usually also contain the specification of the crosscutting relationship, i.e., where an aspect has an impact on other concerns.

Symmetric approaches are useful to separate concerns that are seen as peers and that are rather loosely coupled. In contrast, asymmetric approaches are better suited for systems which have clear core concerns that are augmented by crosscutting concerns.

Linguistic vs. framework-based approaches. *Linguistic approaches* introduce a new language or extend an existing language which allows the description of crosscutting concerns separately. In contrast, *framework-based approaches* try to use existing programming frameworks, i.e., predefined solution patterns, or language features to keep concerns separate. The distinction between linguistic and framework-based approaches is usually made in AOP, but not

for aspect-oriented approaches at the pre-implementation stages of the software process. This is due to the fact that framework-based approaches rest upon programming frameworks.

Domain specific vs. general purpose approach. *Domain-specific* approaches solve a narrow and specific problem, whereas *general purpose* approaches can be used to describe a broad range of problems.

Static vs. dynamic composition. *Static composition approaches* integrate the aspect-oriented artifacts at a stroke into a set of woven set of artifacts. *Dynamic approaches* either perform the composition incrementally, i.e., when needed or they interpret the aspect-oriented artifacts directly, e.g., by using a suitable mechanism, such as Java reflection.

The distinction between whether an aspect-oriented approach employs a static or a dynamic composition is only used for approaches with executable artifacts, e.g., aspect-oriented programming or requirements specifications that are executable. Dynamic composition is usually slower than static composition when executing the artifacts. For approaches with non-executable artifacts, static composition is used.

Software variability support. There is the distinction between whether aspect-oriented approaches are able to handle software product lines (cf. for example [Clem01, Pohl05, Brow96, Kang90, Parn76]) or not. Variable and common software parts can be seen as concerns (see for example [Coly04, Gris00b, Gris00a]) that have to be separated and composed to create product variants.

3.2.2 Approaches to Aspect-Oriented Programming

In the following, a selection of AOP approaches is briefly presented.[21] It will help in understanding the details of aspect-orientation and the approaches used during the other phases of the software process.

AspectJ

AspectJ [Kicz01a, Ladd03] is probably the most widespread AOP approach. Therefore, it is discussed more extensively here. It is a linguistic, asymmetric approach that extends the language Java [Sun 07a] with modularization capabilities for crosscutting concerns. It is a general-purpose aspect language that may be used for a broad range of problems. In AspectJ, a modular crosscutting concern is described by a syntactical feature called *aspect*. An aspect contains *pointcut definitions*, *advice definitions*, *declarations*, and *introductions*.

An aspect specifies the crosscutting relationship to other modules, i.e., classes and other aspects, through *pointcuts*. A pointcut specifies the locations in the program flow, where crosscutting behavior is injected.[22] In AspectJ terminology, these impact locations are called *join points*.

[21] For an extensive discussion, please consult the references.

[22] The specification of the impact location is also called quantification (cf. Section 3.1.5).

A pointcut defines a pattern that match class, field, or method names in the target specifying the join points. A join point can be the *call of a method*, the *body of a method*[23], the *catch part of an exception handler*, the *location where object fields are read or written*, etc.[24] A pointcut can either be identified by a name, or it can be anonymously specified together with an advice definition.

An *advice definition* describes the crosscutting behavior of the aspect. Apart from that, its impact location is specified by a pointcut (as defined previously). Moreover, the advice declares if the crosscutting code is executed either *before*, *after*, or *around*[25] the join point. An advice definition can access the context, i.e., the variables in the scope of the join point.

Last, *declarations* allow modification of the inheritance hierarchy of the classes, define compilation warnings or errors, the precedence of aspects, or the softening of the throw clause of exceptions.

Summarized, AspectJ is very flexible and can be used for a broad range of problems. However, there are also several weaknesses. The name-based join point model of AspectJ results in some deficiencies. A difficulty results from an unintentional quantification of join points when using the pattern matching for names. Therefore, the naming has to be well planned and rigorously realized. Moreover, join points are fragile [Stor05] or vulnerable to changes in the code. Furthermore, the use of the names for defining join points makes the migration of legacy Java applications to AspectJ tedious. Such an application may require invasive changes in the naming of the elements, in order to be used with AspectJ pointcuts.

HyperJ

HyperJ is a *symmetric linguistic* approach that is based on Java and relies on the principle of multidimensional separation of concerns (MDSOC) [Tarr99].[26] MDSOC allows the separation of overlapping concerns according to different decomposition dimensions. A decomposition dimension may be any criterion that is of interest during the development of the software, such as data, features, business rules, etc. In HyperJ, all concerns are handled as peers.

For the composition of a system, HyperJ introduces the concepts of hyperspace, hyperslice, and hypermodule. The hyperspace of an application consists of all classes of all decomposition dimensions. A *hyperslice* is a set of distinct conventional Java classes and the methods of these classes which together describe a *particular concern* of the software.

The classes of different hyperslices may overlap by having the same methods.[27] A method which is common to several concerns has to be declared in a class of one concern. The corresponding class of the other concerns need to declare this method as abstract. This must be done in order to have a declaratively complete, i.e., self-contained, implementation of a concern.

Furthermore, HyperJ allows the separation of tangled concerns in existing legacy applications by defining corresponding hyperslices. This approach is called the *mix-and-match* principle.

[23] This applies also to constructors.
[24] For a detailed discussion, see [Ladd03].
[25] An *around* specification may also result in a substitution of the join point (cf. [Ladd03]).
[26] MDSOC is based on the idea of subject-oriented programming [Harr93].
[27] Methods are the smallest decomposition unit in HyperJ.

Mix-and-match enables the tangled concerns to be separated and to be recomposed again in a new application. Apart from handling legacy software, this feature predestines HyperJ for the handling of software variability and software product lines.

A *hypermodule* defines how a set of hyperslices is composed into a system or a subsystem. Hypermodules can also be viewed as hyperslices and can therefore be nested again in other hypermodules [Tarr99]. There are various rules for composing hyperslices into hypermodules. For example, the rule *mergeByName* defines that methods with the same name are merged. The order in which the methods are merged in the resulting hyper module can be specified. Apart from *mergeByName* composition rules, there are other rules, such as the overriding of units, or the merging of elements with an unequal name.

HyperJ is an aspect-oriented general purpose approach, which allows a system to be decomposed into a set of peer concerns. Due to its ability to separate concerns, it is also well-suited to handle legacy systems without invasive changes and it allows the handling of variability in software systems. However, the composition rules can become rather complex, because the composition definition is separated from the modules of the concerns.*In contrast, the composition rules are speci ed in AspectJ by pointcuts de ned in the corresponding aspect.*

The definition of the join point model is more coarse-grained than AspectJ, as the smallest unit for merging are methods, which makes HyperJ less exible than AspectJ. Furthermore, the join point model is based on the matching of names, resulting in the same problems that arise from AspectJ.

Adaptive Programming and DJ Library

Adaptive programming [Orle01, Lieb01, Lieb97] is a hybrid approach which unifies framework-based as well as linguistic elements. Adaptive programming is domain-specific and allows the problem of traversing data structures to be handled in an adequate way.

The traversal of data structures can be implemented in various ways. In object-oriented systems, it is possible to add a traversal method to each class definition of the objects that are part of the data structure. However, in this case the traversal functionality is scattered over different classes. The visitor pattern [Gamm95] tries to overcome this problem by a generic interface for calling a visitor that incorporates all processing logic. The class hierarchy of the objects in the data structure only implements the visiting strategy (accept method). However, although the traversal methods are less scattered than with the previously described approach, changes to the processing logic are still tedious because of the law of Demeter [Lieb89].

The Law of Demeter states that a method in a class should only have limited knowledge of the class structure, in order to be easily maintainable. However, following this law results in a number of small methods which use dynamic binding in order to reduce knowledge of the class structure. As a consequence, the number of methods increases and there are problems in maintaining and understanding such classes. This is also the case for visitor implementations.

Adaptive programming overcomes these problems. DJ [Orle01] is a library which allows the traversal of data structures using the re ection mechanism of Java.[28] For using DJ, the traversal

[28]In contrast to DJ, the previous approach called Demeter/Java [Lieb97] needed static compilation before the

strategy as well as the processing of the data structure nodes must be defined.

Other Programming Approaches

There are other programming languages which are brie y sketched in the following. *AspectC++* [Gal01] provides similar functionality to AspectJ, but using the syntax of the C++ language. The *AspectWerkz* [Bone04, Vass04] approach is based on Java and uses the re ection mechanism and annotations[29] or XML definitions.[30] The *JBoss AOP Java Framework* [Burk04] is similar to AspectWerkz.[31]

Object-oriented frameworks, such as the Aspect Moderator Framework (AMF) [Cons00], try to overcome the problem of the scattering and tangling of crosscutting concerns by employing only object-oriented techniques. Usually, framework-based approaches are domain specific and therefore solve the separation of concerns for one specific issue. For example in [Cons00], concurrency control is handled by such a framework.

Meta-object protocols[32] [Kicz91] are based on re ection and can also be used to cope with the separation of concerns. They enable the dynamic manipulation of the communication between objects on the meta-level of the programming language and even make it possible to change the behavior of the software at the runtime. Meta-object programming allows, for example, crosscutting concern code to be executed before or after a method call. The *Java re ection capabilities and dynamic proxies* [Blos00] are a simple meta-object protocol approach in Java. A dynamic proxy can be used to intercept method calls and manipulate and change the behavior of the software. However, Java has no full- edged support for implementing meta-object protocols.

Composition lters [Berg01, Akc 92] are similar to meta-object protocols in their mode of operation. Filters are implemented in an object-oriented manner, work dynamically and allow filtering of the input and output messages of objects, i.e., their interception and redirection. A composition of filters can be used to create a filter chain composition resulting in complex filter operations. The separation of concerns is achieved by assigning a concern to a filter that redirects the messages addressed to objects of a specific concern.

3.2.3 Approach to Software Design

Some of the various existing aspect-oriented design approaches [Chit05, p. 154ff] are summarized brie y in the following. Some are closer to the design, others closer to the architecture. In

---

mechanism could be used.

[29] Abstract annotations [Ladd05] can be used to denote possible join points in the modules that are potentially crosscut. The use of annotations avoids the problem of fragile join points which results from the use of names for identifying join points. However, the use of annotation approach is invasive, because the potential join points need to be denoted in the source code of the target.

[30] The project has joined the AspectJ project in 2005.

[31] The JBoss AOP Framework as well as AspectWerkz are offsprings of web application server products. The former is part of the JBoss (http://www.jboss.com) application server and the latter is part of the Bea Weblogic server (http://www.bea.com)

[32] Meta-object protocols originate in the Common Lisp Object System (CLOS) [Keen88].

the following, a brief, but incomplete overview is given of the existing approaches.[33]

Aspect-oriented design modeling (AODM) [Stei02] was originally developed to support the AspectJ language in the design phase. Later it was extended to support other asymmetric AOP languages. It is based on UML and supports the modeling of systems by parameterized collaboration diagrams containing classes that specify the crosscutting structure of a system. The crosscutting behavior can be described in terms of use cases and sequence diagrams.

Theme/UML [Bani04c, Clar05] is the design part or the Theme approach (cf. Section 3.2.5). It maps the crosscutting and base *themes* to UML artifacts. Base themes are represented as packages, crosscutting themes (aspects) as *parameterizable packages* in UML. The parameters of the package indicate the join points for aspects. The crosscutting behavior is specified by sequence diagrams contained in the theme packages.

CoCompose [Wage02] is a symmetric design approach that is not based on UML. It allows aspect-oriented designs to be specified in terms of features. Modeling is done using a graphical language. Furthermore, a design algebra makes it possible to map the design to a specific language.

UML for Aspects (UFA) [Herr02] is a symmetric design approach based on UML and extends the UML package construct. An *aspect package* construct is introduced which acts as a facade that may have methods and attributes, representing distinct attributes and methods from the classes contained in the package. For defining the crosscutting relationship between an aspect package and a target package, the aspect package must be inherited a connector package. It binds the aspect package to the packages of the crosscut concern.

The *Aspect Modeling Language* [Groh04] is based on the UFA notation. It provides a subset of the AspectJ language elements and uses packages to describe aspects and base packages. The composition is done by connector packages.

UMLAUT [Ho00] is a weaving tool which can be used to perform compositions of aspect-oriented models based on a UML notation.

3.2.4 Approaches to Software Architecture

Several aspect-oriented architectural approaches have been proposed in the past. A small selection of aspect-oriented approaches are described in the following. A more extensive discussion of them as well as of conventional software architecture approaches can be found in [Chit05, p. 114–153].

The *Perspectival Concern-Space* [Kand03] is an aspect-oriented approach using the UML language. It introduces a UML meta-model extension which introduces perspectives for the multidimensional decomposition of concerns and the specification of join points where the concerns are composed.

DAOP-ADL [Pint03] is an XML-based architecture description language that is able to describe aspects, components and interconnections. The interconnections describe the composition rules for aspects and components.

[33] A more detailed discussion and overview can be found in [Chit05]. This survey also discusses conventional design approaches.

AGOA [Kule04] is an aspect-oriented architectural approach which was originally introduced for the development of multi-agent systems. However, the concepts are more general and can be used in other domains. The method supports the development at all stages of the software process. It uses feature-based modeling at the requirements stage and UML component-based modeling at the architectural stage.

The *TranSAT* [Bara04] approach has two main goals: the reusability of aspect-oriented architecture artifacts and the introduction of new concerns without breaking the consistency of the architectural description. The main constructs of the approach is the so-called architecture pattern which consists of an architecture plan, a join point mask, and a set of transformation rules.

3.2.5 Approaches to Requirements Engineering

As discussed in Section 3.1.6, the earlier concerns are separated, the earlier the advantages from the separation can be exploited. Therefore requirements approaches play a key role in the effective handling of crosscutting concerns. The current aspect-oriented approaches are sketched in the following. A more detailed discussion, as well as an evaluation of aspect-oriented and conventional requirements engineering approaches can be found in Appendix A. The gaps in the existing aspect-oriented requirements approaches are summarized in Chapter 4.

Use Case Approaches

AOSD/UC. Aspect-oriented software development with use cases (AOSD/UC, cf. [Jaco03] and [Jaco05]) proposes an extension to the use cases method [Jaco92]. The approach can be employed throughout all phases in the software process and introduces new aspect-oriented symmetric and asymmetric constructs. The two new main elements are pointcuts and use case slices. It extends the UML use case constructs with aspect-oriented elements, such as pointcuts. It introduces *use case slices* to represent the artifacts belonging to a use case at a particular stage of the software development. *Use case modules* are containers for artifacts refined during the software process. Operationalized non-functional requirements are used by so-called *infrastructure use cases*.

SMA. Scenario modeling with aspects (SMA) [Arau04, Whit04] is an approach for creating more consistent and complete use cases by modeling base and crosscutting scenarios separately from each other. A composition mechanism generates state machines from the elicited scenarios. These state machines help to validate the modeled use cases by simulation.

AUCDA. Aspectual use case driven approach (AUCDA) [Arau03] is another use-case-based aspect-oriented approach. It is similar to AOSD/UC [Jaco05] and aims at identifying and describing crosscutting non-functional (called quality attributes) and functional concerns at the requirements stage. However, in contrast to AOSD/UC, non-functional requirements are not handled by infrastructure use cases. A template, proposed in [More02], is introduced, which facilitates the identification of non-functional requirements.

Goal Approaches

ARGM. Aspects in Requirements Goal Models (ARGM) [Yu04] is a goal-oriented approach which is based on the non-functional requirements framework [Chun00]. Goals and soft-goals can be decomposed into subgoals and sub-soft-goals, respectively. This is done iteratively until the decomposition is reduced to a task. (Sub)soft-goals can be related to operationalizations. Correlations can represent in uences between (sub)soft-goals and goals. Together they form a goal/soft-goal interdependency graph, which is represented as a so-called *V-graph*. The approach aims at identifying aspects during goal-oriented requirements analysis. Aspects are identified as tasks with many links satisfying different goals.

Other Approaches

AORE. Aspect-Oriented Requirements Engineering (AORE) [Rash03] with Arcade is an approach which can be used with any requirements process/technique. In [Rash03], the approach is used together with the PREView approach [Fink92, Fink96]. AORE aims at the modularization of crosscutting concerns and the creation of a consistent requirements specification document. *Aspectual requirements* are similar to external requirements in PREView. They crosscut user requirements derived from various viewpoints. It uses an XML-format to store the artifacts.

COSMOS. The *Cosmos* approach [Stan02, Sutt03, Sutt04] is based on the principle of the multi-dimensional separation of concerns [Tarr99], where the concern space consists of a set of peer concerns that may overlap. Cosmos proposes an artifact-independent way of modeling concerns. It provides a general concern-space modeling schema for the classification of concerns, their relationships, etc. After a concern has been identified, it is assigned to one or more *concern categories*. Furthermore, the relationships to other concerns are identified and classified [Sutt04]. A concern is either logical or physical. A *logical* concern is of a conceptual nature whereas a physical concern deals with real world artifacts.

CORE. Concern-oriented requirements engineering (CORE) [More05a, More05b] extends the AORE with Arcade approach [Rash03] by the concepts of the multidimensional separation of concerns [Tarr99]. CORE decomposes requirements using different concerns as decomposition criteria. It therefore introduces, like the Cosmos approach [Stan02], a meta-concern space which models the concerns of a software system and their relationships. An instantiation of the corresponding concerns allows the assignment of the requirements belonging to the concern. A set of composition rules describes the impact of a concern on the requirements of other concerns. Similarly to AORE, CORE uses XML for the representation of its artifacts, i.e., the meta-concern structure, the concern artifacts, and the composition descriptions.

AOREC. Aspect-oriented requirements engineering for component-based software systems (AOREC) [Grun99, Grun00] provides a classification schema for categorizing the systemic aspects of components. In AOREC, an aspect is a characteristic of a system which consists of components providing or requiring services. The main artifacts for documenting aspects are

diagrams. The diagrams describe the components, the related aspects and the corresponding relationships. Furthermore, there are textual descriptions that additionally describe the functional and non-functional requirements of the system. A so-called aggregated aspect describes a group of interrelated components.

Theme/Doc. Theme/Doc [Bani04c, Bani04a, Clar05] is the requirements part of the Theme approach. In contrast to other approaches, such as ARGM, the Theme approach has been designed from scratch as an aspect-oriented approach and originates in subjective programming [Harr93]. It aims at the identification and further handling of crosscutting concerns identified from natural language requirements. Theme/Doc analyzes the requirements statements of a software, thus it is applied at a later stage in the requirements process, when the requirements have already been elicited. Theme/Doc has a focus on actions, therefore it is less suitable for detecting non-functional crosscutting concerns. The analysis relies on the use of a tool, and the artifacts produced by the tool are based on a manually compiled list of *action words* and *entities*. Action words are derived from the verbs in the natural language requirements specification and are called *themes*. A theme can be seen as the “encapsulation of a concern” [Clar05]. *Entities* (nouns) related to the action words can be seen as objects on which the actions rely.

3.3 Criticism of AOSD

In the past, there have been various discussions about the usefulness of AOSD concepts. Usually, particular AOSD topics or approaches at a particular stage in the software engineering process are subject to criticism. However, most insights arising from these discussions can be transferred to the whole field of AOSD. In the following some criticisms from the past are discussed, including *the application of AOSD concepts to the requirements stage*, the *understandability of AOSD artifacts*, the *breaking of the principle of information hiding*, and *fragile join points*.

3.3.1 Criticism of AOSD at the Requirements Stage

Better use viewpoints instead of aspect-oriented approaches. Nuseibeh [Nuse04] proposes that aspects should not be identified too early, but that the concerns of a software system can be analyzed in order to understand the resulting requirements. Therefore an adequate software process must provide the means for identifying crosscutting concerns. The approach suggests the use of a viewpoint-oriented approach for doing so, as viewpoints provide the means for handling overlapping requirements, i.e., *crosscutting requirements* of several stakeholders.

However, viewpoint-oriented approaches only address conflicts in requirements which are caused by different viewpoints *before* the requirements are negotiated between the stakeholders. Thus, they do not identify and handle the actual crosscutting concerns, so that they exist also after the negotiation. Hence, viewpoint-oriented approaches do not adequately solve the crosscutting concern problem.[34]

[34] An exception is the PREView approach, which allows the identification and handling of non-functional cross-

Aspects in domain models. Steimann [Stei05] mistrusts the existence and the meaningfulness of aspects in domain models, i.e., in models resulting from the early phases of the software engineering process. Several arguments are brought forward to support this hypothesis. The work claims first that the term *aspect* is used with different meanings by various approaches in the field. The work tries to argue that these concepts are better covered by non-aspect-oriented constructs or that they are not part of the domain modeling.

It is argued that aspects can be realized by *roles* by using a generalization relationship between different types in an object-oriented system. This means that a super type can denote a role, such as *Billable* or *Serializable*. A role describes therefore a specific concern. A role can overlap, or "crosscut", other roles, because concerns can overlap. An overlapping of roles results in a multiple inheritance.

Even though this argument is correct and roles can be used for expressing crosscutting concerns, this should not be considered for domain models. First, inheritance is usually not extensively used in domain models, because domain models need to be simple to understand for non-expert stakeholders. Furthermore, introducing crosscutting concerns through roles is difficult, since inheritance is a very coarse-grained way of describing crosscutting behavior. The smallest unit for defining the behavior of crosscutting concerns in an inheritance hierarchy are methods. Furthermore, multiple inheritance results in complex relationships between classes, which is a reason that it has never been broadly used. Moreover, multiple inheritance hampers the understandability of a domain model. Nevertheless, roles in domain models may be an appropriate means for the description of non-crosscutting concerns in certain situations.

The argument in [Stei05] further implies that the aspect examples usually discussed in the literature, such as *authentication*, *caching*, *distribution*, *logging*, etc. are programming aspects rather than aspects of the domain. This is true to some extent, as they represent behavior which can be mapped from a crosscutting concern to a corresponding implementation. However, the origins of the instantiated aspects are the corresponding high-level non-functional requirements, or concerns, which cut across other concerns. Non-functional requirements can be seen as part of the domain. This is due to the fact that the implementation of the domain will not work satisfactorily if the NFRs are neglected. For example, take the requirements for an online shopping scenario. Ignoring the non-functional requirement which states that the maximum response time of an application must be below two seconds results in an unusable application. This fact is also agreed upon [Stei05]: "*..., that something is classi ed as a non-functional requirement does not preclude it from being part of a domain model, ...*" Furthermore, functional aspects at the programming level can have their counterpart at the requirements stage, too.[35] Therefore, it makes sense to express aspects as (high-level) aspects in a domain model.

Steimann [Stei05] also discusses aspects of modeling, which are called facets in the present work (cf. Section 2.3.2). Facets can also be seen as concerns. They are not overlapping, i.e., crosscutting, concerns. Therefore, they can be represented independently from each other, e.g., as separate diagrams in UML. As they are not crosscutting, they do not need to be represented

cutting concerns [Fink92, Fink96] for a small number of concerns. Thus, it partially handles crosscutting concerns. See also Section A.2.1 in the appendix.

[35] See Example 3.2, where an instance of a functional crosscutting concern is given.

separately by an aspect.

Finally, [Stei05] claims to supply evidence for the non-existence of aspects in domain models by giving a semi-formal proof about the non-existence of the second-order quantification in domain models. The quantification mechanism (cf. Section 3.1.5) of aspects, especially at the programming level, is often specified in the form of second-order predicates which define *where* and *when* an aspect crosscuts the artifacts of another concern. The paper argues that *"modeling languages are usually rst-order languages . . . "* [Stei05, p. 176] and therefore concludes that the second-order quantification mechanism does not allow description of aspects in domain models. There are two counter-arguments to this "proof": first, the argument neglects the existence of aspect-oriented approaches which do not use second-order constructs for quantifying the impact location of an aspect. There is not necessarily a need for second-order quantification, thus, the crosscutting relationship between an aspect and the target can be specified by a first-order construct. Second, if it is useful to have second-order constructs in a modeling language, why should it not be part of the modeling language? This claim re ects simply the *pragmatics principle* in modeling (cf. Section 2.3.1). Other counterarguments to the claims in [Stei05] have been published in [Rash06].

3.3.2 Problems with the Understandability of Aspect-Oriented Constructs

In [Stei06, Coly06, Cons04], the understandability and readability of aspect-oriented programs is criticized. The argument in [Stei06, Cons04][36] is based on the seminal essay *Goto Statement Considered Harmful* by Dijkstra [Dijk68]. Dijkstra highlights the fact that when using *Goto* statements, the static and dynamic structure of a program differ strongly from each other. This effect leads to worse understandability compare to the use of structured statements. The same effect can be observed in AOP programs.[37] The papers [Stei06, Cons04] argue that the static structure of an AOP program strongly differs from the dynamic (i.e., the executed) structure of the program. This effect is sometimes amplified by the fact that the impact of an AOP aspect can sometimes only be identified dynamically. Both problems contribute to a poor intelligibility of the source code.

The second problem which affects understandability is that the modularization of crosscutting concerns may lead to many different aspects and many crosscutting relationships, especially when using approaches that propose a multidimensional separation of concerns, such as HyperJ [Ossh01]. This effect is called *fragmentation* in the present work. A too high fragmentation caused by the separation of concerns can affect the understandability, as the interrelationship between the different concerns cannot clearly be understood any more. The problem results from the fact that for the full understanding of a system, an aspect or a component has to be studied together with the modular artifacts that are related with it (cf. [Meie05] and also the term *interrelated focus* in Section 3.1.7). Thus the more relationships there are between the concerns,

[36] The paper [Cons04] aimed at provoking a discussion at a panel session.

[37] Moreover, in [Clar84], it was proposed to use a statement *come from* rather than *go to*, which is in fact quite similar to the mechanisms used in aspect-oriented programming today. However, this was meant as an April fool's joke.

the more complex it is to understand the interrelationship between the concerns.[38]

The problems of a decreased understandability of aspect-oriented code can also be observed in the aspect-oriented artifacts at other stages of the software process, for example in aspect-oriented requirements documents or models. The reader of such an artifact usually analyzes it sequentially. When reading aspect-oriented artifacts, he has to cope with many dependencies to understand the overall meaning. However, the reader of aspect-oriented elements cannot cope with too many dependencies at a time.

3.3.3 Breaking the Principle of Information Hiding

According to [Stei06, Coly06][39], using AOP breaks the modularity and the data encapsulation of crosscut modules. This is due to the fact that an aspect needs to access the internal structures of a crosscut module. The aspect can also interfere with the program ow of the crosscut target. Moreover, the access to the context of the crosscut target allows the aspect to globalize the variables of the target. Both break the information hiding principle [Parn72]. Apart from that, aspect-orientation can void the principle of design by contract [Meye92], as the interference with the ow of a target module may inject side effects which result in unpredictable behavior. This problem is known as *aspect interference*. All problems related to the breaking of the modularity through aspects can also be observed in aspect-oriented approaches at the design, the architectural and the requirements stage. For example, an aspect may also break the information hiding principle in a high-level component architecture.

In AOP, the problem has been addressed by so called open-modules [Aldr04, Aldr05] or abstract annotations [Ladd05], which allow modules to be crosscut at predefined points only. Similar constructs can be introduced at other stages of the software process. For example, the *extension point* construct in AOSD/UC provides this kind of mechanism. However, [Stei05] criticizes the decreased exibility of aspects, as the use of annotations leads to a reduction of obliviousness and a loss of exibility.[40]

3.3.4 Fragile Join Points

There have been various criticisms, e.g., [Stei06] and [Stor05] that aspect-oriented implementations are fragile to the evolution of the code. This is due to the fact that most approaches use names to refer to join points. As names can be subject to evolution, they are fragile and therefore are not well suited for the identification of join points [Stor05]. Therefore this might result in referencing units that do not exist, or in accidentally referencing units by an aspect, which in turn leads to unintended behavior. The problem of fragile join points is also a problem at earlier stages of software engineering. For example, AORE [Rash03] also uses a naming schema for referencing concerns.

[38] As discussed in Section 3.1.7, this problem can be mitigated by using a composition mechanism which allows switching between a view with composed crosscutting concerns and a view with separated crosscutting concerns.

[39] Beuche and Beust [Coly06, p. 73] do not explicitly mention the breaking of the modularity principle. However, they observe the problems that *"input/output is no longer suf cient to understand its work as part of the system"*.

[40] Steimann [Stei06] criticizes that the exibility of aspects is reduced by using annotation or something similar.

There are different ways to cope with fragile join points. Some of the approaches introduce predefined locations in the modules, such as abstract annotations [Ladd03], that can be crosscut. Other approaches try to overcome the problem by describing the semantics of the target location, such as [Chit07] or [Oste05].[41]

3.3.5 Further Criticism

There are numerous claims, e.g., in [Kicz01b], that AOP techniques can be used to describe and enforce contracts of software modules, following the principle of design by contract [Meye92]. However, in [Balz05], it is clearly shown that contracts of modules should be documented in the modules they describe. Nevertheless, when using AOP techniques to describe and enforce contracts, the contracts are inevitably separated from the code which is not the intention of the original concept.

3.4 Discussion

Aspect-oriented techniques can help to improve various qualities of software artifacts, such as the understandability and the traceability of requirements. However, aspect-orientation also breaks or weakens well tried and established principles, such as information hiding or design by contract. As a consequence, the understandability of artifacts may be hampered. Therefore, the use of aspect-oriented techniques for describing a problem only pays off if the advantages prevail or at least compensate the introduced disadvantages.[42] As a consequence, the use of aspect-orientation for simpler problems is probably not profitable. Nevertheless, aspect-oriented techniques can help to simplify more complex problems, such as software product lines.

[41] However, the use of such mechanisms has also been also criticized [Stei05] (cf. Section 3.3.2).

[42] Therefore, the present work investigates means to mitigate the disadvantages of aspect-oriented software artifacts.

Chapter 4

Motivating a Novel Aspect-Oriented Requirements Engineering Approach

As discussed in Section 3.1.6, the requirements engineering phase is predestined to identify and separate crosscutting concerns. Even though many aspect-oriented requirements approaches have been proposed all of them have one or more shortcomings. This chapter discusses the existing problems and identifies the desirable characteristics of aspect-oriented requirements approaches in Section 4.1. Section 4.2 summarizes the desired qualities of a new approach.

4.1 Gaps in the Existing Approaches

As outlined in the previous chapter, a good separation of concerns is desirable as early as possible, thus, if feasible, it should be established during the requirements phase. Even though, conventional requirements approaches can be used to represent crosscutting concerns, they usually result in tangled and scattered concerns. Therefore many aspect-oriented approaches have been proposed in the past, as discussed in Section 3.2.5.

Apart from the actual ability to separate concerns, there are several other characteristics which are desirable in a requirements approach. The resulting list of characteristics are detailed in Table 4.1.

Table 4.1: Criteria for the evaluation of aspect-oriented requirements approaches.

Criteria	**Description**
Traceability	How well can a requirement be traced through the software process? How well can a software artifact be associated with a requirement and how well can be the sources of requirements be identified?
Composability	Is there explicit support for *decomposing crosscutting concerns* and for *composing* them into a set of integrated artifacts? Is there a mechanism which allows automatic composition?

Continued on the next page ...

Criteria	**Description**
Modifiability and Evolvability	How readily can *changes* be introduced in the requirements artifacts? Are there features which support a change in the requirements document, such as *integrity checking* mechanisms?
Scalability	How well does the approach scale? Is the *process* provided capable of handling the requirements of a small scale as well as large scale project? Are the corresponding *artifacts* describing the requirements able to describe small scale as well as large scale software?
Understandability	How difficult is it for any stakeholder to understand the approach? Are the artifacts produced by the approach *easy to understand*, can they easily be used to *communicate with a stakeholder*, e.g., during validation?
Trade-off	How well is the identification and resolution of trade-offs between different overlapping concerns (non-orthogonal aspects) supported by the approach?
Mapping	How well can the requirements artifacts be mapped to following stages? Does the approach provide a *mapping* for all requirements? Does it allow a simple mapping?
Verification and Validation	How easy is it to conduct a validation or verification with the artifacts resulting from the requirements process?
Concern handling	How well is the *identi cation and separation of functional and non-functional crosscutting concerns* supported? Are all concerns *treated* in the same way or are there differences between non-functional and functional concerns? Can both types be represented adequately?

The remainder of this section will discuss these characteristics in more detail, and it will show how far the existing requirements approaches satisfy them. The analysis of the present work is based on the surveys of aspect-oriented approaches in [Bono04, Chit05]. The present work extends this analysis by investigating some other characteristics and by clarifying some issues.

The gaps found in the surveyed approaches are summarized in the following, whereas the detailed survey itself and the analysis of the corresponding characteristics for each approach can be found in Appendix A.

Traceability. The support for traceability comprises the recording of the *source* of the original requirements, as well as the source for the changes in a requirement. *Forward traceability* allows tracing the refinement of requirements through the resulting architectural, design and implementation artifacts. *Backward traceability* allows the artifacts created in later stages of the software process to be traced back to the originating requirements. *Cross referencing* allows the dependencies of artifacts to be traced between the requirements on the same level of abstraction/refinement.

Many of the approaches discussed in the Appendix A do not directly or only partially address this issue, e.g. the ARGM Approach discussed in Section A.3.2. However, some of them can be more easily extended for traceability than others.

Composability. An aspect-oriented approach must provide a way to decompose and modularize crosscutting concerns. A composition must allow the investigation of conflicts between aspects (cf. trade-off analysis). An automatic composition mechanism can support the understandability of the aspect-oriented artifacts (cf. Section 3.1.7). Nevertheless, at the time of writing, none of the requirements approaches in Appendix A explicitly mentions an automatic composition of the requirements artifacts. However, for some of them a composition mechanism can be quite easily implemented. For example, the AORE approach (cf. Section A.3.1) could use a simple XSLT-based [W3C 07] composition mechanism.

Changeability/Evolution support. Aspect-oriented approaches usually support better modifiability and evolvability than conventional approaches, as they separate crosscutting concerns. Furthermore, the elicitation/evolvability of a requirements specification may be supported by further mechanisms, such as consistency checking mechanisms and language support for the evolution. In some of the approaches presented in Appendix A, changes to artifacts are problematic, e.g. in the ARGM approach (cf. Section A.3.2), Furthermore, several of the approaches do not facilitate the checking of the artifacts' consistency.

Scalability. The artifacts of a requirements approach as well as the proposed process should be scalable. Several approaches discussed in Appendix A do not adequately address the *scalability*[1] issue. For example, the graphical representation of I*, NFR, ARGM, the use case diagrams of the use case based methods, the graphical representation of Theme/Doc, as well as the graphical representation of AOREC, do not scale well. Furthermore, AORE and CORE use an XML format as the representation of the concerns and the composition description, which tends to become large and unclear, the more requirements are represented.[2] There are various means to achieve scalability. For instance, the process must allow multiple engineers to work on the requirements in parallel and the representation of the artifacts must provide a way to abstract parts which are not in the current focus of interest.

Understandability. A requirements approach should be well suited for communication with (non-expert) stakeholders, as they are strongly involved in the requirements process. Therefore the language used for communicating the requirements should be *easy to understand*. There are various factors which influence understandability: (i) the representation must be scalable (see above), because too much information at the same time is too difficult to comprehend, (ii) the concepts used must be comprehensible for the stakeholders, and (iii) the different artifacts must be easy to interrelate. Usually, an approach which uses concepts that are related to the domain, such as objects or workflows, is easier to understand than an approach that uses more abstract concepts and therefore supports issues (ii) and (iii).

The approaches discussed in Section 3.2.3 neglect understandability. The use case based approaches, such as AOSD/UC discussed in Appendix A, are more intelligible for non-expert

[1]The scalability issue is strongly related to the understandability issue.

[2]This problem is marginally mitigated by the use of a tool which is presented in [Chit06] (cf. the paragraph about understandability).

stakeholders than the goal-oriented approaches. However, they are hampered by the lack of scalability, which in turn may lead to problems of understandability. AORE and CORE use XML which is in fact designed to be human-legible but not necessarily easy to understand (cf. for example [Brab05]). There are three reasons for this. First, bigger XML files are rather tedious to read and interpret. Second, there may be many different XML files for each concern, each crosscutting concern, as well as for each composition description. Third, XML has a at text structure, which describes cross references by text identifiers and references. However, they are not easy to grasp, as they have to be tediously followed and interpreted by the reader of an artifact. Moreover, the use of the tool presented in [Chit06] does not satisfyingly mitigate the understandability problem, as a requirement and the crosscutting concerns are separated from their context and therefore difficult to relate. Instead of using a text-based XML representation, a graphical representation, such as UML, is preferable to a textual description of relationships and the crosscutting artifacts. A graphical representation facilitates a simpler understanding of the interrelationships in a requirements specification.

Some of the approaches discussed involve parts of the UML language (e.g., the use case based approaches). Even though it is advantageous to have a graphical representation of the requirements, UML has some major drawbacks. It uses multiple sublanguages which describe each facet separately (cf. Section 2.3.2). The facets of the system have to be integrated individually into an overall model by the reader of the model in his mind. For this purpose, each facet model contains "glue information" which gives a hint on how to integrate two different facets. For example, a class may indicate the name of the package to which it belongs. In turn, this helps to identify how the class is integrated in the overall package structure of a package diagram.

Figure 4.1 illustrates the number of integration operations that have to be performed in UML. Each of the 13 facets can be combined with maximally 12 other facets, which is described by the X and Y axis of the cube shown. Thus using all 13 diagram types forces the reader of the model to perform up to $\binom{13}{2} = 78$ or more integration operations in mind to comprehend the overall model of the system. This can be demanding already for small models consisting of only a few facets. Thus, a model in a non-integrated language, such as UML, contains more problem-exogenous complexity (cf. Section 1.1) than a model described in an integrated modeling language.

An aspect-oriented modeling language which is based on UML can multiply the complexity effect. Usually, crosscutting concerns impact various types of system facets, e.g., they affect the structural description, the behavioral description, or the relationships between the artifacts. In the worst case, a crosscutting concern cuts across all facets. For an aspect-oriented approach based on UML, the number of integration operations in mind is multiplied for each additional crosscutting concern described as an aspect in the system. This is visualized by the Z axis of the cube in Fig. 4.1. Nevertheless, an aspect-oriented approach should try to reduce the intellectual effort needed to understand the artifacts to a minimum. Consequently, an integrated modeling language (cf. Section 2.3.2) may be more suitable as a base for an aspect-oriented modeling approach than the non-integrated UML.

Even though an aspect-oriented model in an integrated language may be more understandable than one in UML, it can still be complex (cf. Section 3.1.7). To comprehend the relationships between the various concerns, the approach must provide a means to switch between the aspect-oriented and the conventional representation according to the representation which is better suited

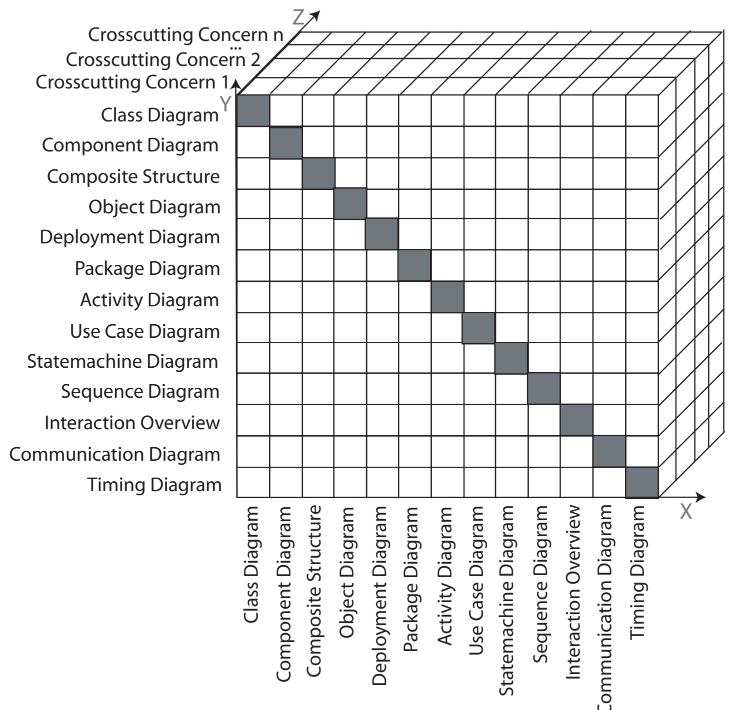


Figure 4.1: Illustration of the number of integration operations needed to compose different facets described by an aspect-oriented UML approach. The X and the Y axis span the space for the number of integration operations needed for the facets described by UML. The Z axis expands that space by an additional dimension requiring additional effort to integrate the overall model.

to the task at hand.

Furthermore when using an aspect-oriented visual modeling approach, the approach should be able to handle the layout of the model when composing the aspect-oriented model to a conventional one[3]. This is important, as otherwise the advantages from the composition get lost.

Trade-off analysis and decision support. Trade-off analysis and decision support is another characteristic that should be supported by an aspect-oriented approach. An approach should support the detection as well as the resolution of con icts between crosscutting concerns. However, some of the approaches discussed in Appendix A do not satisfactorily tackle this problem. For example, AOSD/UC discussed in Section A.3.3 only provides minimal support for handling trade-offs. A trade-off analysis is not provided. The AUCDA and SMA approaches (cf. Section A.3.5 and A.3.4, respectively) do not deal with the trade-off problem at all.

[3] See the composition characteristics discussed above.

Mapping. The mapping of artifacts from the requirements stage to artifacts at later stages should be as easy as possible. The approach should support the object-oriented paradigm, since it is the most usual. Therefore, aspect-oriented constructs should be built on top of it. Aspect-oriented artifacts should be easily mappable to aspect-oriented artifacts at later stages or at least to an object-oriented paradigm, which is the most popular way of implementing systems today.

Several of the approaches discussed, e.g., the SMA approach delineated in Section A.3.4, do not explicitly or only partially disclose how the resulting artifacts are mapped from the requirements phase to later stages.

Validation/Veri cation. There are static and dynamic means to validate/verify software (cf. Chapter 2 and especially Section 2.1.2). The validation/verification of requirements artifacts should be as easy as possible. A simple static validation/verification implies that the artifacts used need to be easy to understand and scalable. In contrast to static validation, dynamic validation is easier, as it allows automatic interpretation and/or checking of the artifacts, e.g., by simulation. However, dynamic validation is often more costly, as the problem domain must be formalized first. Most of the approaches discussed in Appendix A allow only a static verification, e.g., AOSD/UC or the AUCDA. Other approaches partially support dynamic verification, such as AORE with the PROBE framework.

Concern treatment. An aspect-oriented requirements approach should be able to handle all concerns equally. It should be possible to identify, separate and describe crosscutting as well as non-crosscutting concerns, whether they have a non-functional or a functional origin. Nevertheless, some of the approaches, such as SMA or AOREC (cf. Section A.3.4 and A.3.8) do not treat functional and non-functional crosscutting concerns equally.

4.2 Proposal for a New Aspect-Oriented Requirements Engineering Approach

All of the evaluated approaches neglect the understandability of requirements artifacts to some extent. Therefore, the present work proposes a new aspect-oriented approach that will concentrate on the understandability of aspect-oriented requirements, but also copes with other issues identified in this chapter. To simplify its development, an existing requirements approach will be used as a basis. It should facilitate understandability by satisfying the following characteristics:

1. The approach must provide a graphical representation. Modeling languages allow the visualization of important information such as relationships, including crosscutting relationships, which makes this information more comprehensible.

2. The modeling language must use an integrated visualization, as it reduces the problem-exogenous complexity. It helps to ease the understandability when a reader of the requirements artifacts needs to interpret the relationships between the concerns or needs to understand the overall meaning of a model.

3. A modeled system should be easy to understand for any stakeholder, which especially demands an easy and straight-forward description of the domain model.

4. The approach must scale well. This can be achieved through a mechanism which helps to abstract from elements that are not at the focus of interest, and by the support through a process.

5. It must allow switching between the aspect-oriented view and the conventional view. This facilitates the understandability of the relationships between the crosscutting concerns and helps to detect conflicts.

The approach must satisfy the following characteristics, or, at least, it should facilitate a simple implementation to satisfy them:

6. Non-functional and functional concerns must be handled equally and adequately.

7. The approach should avoid or mitigate the problems introduced by aspect-orientation, such as fragile join points and the violation of the information hiding concepts.

8. There should be a means for a dynamic validation, e.g., by simulating a requirements model.

9. The source of requirements as well as the source of changes should be recordable. Artifacts should be traceable throughout the software process.

10. Artifacts should be easily modifiable. Preferably, an automatic mechanism should help to maintain the consistency of the model after changes.

11. The approach must allow the recognition of trade-offs between crosscutting concerns impacting the same target and provide a way to resolve these conflicts.

12. Artifacts from the requirements stage must be easily mappable to the following stages of the software process.

The remainder of this work deals with the implementation of an aspect-oriented approach which satisfies the above characteristics. It is based on the object-oriented, integrated modeling language ADORA [Joos99, Glin02b], since ADORA satisfies several of the characteristics proposed above. The following Chapters 5 and 6 introduce the ADORA language and define it more formally. The chapters following after this introduction deal with the aspect-oriented extension of the ADORA language and its validation.

Chapter 5

Basics of the ADORA Approach

This chapter introduces the modeling language ADORA. It will be used as a means for demonstrating how to introduce aspect-oriented language constructs in a requirements modeling language. The ADORA language comprises several language features (cf. Section 4.2) which are desired when introducing aspect-oriented constructs.

ADORA is an object-oriented method for the **A**nalysis and **D**escription **O**f **R**equirements and **A**rchitectures. The method has been developed in the Requirements Engineering Research Group at the University of Zurich and is still evolving [Joos99, Glin02b, Bern02, Xia04, Seyb06a, Rein08]. In this chapter, the concepts and the language elements of the ADORA approach are presented in detail.

The ADORA language is used for modeling requirements and architectural design specifications on the logic level [Glin02b]. It can be used for describing models of functional requirements at a later stage in the requirements process. Non-functional requirements can be included as textual annotations in ADORA models. Furthermore, ADORA models are supposed to evolve within an incremental requirements process, as outlined in [Seyb04a, Seyb06a].

The ADORA language aims at the elimination of modeling problems and anomalies which occur when using other modeling languages, such as UML [OMaG03b]:

- **Loosely Coupled Language:** UML is a loosely coupled language. Basically, UML consists of a set of independent sub-languages which are neither integrated on the level of the language nor on the level of visualization. The resulting problems are twofold:
  - The lack of integration on the language level leads to more redundancy in the language. Thus, many constraints are needed to enforce horizontal consistency [Mens05, Cram07], i.e., the consistency between the sub-models.
  - The lack of an integrated visualization of the elements leads to models which are difficult to understand, as the reader of the model[1] must integrate the loosely coupled and separately visualized sub-models in mind, which can be a demanding task already for small models.

[1] The reader of the model belongs to the group of stakeholders involved in the requirements or architectural phase of the software.

- **Class Models:** UML uses class models as the central modeling element. Class models have poor capabilities for expressing the context of their instances. For the reader of a model, it is easier to work with instances instead of classes themselves, because they are more concrete, i.e., they describe the system to be built in a more tangible way. This problem is not satisfyingly solved by UML.

- **Decomposition:** Another problem in UML is the poor aptitude for model decomposition. Decomposition is a means of coping with the complexity of a system. The support for decomposition in the UML language up to version 1.5 is only rudimentary (e.g. by package constructs or within state diagrams). Even though newer versions of UML provide better support, the support is still not optimal.

The language concepts of ADORA aim at the elimination of these problems. The remainder of this chapter elaborates on the language. Section 5.1 discusses the language concepts in detail, whereas Section 5.2 gives an overview of the syntax and semantics of the language. The language definition of ADORA is delineated in the next Chapter 6. It will be the basis for the introduction of the aspect-oriented modeling elements in Part II of this work.

5.1 Language Concepts of ADORA

As summarized above, the ADORA language overcomes the problems discussed above through the following concepts [Glin02b]:

- The ADORA modeling language is based on an integrated modeling language concept (cf. Section 5.1.3).

- It is based on *abstract objects* (i.e., prototypical instances) rather than on a class model (cf. Section 5.1.1), which permits decomposition and facilitates the integration of the language.

- ADORA models are hierarchically decomposable (cf. Section 5.1.2).

- The ADORA language employs a view concept which allows combining the representation of different facets[2] of the system (cf. Section 5.1.4) in the same diagram.

- ADORA applies three mechanisms for abstracting models (cf. Section 5.1.4), i.e., for setting a particular focus of interest on a given set of elements.

- The language supports an adaptable degree of formality (cf. Section 5.1.5), which allows models to be described either informally, semi-formally, or formally.

- ADORA supports the controlled evolution of requirements by particular model elements (cf. Section 5.1.6).

[2] Throughout this work, the term *facet* is used instead of the term *aspect* so as not to confuse the reader when talking about aspects in the sense of aspect-orientation.

5.1.1 Modeling with Abstract Objects

The main structure of the ADORA modeling language is based on *abstract objects*, rather than on classes or types, which are the basis for most of the other object-oriented modeling languages, such as UML [OMaG03b]. Abstract objects are prototypical instances of types. They do not represent objects at runtime.[3] Hence, they do not have concrete values stored in their attributes. Nevertheless, they designate the concrete instances and their location in the overall object structure of a system which is expressed by a part-of relationship.

Class models are not as powerful as object models in expressing composition relationships (cf. Section 5.1.2 and [Joos99]). Hence, it is not possible to say in which context an instance of a class is used. This is illustrated in Fig. 5.1. Figure 5.1 (a) shows an object decomposition describing an electronic management system of a library. The system consists of several elements which are modeled by abstract objects denoted by rectangles. A library system consists of an *Authorization* object, a *BorrowManager* object, a *UserAdministration* object and a *BookAdministration* object. Each of the latter three objects contains an instance of a *Logger*, i.e., a separate logging object is part of each of these components. They may be used to log, for example, security related events. In this example model, the context of each object is clearly apparent.

In contrast to Fig. 5.1 (a), Fig. 5.1 (b) shows the same system described by a UML class diagram. In the class diagram the context of the instances is hard to identify. Only the attributed cardinalities of the composition give a hint about the number of instances which are present at runtime. Therefore, the connection between the different objects is more difficult to comprehend than in the object model.[4]

Hence, the use of object structures instead of classes results in more expressive models and fosters understandability, which is especially important for requirements models, as they also have to be read and understood by the non-expert stakeholders of a project.

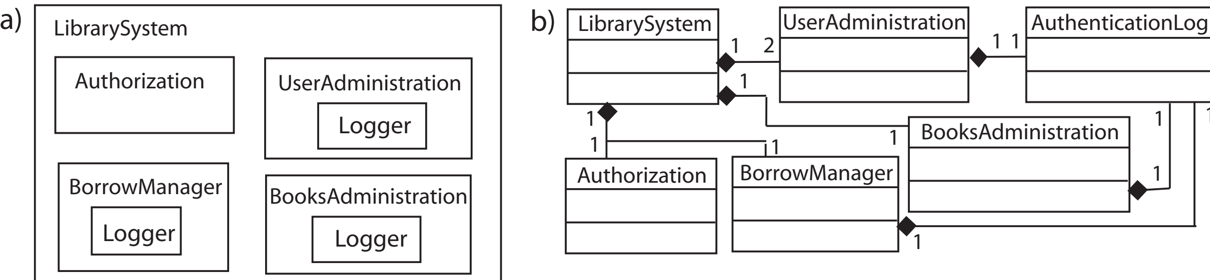


Figure 5.1: An object composition of a library system in (a) vs. the corresponding class representation of the same system in (b).

[3] However, when executing an ADORA model, a concrete instance of an abstract object is created [Seyb06a].

[4] In versions 1.0 to 1.5 of UML, the concept of abstract objects is known too, but it is not used coherently. In UML 2.0, the concept of abstract object is extended and integrated more coherently as so-called structured classes.

5.1.2 Hierarchical Decomposition

Hierarchical decomposition[5] is the main means of handling big models and coping with their complexity. Furthermore, the use of hierarchical decomposition facilitates the concepts of information hiding [Parn72] and increases the maintainability of components.

The hierarchical decomposition of a model means to divide it into highly cohesive parts which are in a part-of relationship with other parts. It is enabled by the use of abstract objects and the part-of relationship which is expressed by nesting abstract objects. Each object in the decomposition hierarchy is encapsulated and recognizes a set of responsibilities. An object may provide a service to other objects that may be accessed through a well-defined interface.

For example in Fig. 5.1 (a), the *LibrarySystem* consists of several components, e.g. the *BorrowManager* component. In turn, the *BorrowManager* consists of the *Logger* component and the *BorrowManager* might use the services provided by the *Logger*.

5.1.3 Integrated Modeling Language Concepts

ADORA is based on an *integrated modeling concept*, i.e., different facets of a software system are defined in one coherent modeling language, which in uences the language design and the visualization of models:

i. **Language Design:** In contrast to non-integrated modeling languages, integrated languages are based on a coherently well-defined meta-model. This means that the language is planned from the beginning as one integrated language [Joos99, Xia04], rather than being composed from separately designed sub-languages where each of them is representing a facet, such as behavior, static structures, use cases, etc. Non-integrated modeling languages contain more redundancy than integrated ones [Joos99, p. 76] and need therefore more constraints to enforce consistency between the different facets.

ii. **Model Visualization:** In non-integrated modeling languages, such as UML, the facets of a model must usually be visualized separately. Conventional modeling tools handle this problem very often by describing different parts of a model in different diagrams, i.e., there is no integrated visualization and the diagrams are connected through some kind of linking mechanism. The navigation through the model structure has to be done by following links to other model elements, usually causing a so-called explosive zooming [Seyb03]. An explosive zoom displays the newly shown model parts in a separate window or area and the reader of the model easily loses the context of the element. Thus, explosive zooming makes it hard to understand how the visualized parts are related to each other.

ADORA does not suffer from these problems as it is based on an integrated modeling language. The particular facets of the system model are represented as *views*. A view can be seen as a projection (in the mathematical sense) on the model. For the visualization of a model, views

[5] A hierarchical decomposition is a composition relationship in the terminology of UML class model. An aggregation relationship in the sense of UML has to be modeled in ADORA either by a decomposition or by an association (cf. Section 5.2.2).

are displayed in combination with each other in an integrated way (see Fig. 5.2). There are six different views in ADORA, which are discussed in detail in Section 5.2:

1. **The Base View:** The base view visualizes the static structure of the model. It consists of *abstract objects* and sets of abstract objects which are just called *object sets* in the following. Objects are visualized as rectangles and object sets as a stack of rectangles, respectively. For instance, the rectangle with the name *LibrarySystem* in Fig. 5.2 represents an abstract object, whereas the stack of rectangles with the name *User* denotes an object set.

2. **The structural view:** The structural view comprises associations connecting components. An association denotes either a structural relationship or a communication channel between objects. They are visualized as lines which may have two directed roles. A bold association is called *abstract* and represents a set of hidden associations or indicates an association which is not fully evolved at this time. Figure 5.2 illustrates the use of associations. The line between the object *BorrowManager* and *Authorization* denotes an association with the two roles *AuthenticateBorrowing* and *AuthorizeBorrowing* which is used for the communication between both objects.

3. **The behavior view:** The abstract behavior of objects is described by means of statecharts [Hare87]. A statechart consists of states and transitions. They are denoted in ADORA by rounded rectangles and arrows, respectively. For instance, in Fig. 5.2 the behavior of the object *Authorization* is shown. For example the rounded rectangle named *Wait* is a state and the arrow connecting to the state *UserInfo* is a transition.

4. **The user view:** The user view describes the use cases, which are called *scenarios* in the ADORA approach. A use case is modeled by a *scenariochart* [Xia04] and describes the interaction between the system and an environment object. It is visualized as an ellipse and can be decomposed into sub-scenarios forming a scenario tree. In Fig. 5.2, the rounded rectangle *BorrowBooks* in the *BorrowManager* component denotes a scenario which is connected with three sub-scenarios. An association connects the root scenario with the *LibraryUser* environment object in the system context.

5. **The context view:** The context view describes the actors in the context of the modeled system. Actors are called environment objects in ADORA. The environment objects are denoted by hexagons. Figure 5.2 illustrates several environment objects of the library system, e.g., the *LibraryUser*.

6. **The functional view:** The functional view defines additional properties, such as attributes and operations for an object or object set. This view is not visualized together with the other views. Its information is displayed separately. An example of it can be found in Section 5.2.6.

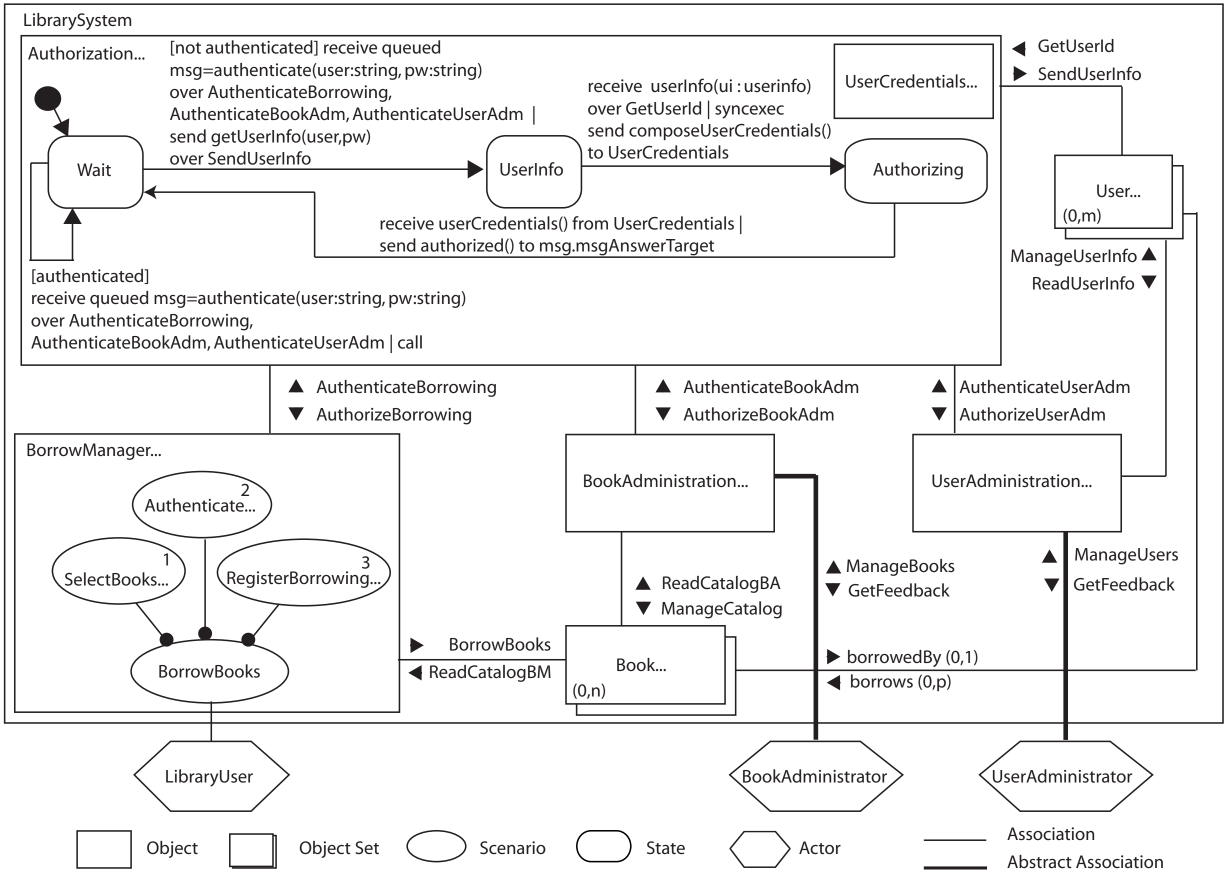


Figure 5.2: Parts of a library system modeled in ADORA, based on [Meie06].

5.1.4 Visual Abstraction Mechanisms

The use of an integrated modeling language facilitates better model understandability [Bern99b]. However, an integrated model tends to result in a cognitive overload if every part is visualized at the same time. This is due to the fact that already small models with a few elements may have rather large and complex graphical representations with a lot of details. Nevertheless, there are many elements which need not to be visualized, as they are not at the focus of interest of the model reader. Hence, it is desirable to have a mechanism reducing the cognitive overhead by just displaying the parts in the model that are of interest. For this purpose, the ADORA language provides three *visual abstraction mechanisms* which are applied to the integrated models.

1. The *vertical model abstraction* applies to the elements which are nested into another node, i.e., which are a part of another node. Applying a vertical abstraction to a node hides or shows these inner elements. For example, the details of the *Authorization* component in Fig. 5.2 can be hidden if they are not

   Applying a crosswise abstraction visualizes just specific concerns or viewpoints of the

model. For example, hiding the components *UserAdministration*, *BookAdministration*, *Book*, *User*, the environment objects *BookAdministrator* and *UserAdministrator* results in a model representation showing only the components *BorrowManager* and *Authorization* as well as the environment objects. The remaining elements deal with the borrowing of books.

Algorithm. From a technical point of view, all three abstraction mechanisms are based on toggling the visibility of one or more model elements. This operation can be done by using a logical fisheye zoom algorithm which is described in [Bern02]. The algorithm hides particular model elements and therefore reduces the cognitive overhead. It allows the hiding of elements and then shrinks the freed-up space in the diagram. Conversely, it creates enough free space, when a model element is shown again.

The aim of the algorithm presented in [Bern02] is to preserve the layout of the modeler's mental map [Eade91] despite the changes in the model visualization caused by the hiding of an element. Note that the layout resulting from the mental map is also called secondary notation [Petr95]. However, the algorithm in [Bern02] has deficiencies as it is unstable in certain situations. This problem leads to major changes in the secondary notation when executing the inverse of a zoom operation. To overcome these deficiencies, the algorithm was radically redesigned in [Rein07].

Fisheye algorithms, such as the one presented in [Rein07], can be used not only to abstract a model but also to implement a smart editing that allows the automatic allocation of space when inserting a new model element and the disposal of freed space when deleting a model element [Seyb03].

Orthogonality of the abstraction mechanisms. The abstraction mechanisms presented work independently of each other, e.g., if a state *A* is hidden by the crosswise abstraction mechanism, and then the behavior view is hidden, *A* stays hidden if the behavior view is shown again.

Indicating partial view. Zoomed out components, states and scenarios indicate their hidden content, by adding a trailing ellipsis symbol (...) to their name. Furthermore, there may be so-called abstract associations indicating associations with a hidden source and/or target node. Both abstraction indicators are discussed in Section 5.2 in more detail.

5.1.5 Variable Degree of Formality

In ADORA, all language constructs support a *variable degree of formality*. Thus, elements can be described either informally, semi-formally, which is a mix of formal and informal elements, or formally. Therefore, the description of model parts can range from informal natural language specifications to formal models. Nevertheless, a mixed description consisting of formal and informal elements is employed most of the time. For example, during the evolution of software requirements towards a complete and formal software specification, formal as well as informal

elements can be contained in the model. An evolutionary requirements process can be supported by a semi-formal simulation of the model as described in [Seyb06a].

Furthermore, the ability to specify system parts with a variable degree of formality helps to handle the varying risks of system parts. Parts that have a higher risk can be modeled formally, whereas low risk parts can be modeled informally and need less effort [Seyb06a].

5.1.6 Requirements Evolution Support

The software process during the requirements specification phase is often evolutionary, because requirements are not clear and complete from the beginning and have to be elicited and refined first. Hence, a requirements language should support a planned evolution by an adequate mechanism. ADORA supports an evolutionary requirements process by providing language constructs allowing the modeler to mark particular model parts as partial [Xia04], i.e., as *intentionally incomplete*.

The missing information of partial elements is intended to be completed later in the modeling process. By using these partial constructs, the understandability of the model is increased because it helps to distinguish between intentional and unintentional incompleteness in the specification. Furthermore, it helps in tracking the incompleteness through the whole semi-formal simulation process as described in [Seyb04a].

In ADORA, the evolution of requirement models is supported by the so-called partial indicator for components, states and scenarios. If the partial indicator is set, it is also visualized by an ellipsis added to the name of the partial element. Furthermore, there is the so-called abstract association[6] which indicates the intentional incompleteness of it. Both elements, the partial indicator and the abstract association, are discussed in detail in Section 5.2. Note that there are also partially *viewed* models, which are the result of applying the abstraction mechanisms described in Section 5.1.4. Partial models and partially viewed models are denoted by the same syntactical elements, i.e., the partial indicator and abstract associations.

5.2 Overview of the ADORA Language

In this section, the language elements of the ADORA language are discussed in more detail according to their view membership.

5.2.1 Base View

The basic structure of an ADORA model consists of abstract objects, which are prototypical instances, as discussed above. An object set is a set of similarly typed objects. The superordinate concept of abstract objects and object sets is called *component*, i.e., components represent both abstract objects and object sets. A component comprises a cardinality denoting the minimal and the maximal number of objects that exist. It is denoted by a pair (a, b), where a is the minimum

[6] More precisely, a partial association is a so-called *manually abstracted* association.

and b is the maximum cardinality. A component which has $(1,1)$ as cardinality is an abstract object, because exactly one instance exists. Abstract objects are denoted by a rectangle, such as the object *BorrowManager* in Fig. 5.3, which shows the same system as in Fig. 5.2 but with a different focus on the system. The name is either centered or at the upper left corner of the object, depending on whether the object contains parts or not. The cardinality tuple of an abstract object is not displayed.

In contrast, object sets are extensions, i.e., they comprise a set of similarly typed objects and have a cardinality different from $(1,1)$. An object set is represented by a shadowed rectangle, as illustrated in Fig. 5.3 by the component *User*. An object set shows the cardinality tuple in the lower left corner. In contrast to an abstract object, an object set can comprise operations which are used to manage the extension, e.g. to delete and create objects.

Structure of a component. The structure of a component is twofold: it comprises a set of *properties* and a set of *parts*. The *properties* consist of attributes, operations, values of so-called standardized properties[7], and directed relationships. The first three elements are described in the functional specification of an object, which is discussed in Section 5.2.6. The directed relationships originating in a component are either transitions (a component can be a part of a statechart, cf. Section 5.2.3) or associations (cf. Section 5.2.2).

An object can be of a specific type, which helps to avoid redundant structures in objects. The type name is indicated after the component name, separated by a colon.[8] An example of an object with a specific type is given by Fig. 5.3 where the object *UserAdministration* contains an object *Logging* which is of the type *Logger*. Furthermore, an object might be agged as external. An *external* object is similar to an environment object. However, it is embedded in the system, and therefore, it is a part of it. External components are denoted by the type *external* and do not have an inner description, i.e., a functional specification or a behavior description (cf. Section 5.2.5).

Decomposition. A component can contain other components as *parts*. These parts can contain in turn other components, thus a component can be decomposed recursively, creating a hierarchical decomposition of components. Moreover, there may be other parts embedded in components, such as states (cf. Section 5.2.3) and scenarios (cf. Section 5.2.4) which can also be decomposed hierarchically as statechart and scenariochart, respectively. The graphical representation of the part-of relationship in ADORA is strict, i.e., neither components nor other decomposed elements may be part of multiple elements, nor may an element contain itself [Glin02b].[9] In Fig. 5.2, the decomposition is illustrated by the *LibrarySystem* object which comprises several other objects. Furthermore, there are examples of other object parts, such as scenarios, represented by ovals, in the object *BorrowManager*, or the states, represented by rounded rectangles, in the object *Authorization*.

[7] All standardized properties are defined in an directory structure which is discussed in Section 5.2.7.

[8] Types are defined in the type directory which is discussed in Section 5.2.7.

[9] An exception to the rule of being part of more than one composite is given with scenariocharts (cf. Section 5.2.4). They form their own decomposition hierarchy besides their membership in the decomposition hierarchy of components. Thus a scenario can be part of more than one composite.

Abstracting components. The modeler has the ability to hide and show the inner parts of a component using the abstraction mechanisms discussed above. A component whose inner structure is completely or partially hidden is indicated by a name with an ellipsis. This abstraction indicator helps the reader of the model to distinguish the abstracted components from non-abstracted ones.[10] Figure 5.3 illustrates the library system with an abstracted component *Authorization*.

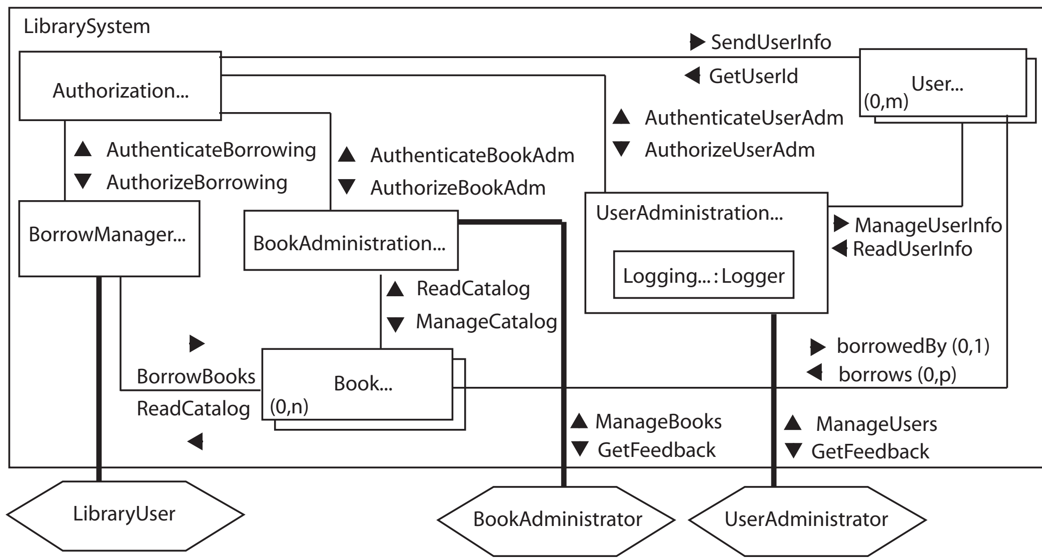


Figure 5.3: The library system (cf. Fig. 5.2) with a different focus on the model. The component *Authorization* is abstracted, whereas a part of the component *UserAdministration* is shown.

5.2.2 Structural View

The structural view combines the base view with the associations. Associations are unidirectional binary relationships, which can connect different types of ADORA elements. The connected elements are called the constituents of the association. An association can either connect two components, an *external* component and a scenario, or a scenario and an environment object. An association may be assigned to two roles, one from the source to the target (source-to-target role) and the other one from the target to the source (target-to-source role).

Application of Associations

There are two purposes for associations in a model. First, they can denote a structural relationship which is used for describing a connection between two data entities, as done in the

[10] In fact, denoting the abstraction by three trailing dots is also applied to other elements which are hierarchically decomposable, such as states and scenarios.

entity relationship modeling with relationships. Therefore, associations can be associated with cardinalities indicating the minimal and the maximal number of instances that take part in the association. For instance in Fig. 5.3, the object sets *Book* and *User* are connected by a structural association. The role *borrows* expresses that a user of the library can borrow between zero and *p* books, where $p \leq n$. The role *borrowedBy* denotes that one book can have zero or one user borrowing it.

The second purpose of an association is its use as a communication channel over which a source and a target element exchange messages with each other. The source element can send a message to the target by addressing it in the corresponding source-to-target role. An answer may be received over the same association. In this case, the association must define a target-to-source role which can be addressed to get the answer. By using associations as communication channel, it is possible to decouple components from each other, which is important for properly realizing the information hiding principle [Parn72]. Moreover, the decoupling enables the modeling of contracts between components.

Figure 5.3 illustrates the use of an association as a communication channel. The association with the roles *AuthenticateBorrowing* and *AuthorizeBorrowing* between the component *Borrow-Manager* and the component *Authorization* are used a as communication channel.

Abstract Associations

There are so-called *abstract associations* which are visualized as bold lines. They occur in two cases:

1. A so-called *calculated abstract association* denotes one or more concrete associations. It is introduced if a constituent of an association is hidden [Xia04, Section 2.2] in the view of the model.
2. A so-called *manually abstracted association* is given if the association is partial, i.e., it is intentionally incomplete. In this case, the *abstract* indicator denotes an unfinished evolution of the association [Xia04, Section 3.5].

Calculated abstract associations. The following situation illustrates when a *calculated abstract association* occurs. Suppose a component *B* which is part of a component *A* and a component *D* that is contained in a component *C*. *B* and *D* are connected by an association α, as shown in Fig. 5.4 (a). An association is hidden if at least one of its constituents is hidden. Nevertheless, the information expressed by an association is crucial to the understanding of the model and must be preserved when applying an abstraction to the model. Thus, in the example, when hiding the component *B* or *D* or both, a calculated abstract association is generated by the view of the model and shown instead of α. Hence, if *B* is hidden, as shown in Fig. 5.4 (b), the abstract association is drawn between its constituents *A* and *D*. *A* and *D* are derived from the constituents of α. *D* is the constituent of α that is still visible, whereas *A* is the next direct or indirect parent of α's hidden constituent *A*. Correspondingly, the abstract association is calculated when *D* is hidden, or *B* and *D* are hidden together. Both situations are illustrated by Fig. 5.4 (c) and (d), respectively.

A calculated abstract association β may represent a set of concrete associations δ. In this case, all label parts of β, i.e., the cardinality, the name and the direction of β are displayed only if the represented label parts of all associations in δ are consistent. Thus, β has only a role name with a direction if all associations in δ have the same role names and the same direction. Moreover, the cardinality shown by β must not be more restrictive than the least restrictive cardinality of the represented association [Glin02b]. If there is an inconsistency, the corresponding label part is left empty for β.

Manually abstracted associations. Compared to calculated abstract associations, manually abstracted ones denote a partial, i.e., an unfinished, evolution. In this case, the abstract indicator has to be set manually by the modeler. There are two reasons for an association to be partial: either the source or the target of the association is not evolved yet, or the manually abstracted association represents a set of associations which are not evolved yet. The label of a manually abstracted association has to be set and maintained manually.

Hierarchical relationship between abstract associations. Components can be in a hierarchical decomposition relationship. As components may be connected by associations, the hierarchal decomposition is also re ected by so-called super- and sub-associations re ecting the hierarchy of their constituents. It is meaningful to express this relationship between associations, which is done by so-called interrelationships denoted by a dashed line. The super-association of a sub-association is defined by the nearest abstract association in the decomposition hierarchy.[11] There are several constraints that have to be fulfilled by the interrelationship between a super- and a sub-association, which are discussed extensively in [Joos99, Xia04]. Fig. 5.4 (e) shows another situation of a model where an interrelationship arises between an abstract super- and a sub-association. It is displayed by the dashed line connecting both associations.

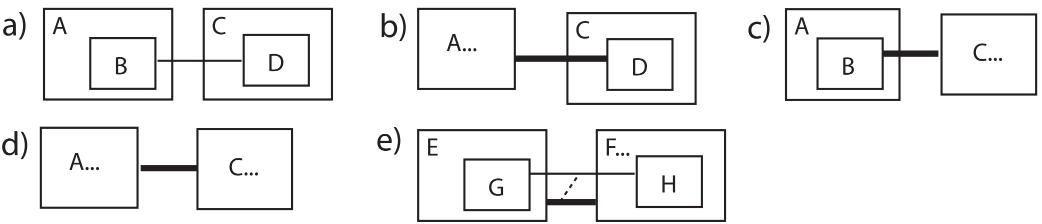


Figure 5.4: Examples of associations, abstracted associations and interrelationships. In (a), an initial situation is given. In (b) - (d) the model is shown after executing some zooming operations. In (b), component *A* is zoomed out and in (c), *D* is hidden. Correspondingly, an abstract association is displayed. In (d), *B* and *D* are hidden. Finally, (e) shows another situation illustrating the case where an interrelationship is shown.

[11] An elaborated definition of super-associations can be found in [Joos99, Section 5.3.3].

5.2.3 Behavioral View

Each component in an ADORA model can contain an event-based behavior description based on statecharts, i.e. *hierarchically decomposable state machines*. Statecharts were originally proposed by Harel [Hare87].

Syntactical Elements

As outlined above, *states* and *transitions* form the behavior description of a component. States describe the condition of being, whereas transitions designate the change from one state to another state. Figure 5.5 illustrates what a behavior description looks like by showing another view of the library system from Fig. 5.2. In this model, the component *Authorize* shows some details of its behavior description. States are shown as rounded rectangles, transitions as solid arrows.

A state can be decomposed hierarchically, which allows coping with the complexity of a behavior description. A decomposed state is called a *complex state* and contains in turn a statechart.

A state might have incoming and outgoing transitions, thus, a statechart is a directed graph, where the vertexes of the graph are represented by the states, the abstract objects, and the object sets. Transitions designate the directed edges. A set of states connected by transitions is called a *state group*. In fact, a state group is a *connected component*[12] [Cap93, p. 77] in a statechart. Concurrent behavior is modeled by several state groups within the same component.

Apart from states, components may also be a part of a behavior description. In this case, they are connected by a transition to a state group and act as complex states. Components which act as states can be used to model the life cycle of a component explicitly. They can also be used to describe abstract concepts like operational modes (on, off, operating, startup, etc.) [Glin02b]. An in-going transition denotes the component's instantiation, whereas an out-going transition specifies its destruction. Moreover, a component which is part of a statechart can have a start state indicator, denoting it as the initially active state of the corresponding state group. An example is given by Fig. 5.6 where a component in the library system models the on/off mode of the *BorrowManager*'s bar code reader component. The component *On* denotes an operational mode. In the *on* mode, several properties and elements are available (modeled by the *On* component) which do not exist in the *off* mode.

In statecharts, transitions may be associated with a label that specifies under which condition the transition is triggered. The triggering condition may include a *guard* and a specification for a *received message*. Moreover, the label may specify an action part which either calls an operation or executes a sequence of actions. A called *operation* is defined in the functional specification (cf. Section 5.2.6) of the component and executes a sequence of actions. The operation may be either executed quasi-synchronously or asynchronously (cf. [Glin95, Glin02b, Glin02a, Seyb06a]). If the action part defines a sequence of actions, it is embedded in the transition label itself. The actions may, for example, generate new messages which trigger further transitions.

Messages may be sent over three different types of *communication channels*. First, they can be sent over associations to another component. The association and the direction of the message sent are specified by the referred role names in the send statement. Second, messages can be

[12] Connected components of a graph must not be confused with the concept of ADORA components.

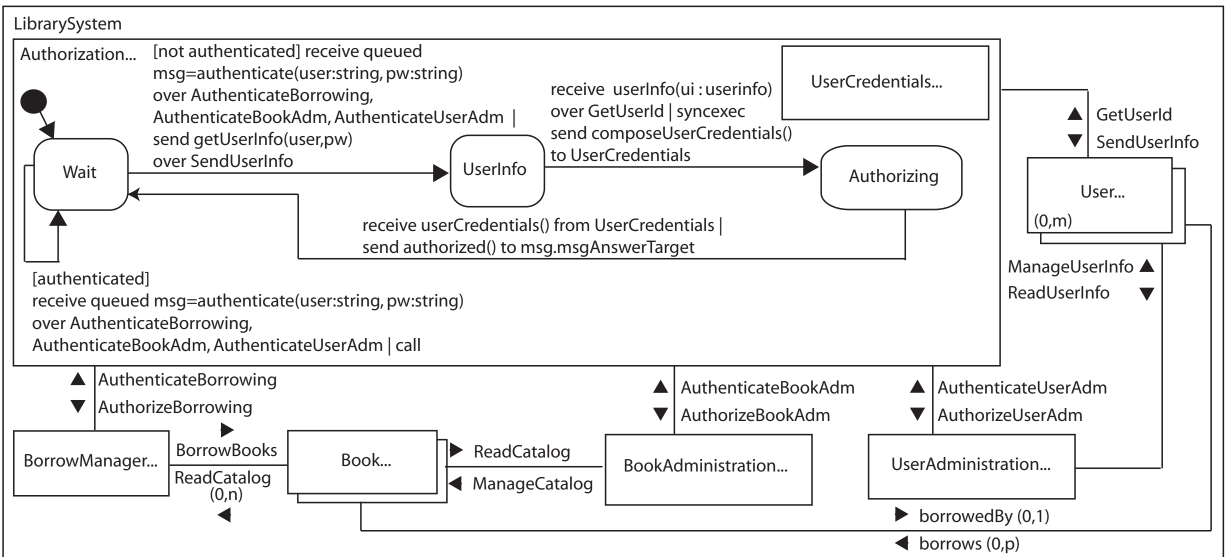


Figure 5.5: Illustration of the behavioral description. The model shows the behavior description of the *Authorization* component.

sent to any direct or indirect child or to the direct parent by specifying the name of the target component. Third, it is possible to broadcast a message by specifying no target association role or target components. Broadcasts have a local scope, thus, a message is only delivered within the object in which the message was created [Glin02b, Glin02a].

Execution Semantics, Concurrency and Redundancy Avoidance Mechanism.

The execution semantics of ADORA statecharts differs from the usual execution semantics of other approaches such as UML or the statechart approach. ADORA statecharts have a quasi-synchronous timing semantics [Glin95, Joos99, Glin02b, Glin02a, Kreb04, Seyb06a]. Although the previous work extensively discusses the execution semantics, there are two open issues in ADORA execution: *race conditions* and *redundancy* in behavior descriptions. In [Meie09a], the problems of race conditions and unnecessary redundancy in the behavior description of ADORA are discussed in more detail and a possible solution to both problems is introduced. The present section summarizes the contents of this work brie y.

A very common situation in ADORA models is when several components access a service provided by another component. Fig. 5.5 exemplifies a service-providing component (in the following simply called server component). *Authorization* provides an authentication service for all components that are connected by an association. The components, such as *BorrowManager*, send their authentication messages to the authorization component. The requests for an authentication are processed and the answer, whether the authentication was successful or not, is sent back.

However, the language specification and the execution semantics proposed in the previous

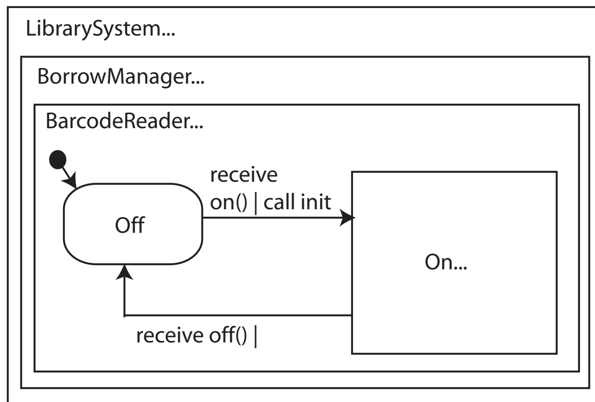


Figure 5.6: Illustration of the life cycle modeling of objects. The model shows the behavior description of the *BarcodeReader* object which may be either in *off* or in *on* mode.

work on ADORA is not able to cope properly with server components.[13] In the case that a server component accesses a commonly used resource, a so-called race condition (cf. for example [Tane97, p. 57 ff]) may occur. Race conditions aw the intended semantics of the model and therefore make it unclear. Apart from race conditions, behavior description may result in a lot of redundancy and unclear behavior descriptions. This is due to the fact that communication channels, such as the role of an association, are statically referred to by the transition labels. As each receive statement of a transition could only receive messages from one channel, the corresponding behavior had to be duplicated before a solution to the problem was proposed in [Meie09a]. Note that the redundancy and the race conditions problem are strongly intertwined [Meie09a].

In [Meie09a] a solution for the concurrency problem is presented. It is based on a simplified monitor mechanism [Hoar74, Hans75] and allows a *critical section* to be defined within a statechart. The critical section is *entered* when a state S is left by an outgoing transition that is marked as *queued* and triggered by a message m. The critical section is left, if the state S is reentered, i.e. the critical section describes a cycle in the statechart. Any message having the same signature as m is queued if the statechart is in the critical section. Any other message is processed normally. After the critical section is left, the next message from the queue is removed and processed.

A solution to the redundancy problem is also presented in [Meie09a]. It proposes a mechanism which allows the storing of the meta-information of a message and the answer channel of a message to be derived dynamically. By doing so, there is no longer any need to redundantly define the same behavior for multiple communication channels.

The example in Fig. 5.5 illustrates the use of both mechanisms. The message *authenticate(user:string, pw:string)* triggering the transition between the state *Wait* and the state *UserInfo* is marked as *queued* and indicates the entrance of the critical section. The triggering of the transition between the state *Authorizing* and *Wait* denotes its exit. In the example model, the queued

[13]However, a precise semantics is especially important when an ADORA model needs to be formally modeled.

message *authenticate* can be received over various channels. This is indicated by the channel name list in the receive statement of the corresponding transition. The meta-information of the message is assigned[14] to a variable *msg* when triggering the transition. This meta-information can be accessed later and is used for dynamically determining the answer channel of the message. In the figure, the label of the transition which exits the critical section contains an action that determines the answer channel by involving the expression *msg.msgAnswerTarget*. The corresponding send statement uses the dynamically determined channel to send an answer back to the service consumer.

5.2.4 User View

The user view specifies the interaction between the modeled system and an environment object [Glin02b, Seyb06b]. Thus, it provides the use cases for a system, i.e., a high-level protocol of the logical interaction. It defines the order in which the stimuli from an environment object are injected into the system, and defines the order in which the corresponding system responses occur. However, the user view does not deal with a user interface design.

Use cases are called scenarios in ADORA. Scenarios have a name and can be decomposed into sub-scenarios forming a tree. Such a scenario tree is called a scenariochart [Glin02b, Xia04] and is similar to a Jackson JSP diagram [Jack75]. In the scenario tree, *directed* scenario connections link the nodes. Directed connections are necessary due to the fact that such a scenario tree has not necessarily a top-down and left-right order in an ADORA model. A scenario node is represented by an ellipse, whereas a scenario connection is visualized as a line with a dot at one end. The dot indicates the parent of the two nodes taking part in the connection. A set of scenario nodes linked together by scenario connections are also called a scenario group.[15]

The decomposition hierarchy of a scenario is orthogonal to the decomposition hierarchy of the components. As each node in a scenariochart can communicate with a particular part of the system, it is embedded in a component. The children of a scenario that is embedded in a component C must also either be embedded in C or in a child, grandchild or any descendant of C.

Scenariocharts are also used to drive the simulation of reactive systems [Seyb06a]. A scenariochart is interpreted during the simulation by an in-order traversal of its nodes [Seyb06b]. Each node may contain so-called transformation expressions. A transformation expression denotes either an injection of a stimulus into the component in which the scenario is embedded, or it denotes the awaiting of a response from the component. As soon as a node containing a transformation expression is encountered during the traversal, it is executed.

Each node is of a particular type used to control the way in which it is visited. The type is visualized by a specific symbol in the oval of the scenario. There are four types: *Root*, *Sequence*, *Alternative* and *Parallel*. In Fig. 5.7 (a), the visualization of the scenario types is illustrated.

The siblings, i.e., nodes with the same parent, in a scenariochart node must have the same type, in order to be meaningful. The *Root* type is used to denote the root of a scenario tree and

[14] See the expression *msg=authenticate(user:string, pw:string)*.

[15] *Scenario group* is analogously defined to the term *state group* (cf. Section 5.2.3).

is indicated by the absence of a type symbol. The types *Sequence* and *Alternative* have the same semantics as in the JSP diagrams. The symbol of a *Sequence* node is displayed by a number indicating the order of the node in the sequence. This is required due to the fact that scenariocharts are not necessarily drawn from top-down and from the left to the right. In a simulation, a sequence of nodes is visited in a sequential order, specified by the ordering numbers.

In contrast to JSP diagrams, iteration is not described by a separate node type but rather as a property of all node types. Thus, each node may additionally have an iteration indicator which is denoted by an asterisk (*). Each node with an iterator indicator may also contain a condition which must evaluate to *true* in order to execute the node. Whenever the condition is satisfied, the sub-tree of the node with the iterator property is visited, otherwise the execution of the node is finished.

Nodes of the type *Parallel* are not part of the original JSP diagrams. Their symbol consists of two vertical, parallel lines (||). A scenario node S which has child nodes of the type *Parallel* executes its children concurrently. The execution of S is finished if the execution of each child node is finished.

Each node in the scenariochart has a guard, which is a Boolean expression that must be satisfied in order that the sub-tree is visited. If no expression is specified, the default value of a guard is *true*. The guard is also used to determine whether an *Alternative* scenario is executed or not. An alternative is visualized by a circle symbol (o). From a set of alternative nodes, the node whose guard evaluates to *true* is chosen to be executed. If the guard of more than one alternative scenario evaluates to true, one of them is chosen non-deterministically.[16]

Figure 5.7 (b) shows an alternative view of Fig. 5.2 and illustrates an example of a scenariochart. In the object *BorrowManager*, the use case for borrowing books is given by the four scenario nodes *BorrowBooks*, *SelectBooks*, *Authenticate* and *RegisterBorrowing*. The latter three are sub-scenarios of *BorrowBooks* and denote a sequence in the use case. Each of these sub-scenarios contains transformation expressions which describe the stimuli and the responses sent to and received from the system during the execution of the use case.

5.2.5 Context View

The context view contains the objects in the environment of a system. An environment object is denoted by a hexagon in ADORA and represents either a person, an actuator, a sensor, or another system. It can have a cardinality which indicates the minimum and maximum number of instances that interact with the system. Environment objects with a cardinality different from (1,1) are visualized by a stack of hexagons.

Environment objects interact with the modeled system by injecting one or more stimuli per scenario and awaiting the corresponding system reactions. The protocol of this interaction is specified by scenariocharts (cf. Section 5.2.4) which are connected by an association with the environment object. During the interactive simulation of an ADORA model, the roles of the environment objects are played by the user driving the simulation [Sche04, Seyb06a]. Figure 5.7 (b) illustrates the environment objects *LibraryUser*, *BookAdministrator* and *UserAdministrator* of

[16] In an interactive simulation, this choice is made by the simulation-driving user [Seyb06a].

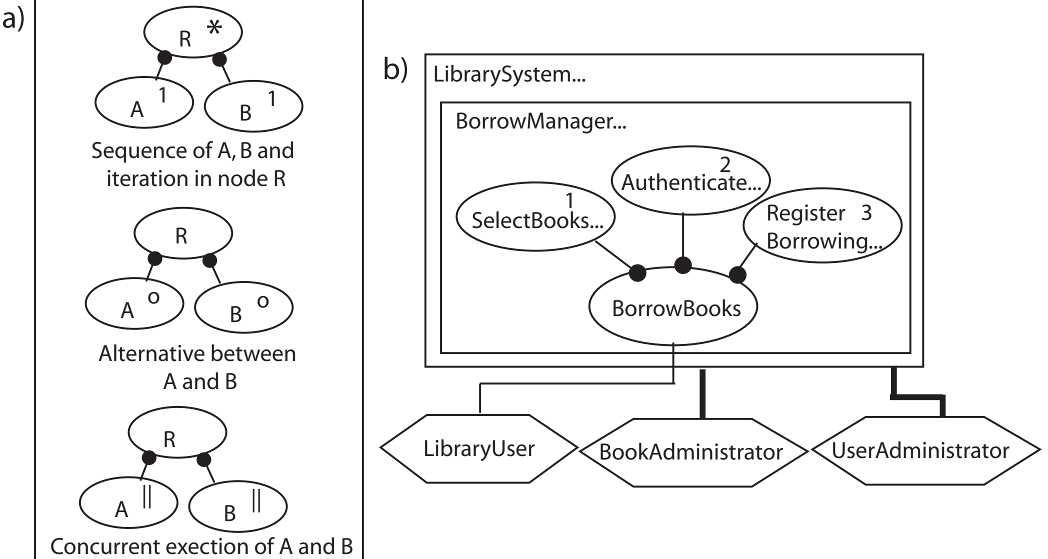


Figure 5.7: An example of the scenario syntax and all syntax elements of scenarios. Figure (a) shows all syntax elements for describing sequences, iterations and concurrent executions of scenarios. Figure (b) shows a different view of the model from Fig 5.2.

the library system.

Furthermore, a so-called *external component* is similar to an environment object. External components can be used to denote sub-systems which are black-boxes, e.g. third party components. They are components which have the reserved type *external* and which are directly embedded in the system instead of being placed in its context. They do not specify any internals, such as behavior. Hence, external components are neither full- edged environment objects nor full- edged components but some hybrid form of both.

5.2.6 Functional View

Each component in an ADORA model can contain a so-called *functional speci cation* which defines the properties of an object. The grammar for the functional specification is defined in the ADORA grammar which is discussed in detail in the next chapter. A functional specification can consist of the following properties:

- **Export Declaration:** The export declaration in a functional specification of a component *A* defines the elements which are visible to components other than *A*.
- **Import Declaration:** The import declaration of the functional specification of a component *A* defines the elements of other components which are accessed by *A*.
- **Invariants:** The invariant section is part of the component's contract and consists of one

or more expressions describing a logic predicate which must hold before and after the execution of an operation.

- **Standardized Properties:** Standardized properties are user-definable structures for stating goals, constraints, configuration information, notes, etc. [Glin02b].
- **Data Type Declarations:** A data type declaration defines a new data type which can be used to encapsulate or structure data.
- **Attribute Declarations:** An attribute is a storage which can be used to save and retrieve a data value belonging to a particular object.
- **Operation De nition:** Operations can define sequences of actions[17]. Furthermore, they may define contract elements, such as pre- and postconditions.

These elements are explained in the following, however, the syntax for the functional specification is given and explained in Section 7.8.

Import/Export declarations

A component can import and export elements defined in its functional specification. Exported and imported elements may be either *attributes* or *named data types*. Exported elements can be accessed by components which declare them in their import statement.

Exported elements are listed by their unqualified name in the *provides* section. In contrast, imported elements are listed in the *requires* section with a qualified name[18], which may qualify the variable either relatively or absolutely.

An example for the requires and provides section can be found in Listing 5.1, where the attribute *userName* is exported and the attribute *systemState* and the corresponding data type *SystemStateEnumeration* are imported from the object *LibrarySystem.BorrowManager*.

Listing 5.1: An example of provides/requires section.

```
1 provides userName;
2 requires LibrarySystem.BorrowManager.systemState,
3          LibrarySystem.Borrowmanager.SystemStateEnumeration;
```

Invariants

Creating axiomatic definitions [Hoar69, Dijk76] from software requirements and using them for various purposes has been well known for a long time in computer science. Hoare [Hoar69] uses pre- and postconditions for verifying program code. Perry [Perr87] extends this concept with *obligations, which describe conditions that must eventually be satis ed*. Meyer [Meye92] adds invariants to the concept for defining software contracts.

[17] An action can be the sending of events or the assigning of values to variables.

[18] A qualified name contains an access path to the element (cf. Section 6.1.5).

In ADORA, axiomatic specifications are used to declare the contract of components [Joos99]. The elements of a software contract are specified optionally and become manifest in pre-, post-conditions, and in invariants. Pre-, postconditions may be part of a component's operation and will be discussed later.[19]

The contract specification of ADORA components may also contain *invariants* [Seyb06a]. Invariants allow the defining of predicates on the state of a component. The state is defined by the values of the component's properties. During the runtime, the changing of a component's state space may depend on the current state of the component. Hence, certain properties of a component in a particular state may not be changed, as otherwise the component may reach an inconsistent state [Glin07b]. A way of avoiding this problem is to devise predicates, so-called invariants, which must be satisfied to re ect that a module is in a consistent state. They must hold before and after the execution of an operation which modifies the module's state.

There are two outcomes of using invariants. First, they can be dynamically checked together with the other elements of a component's contract, e.g., during the simulation run of a model, which helps to reveal mistakes during the modeling process and incomplete specifications. Second, invariants may cause a simplification of the postconditions, which in turn leads to a reduction of redundancy in the contract specification. This is due to the fact that commonly shared parts in the post-condition of different operations may be assimilated by the invariant [Glin07b].

Listing 5.2 illustrates an invariant for a component of the library system. It prescribes that the current value of the attribute *userName* has to be empty if the aspect is in its initial state *Wait*. In contrast, the value of the attribute *userName* may not be empty if the module to which the invariant belongs is in any other state.

Listing 5.2: A example of an invariant which may be part of the library system.

```
1 inv not isActiveState("Wait") −> userName != "";
2         isActiveState("Wait") −> userName == "";
```

Standardized Properties

Standardized properties are user-defined structures and annotate the objects with additional structured information, such as goals, constraints, configuration information, notes, etc. Thus, they are a way to adapt and extend the ADORA language in a controlled way for different projects, applications domain, or persons, i.e., they are descriptive stereotypes according to the definition in [Bern99a]. The form of a standardized property is defined in the corresponding directory (cf. Section 5.2.7). Each property adheres to either a primitive or a user-defined data type. Examples of a property definition are given in lines 19 and 20 in Listing 5.3.

Listing 5.3: Examples of data types, standardized properties and attributes.

```
1 data type
2   UserId : string;
3
```

[19] The concept of obligations is not employed by the ADORA language.

```
4   AuthenticationResult :
5   (
6     AUTHENICATING, AUTHENTICATION_FAILED,
7     ACCOUNT_LOCKED, AUTHENTICATED
8   );
9
10  AuthenticationCounter : constrained boundary : integer (boundary >= 0 and boundary <= 3);
11
12  LogEntry : structure of (
13    userName : UserId;
14    numberOfAttempts : AuthenticationCounter
15  );
16
17  LoginLog : list of LogEntry;
18
19  property author "D. Cunningham";
20  property goal 1 "Identify library users.";
21
22  attributes userame : string;
```

Data Type Declarations

Apart from the primitive data types *integer, real, boolean, id, string* and *time*, ADORA offers the modeler the possibility of composing more complex or restricted data types for reasons of convenience [Joos99, Seyb06a]. Applying these user-defined types helps to simplify the behavioral descriptions of components, as the number of messages exchanged between objects can be reduced significantly.

- **Primitive Type:** A primitive data type, i.e., integer, string, etc. can be redefined by aliasing it. An example for the aliasing of a primitive data type is given by line 2 in Listing 5.3, where the alias *UserId* is defined for the type string.

- **Enumeration Type:** An enumeration type allows the specification of a set of enumeration items, similar to the Pascal language [Wirt70]. An enumeration type can be used to denote and to enumerate things. An example of an enumeration can be found in lines 4–8 of Listing 5.3, where the enumeration type *AuthenticationResults* is defined.

- **Designed Type:** A designed type is one of the following types:

  – **Constraint Type:** A constraint type specifies a new primitive data type which is restricted in its domain. Line 10 in Listing 5.3 gives an example of a constrained type named *AuthenticationCounter* which restricts an integer within a given range.

  – **Structure Type:** A structured type consists of an arbitrary number of values which may be of various types. It is the same as the record type known from Pascal. A

structured type consists of fields which are either of a primitive, an enumeration, a list, or another structured type. In lines 12–15 of Listing 5.3, an example of the definition of the structured data type *LogEntry* is given. The type can be used to store a record entry of an authentication log.

– **List Type:** A list type defines an array of values whose elements either belong to a particular primitive, enumeration, list, or a structured type. Line 17 in Listing 5.3 illustrates the declaration of a list type consisting of structure type values.

Attribute Definitions

In object-oriented systems, an attribute is part of each instance of a class and represents it with a stored, individual value. An attribute has a specific type and can be accessed by a name. It does not have its own identity as an object has, i.e., an attribute and its value are under the full control of the object. In line 22 of Listing 5.3, an example of the definition of the attribute *userName* is given. It is of the type *string*.

Operation Definitions

Operations are a means for processing events in a message based behavior model. There are three purposes of operations in ADORA:

1. **Definition of synchronous events:** There are two types of operations in ADORA with respect to the duration of their execution. Operations execute either asynchronously or synchronously. An asynchronous operation needs an undetermined amount of time to execute, whereas a synchronous operation does not need time to perform and returns immediately [Joos99, Meie09a]. The default handling of an event is asynchronous. If an event has to be processed synchronously, it has to be defined explicitly by a corresponding operation [Seyb06a].

2. **Pre-/Postcondition:** As stated in the section about invariants above, it is useful to specify modules based on an axiomatic definition. An operation can have a contract specification consisting of a pre- and postcondition.[20]

3. **Simplified behavior description:** Operations can be used to define a block of actions which is executed as a reaction to an event occurring in the system. Operations are useful to simplify statecharts.[21] Nonetheless, this means of simplifying behavior descriptions has to be used carefully, as it may also lead to a loss of graphically visualized information in the model.

In the following, the structure of operations is discussed. Listing 5.4 exemplifies an operation defined in the functional specification of a component. This operation might be used by the

[20]Obligations are currently not supported by the ADORA language and therefore not discussed in the following.

[21]Instead of modeling sequence transitions which each execute one single action, the executed actions can be united in an operation.

authentication mechanism of the library system introduced before. It adds a log entry after each attempt to authenticate.

Operation signature. As mentioned above, there are two different types of operations. Synchronous operations may have one or more output values apart from input values. In contrast, asynchronous operations may only have input values. Depending on the timing operation, the signature of an operation looks different. Listing 5.4 illustrates a synchronous operation.

Local variables. An operation may define local variables, used as temporary storage for calculations. In the example of Listing 5.4, a local variable *len* is employed to save the length of a field containing log messages.

Pre- and postconditions. Pre- and postconditions are two elements of the component's contract.[22] The precondition describes a predicate which has to be satisfied by the sender of a message. The postcondition defines the results of the executed operation. It is asserted in the case the precondition has been satisfied.

Both the pre- and postconditions are predicates formed by one or more expressions which evaluate to a Boolean result. They can built upon the attributes of the module, the local variables and the arguments of the operation.If there is more than one expression, they are separated by a semicolon which has the meaning of a logical *AND* operation.

Line 5 in Listing 5.4 exemplifies a precondition. It demands that the user name of a log entry to be added may not be empty. The lines 6–9 describe the postcondition of the method. Line 7 of the postcondition assures that the list index of the log is increased. The second line ensures that the last element is the newly appended log entry. Line 9 asserts the output parameter of the operation returns the index of the log's last element.

Statements of an operation. The statements section describes a block of actions that are executed by the operation. A statement may be either the assignment of an evaluated expression to a variable, the call of a meta-function, or the sending of an event.

For example, lines 10–13 in Listing 5.4 describe the sequence of statements which stores a log entry. In line 11, the current length of the field containing the log entries is read and saved in the local variable *len*. Line 12 adds the new log entry at the end of the log. Finally in line 13, the index of the current element is assigned to the output parameter.

Listing 5.4: An example for the definition of a synchronous operation.

```
1 syncoperation addLogEntry(in entry : LogeEntry, logTime : time; out index : integer)
2   var
3     len : integer;
4   pre
5     entry.userName != "";
6   post
```

[22] The invariant is another part of the contract which was already discussed above.

```
 7      log.length@pre + 1 == log.length;
 8      log[log.length@pre] == entry;
 9      index == len;
10    statements
11      len = log.length;
12      log[len] = entry;
13      index = len
14  end operation addLogEntry
```

5.2.7 Additional Structures

There are three additional language parts in an ADORA model which are not described by the different views discussed in the sections above. These are directories which contain the definitions of *types*, *stereotypes* and *standardized properties*. These structures are not fully elaborated in the current definition of ADORA and the concepts behind them have not been investigated in detail so far. Therefore they are a focus of interest for future research on ADORA. Nevertheless, they are discussed brie y in the following:

- **Type Directory:** The so-called *type directory* [Joos99] consists of type definitions. The use of *types* helps to avoid similar structures and redundancy in components. For example, all *Logger* objects in Fig. 5.1 might have the same type, as all of them define the same functionality. A type comprises the attributes and operations of all objects and object sets of this type. Furthermore, it can define the behavior and its provided services. However, a type neither defines the associations with other objects and object sets, nor does it specify the embedding of the instances of this type in the decomposition hierarchy.

  Types may be organized in a sub-typing hierarchy, i.e., a super-type describes the common properties and the part-of relationships to components of one or more sub-types. All types have a common root super-type.

  In the base view, the type of the component is denoted by appending the type's name to the name of the component separated by a colon. However, an object need not to have an explicitly defined type. In this case, it implicitly inherits the properties and the inner structure of the root type.[23]

- **Standardized Properties Directory:** The *standardized properties directory* consists of a set of templates. Such a template specifies what a *standardized property* (cf. Section 5.2.6) looks like. A template consists of the property's name and the type of the value that can be assigned to the name. The templates specified in this directory are accessible to all objects of the model.

[23]The root type defines an empty component.

- **Stereotype Directory:** The *stereotype directory* defines restrictive stereotypes [Bern99a].[24] They can be used to formally constrain the used language elements. The constraint expression is given by a logic predicate.

[24] In [Joos99], the stereotype directory comprises both restrictive and descriptive stereotypes. In the ADORA language version of [Glin02b] the descriptive stereotypes were removed from the stereotype construct and introduced as separate standardized property language construct.

Chapter 6

Analyzing and De ning the ADORA Language

There are various ways of describing graphical modeling languages. In [Xia04] several different methods are discussed, such as the usage of string (textual) grammars, graph grammars or graphical meta-models based on class modeling. According to [Xia04], graph grammars and class meta-models tend to be very large and are therefore difficult to understand and tedious to read for human beings. In contrast string grammars [Rech97, p. 81] are easier to read and understand. Therefore, a context-free string grammar is chosen as a basis for the definition of the ADORA language. The ADORA grammar is discussed in Section 6.1 of this chapter.

In handling textual ADORA models, they need to be processed in various ways. Amongst other things, ADORA models need to be analyzed, represented, transformed, and checked to see if they are well-formed. The formal processing of ADORA models requires a formal representation. This work uses concrete syntax trees to represent textual models. Furthermore, a set of functions is needed which operate on these syntax trees. The concrete syntax trees and a set of basic functions applied to on these syntax trees are discussed in detail in Section 6.2.

A context-free grammar is powerful enough to describe the (context-free) syntax of the ADORA language, but it is not able to express the fact that a given element can only occur in the context of particular elements of the language. To express this fact, a context-sensitive [Rech97, p. 81] grammar is actually needed. To extend the context-free grammar to a context-sensitive one, additional predicates called *language constraints* are introduced, which are covered in Section 6.3.

Even though the definition of the ADORA language is done textually, the actual models are represented graphically. This is necessary because large textual representations of ADORA models are difficult to read. For this reason, the main elements of the language, such as abstract objects and object sets, are mapped to graphical counterparts. This mapping is done automatically by a modeling tool. In Section 6.4, the mapping of textual ADORA models to graphical elements is brie y discussed.

Furthermore, the so-called dynamic semantics of a modeling language describes the meaning of the model elements at runtime. A description of the dynamic semantics of ADORA can be found in [Seyb06a, Xia04, Bern02, Glin02b, Joos99]. However, the runtime semantics is

rather irrelevant for this work, as all newly introduced aspect-oriented elements do not have a dynamic semantics. They will rather be mapped by a model transformation (a so-called weaving semantics) to conventional elements which have a dynamic semantics. Therefore, the dynamic semantics is not discussed explicitly in this work.

6.1 ADORA Grammar

In this section, the structure of the context-free ADORA grammar is discussed. This grammar is based on the previous work in [Joos99, Bern02, Glin02b, Xia04, Seyb06a].

The definition of the ADORA modeling language consists of two parts: (i) a language for describing the actual model information and (ii) a description of the corresponding graphical representation[1]. Issue (i) is the subject of this chapter. The language definition will be outlined by discussing particular parts of the ADORA grammar. The full grammar can be found in Appendix B. Issue (ii) is beyond the scope of the present work and therefore will not be discussed.

The remainder of this section elaborates on several facets of the ADORA language. In Section 6.1.1, an introduction to the ADORA grammar is given. The subsequent Section 6.1.2 shows the general grammar structure and its application to textual models. In Section 6.1.4, the textual representation of different relationships is discussed in detail. The following Section 6.1.5 deals with the mechanisms for identifying and referencing model elements and Section 6.1.6 delineates how connections are represented by textual ADORA models. Finally, Section 6.1.7 covers the non-graphical elements of the language.

6.1.1 Grammar De nition

A textual ADORA model adheres to the context-free grammar $\mathcal{G}$ which is specified as four-tuple in Definition[2] 6.1. N is the set of non-terminal elements, T the set of terminal elements, P the set of grammar production rules, and $S \in N$ a non-terminal element indicating the start rule of the grammar. The elements contained in N and T are used to compose the grammar rules in P. Non-terminals N are indicated by a leading upper case letter. The terminal elements T are enclosed by double quotes and written in bold typewriter letters, or they are indicated by angle brackets when used in a grammar rule.

$$
\begin{aligned}
\mathcal{G} &= (N, T, S, P) \\
N &= \{\mathit{SpecificationDefinition}, \mathit{Model}, \ldots, \mathit{Cardinality}, \mathit{InformalDescription}\} \\
T &= \{\text{“}\texttt{\textbf{partial}}\text{”}, \text{“}\texttt{\textbf{specification}}\text{”}, \ldots, <\mathit{INTEGER_LITERAL}>, \ldots, \text{“}\texttt{\textbf{(}}\text{”}, \text{“}\texttt{\textbf{)}}\text{”}\} \\
S &= \mathit{SpecificationDefinition} \\
P &\subseteq N \times [N \cup T]^*
\end{aligned} \tag{6.1}
$$

[1]The elements of the graphical representation include several attributes, e.g., the position and visibility of nodes, etc.

[2]It is based on the general definition for grammars, cf. [Rech97] and [Cap93]. The asterisk in $[N \cup T]^*$ denotes the Kleene Star operation. V^* is the set of arbitrarily concatenated sequences of the elements of V. Furthermore, V^* contains the empty element ε.

Table 6.1: Excerpt of the ADORA EBNF grammar showing the most important production rules.

Production Name	Production Rule
SpecificationDefinition ::=	("**partial**")? "**specification**" (SpecialIdentifier)? Model (Representation)? "**end**" "**specification**" (SpecialIdentifier)? <EOF>
Model ::=	"**model**" ((ComponentDefinition | EnvironmentObjectDefinition | AspectDefinition))* TypeDefinitions PropertyDefinitions StereotypeDefinitions "**end**" "**model**"
ComponentDefinition ::=	("**partial**")? ("**external**")? ("**start**")? "**component**" ComponentName UniqueModelElementIdentifier (Cardinality)? ("**is**" InheritedType)? ComponentParts FunctionalSpecification (ComponentConnections)? "**end**" "**component**" ComponentName
ComponentName ::=	SpecialIdentifier
EnvironmentObjectDefinition ::=	("**partial**")? "**environment**" "**object**" EnvironmentObjectName UniqueModelElementIdentifier (Cardinality)? (EnvironmentObjectConnections)? "**end**" "**environment**" "**object**" EnvironmentObjectName
ComponentParts ::=	("**consists**" "**of**" (ComponentDefinition | StateDefinition | ScenarioDefinition | AspectDefinition)+ "**end**" "**consists**" "**of**")?
StateDefinition ::=	("**partial**")? ("**start**")? "**state**" StateName UniqueModelElementIdentifier StateParts (StateConnections)? "**end**" "**state**" StateName
StateName ::=	SpecialIdentifier
StateParts ::=	("**consists**" "**of**" (StateDefinition)+ "**end**" "**consists**" "**of**")?
ScenarioDefinition ::=	("**partial**")? ScenarioType "**scenario**" ScenarioName UniqueModelElementIdentifier ("**on**" GuardPart)? ("**iteration**" Expression)? (ScenarioConnections)? (TransformationElements)? "**end**" "**scenario**" ScenarioName
TransitionDefinition ::=	("**partial**")? "**transition**" UniqueModelElementIdentifier "**to**" ElementReference (SimpleTransition | DecisionTableTransition)?
AssociationRoleDefinition ::=	Role "**of**" "**association**" ElementReference
Role ::=	"**role**" RoleName (Cardinality)?
AssociationDefinition ::=	("**partial**")? "**association**" UniqueModelElementIdentifier "**to**" ElementReference Role
RoleName ::=	SpecialIdentifier
ScenarioConnectionDefinition ::=	"**scenarioconnection**" UniqueModelElementIdentifier "**to**" ElementReference
FunctionalSpecification ::=	("**functional**" "**specification**" (Provides)? (Requires)? (Invariants | DataTypeDeclarations | AttributeDefinitions | OperationDefinition)* "**end**" "**functional**" "**specification**")?
InformalDescription ::=	<INFORMAL_DESCRIPTION>

The grammar of ADORA as well as an explanation of the EBNF symbols are given in Appendix B and an excerpt of the most important rules can be found in Table 6.1.[3] The syntax rules are defined in an Extended Backus Naur Form (EBNF) [ISO 96].

6.1.2 Applying the Grammar to Textual ADORA Models

The so-called main production rules describe those modeling elements in the language which have a graphical representation and which are uniquely identifiable (cf. Section 6.1.5) in the textual model. Furthermore, they have a name with the suffix *De nition*. The name of the rule indicates the ADORA element which is described, e.g., *ScenarioDe nition* defines what a scenario node looks like in textual representation. Table 6.1 exemplifies some of the rules.[4]

Apart from the main production rules, there are several auxiliary rules which help to compose the textual description of a main element. For instance, the production rule *ComponentDe nition*[5] refers to the auxiliary production rule *ComponentName* which defines what a component name looks like.

Listing 6.1: Excerpt of a textual model example based on the model in Fig. 5.2.

```
 1 speci cation LibrarySystem
 2   model
 3     component LibrarySystem '1.1' (1,1) is type Root
 4       consists of
 5         ...
22         partial component BorrowManager '1.1.3' (1,1)
23           consists of
24             root scenario BorrowBooks '1.1.3.1'
25               connections
26                 scenarioconnection '1.1.3.1.s.1' to '1.1.3.2'
27                 ...
28               end connections
29             end scenario BorrowBooks
30             ...
46         end component BorrowManager
47         ...
56         partial component BookAdministration '1.1.5' (1,1)
57           connections
58             role GetFeedback (1,1) of association '1.3.a.1'
59             association '1.1.5.a.1' to '1.1.2' role ManageCatalog (1,1)
```

[3] This appendix also contains the ADORA grammar comprising the aspect-oriented extensions of this work. A discussion of these extensions can be found in Chapter 7.

[4] The main grammar rules are *Speci cationDe nition*, *ComponentDe nition*, *EnvironmentObjectDe nition*, *StateDe nition*, *ScenarioDe nition*, *TransitionDe nition*, *AssociationRoleDe nition*, and *ScenarioConnectionDe nition*.

[5] Abstract objects and object sets are expressed by the same grammar rule *ComponentDe nition*.

```
60              association '1.1.5.a.2' to '1.1.6' role AuthenticateBookAdm (1,1)
61            end connections
62          end component BookAdministration
63
64          partial component Authorization '1.1.6' (1,1)
65            consists of
66              start state Wait '1.1.6.1'
67                connections
68                  transition '1.1.6.1.t.1' to '1.1.6.2'
69                    [not authenticated]
70                    receive queued msg=authenticate(user : string, pw : string) over
71                      AuthenticateBorrowing, AuthenticateBookAdm,
72                      AuthenticateUserAdm | send getUserInfo(user, pw) over SendUserInfo
73                      ...
96            end consists of
97            functional speci cation
98              ...
100             operation authenticate(user : string, pw : string)
101                 statements
102                   send authorized() to msg.msgAnswerTarget
103             end operation authenticate
104           end functional speci cation
105           ...
111         end component Authorization
112       end consists of
113     end component LibrarySystem
114     ...
121     environment object BookAdministrator '1.3'
122       connections
123         association '1.3.a.1' to '1.1.1.5' role ManageBooks (1,1)
124       end connections
125     end environment object BookAdministrator
126     ...
133 end speci cation LibrarySystem
```

The further discussion of the grammar is exemplified with the textual model that is based on the *LibrarySystem* contained in Fig. 5.2. Listing 6.1 shows an extract of this model which is presented fully in Appendix D.

6.1.3 Representing Informal Elements

Besides the formal syntax, a textual ADORA model may contain informal descriptions of the language elements, which allows models to be specified with a variable degree of formality [Glin02b, Seyb06a]. An informal description looks like the example shown in Listing 6.2 showing a part of the model in Appendix D. The grammar rule *InformalDescription* in Table 6.1 specifies the

form of a informal description by the regular expression *<INFORMAL_DESCRIPTION>*[6]. An informal description is introduced with /# and ended with #/.

Any language element may be annotated with an informal comment. Informal comments may be placed anywhere in a textual model but this fact is not expressed explicitly, as it would cause the grammar to become bloated. However, only the informal elements placed in front of a textual main element are evaluated by the evolution and simulation mechanism presented in [Seyb06a] and therefore relevant for the informal specification.

Listing 6.2: Example of a informal description: The object set is described by an informal comment.

```
1   /# The set of books describes all books in the library.
2       A book object comprises the attributes ID, author,
3       title and ISBN #/
4   partial component Book '1.1.2' (0,n)
5   connections
6     role ReadCatalog (1,1) of association '1.1.5.a.1'
7     association '1.1.2.a.1' to '1.1.3' role ReadCatalog (1,1)
8     association '1.1.2.a.2' to '1.1.1' role borrowedBy (0,1)
9   end connections
10 end component Books
```

6.1.4 Expressing Model Relationships by Nesting Textual Models

Expressing relationships between elements in a graphical model is an important issue. One schematic means of relating one element to another is to *nest* them, which can be adopted for textual models: by nesting text parts, they can be related to each other. In textual ADORA, there are three types of relationship which are expressed by nesting:

1. **Syntactic Relationship:** The nesting of textual elements is used to relate them syntactically. This helps split up the complex grammar into grammar parts which have a manageable size and, therefore, it helps to cope with the complexity of the textual grammar.

2. **Semantic Relationship:** The nesting of textual elements is used to describe the following semantic relationships:

   (a) **Hierarchical Decomposition:** The hierarchical decomposition (cf. Section 5.1.2) expressing a part-of-relationship is represented by nesting the corresponding textual elements. The part-of relationship is expressed by the grammar rules *Component-Parts*, *StateParts*, *Speci cationDe nition* and *Model* (cf. Table 6.1).

   (b) **Denoting the Source of a Connection:** The relationship between a connection, such as an association or a transition, and its source element is expressed by embedding the textual connection in the textual description of its source element.

[6] The actual regular expression is abstracted in the given ADORA grammar rules. Instead a descriptive placeholder is given.

In the following, the two semantic relationships are discussed in more detail.

Hierarchical Decomposition

The part-of relationship is expressed by the grammar rules *Speci cationDe nition*, *Model*, *ComponentParts*, or *StateParts*. For example in Listing 6.1, the description of the model in lines 2–132, is part of the ADORA specification (*Speci cationDe nition* between lines 1 and 133). For a clear distinction between the parent and its children, most of the nestable elements are separated by a header and a footer. Furthermore, some elements contain the *name* of the represented element in the footer and the header, e.g. the lines 22 and 46 contain the name *BorrowManager* of the component.

Another example for the part-of relationship in Listing 6.1 is given by the component *Authorization* in lines 64–111 which is a part of the component *LibrarySystem*. Furthermore, there are several states and one component which are a part of the component *Authorization*, e.g., the state *Wait* in lines 65ff. The part-of relationship is specified by the production rule *ComponentParts*. An example for a footer and header delimiting an element is given for the state *Wait* in the lines 65ff.

Denoting the Source of a Connection

Associations, transitions and scenario connections are called connections in ADORA. To express the relationship between a connection and its source element, the textual description of the connection is embedded in textual description of its source, as illustrated in lines 59 and 60. In this example, two associations which originate in the component *BookAdministration* are specified. Connections are embedded in the source element by enclosing them with the keywords *connections* and *end connections* (cf. lines 57 and 61). Besides the source-to-connection relationship, there are several other issues concerning the textual specification of a connection, which are discussed in Section 6.1.6.

6.1.5 Identifying ADORA Model Elements

When using nesting as the only means for expressing relationships, redundancy may occur in textual ADORA models if several (semantical) elements, such as abstract objects, states, associations, transitions, etc., have a unidirectional (semantical) relationship to the same semantical element. For example, assume the elements *B* and *C* which have a relationship with the element *A*. If using the nesting of elements is the only means for relating them, *A* has to be duplicated in *B* and *C* to express the relationship between both, which in turn leads to redundancy.

For example, connections have a relationship to their target and their source elements. The target and source elements have in turn a part-of relationship to an element in the decomposition hierarchy. Hence, either the connection, the source or the target element would have to be redundant, if just a textual nesting were used.

To reduce redundancy to a minimum, ADORA uses identifiers and references. An identifier tags a particular semantical language element, whereas a reference points to the element with the

corresponding identifier. Apart from reducing the redundancy in textual ADORA models, identifiers and references also help to identify, find, and extract a semantical element of an ADORA model.

However, the introduction of identifiers and references leads to the need for a context-sensitive language and, therefore, it demands the introduction of additional language constraints, which are discussed in Section 6.3.

There are two different types of identifiers: so-called *generated identi ers* and *names* which are discussed in the following.

Generated identi ers. A generated identifier is unique in the whole model and tags a main element, e.g. an instance of a *ComponentDe nition* (cf. Section 6.1.2). When being referenced, a relationship between two model elements is established. Such a relationship is usually handled by the ADORA tool. Thus, generated identifiers are not meant to be human-readable. They are not shown explicitly by the tool, neither in the graphical, nor in the non-graphical model parts. However, in the textual models of the present work, all generated identifiers are built upon an alphanumeric classification (e.g. '1.1.6' in the example below) in order to be legible for the reader of the present work.

The relationships constituted by a generated identifier and a corresponding reference must always be consistent, as otherwise a model is not meaningful. For instance, the reference which denotes the target of a transition (cf. Section 6.1.6) must always point to an existing and unique target element which has the expected type. To enforce consistency, such a relationship may be checked with a *strictly enforced* language constraint (cf. Section 6.3).

Line 60 of Listing 6.1 shows the textual specification of an association tagged with the generated identifier '1.1.5.a.2'. It has a reference '1.1.6' to an element with the corresponding generated identifier. The referred element is the *Authorization* component which has the identifier '1.1.6' (cf. line 64).

Names. The second type of identifiers are *names*. In contrast to generated identifiers, names are human-readable strings which are assigned manually by the modeler to a given model element. As generated identifiers, names can be referenced. Names and the corresponding references are used for describing *modeler-maintained* relationships in the model. Names occur in model parts which are presented to the modeler as text. For example, a name is the variable defining the functional specification or the name of a component. An example of a name is *BorrowManager* in line 22 of the text model in Listing 6.1.

In previous work about ADORA, e.g., [Xia04, Seyb06a], names were used for establishing relationships in the sense of the tool-maintained relationship described in the section about generated identifiers. However, names are not necessarily unique, and they are subject to evolution [Seyb06a, p. 83ff], [Xia04] of an ADORA model. Hence, names can be imprecise, partial, inconsistent, or even undefined, and therefore they are inappropriate for describing the tool-maintained relationships described above.

Grammar rules for identifiers and names. Generated identifiers and names are described by the four basic grammar rules *Identifier*, *SpecialIdentifier*, *QualifiedIdentifier* and *ExtendedQualifiedIdentifier*. An extended discussion of them can be found in Section B.4 of the appendix.

6.1.6 The Representation of Connections

Connections (transitions, associations and scenario connections) are constituted by a source and a target element. In graphical models, connections are represented by a line drawn between the source and the target. Mapping a connection, its source, and its target element to a text model requires the introduction of identifiers and references to minimize redundancy, as discussed in Section 6.1.5.

The source of a connection is specified by embedding the connection into the source element. For example, the lines 59 and 60 in Listing 6.1 show the associations (*AssociationDefinition*) whose source element is the component *BookAdministration*. In contrast to the source element, the target element of a connection is referred to by an generated identifier. For example, the target of the association in line 59 is determined by the identifier '1.1.2'.

Any of the two constituents of an association may be its source, i.e. the source is chosen arbitrarily. However, associations may be attributed with two roles (cf. Section 5.2.2). One role (*Role*) is associated with the direction from the source to the target, the other one with the opposite direction. The source-to-target role is defined together with the association and therefore embedded in the source. For instance in line 59, the source-to-target role *ManageCatalog* is attached to the textual description of the association with the identifier '1.1.5.a.1'. Note that this role also specifies a cardinality (1,1).

In contrast to the source-to-target role, the target-to-source role is defined by the grammar rule *AssociationRoleDefinition* and is embedded in the *target* element.[7] A role has a reference to the association to which it belongs. The target-to-source role refers to the association with the corresponding generated identifier. For example, line 58 denotes the target-source role for the association '1.3.a.1' (cf. line 123) between the *BookAdministration* object and the *BookAdministrator* environment object.

Furthermore, scenario connections (*ScenarioConnection*, cf. Table 6.1) and transitions (*TransitionDefinition*) are other types of textual connection descriptions. Line 26 and the lines 74ff, respectively, exemplify these elements. In contrast to associations, scenario connections and transitions do not have roles. However, a transition can have a guard, a message reception part, and an operation either in a simple textual (*SimpleTransition*) or a tabular description (*DecisionTableTransition*).

6.1.7 Non-Graphical Language Elements

Apart from the graphical main elements, there are several other model parts which are only textually described, such as stereotype (*StereotypeDefinition*), type (*TypeDefinitions*), and property

[7] More precisely, it is contained in the section describing the connections of the target element. See the rules *ComponentConnections*, *StateConnections*, *ScenarioConnections*, and *EnvironmentObjectConnections* of the ADORA grammar in the Appendix B.

definitions (*PropertyDe nition*). These elements are discussed in Section 5.2.7 and more extensively in the earlier work of [Joos99] and [Glin02b].

Furthermore, there is the so-called functional specification of components (cf. rule *FunctionalSpeci cation* in Table 6.1 and [Joos99, Glin02b, Seyb06a]) which is used in this work for aspect constructs, too. It is discussed brie y in Section 5.2.6 and later with respect to aspects in Section 7.8.

6.2 Formalizing the Model Data Structure and Operations

In the present work, the definition of language constraints and model transformations have a paramount importance. Language constraints augment the context-free grammar rules of the ADORA language given in Appendix B to a context-sensitive grammar, whereas model transformations are needed for the weaving of aspect-oriented models (cf. Chapter 9). Using textual ADORA models directly to define transformation and language constraints is too complicated, ambiguous, tedious, and error-prone.

A more elegant and simpler way is to transform the textual models first into a formal data structure which simplifies the handling of the input. The data structure employed in the present work is the concrete syntax tree[8] [Rech97] of the textual ADORA model. It can be simply derived from the ADORA grammar described above.

The concrete syntax tree can be used to define functions that allow the retrieval of various properties and particular model parts of the language input. By employing these functions, language constraints, as well as model transformations, can be described in a simple way.

The syntax tree data structure as well as the corresponding functions are discussed in the following.

6.2.1 A Data Structure for the Textual Representation of ADORA Models

A concrete syntax tree of a given textual ADORA model is a representation which is derived from the grammar rules given in Appendix B. A syntax tree is ordered. The parent nodes[9] (i.e., the nodes with child nodes) in the tree are labeled with the name of the producing syntax rule. Only non-terminal symbols from the syntax can have children. The children of a parent node are particular elements of the right-hand side of a production rule. A child node may either be labeled with a terminal or a non-terminal symbol. Furthermore, a leaf node is always labeled with a terminal symbol.

[8]Instead, *abstract* syntax trees could be applied. However, concrete syntax trees are closer to the (concrete) ADORA grammar, used throughout this work.

[9]When talking about trees, the common genealogical terminology for trees is used, such as parent, direct descendant (child), ancestor, and descendant.

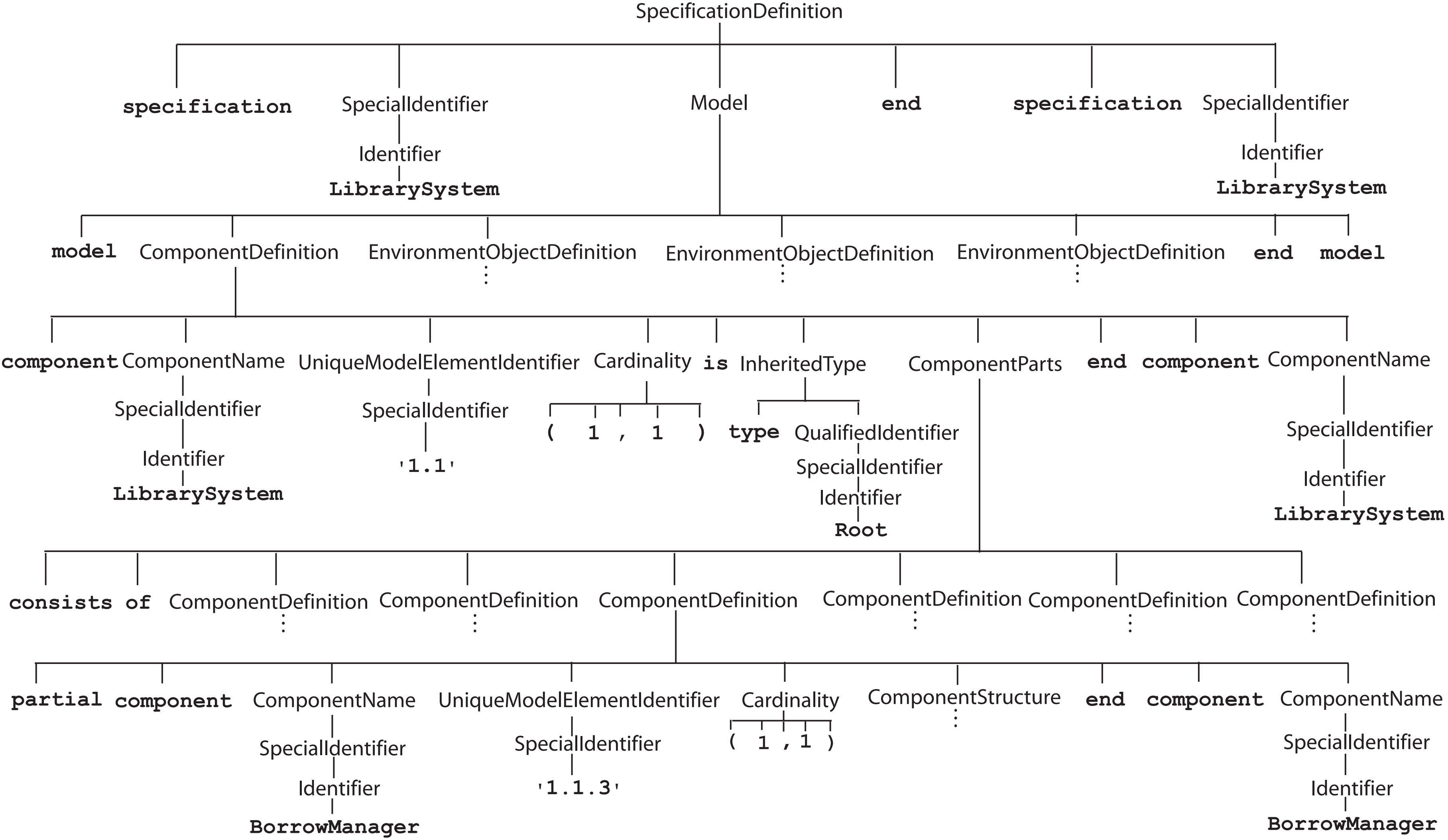


Figure 6.1: An excerpt from the concrete syntax tree of the textual model in Appendix D. It shows mainly the syntax tree of the component *LibrarySystem* shown between the lines 3 and 113.

Figure 6.1 shows an example of a concrete syntax tree representing the textual model given in Appendix D. Note that this example tree is shown partially. Terminal nodes are printed in bold typewriter font, whereas non-terminal nodes are printed normally. For example, line 1 of the textual model in Listing D.2 given in Appendix D.1 is represented by the two left-most branches of the tree's root node which results in the terminals **specification** and **LibrarySystem**, whereas line 133 results in the three right-most branches. Lines 2 to 132 are represented by the subtree with the root node *Model*.

The set of all possible syntax trees is denoted by Δ which is formally defined in Appendix E.1. Each syntax tree in Δ can be represented as nested tuple structure. Fig. 6.2 shows an excerpt from the nested tuple structure of the syntax tree given in Fig. 6.1. Note that the tuple is also partially shown.

(SpecificationDefinition, (**specification**, (SpecialIdentifier, (Identifier, (**LibrarySystem**))), (Model, (ComponentDefinition, (**component**, (ComponentName, . . .), (UniqueModelElementIdentifier, . . .), . . .))), **end**, **specification**, (SpecialIdentifier, (Identifier, (**LibrarySystem**)))))

Figure 6.2: The syntax tree of Fig. 6.1 written as a nested tuple structure.

As discussed in Section 6.1.2, ADORA models and their model elements can contain informal descriptions. An informal comment (*InformalDescription*) may occur at any place in the textual model. If an informal description is discovered in the language input by the parsing function ρ, it is placed as the first child in the syntax subtree of the element following the informal description in the language input. For example, the textual description of the component *Books* given in Listing 6.2 results in the syntax tree illustrated in Fig. 6.3.

(ComponentDefinition, ((InformalDescription, (**# The set of books ...title and ISBN #)**), **component**, (ComponentName, . . .), (UniqueModelElementIdentifier, . . .), . . . , **end**, **component**, (ComponentName, . . .)))

Figure 6.3: An example tuple structure containing an informal comment. The model with the textual description from Listing 6.2 is represented as a tuple structure.

6.2.2 Functions on Syntax Trees

To process the models in a simulation [Seyb06a], in model transformations (cf. Chapter 9), or when checking language constraints (cf. Section 6.3), a set of functions is needed which allow retrieval and manipulation of particular properties of a given ADORA syntax tree. The following two sections brie y introduce these functions. They are more extensively discussed in Appendix E.

Primitive functions

There are several primitive functions that allow retrieval of basic properties and the execution of some simple operations on syntax trees: the function *insert* allows new elements to be added in a tuple of a syntax tree at a particular position, whereas the function *delete* is used to remove an arbitrary element from a tuple. The number of elements contained in a tuple can be determined by the function *arity*. An element in a tuple can be extracted by using the *projection* function. The type of an ADORA subtree is denoted by the label of the tree's root node. It can be retrieved by the function *type*. The function *child* returns one specific child of the given tree's root node, whereas the function *children* returns all children in a tuple. A catalog of all primitive functions, including their formal definitions, can be found in Section E.3 of the Appendix.

Complex functions

Complex functions are built upon the primitive functions discussed above. They are divided into three categories. *Basic functions* allow retrieval of properties of the basic ADORA language as it is defined by the previous work of [Joos99, Glin02b, Bern02, Xia04, Seyb06a]. Their catalog, including their informal definitions can be found in Section E.4 of the appendix.

The second category of complex functions help to retrieve aspect-specific properties of models. They will be used in the main part of the present work, especially for the description of language constraints and the transformation semantics. The aspect-specific functions are described in detail in Section E.5 of the appendix.

The third category of complex functions are used to describe the model transformations for the weaving transformation of aspect-oriented models. They are contained in Section E.6 of the appendix.

6.3 Language Constraints

Most programming and modeling languages need to be formulated with a context-sensitive grammar. However, the definition of context-sensitive grammar production rules is impractical or not feasible. Therefore, these languages are usually defined with a context-free grammar that is complemented by so-called language constraints, resulting in a context-sensitive grammar. Language constraints are predicates that define how an instance of an element is connected meaningfully with another element of the language [Xia04, p. 35]. Elements adhering to both, the context-free syntax and the context-sensitive language constraints, are called *well-formed* [Xia04].

In [Xia04], there are two types of language constraints. One type is expressed by an attributed grammar and is used for describing relationships between *model elements*, whereas the other type describes the well-formedness of *model views* and is specified by a refinement calculus [Morg94], a kind of operational semantics [Plot81].

The use of *attributed grammars* to define the well-formedness of relationships between model elements, as it is done in [Xia04], is rather problematic. Using an attributed grammar is a technique which is preferably employed in implementing compilers of programming languages. An

attributed grammar propagates the evaluated results of a constraint evaluation to the parent grammar rule. Therefore, it normally needs several parsing steps [Paak95, Page 208 ff] to compute whether a constraint is satisfied or not. Each evaluation of a language constraint may therefore rely on the result of a former constraint evaluation step. Hence, using an attributed grammar for the definition of a modeling language can be counter-intuitive and less intelligible, as humans can not manage too many time-delayed dependencies resulting from a multi-pass parsing and constraint evaluation process. Thus, the resulting language constraints are not well suited for being read by humans. However, this is one of the aims of the ADORA grammar. Consequently, to obtain a clearer language definition, a multi-pass process should be avoided.

The present work focuses on the well-formedness of the relationships between model elements rather than on the consistency in the view transitions. Furthermore, constraints are neither attributed to grammar rules nor expressed by an operational semantics. The constraints apply rather to a given (sub)syntax tree of the ADORA model. For example, there are constraints that apply to the subtrees of the types *ComponentDefinition* or *AssociationDefinition*, which may, for example, check if the naming of a given component or an association role is correct.

6.3.1 Time-Dependant Enforcement of Constraints

A constraint must be satisfied at the latest at a predefined point in time. The point in time may vary from constraint to constraint and depend on the kind of model properties that are checked. This point-in-time-dependent checking is necessary in order to gain a higher flexibility during the modeling [Cram07]. Based on the classification in [Cram07], there are mainly two time-dependent types of constraints, which are summarized in Table 6.2: *strictly* and *leniently enforced constraints*. A strictly enforced constraint C_s is checked *before* an operation that may violate C_s is performed. Thus, before executing a potentially violating operation, the result of the operation is anticipatorily calculated but not definitely committed. The operation is not permitted to execute if the constraint is violated. Otherwise the result is committed.

Table 6.2: Points in time for evaluating language constraints.

Type	Checking Time
Strictly Enforced	• Immediately before operation
Leniently Enforced	• Immediately after operation • Before weaving • Before simulation • User-defined point in time

In contrast, a *leniently enforced* constraint C_l can be temporarily violated. However, there may be a point in time when C_l also needs to be satisfied. The point in time depends on the kind of model part that is checked. A constraint may be checked immediately after the execution

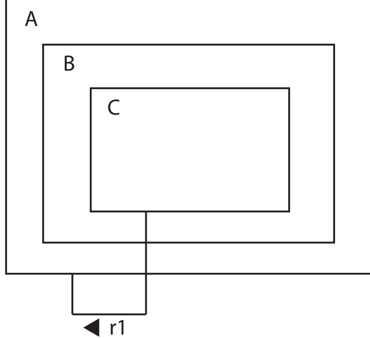


Figure 6.4: Illustration of a constraint violation: no association is allowed between A and C because of constraint C_1. Therefore, the shown association with the role *r1* is actually not allowed.

of a model-altering operation, which has a similar effect to a strictly enforced constraint. Other leniently enforced constraints need to be satisfied before the execution of a model simulation [Seyb06a], and others must be evaluated positively before performing a model transformation (cf. Chapter 9). Finally, there are constraints which are checked at a user-defined time: the user decides individually for each of them when they need to be satisfied.

The consistency of most of the tool-maintained relationships (cf. Section 6.1.5) is enforced by strict constraints, i.e., they are checked immediately before an operation alters the model. For example, connecting an association to another model element involves a tool-maintained relationship. At no time, it makes sense to connect an association to a *state*. Therefore, this constraint is strictly enforced.

The parts of a model which are subject to model evolution, e.g., modeler-maintained relationships, are mostly checked by leniently enforced constraints. For example, the references to a component's *name* (cf. Section 6.1.5) may be wrong until a formal simulation is started.

6.3.2 Example of a Constraint Description

The descriptions of constraints in this work follow a uniform pattern: they consist of a natural language description and a formalized predicate. The predicate is based on the data structure and the functions outlined in Section 6.2. Furthermore, each constraint specifies if it is strictly or leniently enforced. If the constraint checking is lenient, the latest point in time when the constraint must be satisfied is given. An example taken from [Xia04, p. 75f] will illustrate the constraint specification scheme used in this work:

> ...if an object *C* is embedded in another object *A* (it can be one level embedded or several levels embedded), an association connecting *A* and *C* is not allowed...

The situation illustrated in Fig. 6.4 violates this constraint. The textual description of the model shown in Fig. 6.4 can be found in Listing 6.3. The natural language constraint described

above is defined formally in Constraint C_1 where *s* represents the syntax (sub)tree describing the association and *t* the syntax tree of the corresponding model.

Listing 6.3: The model from Fig. 6.4 as textual description.

```
1  speci cation
2    model
3      component A '1' (1,1)
4        consists of
5          component B '1.1' (1,1)
6            consists of
7              component C '1.1.1' (1,1)
8                connections
9                  association '1.1.1.a.1' to '1' role 'GetUserId' (1,1)
10               end connections
11             end component C
12           end consists of
13         end component B
14       end consists of
15     end component A
16   end model
17 end speci cation
```

The predicate of the Constraint C_1 employs three different functions. The function $descendants$ takes the syntax tree δ of an ADORA model. It traverses the syntax tree model and adds each visited subtree to the result set of the function. The function $source$ returns the syntax tree of a connection's source element, the function $target$ correspondingly returns the syntax tree of the target element. These three functions are described in more detail in Section E.4 of the appendix.

$$
\begin{aligned}
source(t,s) \notin descendants(target(t,s)) \wedge \\
target(t,s) \notin descendants(source(t,s))
\end{aligned} \qquad (C_1)
$$

The constraint is strictly enforced, i.e., it is checked before an altering operation is executed. Thus, the model in Fig. 6.4 could not be created.

There are a lot of other constraints for the ADORA language. An extensive introduction to language constraints can be found in [Xia04]. Furthermore, an up-to-date discussion with a constraint catalog for the ADORA language version described in [Seyb06a] is given by [Cram07].

6.4 Graphical Mapping

Textual ADORA models tend to become very large and complex which makes them rather hard to maintain manually. This is due to the fact that the textual ADORA language is designed not to be edited directly by the modeler but constructed and modified in a graphical way. Nevertheless,

there are a few textual parts with which the modeler gets in touch (cf. Section 6.1.7), e.g., the functional specifications of components or the labels of transitions.

The graphical representation is produced by the ADORA tool that maps the textual ADORA models to graphical elements and vice-versa[10]. The mapping defines a graphical representation for each main element of the language which is described by a set of properties, such as the position, size, visibility, etc. Each graphical representation references the corresponding model element by its identifier. Furthermore, there may be more than one graphical representation per model element which enables multi-user editing [Moda03, Seyb06a] or multi-view models respectively.

The graphical representation is not within the scope of this work. However, the mapping between the graphical shapes and the corresponding model grammar rules is discussed brie y. Table 6.3 gives an excerpt of the full graphical mapping taken from the Appendix C which is based on [Seyb06a, Xia04] and discussed extensively in [Xia04].

For each mapped element, the textual representation of the corresponding model parts as well as the name of the applied grammar rule from Appendix B are given. The first row of Table 6.3 shows the graphical mapping for an abstract object. The corresponding textual model extract is produced by the grammar rule *ComponentDe nition*, and is given in the third column, whereas the fourth column shows a rectangular shape representing the abstract object. The name of the object is visualized, whereas the generated identifier of the object is not. In the second row of Table 6.3, the mapping of an object set is described. An object sets is visualized by a stack of rectangles, and its cardinality differs from *(1,1)* and is shown in the lower left corner.

Besides the visualized shapes of the main elements, there are several other properties defined by the grammar which are shown in the graphical representation. For example, in the third row, a partial property of an abstract object is given. It is visualized by an ellipsis following the name.

The fifth column in Table 6.3 shows how an association with a role is graphically represented. The role name *r1* and the cardinality are visualized too, whereas the generated identifier of the association is not shown.

[10]For example, when a model is stored, the graphical representation is tranformed to a textual mapping again.

Name	Grammar Rule	Textual Instance	Graphical Mapping
Abstract Object	Component-Definition	/# Informal Description #/ component *Name Identi er (1, 1)* ... end component *Name*	*Name*
Object Set	Component-Definition	/# Informal Description #/ component *Name Identi er (n,m)* ... end component *Name*	*Name* *(n,m)*
Partial Abstract Object	Component-Definition	/# Informal Description #/ partial component *Name Identi er (1,1)* ... end component *Name*	*Name...*
Association	Association-Definition	/# Informal Description #/ association *Identi er* to *target_identi er* role *Role (m,n)*	Name ➤*Role (m,n)* target_ identifier

Table 6.3: Excerpt of the graphical mapping in Appendix C.

Part II

The Aspect-Oriented ADORA Modeling Approach

This part presents an aspect-oriented modeling approach with the desired characteristics identified in Chapter 4. It is based on the the ADORA modeling language and aims at the reduction of problem-exogenous complexity in requirements and architectural models by providing support for aspect-orientation elements. The approach allows crosscutting concerns to be identified and separated and enables the derivation of a conventional model with a composition mechanism when a woven model is more useful than the aspect-oriented one.

The present part is structured as follows. In Chapter 7, a syntax extension for the current ADORA approach is presented. It is discussed according to the ADORA EBNF syntax definition and the language definitions that are introduced earlier in this work. Chapter 8 discusses an extension for the visualization and abstraction mechanisms presented in Chapter 5, in order to enable it to handle the introduced aspect-oriented modeling elements. Chapter 9 presents a weaving semantics for the defined syntax elements. Finally, Chapter 10 discusses how this modeling approach supports an early identification and separation of crosscutting concerns.

Chapter 7

Aspect-Oriented Language Extension for ADORA

In this Chapter, an aspect-oriented language extension to ADORA is introduced. The ADORA language was chosen as a base language, because it is an integrated modeling language which is better suited for the introduction of aspect-oriented modeling constructs than non-integrated languages (cf. Chapter 4). Apart from that, the language innately possesses several other desired characteristics for aspect-oriented modeling languages. In the following, a motivation and an overview of the extension is given. Subsequently, the new language elements are discussed in detail.

Note that for the sake of completeness, all newly introduced aspect-oriented syntax elements are given in the following. However, in order to understand the concepts only the reading of the first four sections is required. Nevertheless, an interested reader also may read the other sections.

7.1 Motivation

As discussed in the Chapters 3 and 4, decomposing software systems using a one-dimensional modularization criterion may lead to a scattering and tangling of crosscutting concerns. This effect can be also observed in conventional ADORA requirements models. The problem becomes manifest in the structural and semantical redundancy of the model elements. Structural redundancy means that the elements are equivalent in their structure, whereas semantical redundancy means that the elements are equivalent in their semantics. For example, an element *A* is semantically redundant with another element *B* if *A* and *B* are differently represented, e.g. have different names, but have the same meaning or fulfill the same purpose.

The example in Fig. 7.1 vividly illustrates the redundancy resulting from the modeling of crosscutting concerns by conventional means. It shows a more evolved version of the library system already used in Chapters 5 and 6. It contains some parts of the *Authorization* concern which cuts across the system. The abstract object *Authorization* provides an authentication service for the various components. Amongst others, the components *BorrowManager* and *BookAdministration* use this service to authenticate and authorize their users (*LibraryUser* and *BookAd-*

ministrator). For this purpose, the behavior description of the components contain the states *WaitForUserName*, *Authenticate*, *Authorized*, and the corresponding transitions. This part of the behavior is responsible for the communication with the *Authorization* component. It sends the user identification and the password to the *Authorization* component and waits. The *Authorization* then authenticates and authorizes the users of the library system and sends the *authorized()* message back to the initiator of the authorization request.

Each failure to authenticate for an access-restricted functionality is recorded by a corresponding mechanism that is provided by the *AuthenticationLog* component.

Apart from the actual authentication and logging functionality, each component implements another functionality which allows a user to retrieve a forgotten password. The state group consisting of the four states *WaitForHintRequest*, *WaitForHintQuestion*, *WaitForHintAnswer*, and *WaitForHintPassword* in the component *BorrowManager* describes the behavior of this password retrieval mechanism. Note, that the model shown is partially viewed — the same behavior description is also contained in *BookAdministration* but hidden. Apart from the behavior description for the password retrieval, each component contains a scenariochart which describes the communication between the environment object and the password retrieval mechanism. This is indicated by the partially viewed *RequestPassword* scenario in the components *BorrowManager* and *BookManager*.

Summarized, crosscutting concerns can have an impact on all views of a conventional ADORA model:

- **Behavioral View:** In ADORA, crosscutting behavior becomes manifest in semantically equivalent and therefore redundant parts in different statecharts. The redundant behavior is either a statechart fragment or a complete statechart.

  Example (7.1). In Fig. 7.1, a *LibraryUser* has to be authenticated before he or she can borrow books. This behavior is described by the states *WaitForUserName*, *Authenticate*, *Authorized* and the corresponding three transitions. A similar behavior description can be found in the other components, e.g., *BookAdministration*, which are impacted by the same crosscutting concern. The state group consisting of the states *WaitForHintRequest*, *WaitForHintQuestion*, *WaitForHintAnswer*, and *WaitForHintPassword* belong to a complete crosscutting statechart.

- Crosscutting behavior can delegate tasks to other components. There are two different types of delegation. Either the task is delegated to an *child component*, which is also called *embedded component* in the following, or to a *server component*. An embedded component is exclusively used by one impact location of the crosscutting behavior. Thus, different impact locations do not interfere. In contrast, a server component is shared commonly between all impact locations of the crosscutting behavior and therefore it may interfere.

  Example (7.2). In Fig. 7.1, the re exive transition outgoing from the *Authentication* delegates the task of logging a *log* event to the *AuthenticationLog* component. A task delegation to a server component is exemplified by the transition between the state *WaitForUser-*

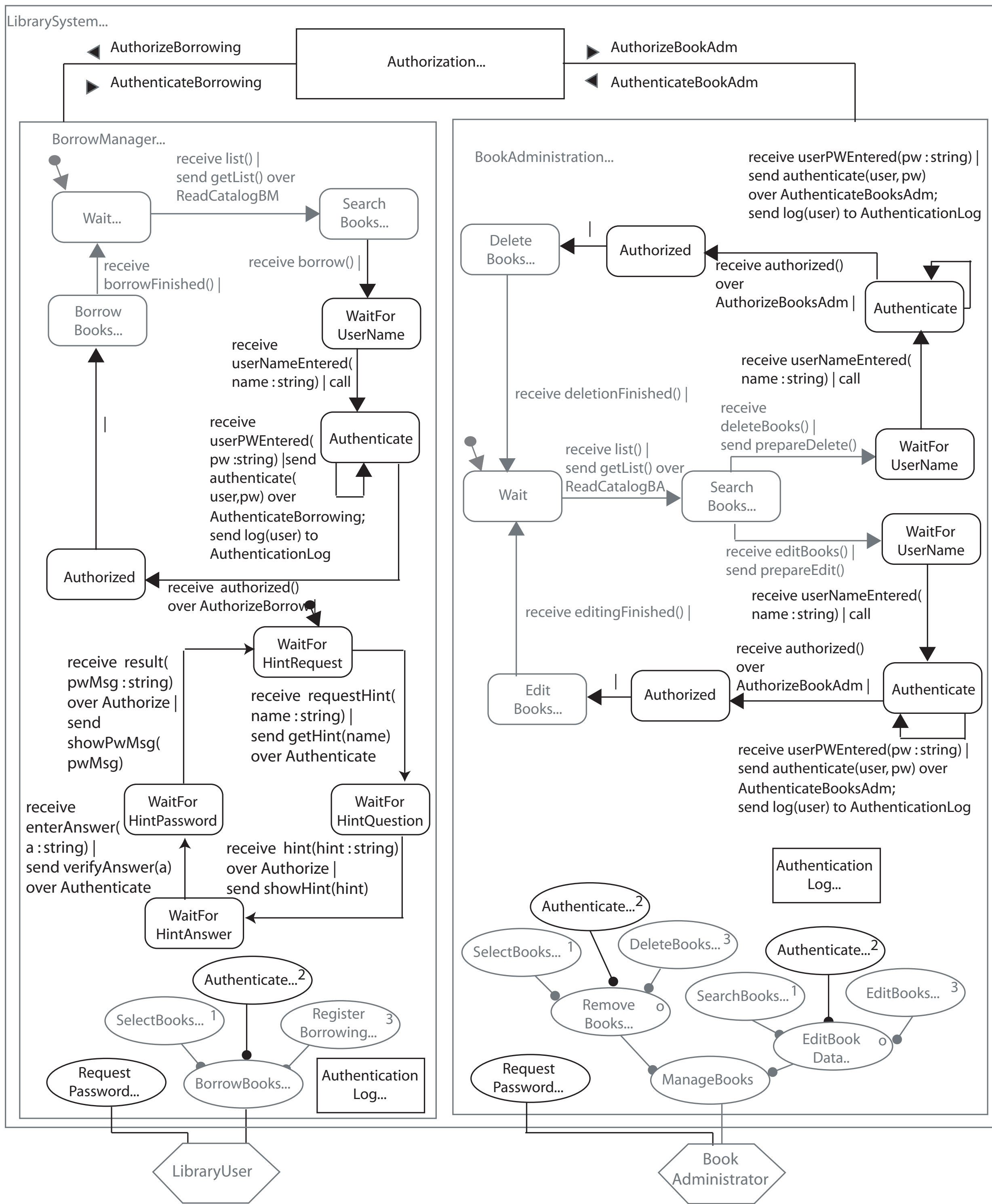


Figure 7.1: A partial view of the library system example showing the authentication concern crosscutting several components of the system.

Name and *Authenticate* which calls an operation[1] that delegates the authentication to the server component Authorization.

In either case of the delegation, redundancy is introduced, which affects one of the following views:

- **Basic View:** In the case that a task is delegated from a crosscutting concern to an embedded component, each component that is impacted by the crosscutting concern contains its own embedded, and therefore redundant, component.

  Example (7.3). In Fig. 7.1, the component *AuthenticationLog* appears in several components of the model. Each of them is used by the authentication behavior to log the authentication attempts independently for each component.

- **Structural View:** In the case where crosscutting behavior delegates tasks to a server component, the same component is shared between all occurrences of the crosscutting behavior. Therefore the server component is not redundant. However, the communication channels between the crosscutting behavior and the server component occur redundantly in the model.

  Example (7.4). In Fig. 7.1, the associations with the roles *AuthorizeBorrowing* and *AuthorizeBookAdm* illustrate the problem. Both are semantically equivalent, i.e., they have the same purpose, namely to facilitate the (same) communication between different impact locations of the crosscutting behavior and the service-providing component *Authorization*.

- **User View:** As crosscutting behavior may demand the interaction with objects in the environment of the system, use cases may also be redundant. Crosscutting scenarios may manifest either in fragmentary scenario descriptions or complete scenariocharts.

  Example (7.5). In Fig. 7.1, this is the case for the scenarios that describe the authentication interaction between the system and the *LibraryUser* as well as the *BookAdministrator*. Consequently, the corresponding scenario trees contain redundant elements, i.e., the node *Authenticate* and its (hidden) sub-nodes. The scenario *RequestPassword* denotes a complete scenariochart that is redundant.

- **Context View:** The environment objects of a system denote roles that interact with the system. An environment object is connected to a scenariochart that describes the use cases which can be performed by the environment object. The role represented by an environment object can be split up into sub roles. In turn sub roles can be associated with sub-scenarios. For instance, the role *LibraryUser* in Fig. 7.1 can be sub-divided into the roles *User who is searching for books*, the *user who is being authenticating by the system*, and the *user who is registering the borrowed books*. In the case that an environment object

[1] The operation is defined in the functional specification and therefore not shown in the graphical representation. The call of an operation is denoted by the keyword *call*.

contains a sub-role which is associated with a crosscutting scenariochart, the sub-role is also crosscutting and redundant. However, a crosscutting role is implicit in a conventional model and therefore it is not directly apparent from the model.

Example (7.6). The environment object *BookAdministrator* in Fig. 7.1 as well as *LibraryUser* contains a sub role *User who retrieves a forgotten password*, which is indicated by the association connecting them with the scenario *RequestPassword*. Even though the redundancy of the role does not in uence the description of the environment object, the meaning of the environment object is extended. As a consequence a part of the environment object's semantics may be redundant compared with other environment objects which comprise the same role.

- **Functional View:** Crosscutting behavior may cause redundancy in the functional view of components. This is due to the fact that elements of the functional view referred to by crosscutting behavior have to be replicated, too.

  Example (7.7). In Fig. 7.1, this is the case for the operation called[2] by the transition between the *WaitForUserName* and the *Authenticate* state. The operation is replicated for each location where the crosscutting behavior impacts.

7.2 Overview of the new Aspect-Oriented Approach

As discussed in chapters 3 and 4, various problems emerge when using a conventional language to describe crosscutting concerns. The redundancy caused by non-modularized crosscutting concerns may bloat software models, which compromises their comprehensibility. Apart from that, the altering of crosscutting elements in such models is a tedious task, as the same changes have to be accomplished for all redundant locations. Furthermore, it is difficult to implement traceability for crosscutting concerns, as their elements are scattered all over the model. Hence, it is desirable to reduce the redundancy as far as possible, which can be achieved by introducing elements that support an aspect-oriented modularization of crosscutting concerns.

A corresponding modeling language has not only to consider the modularization of crosscutting artifacts contained in the behavior description but also the crosscutting elements forming part of other facets, such as the static structure, the relationships to other elements, etc. Therefore, an aspect-oriented extension for a conventional language affects the base view, the behavioral view, the structural view, the user view, and the functional view. Furthermore, to avoid unnecessary bloating of the language, the aim is the aim to introduce new language elements as conservatively as possible.

The remainder of this chapter introduces an aspect-oriented extension for the ADORA language. The extension is based on the work in [Meie06, Meie07] and satisfies several desiderata proposed in Section 4.2. Figure 7.2 is used to illustrate the details of the approach. It shows

[2]The operation call is indicated by the *call* keyword in the action part of the transition. It causes the calling of an operation which is defined in the functional specification of the component.

an aspect-oriented version of the model in Fig. 7.1 and gives an overview of the new language elements by modularizing the crosscutting concern *Authentication*.

Listing 7.1: The *Authentication* aspect from Fig. 7.2 as textual specification.

```
1  aspect Authentication '1.1.7'
2    consists of
3      state WaitForUserName '1.1.7.1'
4        connections
5          transition '1.1.7.t.1' to '1.1.7.2'
6            receive userNameEntered(name : string) | call
7        end connections
8      end state WaitForUserName
9      state Authenticate '1.1.7.2'
10       connections
11         transition '1.1.7.t.2' to '1.1.7.3'
12           receive authorized() over Authorized |
13         transition '1.1.7.t.3' to '1.1.7.2'
14           receive userPWEntered(pw : string) |
15           send authenticate(user, pw) over Authenticate;
16           send log(user, pw) to AuthenticationLog
17       end connections
18     end state Authenticate
19     start state WaitForHintRequest '1.1.7.3'
20     ...
24     end state WaitForHintRequest
25     ...
42     exit Authorized '1.1.7.7'
43     end exit Authorized
44     root scenario Authenticate '1.1.7.8'
45       connections
46         scenarioconnection '1.1.7.8.s.1' to '1.1.7.9'
47         scenarioconnection '1.1.7.8.s.2' to '1.1.7.10'
48       end connections
49     end scenario Authenticate
50     sequence 1 scenario EnterUsername '1.1.7.9'
51       transform input userNameEntered(name : string)
52     end scenario EnterUsername
53     sequence 2 scenario EnterPassword '1.1.7.10'
54       transform input userPWEntered(pw : string)
55     end scenario EnterPassword
56     partial component AuthenticationLog '1.1.7.11'
57     end component AuthenticationLog
58     root scenario RequestPassword '1.1.7.12'
59     ...
66     end scenario RequestPassword
67     ...
```

```
73   end consists of
74   functional speci  cation
75     attributes
76       user : string;
77     operation userNameEntered(name : string)
78       statements
79         user = name
80     end operation userNameEntered
81   end functional speci  cation
82   connections
83     association '1.1.7.a.1' to '1.1.6' role 'Authenticate'
84     joinrelationship '1.1.7.j.1' from '1.1.7.1' to '1.1.3.t.2' before
85     joinrelationship '1.1.7.j.2' from '1.1.7.1' to '1.1.5.t.2' before
86     joinrelationship '1.1.7.j.3' from '1.1.7.1' to '1.1.5.t.3' before
87     joinrelationship '1.1.7.j.4' from '1.1.7.4' to '1.1.3.3' before
88     joinrelationship '1.1.7.j.5' from '1.1.7.4' to '1.1.5.8' before
89     joinrelationship '1.1.7.j.6' from '1.1.7.4' to '1.1.5.11' before
90   end connections
91 end aspect Authentication
```

As discussed in Chapter 6, ADORA models are described by a textual notation that is specified in an EBNF grammar and language constraints. A textual ADORA model can be mapped to a graphical representation. Correspondingly, the graphically visualized aspect module *Authentication* given in Fig. 7.2 is in fact described by the textual model in Listing 7.1 and mapped to a graphical representation.

For the introduction of aspect-oriented modeling elements in ADORA, the textual grammar and the corresponding graphical mapping of the previous work in [Seyb06a] needs to be augmented by several new language elements. In the following, the impact on the ADORA language specification is discussed by presenting excerpts of ADORA EBNF and the complementary language constraints. The graphical mapping for the language elements introduced is presented along with the full extended ADORA grammar in Appendices B and C, respectively.

Furthermore, note that an execution semantics for the new language elements does not need to be specified, as the aspect-oriented elements are not directly executable. However, the language extension presented needs rather to define rather a weaving semantics which describes the rules for transforming an aspect-oriented model into a conventional, i.e,. woven, model. Woven models can then be executed by the semantics defined in [Seyb06b]. The weaving semantics is discussed in Chapter 9.

The following sections describe the newly introduced aspect-oriented elements in more detail.

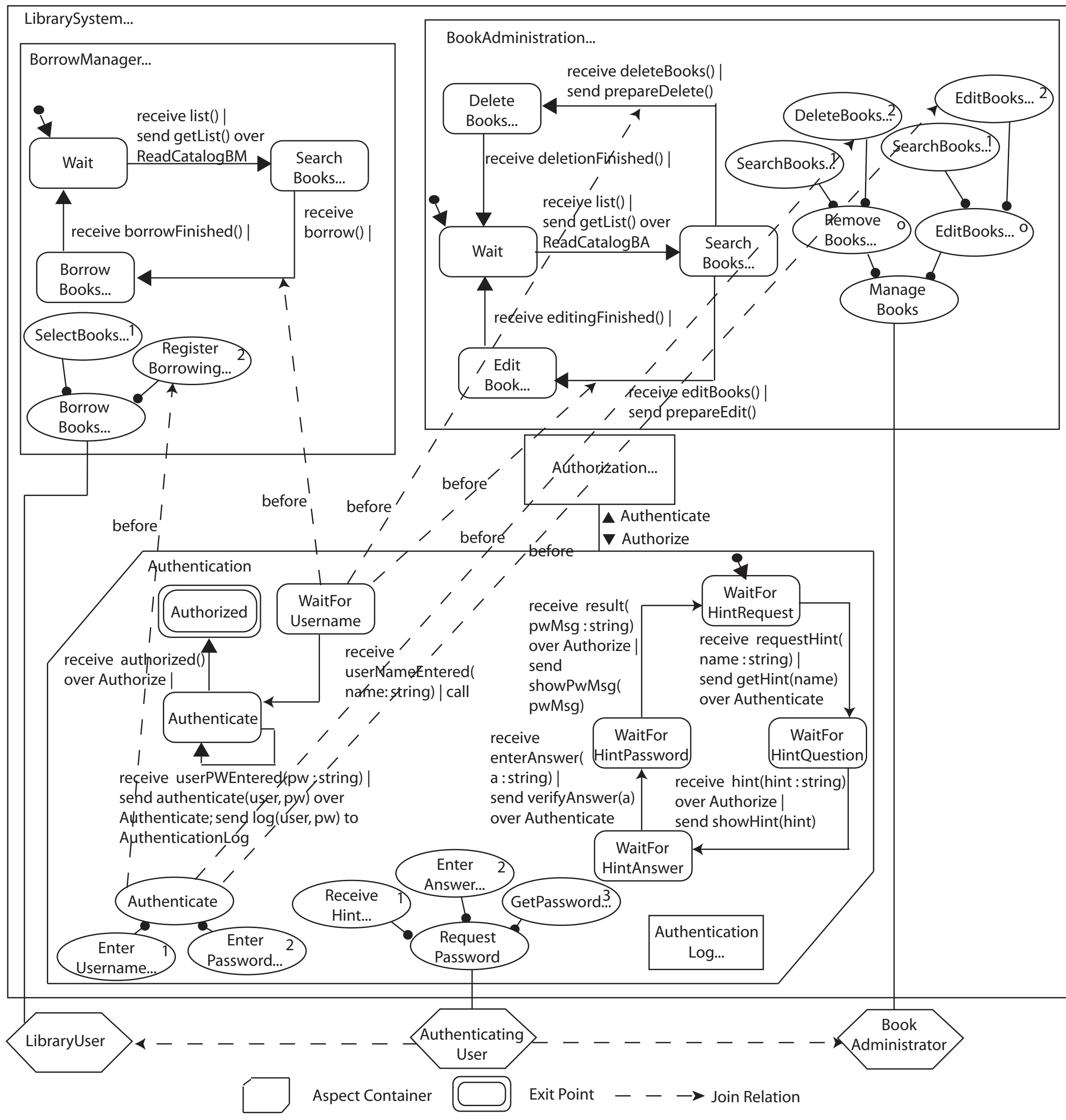


Figure 7.2: An example ADORA model that contains aspect-oriented elements (based on [Meie06])

7.3 Aspect Module

An aspect module[3] encapsulates the elements belonging to a crosscutting concern, such as the crosscutting behavior, crosscutting scenarios and other elements. An aspect module belongs to the main elements (cf. Chapter 6) in ADORA, hence it has a unique identifier and a graphical mapping. The aspect may have a name that helps to indicate its purpose. Like any other element, it may be augmented by informal descriptions. In Fig. 7.2 the beveled rectangle named *Authentication* denotes an aspect module.

Various elements may be encapsulated by an aspect module. For example, it may comprise a behavior description (cf. Section 7.4) and a crosscutting scenario description (cf. Section 7.5). An aspect module can also possess a functional specification describing datatype declarations, attributes, and operations of the crosscutting concern (cf. Section 7.8). Moreover, it may be connected by associations to components, or it may contain components (cf. Section 7.9). Join relationships express the impact of an aspect on other concerns and define the way the aspect is woven with its targets (cf. Section 7.6).

Two language features of aspect modules support the evolution of an aspect-oriented ADORA model. An aspect may be marked as partial (cf. Chapter 5), which expresses the intentional incompleteness and therefore a further model evolution. The other feature is the embedding of aspects in aspects. Embedding an aspect *A* in an aspect *B* expresses a refinement of *A* by *B* (cf. Section 7.9). It is useful for aspects which originate in non-functional requirements. The use of both language features is discussed in detail in Chapter 10.

7.3.1 Grammar Production Rules

Two extensions are needed in order to extend the ADORA EBNF for the support of aspect modules. The first extension concerns the aspect modules themselves and the second the embedding of the aspects in the model.

AspectDe nition Production Rule

The rule for the textual description of an aspect module is presented in Table 7.1 by the production *AspectDe nition*. An aspect module is a main element, thus it contains a name (*AspectName*) and a unique identifier (*UniqueModelElementIdenti er*). The lines 1–91 of Listing 7.1 illustrate the textual description of an aspect module, which adheres to the rule *AspectDe nition*.

The production rule *AspectParts* specifies the elements which can be a *part* of the aspect. Parts may be components, scenarios, states, exit points, and aspects.

Embedding of Aspects in ADORA Models

Aspects can occur at different locations in ADORA models. The tables 7.1 and 7.2 show the grammar production rules for the elements which may contain an aspect as a part. Aspects

[3] In this work aspect modules are also just called *aspects*.

Table 7.1: The EBNF grammar which describes aspect modules. Excerpt from Appendix B.

Production Name	Production Rule
AspectDefinition ::=	("**partial**")? "**aspect**" AspectName UniqueModelElementIdentifier AspectParts FunctionalSpecification AspectConnections "**end**" "**aspect**" AspectName
AspectParts ::=	("**consists**" "**of**" (ComponentDefinition | StateDefinition | ScenarioDefinition | AspectDefinition | ExitPointDefinition)+)?
AspectName ::=	SpecialIdentifier

Table 7.2: Syntax rules describing the embedding of aspects in components and the root of an ADORA model. Excerpt from Appendix B.

Production Name	Production Rule
Model ::=	"**model**" ((ComponentDefinition | EnvironmentObjectDefinition | AspectDefinition))* TypeDefinitions PropertyDefinitions StereotypeDefinitions "**end**" "**model**"
ComponentDefinition ::=	("**partial**")? ("**external**")? ("**start**")? "**component**" ComponentName UniqueModelElementIdentifier (Cardinality)? ("**is**" InheritedType)? ComponentParts BodyElements (ComponentConnections)? "**end**" "**component**" ComponentName
ComponentParts ::=	("**consists**" "**of**" (ComponentDefinition | StateDefinition | ScenarioDefinition | AspectDefinition)+)?

(*AspectDe nition*) may be embedded in the root of an ADORA model (*Model*) or they may also be either a part of a component (*ComponentParts*) or an aspect (*AspectParts*).

7.3.2 Language Constraints

For the sake of consistency in the textual description, the name in the header and the footer (i.e., lines 1 and 91 in Listing 7.1) must be equal.[4] This constraint is leniently enforced at a user-defined point in time. The constraint is given formally in Definition C_2 in Section F.1 of the appendix.

7.4 Crosscutting Behavior

An aspect-oriented method must allow the separation of crosscutting behavior without altering the logical ordering of the transition sequence triggered when executing the system's behavior.

[4]Moreover, all the other constraints on names introduced in the previous work of [Seyb06a, Xia04, Joos99] need to be satisfied, too.

In aspect-oriented ADORA, there are two ADORA constructs which allow this:

1. A so-called *behavior chunk* specifies a fragmentary crosscutting behavior description.

2. A *crosscutting statechart* is a self-contained behavior description which is executed concurrently with the behavior chunks in the aspect.

Behavior chunks. A behavior chunk is a fragmentary statechart which amends the behavior in the target module. It consist of atomic and complex states, components and transitions. The meaning of these elements is the same as for conventional ADORA statecharts (cf. Chapter 5). Nevertheless, a behavior chunk is not a self-contained description of behavior. It does not have a start state and it is not a complete behavior description. Furthermore, the behavior is crosscutting, i.e., it occurs in a conventional model at several locations.

Each behavior chunk contains two types of junctions: so-called entry and exit points. One or more entry points designate where the crosscut behavior enters the behavior chunk. The entry point is denoted as a state or a component with an out-going join relationship (cf. Section 7.6). A behavior chunk has usually just one entry point[5] but does not need to have one. If a behavior chunk has no entry point, i.e., no outgoing join relationship, the crosscutting behavior has no impact on a target.

In contrast to an entry point, an exit point defines where the behavior chunk is left and the crosscut behavior is reentered. There must be *exactly one* exit point which is connected by one or more in-coming transitions with the behavior chunk. An exit point is visualized graphically as a rounded rectangle with a double outline.

Behavior chunks are state groups (cf. Section 5.2.3). In order to constitute behavior chunks, the original definition for state groups needs to be extended. Besides states, abstract objects and object sets, exit points are also allowed as vertexes.

A behavior chunk is defined as state group which contains exactly *one* exit point, and which has *no* node that is marked with a start state indicator.

An aspect may contain one or more behavior chunks. Nevertheless, an aspect that contains more than one behavior chunk may contain too many responsibilities. Too many responsibilities may worsen the understandability of the model. Therefore they should be split up into multiple aspect modules.

For example in Fig. 7.2, the behavior chunk is constituted by the states *WaitForUserName* and *Authenticate*, the exit point *Authorized* and the transitions between them. An exit point is denoted by a rounded rectangle with a double outline. The *WaitForUserName* state is the entry point of the behavior chunk.

Crosscutting statecharts. Another kind of crosscutting behavior is described by so-called crosscutting statecharts. In contrast to a behavior chunk, a crosscutting statechart is a self-

[5] Note that in [Meie06] it is mentioned that an entry point must be unique. In fact, it does not need to be unique. Nevertheless, for the sake of a coherent usage and an easy understandability of the crosscutting behavior, just one entry point should be used for all join relationships outgoing from the behavior chunk.

contained behavior description which is interrelated with at least one behavior chunk. It is concurrently executed with other statecharts and chunks contained in the aspect.

A crosscutting statechart is described by the same syntax and semantics as used for describing the behavior of a component. Hence, it consists of a start state, atomic states, complex states, components and transitions. More formally, a crosscutting statechart is a state group which has exactly *one* start state, *no* entry points (i.e., they do not have join relationships which are outgoing from a state or a component and *no* exit points).

The model in Fig. 7.2 shows a crosscutting statechart whose functionality allows a user to retrieve a forgotten password by requesting and answering a password hint question. The mechanism comprises the states *WaitForHintRequest*, *WaitForHintQuestion*, *WaitForHintAnswer*, and *WaitForPassword*, as well as the corresponding transitions. Apart from the concurrent behavior, a so-called *crosscutting scenariochart* with the root node *RequestPassword* (cf. Section 7.5) is also introduced in this figure. It describes the communication protocol between the corresponding environment object and the crosscutting statechart.

7.4.1 Production Rules

The behavior description of an aspect is specified by the syntax rules in Table 7.3. Aspect-oriented behavior descriptions are similar to conventional behavior descriptions of components, except that they may also contain exit points. Thus, a behavior description of an aspect may be formed of components (*ComponentDe nition*), states (*StateDe nition*), exit points (*ExitPoint-De nition*) and transitions (*TransitionDe nition*). The former three elements are embedded as a *part* in an aspect which is specified by the grammar rule *AspectParts* in Table 7.1. Components and states may in turn contain outgoing transitions.

Transitions connect nodes of the behavior descriptions (i.e., exit points, states and components) to state groups. A transition is specified according to the grammar rule *TransitionDe nition* in Table 7.3 which is referred to by components (in the rule *ComponentConnections*) and states in the rule (*StateConnections*), respectively.

Note that components which have at least one ingoing or outgoing transition belong to a behavior description of an aspect. In contrast, components which have no ingoing or outgoing transitions do not belong to a state group.[6] Consequently, they are not part of an aspect's behavior description and therefore they are seen as embedded components, which is discussed in detail in Section 7.9.

An example of the textual elements of a behavior chunk is given in Listing 7.1 where the lines 3–18 describe the states of a behavior chunk of the *Authentication* aspect. The exit point of the behavior chunk can be found in lines 42 and 43. The lines 5–6 and 11–16 illustrate textual transition descriptions of the behavior chunk. The crosscutting statechart is shown between lines 19 and 41.

[6]This fact is implicit and therefore not apparent from the syntax.

Table 7.3: EBNF syntax rules describing the behavioral view of an aspect. Excerpt from Appendix B.

Production Name	Production Rule
ComponentDefinition ::=	("**partial**")? ("**external**")? ("**start**")? "**component**" ComponentName UniqueModelElementIdentifier (Cardinality)? ("**is**" InheritedType)? ComponentParts FunctionalSpecification (ComponentConnections)? "**end**" "**component**" ComponentName
ComponentConnections ::=	"**connections**" (AssociationDefinition | AssociationRoleDefinition | TransitionDefinition)* "**end**" "**connections**"
ExitPointDefinition ::=	"**exit**" StateName UniqueModelElementIdentifier "**end**" "**exit**" StateName
StateDefinition ::=	("**partial**")? ("**start**")? "**state**" StateName UniqueModelElementIdentifier CompoundStates (StateConnections)? "**end**" "**state**" StateName
StateName ::=	SpecialIdentifier
StateParts ::=	("**consists**" "**of**" (StateDefinition)+)?
StateConnections ::=	"**connections**" (TransitionDefinition)* "**end**" "**connections**"
TransitionDefinition ::=	("**partial**")? "**transition**" UniqueModelElementIdentifier "**to**" ElementReference (SimpleTransition | DecisionTableTransition)?

7.4.2 Language Constraints

Several language constraints need to be specified to get well-formed behavior descriptions of an aspect module. They complement the EBNF rules presented above.

State Groups Must Be Well-Formed

Each state group in an aspect module must be either a well-formed behavior chunk or a well-formed crosscutting statechart. Besides the syntactical correctness, a well-formed behavior chunk must have *exactly one* exit point and *no* start state, whereas a crosscutting statechart must comprise *exactly one* start state and *no* exit point. Figure 7.3 (a) and (b) show situations which satisfy this constraint, whereas (c), (d), (e), (f), and (g) show models which violate it. Situation (c) violates the constraint because a state group must *either* be a crosscutting statechart (indicated by a start state) *or* a behavior chunk (denoted by an exit point) but not both. Situation (d) has two start states, whereas (e) has none. The behavior chunk in situation (f) contains two exit points, which is also not allowed. Situation (g) is wrong because an exit point is missing.

This constraint is leniently checked before the aspect-oriented model is woven. Its formal specification is given in Definition C_3 in Section F.2 of the appendix.

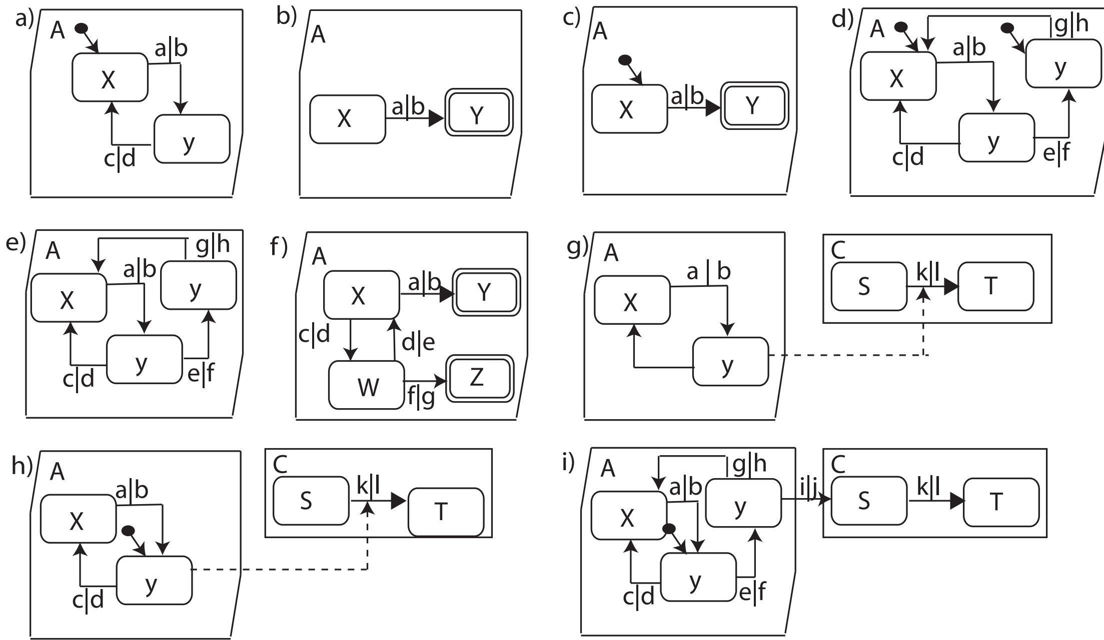


Figure 7.3: Examples violating and satisfying the constraints of an aspect's behavior description. In (a) and (b) no constraint is violated. All other situations violate one of the behavior constraints.

No Out-Going Join Relationships from Crosscutting Statecharts

No state or component which is part of a crosscutting statechart may have an out-going join relationship. In Fig. 7.3, the situation (h) illustrates the violation of this constraint. It connects a crosscutting scenariochart by a join relationship to a transition. This constraint is strictly enforced, i.e., it is checked before a join relationship is inserted in the model. It is formally specified by Definition C_4 in Section F.2 of the appendix.

No Crossing of the Aspect Border by Transitions

The transitions of a behavior chunk or a crosscutting statechart may not cross the border of the aspect module. In Fig. 7.3, situation (i) shows the violation of this constraint. This constraint is strictly enforced and specified formally in Definition C_5 in Section F.2 of the appendix.

Transitions Can Be Connected to Exit Points

In the conventional ADORA modeling language, transitions are only allowed to be connected either to states or to components. However, with the introduction of aspects, transitions may also have exit points as targets. Therefore the corresponding constraint defined in the previous work has to be relaxed. It is given by the formal Definition C_6 in Section F.2 of the appendix. This constraint is strictly enforced.

Other Language Constraints

There are language constraints restricting the behavior view of a component, defined by the previous work in [Seyb06a, Xia04, Bern02, Glin02b, Joos99, Cram07]. They also apply to the behavior description of an aspect.

7.5 Crosscutting User View

As discussed in Section 5.2.4, the user view describes the protocol of the interaction between environment objects and particular components in the system, i.e., the use cases of the system, by means of scenariocharts. Use cases can be crosscutting too, and therefore, they need to be introduced also for aspects. There are two different types of crosscutting use cases:

1. **Scenario Chunks:** The interaction between an environment object and a behavior chunk is described by a *scenario chunk*.
2. **Crosscutting Scenariocharts:** The interaction protocol between a crosscutting statechart and an environment object is described by a conventional scenariochart embedded in the aspect module.

Scenario chunks. Environment objects may interact with a behavior chunk (cf. Section 7.4) of an aspect. A *scenario chunk* is a fragmentary use case of an aspect which describes the interaction protocol between an environment object and a *behavior chunk* of an aspect. It complements a use case description of the crosscut module and can consists of nodes and connections forming a tree. Its syntax and semantics is similar to a scenariochart (cf. Section 5.2.4). Nevertheless, a scenario chunk is not self-contained, i.e., it is a fragmentary description, as it does not describe a full interaction between an environment object and a system part.

Scenario chunks normally interact with a behavior chunk, as they describe a part of the communication between environment objects and the behavior chunk. However, a scenario chunk need not have a corresponding behavior chunk as a counterpart and vice-versa.

A well-formed scenario chunk is a scenario group (cf. Section 5.2.4). In contrast to a scenariochart, a scenario chunk may not be connected by an association to an environment object. The root node of the scenario chunk is rather connected by an out-going join relationship (cf. Section 7.6) to a target scenario node. The target scenario node indicates where the scenario chunk crosscuts the target scenario. However, there is no need to have an out-going join relationship. In this case, the scenario chunk has no impact on a target.

Each node in a scenario chunk has a specific scenario type, which follows the same type schema as for conventional scenariocharts (cf. Section 5.2.4). Note that in [Meie06], it was claimed that the type of the behavior chunk's root node must have the same type as the target scenario node and its siblings. However, there is no need for this constraint, as it would restrict the reusability of a scenario chunk too much. Therefore this constraint is relaxed: the root node of a scenario chunk has to be of the type *Root*. In a woven model, the type of the behavior

chunk's root node is adapted to have the same type as its target node and its siblings. This issue is also discussed in Chapter 9.

Figure 7.2 illustrates a scenario chunk comprising the three scenario nodes *Authenticate*, *EnterUsername* and *EnterPassword*. The latter two nodes represent sub-scenarios of *Authenticate*. It describes the interaction between the internal behavior for the authentication mechanism and the environment objects in the target. The out-going join relationships indicate the target of the (crosscutting) scenario chunk. In the example, there are three target nodes which describe the deletion (*DeleteBooks*), the editing (*EditBooks*), and the borrowing (*RegisterBorrowing*) of the registry records of books.

Crosscutting scenariochart. Apart from scenario chunks, crosscutting scenariocharts can describe the interaction between an environment object and the behavior description of an aspect. A crosscutting scenariochart is a self-contained description of the interaction between an environment object and a crosscutting statechart (cf. Section 7.4). A crosscutting scenariochart may not have any out-going join relationships, but it must be connected by an association from their root node to an environment object.

Figure 7.2 illustrates the use of a crosscutting scenariochart in an aspect. In this example, a mechanism for retrieving lost passwords (cf. Section 7.4) is given, which is specified as a crosscutting scenariochart. As the behavior of this mechanism needs to interact with an environment object, the interaction protocol must be described. In the example, this protocol is specified by the crosscutting scenariochart constituted by the nodes *RequestPassword*, *ReceiveHint*, *EnterAnswer* and *GetPassword*, where the latter three are the sub-scenarios of *RequestPassword*. The root node of the crosscutting scenariochart is connected by an association to the environment object *LibraryUser*.

7.5.1 Production Rules

Table 7.4 contains the set of grammar rules which specify the textual description of scenario nodes (*ScenarioDe nition*) and connections (*ScenarioConnections* and *ScenarioConnectionDefinition*). The textual description of scenario connections are embedded in their source node (cf. rule *ScenarioDe nition*). The nodes of scenario chunks and crosscutting scenariocharts are parts of an aspect module, which is defined by the grammar rule *AspectParts* in Table 7.1.

Lines 44–55 of Listing 7.1 give an example of the textual specification of a scenario chunk. They describe the scenario nodes *Authenticate*, *EnterUsername* and *EnterPassword* textually. The scenario connections between the *Authenticate* scenario and the other two nodes are illustrated by lines 46–47. The definition of the crosscutting scenariochart can be found between the lines 58 and 72.

7.5.2 Language Constraints

In order to enforce well-formed ADORA models, the syntactical syntax elements presented above must be complemented with several language constraints.

Table 7.4: EBNF syntax rules for the user view of an aspect.

Production Name	Production Rule
ScenarioDefinition ::=	("**`partial`**")? ScenarioType "**`scenario`**" ScenarioName UniqueModelElementIdentifier ("**`on`**" GuardPart)? ("**`iteration`**" Expression)? (ScenarioConnections)? (TransformationElements)? "**`end`**" "**`scenario`**" ScenarioName
ScenarioName ::=	SpecialIdentifier
ScenarioType ::=	("**`alternative`**" | "**`sequence`**" (<INTEGER_LITERAL>)? | "**`parallel`**" | "**`root`**" (Cardinality)?)
ScenarioConnections ::=	"**`connections`**" (ScenarioConnectionDefinition | AssociationDefinition | AssociationRoleDefinition)* "**`end`**" "**`connections`**"
ScenarioConnectionDefinition ::=	"**`scenarioconnection`**" UniqueModelElementIdentifier "**`to`**" ElementReference

No Crossing of the Aspect Border by a Scenario Connection

As defined by the language constraint described in [Xia04, p. 52], the connections of a scenario-chart may not cross the border of a component. Analogously, this constraint applies for aspects. The connections of a scenario group may not cross the border of the aspect in which the scenario group is embedded. This means that any scenario child of a root node that is part of an aspect A must also be embedded in A, or in one of A's embedded components.

The figures 7.4 (a) and (b) illustrate this constraint. Situation (a) shows a well-formed model, where two scenario connections do indeed cross the border of the *embedded components* (cf. Section 7.9). However, they do not cross the border of the aspect, therefore the constraint is not violated. In contrast, situation (b) is malformed as a scenario connection crosses the border of the aspect. This language constraint is strictly enforced and formally specified by Definition C_7 in Section F.3 of the appendix.

Well-Formed Scenario Chunks

A well-formed scenario chunk is a scenario group which has to satisfy one the following two conditions. First, if there is a join relationship outgoing from the scenario group, there may be no associations connected to any node in the scenario group. Second, there does not have to be any out-going join relationship, but if there is one, it has to originate from the root node of the scenario group.

Figure 7.4 (c)–(f) describe situations which satisfy or violate this constraint. The model in (c) describes a scenario chunk which does not have an out-going join relationship; nevertheless, it is well-formed. The situation (d), where the root node is connected by a join relationship to a target node, is also well-formed. In contrast, figure (e) shows an malformed model, as the join relationship is not outgoing from the root of the scenario chunk. Situation (f) violates the

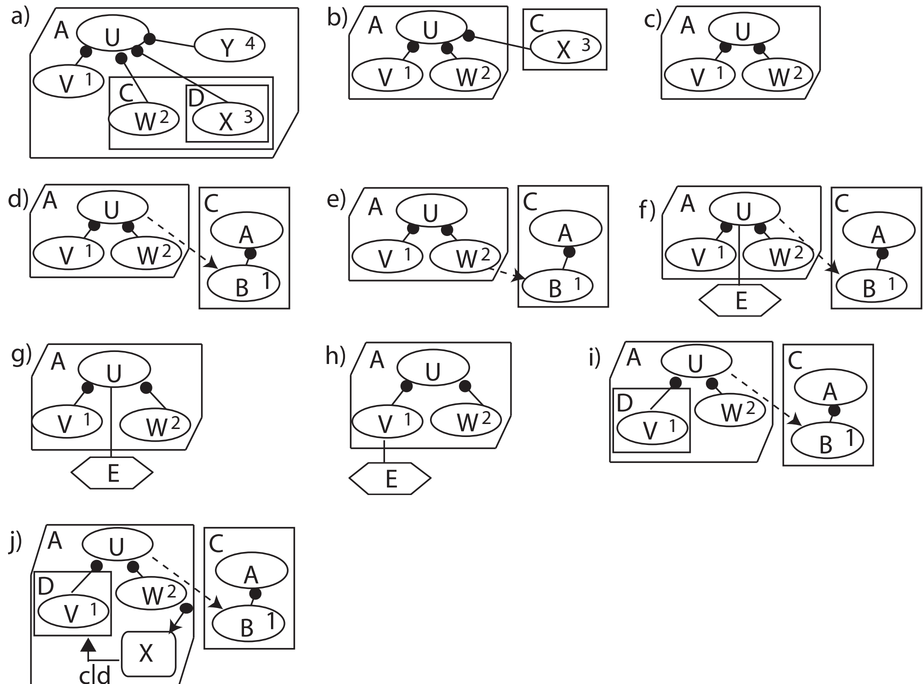


Figure 7.4: Examples violating and satisfying the constraints. Situations (a), (c), (d), (g), and (i) satisfy the user view constraints. All other situations violate one of them.

constraint as the root node is connected by a join relationship and an association. This constraint is strictly enforced and formally specified in Definition C_8 in Section F.3 of the appendix.

Well-Formed Crosscutting Scenariocharts

A well-formed crosscutting scenariochart must satisfy three conditions. First, the root node of the scenariochart must be connected to at least one environment object. Second, no join relationships may originate in the nodes of the scenario group, and third, non-root nodes may not have an association connected to another node.

Figure 7.4 (f)–(h) illustrate situations which violate or satisfy this constraint. Situation (f) neither denotes a well-formed crosscutting scenariochart nor a well-formed scenario chunk, because the root node is connected by a join relationship and an association. The model in (g) shows a well-formed crosscutting scenariochart. In contrast, situation (h) is malformed as it connects the environment object to a non-root scenario node. This constraint is strictly enforced and specified formally in Definition C_9 in Section F.3 of the appendix.

Disallowed Embedding of a Scenario Node in a Component Belonging to a Statechart

A scenario node can be part of components. However, it is not allowed to embed a scenario node in a component which is part of a statechart. This is due to the fact that this may otherwise

lead to contradictions in the modeling, as well as when performing weaving transformations on the aspect-oriented model. Figure 7.4 (i) shows a model satisfying this constraint, (j) a situation which violates it. This constraint is leniently enforced before the aspect-oriented model is woven. Its formal specification can be found in Definition C_{10} in Section F.3 of the appendix.

Other User View Constraints

Apart from the given language constraints above, there are several others from the previous work in [Joos99, Glin02b, Bern02, Xia04, Seyb06a, Cram07] restricting the modeling of scenarios, e.g., that each scenario group must be a tree. These language constraints were defined for the user view of conventional ADORA models, nonetheless they also apply to the user view of aspects.

7.6 Join Relationships

One major concept of aspect-oriented software development is the decoupling of modules by crosscutting relationships (cf. Chapter 3). A crosscutting relationship is unidirectional between two artifacts A and B, where A constrains B and B has no in uence on the way it is constrained by A. In systems where crosscutting concerns are modularized, crosscutting relationships specify where an aspect has an impact on the target, i.e., crosscutting relationships are the main means of decoupling aspects from other modules.

Some aspect-oriented modeling and programming approaches, e.g., AspectJ [Kicz01b], do not specify crosscutting relationships explicitly. Mostly, they use some kind of quantification expression which implicitly defines where the aspect crosscuts the modules of the other concerns. However, for a graphical modeling approach, an explicit definition and the showing of the crosscutting relationship is desirable.

Therefore, aspect-oriented ADORA uses an explicit join point model, where the crosscutting relationships are explicitly defined and shown. Pattern matching or other ways of implicitly denoting a crosscutting relationships is deliberately not used, because this reduces the effectiveness of a visual model and increases the cognitive effort needed to read it. Apart from that, unintentional (i.e., wrong) matches may occur when using pattern matching mechanisms. In the aspect-oriented ADORA approach, crosscutting relationships are called join relationships.

In Fig. 7.2, several join relationships are part of the given model. For example, one of them points from the state *WaitForUserName* to the transition between *SearchBooks* and *BorrowBooks* in the *BorrowManager* component.

Partial vs. concrete join relationships. Join relationships may be partial or concrete. A partial join relationship displays an intentionally incomplete, i.e., not fully evolved, join relationship between an aspect and a target. In contrast, a concrete join relationship denotes a fully evolved join relationship.

Partial join relationships are one of the two syntactical elements supporting the identification, separation and evolution of crosscutting concerns by a guided process.[7] In ADORA, partial is

[7] The other syntactical element is the aspect refinement mechanisms (cf. Section 7.9).

also called *manually abstracted*, or even less precisely, just abstract. Note that there are also visually abstracted join relationships, i.e., join relationships that are shown in a partial view (cf. Section 8.1.2).[8]

A *partial* join relationship allows the connecting of various combinations of source and target nodes. The source node may either be an aspect, a state of a behavior chunk or a node of a scenario chunk, whereas the target is either a component, a state, an association, a scenario node or a transition. As mentioned above, a partial indicator that is set expresses an unfinished evolution of the join relationship between its source and the target element. A detailed discussion of the possible combinations of elements which can be connected by join relationships can be found in Section 7.6.2. Furthermore, the use of abstract join relationships in a evolutionary requirements process is discussed in more detail in Chapter 10.

In contrast, a *concrete* join relationship denotes a finished evolution. It connects concrete pieces of behavior, scenario descriptions, or crosscutting scenarios. This is discussed in detail in the constraints section.

Apart from the partial property, a join relationship may incorporate three additional attributes which are crucial to the meaning of an aspect-oriented model. There is an *ordering descriptor*, a *priority* and a *context map*.

Ordering and priority attributes. The ordering may be optionally specified and indicates the weaving order between the target element and the aspect-oriented element. It is specified by either of the three keywords *before*, *instead*, or *after*. If no ordering is given, the *before* ordering is taken as the default value. A detailed discussion of the ordering descriptor's weaving semantics can be found in Chapter 9.

Besides the ordering descriptor, a join relationship may be attributed with a priority which denotes the precedence of two competing aspects during the weaving process. Two aspects are competing if both target the same element with the same ordering. A priority is a number between 1 and 10, where 1 is the lowest priority and 10 the highest. If no priority is given, 1 is chosen as default. If two aspects with the same priority are in competition, one of them is chosen non-deterministically to be woven first into the target.

Context map. Sometimes aspects need to access information in the context of their target for fulfilling their intended function. For example, an aspect which logs data needs to access the information of the crosscut modules. The problem is illustrated with the situation in Fig. 7.5 (a) where a *Logging* aspect and the crosscut component *BookAdministration* of the partially viewed *LibrarySystem* are given. Suppose that the following requirements are documented by the model presented:

- There should be a mechanism for enabling and disabling the logging aspect, which is realized by the transitions between the state *StartLogging* and the exit point *EndLogging*. They are triggered by a guard evaluating the attribute *logEnabled*.

[8] In general, *abstraction* is the superordinate concept for partial modeling and partial viewing.

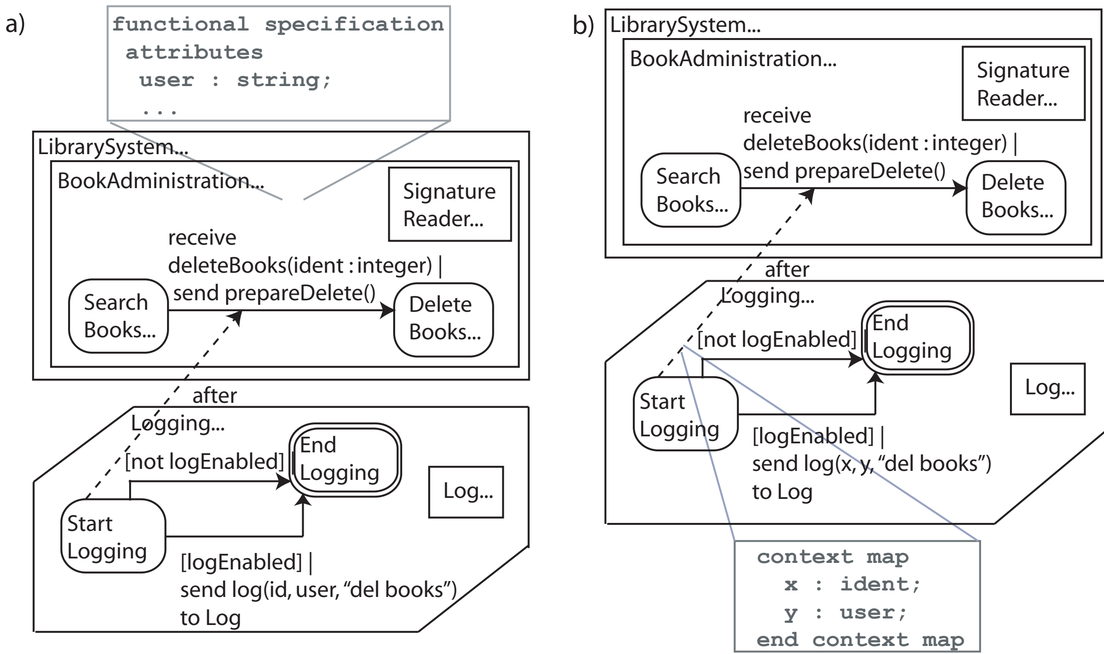


Figure 7.5: Example motivating the usage of a context map.

- The logging must protocol the deletion of book records by journalizing the book *id*s.
- The *user* which committed the deletion must also be recorded.

The recording of this information is done by the *Log* component which is part of the aspect. It adds a log entry as soon as a message *log(elementId : integer, user : string, msg : string)* is received.

On the one hand, a book is identified by the *id* parameter of the message *deleteBooks(id : integer)* which triggers the transition between *SearchBooks* and *DeleteBooks*. On the other hand, the name of the *user* who is currently executing the delete operation has been stored in the attribute *user*. This attribute is part of the functional specification of the *BookAdministration* object. The functional specification is hinted at by the gray box in Fig. 7.5 (a).

For the logging task, the question is how the crosscutting behavior can access the attribute *id* and *user* which are defined in the context of the target *BookAdministration* but not as elements in the aspect. A naive approach for accessing these context elements by an aspect would be to refer to them by simply using the names and the namespace of the target's context. This would be possible, as the elements of an aspect are woven with a target and therefore are actually in the same namespace as the elements in the target. For example, in the crosscutting behavior given in Fig. 7.5 (a), the aspect elements refer directly to the elements *id* and *user* in the send statement (*send log(id, user, "deleteBooks")*). However, this attempt to access the elements in the context of the target *BookAdministration* con icts with the idea that aspects are strongly decoupled from

their targets. Therefore it compromises their reusability, since for each crosscut module, there may be differently named elements which have to be accessed by the aspect.

When referring to elements in the target context, the way to decouple an aspect from its target elements is to use a proxy variable. This proxy variable needs to be mapped to context elements for each target. The mapping is defined for each join relationship individually as a join relationship is the only place where a crosscutting concern and its target are coupled, and where target-specific information, e.g., the ordering of the aspect with respect to the target is introduced. The mapping is defined by an expression referring to context elements. This expression may compute an arbitrary value.

In Fig. 7.5 (b), an example of a mapping for the situation described in Fig. 7.5 (a) is defined. In the *send* statement of the aspect's crosscutting behavior in Fig. 7.5 (b), the proxy variables *x* and *y* are used instead of accessing directly the elements in the context. Furthermore, a mapping table, which is visualized as a gray box[9], is defined for the join relationship. It defines how both these variables are used by the aspect. The mapping consists of a proxy variable *x* and a replacement expression after the colon. The proxy variable is used as a placeholder in the behavior description of the aspect which is substituted by the replacement during the weaving. In the example, the replacement expression consists simply of the variable with the name *id*. This means that all occurrences of the variable *x* in the woven crosscutting behavior are substituted by *id*. Correspondingly, the proxy variable *y* is mapped to the *user* variable.

7.6.1 Production Rules

Table 7.5 shows the syntax rules which define join relationships and the elements belonging to them. The rule *AspectDe nition* in Table 7.1 refers to the *AspectConnections* rule in Table 7.5 which allows an aspect to have between zero and several join relationships.

Join relationships are defined by the rule *JoinRelationshipDe nition*. Join relationships are the only kind of connections in ADORA which have a textual representation that is not embedded in their source element (cf. Section 6.1.4 and 6.1.6).[10] Join relationships are rather embedded in the aspect which is the parent of their source. As a consequence, a join relationship not only references its target but also its element of origin. Hence a constraint is needed which ensures that the source node of the join relationship is a part of the aspect in which the relationship's textual specification is embedded (see below). Furthermore, another constraint for checking the correct type of the source is needed (see below).

A join relationship is a main element of ADORA, i.e., it has a unique model element identifier and is graphically visualized. The textual description of a join relationship contains four optional elements: a partial indicator, a priority (*Priority*), an ordering keyword, and a context map (*ContextMap*, *ContextMapping*).

[9]Note that the mapping is normally not visualized in the graphical model.

[10]This is due to practical reasons when designing the language. Locating the EBNF rule for the join relationship in each actual source element would require the definition of many more language constraints.

Production Name	Production Rule
AspectConnections ::=	("**connections**" (AssociationDefinition | AssociationRoleDefinition | JoinRelationshipDefinition)* "**end**" "**connections**")?
JoinRelationshipDefinition ::=	("**partial**")? "**joinrelationship**" UniqueModelElementIdentifier "**from**" ElementReference "**to**" ElementReference ("**before**" | "**instead**" | "**after**")? (Priority)? (ContextMap)?
Priority ::=	<INTEGER_LITERAL>
ContextMap ::=	"**context**" "**map**" ContextMapping (ContextMapping)* "**end**" "**context**" "**map**"
ContextMapping ::=	Identifier "**:**" Expression "**;**"

Table 7.5: EBNF grammar defining join relationships.

7.6.2 Language Constraints

In order to enforce well-formed aspect-oriented models, it is necessary to complement the syntax rules for join relationships by a set of language constraints. Among its other characteristics, a join relationship is not embedded in its source but rather in the aspect to which it belongs, and therefore, additional language constraints are required. Furthermore, the join relationship syntax rules in Table 7.5 allow the connection of an arbitrary type of source and target element. However, a join relationship is restricted in how it is connected to a constituent node.

Constituents of Non-Partial Join Relationships

In the case of a concrete join relationship, three combinations of the source and target elements are allowed. First as illustrated in Fig. 7.6 (a), a join relationship can connect a component or state of a behavior chunk (cf. Section 7.4) with a transition in the target. Second, a join relationship can connect a scenario chunk node (see Section 7.5) and a scenario node in the target, which is illustrated by Fig. 7.6 (b). Third, two environment objects can be connected by a join relationship (c)[11], which has the meaning that one actor crosscuts the other (cf. Section 7.7). The constraint which handles the possible constituent nodes of non-partial join relationships is strictly enforced and formally specified in Definition C_{11} in Section F.4 of the appendix.

Constituents of Partial Join Relationships

As mentioned above, a join relationship with a partial ag describes an unfinished evolution. In this case, a join relationship can connect the following source and target types:

[11] The black parts of the model illustrate the actual situation. The gray parts may be substituted with any other well-formed model parts.

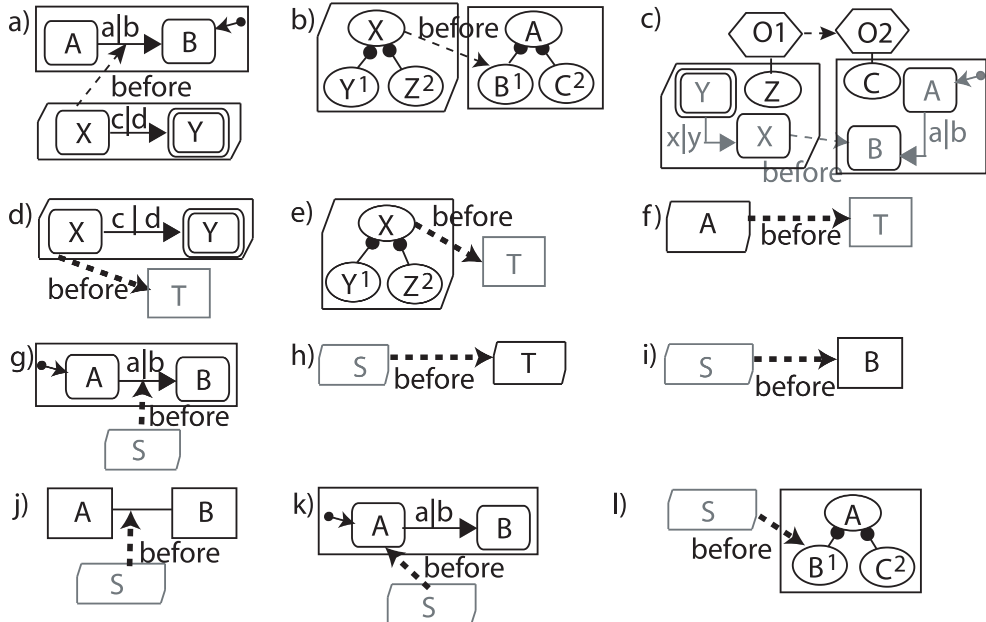


Figure 7.6: Allowed use of partial and concrete join relationships. This figure illustrates the allowed situations for partial and non-partial join relationships.

First, Fig. 7.6 (d)–(f) exemplifies the possible type of sources. The source may either be a state (d), a scenario node (e), or an aspect module (f). In each of these examples the target is a component (printed in gray) and may be substituted with any of the target types defined in the following.

Second, the type of possible target nodes of a partial join relationship are depicted by Fig. 7.6 (g)–(l). The target may either be a transition (g), an aspect module (h), a component (i), an association (j), a state (k), or a scenario (l). In each of these examples, the source is an aspect (printed in gray). However, any type of source node defined above can be used.

Apart from the above defined types for the source and the target of a join relationship, the source must either be a child element of the aspect or the aspect module itself. In addition to the cases described above, if an actor has an outgoing partial join relationship, it can only be connected with another actor. This constraint is strictly enforced and specified formally in Definition C_{12} of Section F.4 of the appendix.

Join Relationships Connecting to Scenariochart Root Nodes

A concrete join relationship may connect a node of a scenario chunk with the *root* node of a target scenario group. This represents a special case as the join relationship must have the ordering *instead*, because *before* and *after* are not meaningful in this situation. Therefore, the correct ordering keyword of a join relationship that points to a root scenario node must be ensured

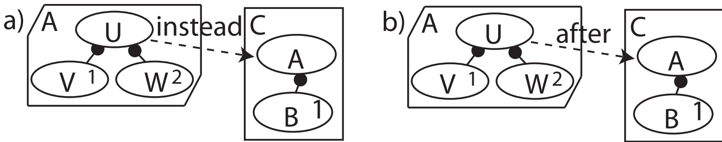


Figure 7.7: Concrete join relationships connected to the root node of a scenariochart. Figure (a) shows a well-formed situation, whereas situation (b) is malformed.

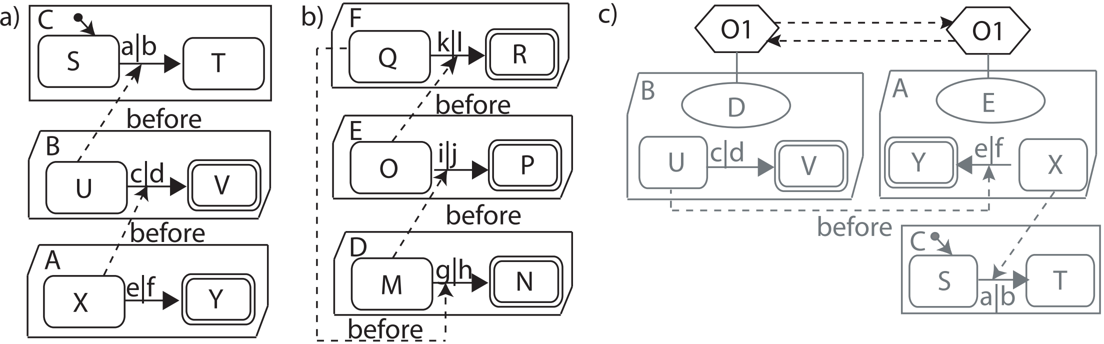


Figure 7.8: Model which exemplifies cyclic that join relationships are not allowed. Model (a) is well-formed and situation (b) malformed. Situation (c) shows a cycle of join relationships between environment objects and is therefore also malformed.

by a language constraint. Figure 7.7 (a) satisfies and (b) violates this constraint. It is leniently enforced before the aspect-oriented model is woven and formally specified in Definition C_{13} in Section F.4 of the appendix.

No Cycles in Join Relationships

Aspects may also crosscut aspects: for instance, a transitive crosscutting of aspects occurs when an authentication must be logged. If the logging functionality is also used at other locations in the system, the logging is (strictly) crosscutting, besides others, the authentication concern. Fig. 7.8 (a) illustrates such a transitive crosscutting, where an aspect *A* crosscuts aspect *B* that in turn crosscuts component *C*.

That an aspect can crosscut aspects implies two facts: (i) The weaving semantics for circular join relationships, as exemplified in Fig. 7.8 (b), is not defined, and consequently, cycles of crosscutting aspects are not allowed. This applies also for circular join relationships between crosscutting environment objects, as shown in the malformed model of Fig. 7.8 (c). (ii) For the same reason, no re exive join relationships are allowed.

The corresponding constraint that ensures non-circular join relationships is strictly enforced and specified formally in Definition C_{14} in Section F.4 of the appendix.

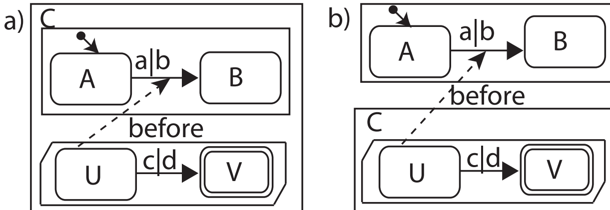


Figure 7.9: A join relationship may not cross the border of the aspect's parent. Model (a) is well-formed as the source aspect and the target of the aspect are located in the same component. Model (b) is malformed as the source aspect has a parent that is different from the parent of the target module.

Border Crossing of a Join Relationship

A join relationship may not cross the border of the aspect's parent. Hence, a target of a join relationship must be a sibling of the aspect or a child of a sibling. Therefore, the situation illustrated in Fig. 7.9 (a) is allowed, whereas the model in Fig. 7.9 (b) is malformed. This constraint is strictly enforced and specified formally in Definition C_{15} in Section F.4 of the appendix.

Priority within the Range of 1–10

According to the grammar production rule in Table 7.5, a priority for a join relationship can be literally any integer. However, the specification above stipulates that the priority must be between 1 and 10. This must be ensured as a constraint. This constraint is leniently enforced before the weaving of the model. It is specified formally in Definition C_{16} in Section F.4 of the appendix.

Well-Formed Context Mapping

The context mapping of a join relationship may contain arbitrary expressions. For a meaningful model, the element names used in the context map must be defined in the corresponding target, which can be solved by using symbol tables (cf. also Section 7.8.2). Furthermore, a context mapping may only be applied to a variable, such as an attribute or a message argument. However, a detailed formal definition of the constraints describing the well-formedness of context-maps is not at the focus of this work.

7.7 Crosscutting Environment Objects

An environment object can be seen as a *role* which is linked by an association with a specific *scenario* of the system. The scenario specifies in which order the environment object sends stimuli to the system and receives the generated responses.

Furthermore, a scenario can be split into a set of logically cohesive sub-scenarios. For example, the scenario *borrowing books* of a library system can consist of the sub use cases *authenti-*

Table 7.6: EBNF grammar rules of the join relationships of crosscutting environment objects.

Production Name	Production Rule
EnvironmentObjectDefinition ::=	("**`partial`**")? "**`environment`**" "**`object`**" EnvironmentObjectName UniqueModelElementIdentifier (Cardinality)? (EnvironmentObjectConnections)? "**`end`**" "**`environment`**" "**`object`**" EnvironmentObjectName
EnvironmentObjectConnections ::=	"**`connections`**" (AssociationDefinition | AssociationRoleDefinition | JoinRelationshipDefinition)* "**`end`**" "**`connections`**"

cate, *choose books*, *checkout books*. Correspondingly, the role represented by the environment object can be split into *sub-roles*. In the library system example, the user who borrows the books may have different sub-roles responsible for the *authentication*, the *selection*, and the *reception of books*.

An environment object may have a *crosscutting sub-role* because it has an association to a crosscutting scenariochart. When describing such a situation with aspects, the crosscutting sub-role is separated from the actual environment object. It is represented by a *crosscutting environment object*, i.e., an environment object which is connected by a join relationship to one or more other environment objects. The join relationship denotes that the crosscutting sub-role actually belongs to another environment object. When weaving, the crosscutting environment object is woven with its targets, which is explained in detail in Section 9.2.6.

Note that even though external components are similar to environment objects (cf. Section 5.2.5), they *are not* in the context but a part of the system. Therefore, they do not actually represent a role and cannot crosscut other environment objects or external components.

An example is given in the library system of Fig. 7.2. *AuthenticatingUser* is a crosscutting sub-role which results from the proper separation of concerns. It is connected with the crosscutting scenariochart describing the interaction with the password retrieval mechanism. This crosscutting sub-role is actually part of other environment objects, such as the *LibraryUser* but due to the separation of concerns, it is separated in the aspect-oriented model.

The join relationships used with crosscutting environment objects may be attributed with an ordering keyword, a priority and a context map. However, these attributes do not have a meaning for the weaving of the model.

7.7.1 Grammar Production Rules

The EBNF rules for describing the join relationships between environment objects is given in Table 7.6. The rule *EnvironmentObjectConnections* defines all connections, including join relationships, which can be outgoing from the environment object.

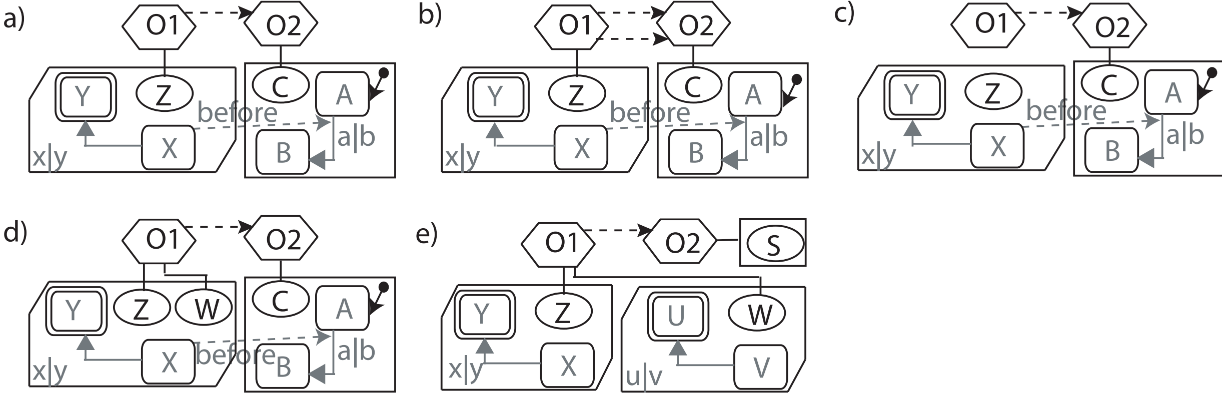


Figure 7.10: Illustration of the constraints for crosscutting environment objects. The situation in Fig. (a) is allowed whereas the models in (b), (c) and (d) are malformed.

7.7.2 Language Constraints

Two constraints concerning crosscutting environment objects and join relationships are defined in Section 7.6.2: there may be no cycles of join relationships between environment objects, and a join relationship outgoing from an environment object may only connect to another environment object. Apart from these two, the following constraints on environment objects are also required to ensure well-formed ADORA models.

Only One Join Relationship between Two Environment Objects

More than one join relationships pointing from an environment object *A* to the environment *B* is not meaningful, and therefore, maximally one is allowed. Fig. 7.10 (a) shows a situation which satisfies this constraint, whereas the model in (b) is malformed. This constraint is strictly enforced and specified more formally by Definition C_{17} in Section F.5 of the appendix.

Crosscutting Environment Objects Must Be Connected To a Scenariochart

A crosscutting environment object must be connected with a *crosscutting scenariochart*. Fig. 7.10 (c) shows a malformed situation, as there is no association between *Z* and *O1*. The corresponding constraint is leniently enforced before the weaving of a model. It is formally described in C_{18} in Section F.5 of the appendix..

No Association of a Crosscutting Environment Object to More than One Aspect

A crosscutting environment object may be connected to more than one crosscutting scenariochart. However, these crosscutting scenariocharts must be contained within the same aspect. This is due to the fact that the role represented by a crosscutting environment object does not usually belong to more than one aspect.

An example of a correct model is given by Fig. 7.10 (d). The crosscutting environment object is connected to two scenarios which are located in the same aspect. In contrast, (e) shows a situation which is malformed, because the scenarios are located in different aspects. This constraint is leniently checked before weaving a model. It is formally described in Definition C_{19} in Section F.5 of the appendix.

7.8 Crosscutting Functional Speci cation

Components are modules in conventional software systems. They may contain a functional specification which allow the definition of properties such as attributes, data type definitions, operations, etc. (cf. Section 5.2.6). Similar to components, aspects are modules encapsulating crosscutting concerns. For example, they contain crosscutting behavior and scenario descriptions which may refer to a set of properties that are common to the whole aspect.

For instance, crosscutting statecharts or behavior chunks are not able to store information, except the one represented by a state. However, more complex information cannot be represented by a state, as this would result in too complex and unmanageable models. Consequently, to retrieve complex data later when it is needed, it is necessary to save more of it in distinct storages, i.e., attributes, of the aspect module. Furthermore, it is convenient to encapsulate a set of actions in operations, which facilitates an easy execution in a behavior chunk or a parallel behavior description. Operations reduce the complexity of behavioral descriptions and lead to more comprehensible models.

Generally speaking, all elements of a component's functional specification (cf. Section 5.2.6) also make sense for aspects. Thus, the elements that may be contained in the functional specification of an aspect are:

- **Export Declaration:** The export declaration defines the elements which are visible outside of the aspect.
- **Import Declaration:** The import declaration defines the elements which are imported from other functional specifications.
- **Invariants:** The invariant section is part of the aspect's contract and consists of one or more expressions describing a logic predicate which must hold before and after the execution of a crosscutting operation.
- **Standardized Properties:** Standardized properties allow the definition of user-definable structures for stating goals, constraints, configuration information, notes, etc.
- **Data Type Declarations:** A data type declaration defines a new data type which can be used to encapsulate or structure data. Thus, they help to simplify the handling of data values.
- **Attribute Declarations:** Attributes are storages which can be used by the aspect to save and retrieve data values.

- **Operation De nition:** Operations define sequences of actions (send events or assignments). Furthermore, they might define contract elements, such as preconditions and postconditions.

Examples for these elements and a deeper discussion of them is given Section 5.2.6 where the functional specification for components is discussed in detail. An example of a functional specification in an aspect can be found in lines 74–81 of Listing 7.1.

7.8.1 Grammar Production Rules

The following discussion sketches the syntactical structure of the functional specification elements brie y. In Table 7.7, the production rule *FunctionalSpeci cation* describes the functional specification of both aspects and components. It references rules for all elements listed above: imported and exported elements, invariants, data type declarations, attribute and operation definitions. The import and export section are declared at the beginning of the specification, the other elements can follow in an arbitrary order.

Import and Export Declaration

An aspect may specify imported and exported elements in its functional specification. Imports and exports are specified by the rules *Requires* and *Provides* in Table 7.7. As they are eventually woven into the target elements, they have to be seen with respect to the namespace of the aspect's target elements. For example, suppose an aspect *A* crosscuts the component $C_0, C_1, \ldots, C_n$. The elements provided or required by *A* are seen as if they were provided or required directly by the component C_i, where $0 \leq i \leq n$.

Provided elements are listed by their unqualified name in the provides section of the aspect. In contrast, required elements are listed in the requires section with a qualified name, i.e., a name containing an access path to the element (cf. Section 6.1.5). The access path of provided elements is either relative or absolute. The aspects have an impact on various components that may be embedded at different levels of the decomposition hierarchy. Therefore, a relative qualified name is rather inadequate. As a consequence, imported elements are better specified by their fully qualified name.

Nonetheless, importing and exporting elements may enormously increase the coupling between aspects and other modules. Hence, the use of this language feature is discouraged.

Invariants

Aspects are *modules* of crosscutting concerns, whereas components are *modules* of conventional concerns (cf. Chapter 3). Both types of modules share a lot of characteristics. Amongst others, both of them have a specified behavior and a well defined interface. Consequently, the concept of the axiomatic specification for components can be applied to aspects, too. It is possible to define invariants, i.e., predicates which have to be true before and after an operation of an aspect is executed. In Table 7.7, the production rule *Invariants* describes how an invariant has to be composed.

Table 7.7: EBNF grammar rules for the functional specification of an aspect.

<table>
<tr><th>Production Name</th><th>Production Rule</th></tr>
<tr><td>FunctionalSpecification ::=</td><td>(“functional” “specification” (Provides)? (Requires)? (Invariants | DataTypeDeclarations | AttributeDefinitions | Property | OperationDefinition)* “end” “functional” “specification”)?</td></tr>
<tr><td>Provides ::=</td><td>“provides” Identifier (“,” Identifier)* “;”</td></tr>
<tr><td>Requires ::=</td><td>“requires” QualifiedIdentifier (“,” QualifiedIdentifier)* “;”</td></tr>
<tr><td>Invariants ::=</td><td>“inv” (Expression “;”)+</td></tr>
<tr><td>DataTypeDeclarations ::=</td><td>“data” “type” (DataTypeDeclaration)+</td></tr>
<tr><td>DataTypeDeclaration ::=</td><td>DataTypeName “:” DataTypeDefinition “;”</td></tr>
<tr><td>DataTypeDefinition ::=</td><td>(PrimitiveType | EnumerationTypeDefinition | DesignedTypeDefinition)</td></tr>
<tr><td>AttributeDefinitions ::=</td><td>“attributes” (VariableDefinition “;”)+</td></tr>
<tr><td>VariableDefinition ::=</td><td>VariableNameList “:” DataTypeReference (“=” ExtendedTypeLiterals)?</td></tr>
<tr><td>VariableName ::=</td><td>Identifier</td></tr>
<tr><td>VariableNameList ::=</td><td>VariableName (“,” VariableName)*</td></tr>
<tr><td>Property ::=</td><td>“property” Identifier (<INTEGER_LITERAL>)? TypeLiteral “;”</td></tr>
<tr><td>OperationDefinition ::=</td><td>(AsyncOperationSignature | SyncOperationSignature) (“is” “set” “operation”)? (LocalVariableDefinitions)? (PreConditions)? (PostConditions)? (Statements)? “end” “operation” OperationName</td></tr>
<tr><td>OperationName ::=</td><td>Identifier</td></tr>
<tr><td>AsyncOperationSignature ::=</td><td>“operation” OperationName “(” (Parameters)? “)”</td></tr>
<tr><td>SyncOperationSignature ::=</td><td>“syncoperation” OperationName “(” (SyncParameterList)? “)” (“:” DataTypeReference)?</td></tr>
<tr><td>LocalVariableDefinitions ::=</td><td>“var” (VariableDefinition “;”)+</td></tr>
<tr><td>PreConditions ::=</td><td>“pre” (Expression “;”)+</td></tr>
<tr><td>PostConditions ::=</td><td>“post” (Expression “;”)+</td></tr>
<tr><td>Statements ::=</td><td>“statements” Statement (“;” Statement)*</td></tr>
<tr><td>Statement ::=</td><td>(GuardPart)? (Assignment | MessageSend | MetaFunction)</td></tr>
<tr><td>Assignment ::=</td><td>ExtendedQualifiedIdentifier “=” Expression</td></tr>
<tr><td>MetaFunction ::=</td><td>(StructureMetaFunction | QueryMetaFunction | ObjectSetMetaFunction)</td></tr>
</table>

Data Type Declarations

Like aspects, these can contain behavioral descriptions. As for the components, the use of user-defined data types in aspects is helpful to simplify the events of the behavior (cf. Section 5.2.6). In Table 7.7, the syntax for user defined data types is given by the rules *DataTypeDeclarations*, *DataTypeDeclaration* and *DataTypeDe nition*. The rule *DataTypeDe nition* refers to the rules *PrimitiveType*, *EnumerationTypeDe nition*, and *DesignedTypeDe nition*.

Attribute De nitions

The idea of attributes can be transferred to aspects, thus an aspect can contain attributes defining a storage for complex data. In Table 7.7, the production rules *AttributeDe nition* and *VariableDefinition* describe what an attribute definition looks like.

Operation De nitions

Aspects benefit from the specification of operations because they help to simplify behavior descriptions. Table 7.7 shows the syntactical structure of operations. The production rule *OperationDe nition* specifies the syntax of an operation in ADORA. The *operation signature* is specified by the rule *OperationDe nition*. Synchronous operations may have one or more output values beside input values. In contrast, asynchronous operations may only get input values. This fact is re ected by the two production rules *AsyncOperationSignature* and *SyncOperationSignature*.

An operation can be working on an object set rather than on a single element. In this case, it is an *object set operation* which may call object set meta functions[12], such as *create* and *dispose* that manipulate the object set. However, this kind of operation is rather meaningless for aspects, as their use implies that all target modules of the aspect must be object sets.

Furthermore, an operation may define *local variables*, used as temporary storage for calculations. In Table 7.7, the production rules *LocalVariableDe nitions* and *VariableDe nition* describe what such a local variable definition looks like. *Pre- and postconditions* are specified by the rules *PreConditions* and *PostConditions*. The production rules *Statements* and *Statement* define how the block of statements is composed. A statement can consist of a variable assignment, the sending of a message or the call of a meta-function.

7.8.2 Language Constraints

There are several language constraints belonging to the syntax rules of the functional specification. Most of them involve a symbol table, as it is used in [Seyb06a, Section 6.1]. However, these constraints are not at the focus of this work, as they were the subject of previous work on the functional specification [Joos99, Glin02b, Xia04, Seyb06a].

[12] See the production rule *ObjectSetMetaFunction* in the full ADORA grammar in Appendix B.

7.9 Aspect Decomposition

The decomposition of an aspect is desirable for the following two reasons: first, for improving the cohesion of an aspect, which has an in uence on the understandability of the model, and second for the refinement of an aspect during the evolution of a model.

Improving cohesion. As discussed in Chapter 3, aspect-oriented modularization techniques decrease the coupling of the software artifacts at any stage of the software process. However, just introducing a new kind of module for crosscutting concerns is not enough because an aspect may comprise a wide spectrum of responsibilities that reduce its cohesion [Scha96, p. 138, ff.] and which in turn increases its complexity. As a consequence its understandability is hampered. Thus, it makes sense to decompose a complex aspect into a set of smaller modules, each with a higher cohesion, which fosters also the understandability of the model. The cohesion problem is mastered by introducing the following two decomposition mechanisms for aspects:

1. **Server Components:** A server component may provide a service for an aspect which is accessed over an association[13]. The same server component instance is shared amongst all target modules of a crosscutting concern.

2. **Embedded Components:** An embedded component is a part of an aspect. Consequently, each target module contains its own embedded component. The aspect and the embedded component communicate over the part-of relationship.

Server components differ also from embedded components by their number of woven instances. A server component has only one instance in the woven system, i.e., the crosscutting concern uses exactly one instance. In contrast, there is an instance of an embedded component *for each* module that is crosscut.

As a consequence, the use of a server component and an embedded component has a different semantics. A server component may be used if a service is used in collaboration with all locations impacted by a crosscutting concern. Thus, the server component provides a common state and common attributes to all crosscut modules. On the other hand, embedded components are used if each crosscut module is amended by a independent crosscutting part which has its own state and attributes. An embedded component is therefore independently used by each crosscut module.

The use of server components creates the need for synchronizing the behavior. This is due to the fact that a server component provides its service to several different crosscut targets in the woven system which concurrently access it. The behavior description in the server component must take this into account by synchronizing the behavior where needed, as otherwise the results may be unpredictable. In the cases where a synchronization of the concurrent behavior is needed, the ADORA synchronization mechanism (cf. Section 5.2.3 and [Meie09a]) can be used. A deeper discussion of this issue and its relation to the weaving of models can be found in Section 9.2.8.

[13] In [Meie06], a restriction on such associations is postulated. They may only be connected to an abstract object and *not* to an object set. However, in this work, this restriction is relaxed, thus it is also allowed to use object sets as server components.

Furthermore, note that embedded components of an aspect may not be connected by associations to other components that are outside the aspect, i.e., an association may not cross the border of the parent aspect. However, embedded components of the same aspect may communicate over associations with each other.

Figure 7.2 illustrates the differences between server and embedded components. The component *AuthenticationLog* is an embedded component which is part of the aspect *Authentication*. For each crosscut component there is an instance of the embedded component *AuthenticationLog* in the woven system, which logs the authentication events independently for each crosscut component. On the other hand, the component *Authorization* is a server component which provides a service to the aspect *Authentication*. In contrast to the embedded component *AuthenticationLog*, there is exactly *one* instance of the server component *Authorization* which has one common state for all locations impacted by the crosscutting concern.

Aspect re nement. Like any other model parts, aspects may be subject to evolution. They may evolve from an abstract level to a more concrete level during the software process. Aspect refinement is another kind of decomposition mechanism which supports and guides the evolution and which improves the understandability of the evolved model. It is mainly used for evolving and particularizing non-functional requirements [Meie07].

To express an aspect on different levels of evolution, an aspect may be refined, i.e., specified, by embedding more concrete and specialized aspects in it. For example, an aspect *B* embedded in the aspect *A* means that *B* is evolved from *A*. *B* satisfies parts of the more general requirements specified by *A*, and therefore, it is more concrete than *A*. The aspect *B* does not necessarily detail the whole aspect *A* but just a part of it. There may be more than one sub-aspects which particularize together the parent aspect. Furthermore, there may be an arbitrary number of aspect nesting levels, i.e., the sub-aspects may be decomposed and detailed further. The innermost aspects represent the most concrete ones.

An aspect comprising sub-aspects may not contain any other elements which are directly embedded. Thus, for example, scenario chunks or behavior charts are not allowed in a refined aspect. Figure 7.11 (a) illustrates an example of an aspect refinement as it might occur during the model evolution of the library system. The abstract *Security* aspect can be refined, for example, to the more detailed aspects *Logging* and *Authentication*. A more detailed discussion of the evolution of aspects in ADORA models can be found in Chapter 10.

7.9.1 Grammar Production Rules

There are several production rules in the ADORA grammar which are involved in the decomposition of aspects. Table 7.1 contains the rule *AspectParts* which defines the refinement of aspects by referring the rule *AspectDe nition*. Furthermore, the fact that server components can be connected by associations with aspects is specifically defined in several grammar rules throughout the ADORA grammar. The grammar rule *ComponentParts* in Table 7.2 and the grammar rule *Model* in Table 6.1 describe where server components (i.e., *ComponentDe nitions*) can be embedded. Moreover, the rule *AspectConnections* in Table 7.5 shows the grammar rules specifying

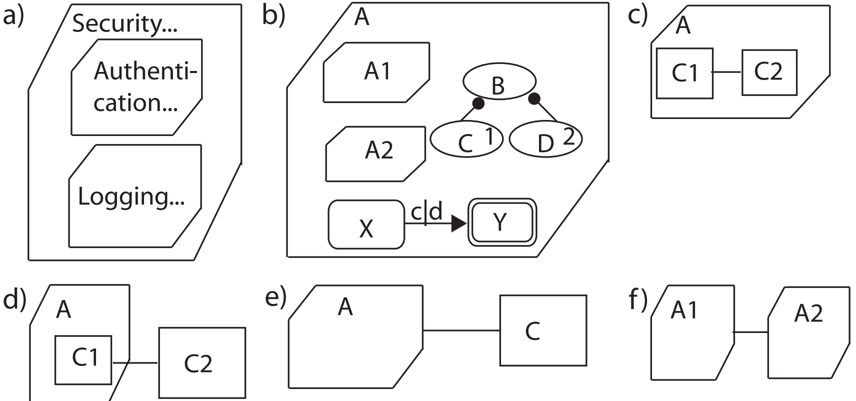


Figure 7.11: Illustration of the language constraints for the aspect refinement. Situations (a), (c) and (e) are well-formed, whereas the models in (b), (d) and (f) are ill-formed.

the associations (*AssociationDe nition*) or association roles (*AssociationRoleDe nition*) which are used to connect an aspect module and a server component.

7.9.2 Language Constraints

The syntax rules discussed above are not self-contained for describing a well-formed ADORA model. They have to be complemented by several language constraints which are given in the following.

Aspect-Re ned Aspect Modules May Contain only Aspect Modules

An aspect module which is refined by aspect modules may only comprise aspects as parts. Thus, besides embedded aspects, no Scenario nodes, no states, no exit points and no components are allowed.

Figure 7.11 (a) illustrates a refinement situation which satisfies this constraint. In contrast Fig. 7.11 (b) violates this constraint, as the aspect *A* contains the aspects *A1* and *A2* as well as a scenario and a behavior chunk. This constraint is strictly enforced and formally specified by the Definition C_{20} in Section F.6 of the appendix.

Associations Originating within an Aspect May not Cross the Border of the Aspect

An association whose source is a direct or indirect child of an aspect must have a target which is a direct or indirect child in the same aspect. Hence, an association which is connected to an element that is part of an aspect is not allowed to cross the border of the aspect.

Figure 7.11 (c) shows a well-formed situation for using an association within a decomposed aspect *A*: the components *C1* and *C2* are linked by an association. In contrast, Figure 7.11 (d) shows an malformed model, as the association between *C1* and *C2* crosses the border of

the aspect. This constraint is strictly enforced and formally expressed by the Definition C_{21} in Section F.6 of the appendix.

Components and Aspects Must Be Connectable

In the conventional ADORA language, only elements of a specific type can be connected by an association. This constraint must be overridden in order to take into account aspect modules which may also take part in an association. Thus, associations may either connect a component with a component, an aspect with a component, a scenario with an environment object, an environment object with an environment object, or a scenario with a component having its external property set. All other combinations are not allowed.

Figure 7.11 (e) shows a well-formed situation which has an association between an aspect *A* and a server component *C*. In contrast, the situation in (f) shows a malformed model, as associations between aspects are not allowed. This constraint is strictly enforced and formally defined in Constraint C_{22} in Section F.6 of the appendix.

Other Constraints

With regard to associations, there are several language constraints from the previous work [Joos99, Glin02b, Xia04, Seyb06a, Cram07], e.g. correctly referred role names, etc., which apply also to the associations between server components and aspects.

7.10 Summary and Discussion

This chapter has demonstrated how to extend the existing requirements specification modeling language ADORA with an aspect-oriented extension. It has been shown that an extension impacts all different facets of a model, such as the behavior, the structure or the environment of the system.

The approach presented aims at providing a better way to represent crosscutting concerns in requirements models by specifying the elements of a crosscutting concern in an *aspect module*. Consequently, the understandability can be improved and the elements of the crosscutting concern can be maintained more effectively and efficiently. On the other hand, the aspect-oriented modularization may introduce additional complexity, which decreases the understandability. Therefore the presented language extension is designed to allow switching between the aspect-oriented and the conventional view of the system, in order to provide a more convenient view of the modeled system. This will be discussed in Chapter 9.

The aspect-oriented and the conventional representation of a model are the means for reducing the problem-exogenous complexity and thus for overcoming the communication gap between the different stakeholders of a project.

Furthermore, the presented syntactical elements support a controlled evolution and a gradual completion and formalization of the aspect-oriented model, which is discussed in Chapter 10.

The aspect-oriented approach presented can be categorized according to the categories in Section 3.2.1. It is a *linguistic*, rather *asymmetric*[14], *general-purpose* approach which uses a static composition mechanism. The choice of a linguistic, general-purpose approach seems to be logical for describing software requirements. However, the reason for choosing an asymmetric approach needs to be commented on.[15]

There is work, such as [Ossh01, Harr02, More05a, More05b] and [Stan02, Sutt03, Sutt04] which argues that core and crosscutting concerns should be seen as symmetric. They regard crosscutting and non-crosscutting concerns as peers and propose to handle concerns uniformly, i.e., by the same modular descriptions. However, aspect-oriented descriptions inherently deal with asymmetric descriptions.

There are two reasons for this. First, the (unidirectional) crosscutting relationship is inherently asymmetric. Second, the description of overlapping concerns is also inherently asymmetric. It is purely a matter of definition which concern of a given set of overlapping concerns contains the overlapping, i.e., the crosscutting, parts. Thus, in the end, one of the modules representing the overlapping concerns must contain the overlapping part with the least redundancy as possible. Thus, there are always modules which are not self-contained parts and therefore asymmetric.

Describing asymmetric modules with symmetric constructs leads to gaps in the concern description which have to be filled by additional and artificially introduced elements that act as a makeshift to ensure that the crosscutting concern modules are self-contained.[16] These additional constructs may lead to a higher effort when handling crosscutting concerns and can worsen the understandability of a description. For this reason, the present work uses an asymmetric description.

[14]The visualized specification of the crosscutting relationship is separate from the aspect, which is rather typical for symmetric approaches. Nevertheless, comparing the structure of aspect modules with the structure of components indicates clearly an asymmetric approach.

[15]The reason for the use of an asymmetric approach is argued in Section 9.6.

[16]This is the case for the modules in the HyperJ approach, where each aspect is described by conventional Java classes. As a consequence, the Java classes must be made artificially self-contained by adding abstract methods (cf. Section 3.2.2).

Chapter 8

Visualization of Aspect-Oriented Model Elements

ADORA is a integrated visual modeling language (cf. Section 5.1.3), and therefore, the visual representation of models can become fairly large if all its elements are shown at the same time. To cope with the size of the model, it is possible to apply abstraction mechanisms to it. They allow the reduction of the size of the model's visual representation by hiding elements which are not at the focus of interest. Elements can be *vertically abstracted*, i.e., the child elements of a node can be zoomed out. The *horizontal abstraction mechanism* allows the elements that belong to a particular facet of the model to be hidden, and the *crosswise abstraction mechanism* allows toggling the visibility of arbitrary elements.

The changes performed in the view when applying one of these three abstraction mechanisms is described by a *view transition semantics* [Xia04]. The following discussion sketches the extension of the view transition semantics for the aspect-oriented language elements introduced in the previous chapter. However, the present approach does not focus on the visualization of models and, therefore, the following discussion does not deal with details. Furthermore, no *formal* description is given. An interested reader may read the corresponding parts in [Xia04].

The remainder of this chapter is structured as follows. Section 8.1 illustrates how the semantics of the three abstraction mechanisms is extended so as to be capable of handling the view transitions of aspect-oriented elements. In Section 8.2, two new views are introduced. They contain the aspect-oriented language elements presented in the previous chapter. Furthermore, it is also possible to weave an aspect-oriented model and to create the conventional view. However, this topic is discussed in the next chapter.

8.1 Applying Abstractions to Aspects

To support the ADORA abstraction mechanisms, it is necessary to define a view transition semantics. A view transition semantics specifies the effects on the representation of the aspect-oriented language elements when one of the three abstraction mechanisms is applied. For conventional models, the following two view transition rules apply:

- A hidden node *N* which is part of another node *A* is indicated by an ellipsis after the name of *A*. Bear in mind that the ellipsis is used for partially viewed models, as well as for intentionally incomplete models.

- There are connections which are allowed to cross the border of the parents of their constituting nodes. As soon as one of the constituting nodes is hidden, the relationship becomes implicit. For conventional ADORA language elements, this is the case for associations, for scenario connections, and for transitions. The information expressed by an association is valuable, as it describes a relationship between two components. Therefore, there arises the concept of a calculated abstract association (cf. Section 5.2.2). A calculated abstract association preserves this information as far as possible.[1] by visualizing the implicit relationship as a bold line between the parent of the hidden constituent and the other constituent.

Analogously to the conventional ADORA modeling elements, hidden elements in an aspect module or the hidden aspect module itself must be indicated in the visual representation of the model. Like associations, join relationships are allowed to cross the border of the parents of their constituent. As the information of the join relationship is valuable to the reader of the model, a concept similar to the calculated abstract association is used.

8.1.1 View Transition Semantics for Aspect Modules

The children of an aspect module can be hidden by any of the three abstraction mechanisms mentioned above. Hidden nodes of an aspect are indicated by a trailing ellipsis in the aspect's name. Figures 8.1 (a)–(e) illustrate several situations where one or more nodes of an aspect are hidden. Situation (a) shows an aspect with no nodes hidden. In (b), a crosswise abstraction is applied, where the node *X* of the behavior chunk is hidden. The ellipsis after the name of the aspect indicates that one or more elements are hidden, i.e., in this case, this is caused by *X*. In situation (c), the two sub-nodes of the scenario chunk are hidden, whereas in (d), the scenario view is hidden by a horizontal abstraction. Finally in (e), the content of the aspect is zoomed out (vertical abstraction).

Furthermore, an aspect itself may be hidden. In this case, the parent of the aspect indicates the hidden aspect by a trailing ellipsis.[2] Figure 8.2 illustrates this case. In (a) the initial situation is given, whereas in (b) the aspect module is hidden and the ellipsis in the name is shown instead.

The view transition semantics for the conventional ADORA language is defined formally by the formalism presented in [Xia04, Section 3.4]. Since the semantics of the view transition for the aspect-oriented elements behaves similarly to those for the conventional ADORA model elements, the view transition semantics rules can correspondingly be applied to the aspect-oriented elements of the language.

[1] However, there is no abstract scenario connection or abstract transition. The information about scenario connections and transitions is not perceptible from the model representation when one of the two constituents is hidden.

[2] An exception to this rule exists for the root node of a model. The root node does not indicate any of its hidden child nodes.

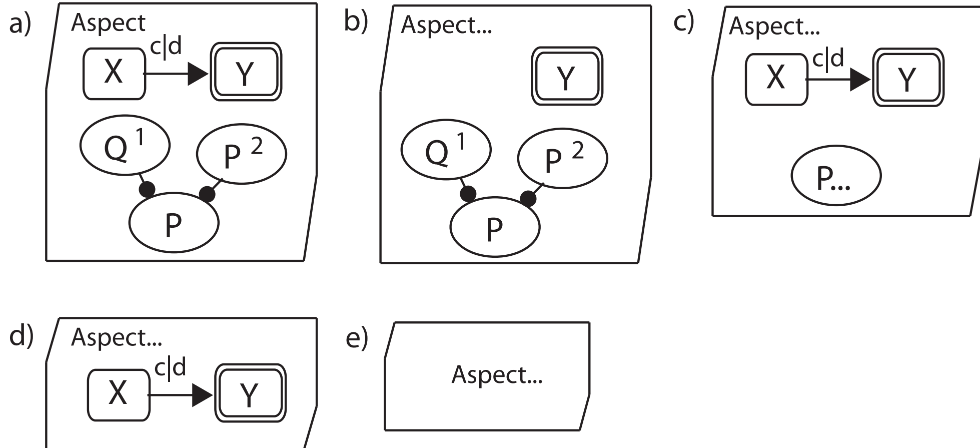


Figure 8.1: Several situations illustrating the hiding of a node in an aspect.

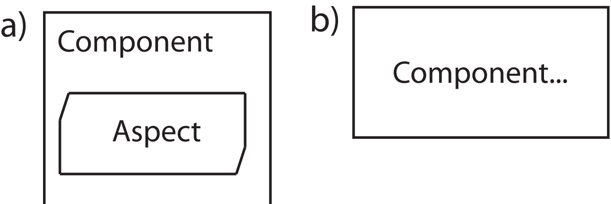


Figure 8.2: View transition when hiding an aspect module.

8.1.2 Join Relationships

When a constituting element of a join relationship, i.e., a scenario node, a state, or a transition, is hidden, the corresponding join relationship can no longer be shown, as it would then be half dangling. The model shown in Fig. 8.3 (a) is used to illustrate the problem. The constituents of the join relationship are the state *A* in the *Aspect* and the transition with the label $a \mid b$. Suppose that *A* is hidden by applying one of the abstraction mechanisms mentioned above. In this case, the join relationship cannot be shown any more.

Actually, this is the same problem as with associations. A join relationship crosses[3] the border of the parents of its constituents.[4] For example, when hiding the constituent *A* in Fig. 8.3 (a), the parent of *A*, i.e., the *Aspect*, is still visible and has an implicit relationship with the other constituent of the join relationship, i.e., the transition $a \mid b$. However, this relationship is no longer visible. Therefore it is meaningful to introduce a substitute to preserve this information. This substitute is called *(calculated) abstract join relationship* and is drawn between the parent of the hidden constituent *A* and the other constituent as a dashed *bold* arrow. The situation is

[3] In fact, a join relationship *must* cross the border of the aspect module, to be well-formed (cf. Section 7.6.2).
[4] In the example, the parents of the constituents are *Aspect* and *Component*.

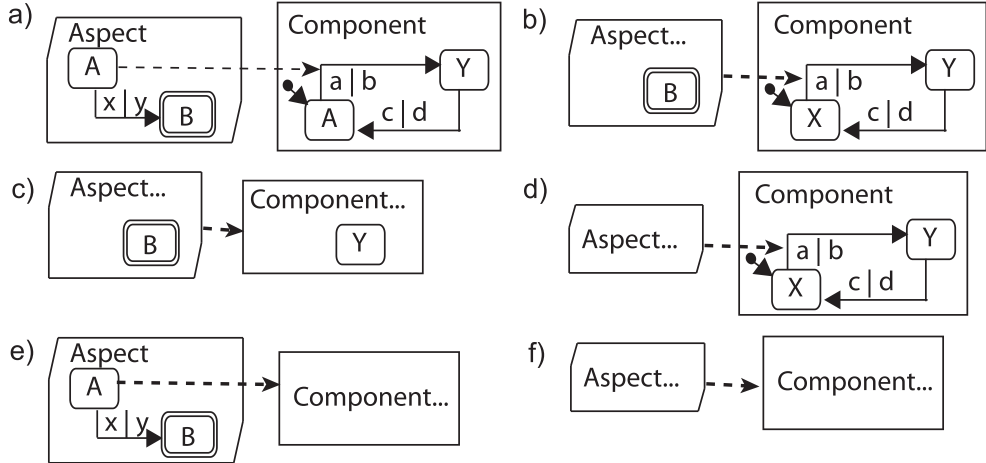


Figure 8.3: Examples of calculated abstract join relationships.

depicted in Fig. 8.3 (b), where the abstract join relationship is drawn between *Aspect* and the transition *a* | *b*.

The situations (c)–(e) may occur analogously. In (c), the transition *a* | *b*, is also hidden. In situation (d), the content of the aspect is zoomed out. A similar situation is given in (e) where the target component is zoomed out. Finally in (f), the content of both the aspect and the component is hidden.

The hiding of nodes may result in more than one abstract join relationship between an aspect and a target component. Suppose the situation in Fig. 8.4 (a), which shows an aspect that has two outgoing join relationships impacting a target *Component*. When zooming out *Component*, two abstract join relationships result, because the target constituent of both join relationships are hidden, which is shown in Fig. 8.4 (b). A similar case is shown in (c), where the content of the aspect is hidden. Since there may not be more than one abstract join relationship between two elements, zooming out the *Component* in situation (c) results only in one abstract join relationship as shown in Fig. 8.4 (d).

Furthermore, a calculated abstract join relationship does not have any properties, such as an ordering keyword or a context map.[5]

As mentioned above, calculated abstract join relationships behave similarly to calculated abstract associations (cf. Section 5.2.2). A formal description of the refinement calculus for associations can be found in [Xia04, Section 2.2 and Section 3.4]. This calculus can be adopted for join relationships. However, in contrast to associations, there are no interrelationships (cf. page 5.2.2) between abstract join relationships that are constituted by nodes on different levels in the decomposition hierarchy.

[5]There is only one case when an abstract join relationship, representing one or more hidden join relationships, might show properties: when all properties of all the represented (concrete) join relationships are equal. However, since this is a rare situation, it is neglected by the view transition rules.

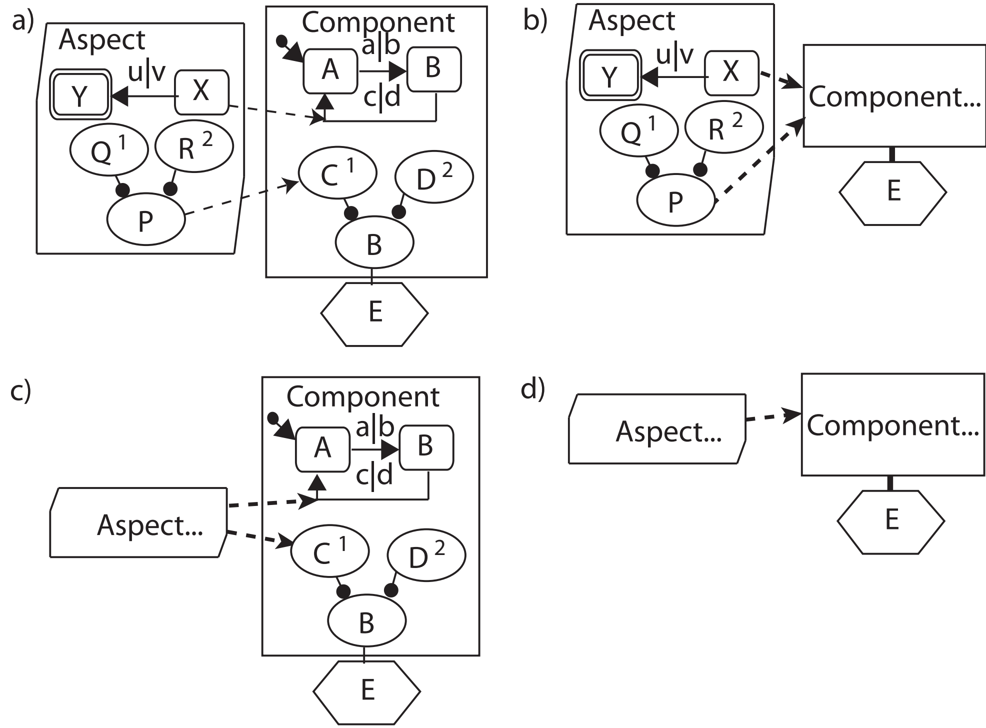


Figure 8.4: Further examples of calculated abstract join relationships.

8.2 Extending the View Concept

The language elements introduced in Chapter 7 are a major extension to the ADORA language. They introduce two new facets to the modeling language which result from the modularization of crosscutting concerns. These facets are visualized by the corresponding views:

- The *join relationship view* consists of all join relationships in an aspect-oriented model. A join relationship explicitly represents a crosscutting relationship. A large number of such relationships can make a model more complex. To prevent the modeler from suffering a cognitive overload, the join relationship view can be hidden with the horizontal abstraction mechanism.

- The *aspect view* contains all aspect modules of a model. The horizontal abstraction is employed to hide all aspect modules and their child elements. After hiding this view, the model shows only the core concerns.

  Since join relationships originate from aspects, hiding the aspect view also hides the join relationship view. Moreover, associations between aspects and server components are also hidden. The server components (cf. Section 7.9) are also hidden in the case that they are only connected to aspects *but not* to components of the core concerns. In this case, the server component is a pure decomposition of one or more aspects and, therefore, it belongs

to one or more crosscutting concerns and is also hidden. However, if server components are also connected by associations to components of the core concern, they are not hidden.

Figure 8.5 illustrates hiding the aspect and the join relationship view. In (a) the initial situation is shown, Fig. (b) shows the model of Fig. (a) with a hidden join relationship view. In (c), the aspect view, and, consequently, the join relationship view are hidden.

Note that the existing views can be used independently of the aspect view. For example, hiding the behavioral view and subsequently hiding and showing again the aspect view still does not show the behavioral view.

Furthermore, the conventional views defined by the non-aspect-oriented ADORA approach can also be applied to aspect elements. An aspect can contain behavioral elements, structural elements, and elements of the user view. They can be hidden and displayed by applying the corresponding horizontal abstraction mechanism. Figure 8.5 demonstrates the application of the horizontal abstraction mechanism for the conventional views in aspect modules. In (d) the user view is hidden, whereas in (e) the behavioral view is not shown. The hiding of structural view elements in an aspect module is not illustrated here, but works analogously.

8.3 Discussion

The aspect-oriented elements of the present approach fit well into the visualization of ADORA. However, it is the view concepts especially that can be extended in order to provide a more fine-grained abstraction mechanism for the filtering of the aspect-oriented elements. These view concepts only work on the set of aspect modules and join relationships. That means that the aspect modules and/or the join relationships can be hidden altogether but not particularly for one specific crosscutting concern.[6] Nevertheless, it would make sense to abstract just one or more particular concerns and not only all crosscutting concerns together.

An extension of the view concept may allow an assignment of each of the aspect modules and components to one or more particular concern. The aspects usually belong to one or more *crosscutting concerns*, the components either to one or more *crosscutting or core concern*.[7] The abstraction mechanisms can be applied to one or more specific concerns, allowing the concealment or display of the modules belonging to a specific concern.

Using this type of mechanism allows the independent visualization of particular concerns and therefore results in similar advantages to those of having a multidimensional concern space, such as the one used in CORE [More05a, More05b] (cf. Section 3.2.5). However, this extension may become a focus of interest for future research on the present approach.

[6] Note that a crosscutting concern may consist of different aspect modules and (server) components.

[7] Note that there may be more than one core concern.

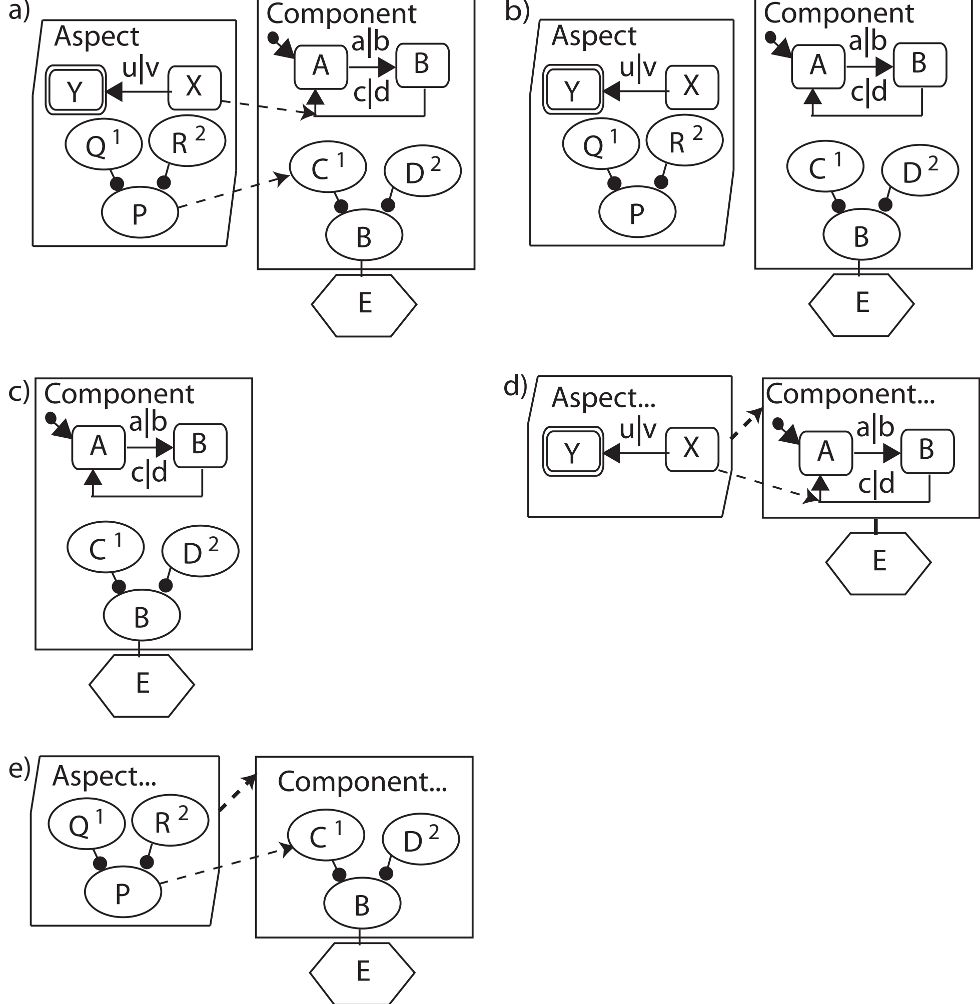


Figure 8.5: Illustration of the effects when hiding the behavioral view, the user view, the aspect view, and the join relationship view. Fig. (a) shows all views of the model, in (b) the join relationship view, and in (c) the aspect view is hidden. In (d), the user view is hidden and in (e) the behavioral view.

Chapter 9

Composing Aspect-Oriented ADORA Models

Sometimes, it is necessary to transform an aspect-oriented to a conventional ADORA model. The process of such a transformation is called weaving. There are mainly two purposes for the weaving of an aspect-oriented ADORA model:

i. In certain situations, an aspect-oriented ADORA model introduces additional complexity compared with conventional models (cf. Section 3.1.7). Therefore, it is desirable to switch between the aspect-oriented and the conventional representation of a model.

ii. Some of the aspect-oriented language elements do not have an execution semantics. Thus, they cannot be executed directly, e.g., in a simulation. A weaving transformation has to be performed first, which results in a conventional model.

Purpose (i) has already been discussed extensively in Section 3.1.7. The modularization of crosscutting concerns and their separated visualization helps in understanding crosscutting artifacts better, because they are disentangled in an additional modularization dimension. Consequently, changes to crosscutting concerns are easier to accomplish because their elements are not scattered across the whole model. However, depending on the situation, aspect-oriented models can be more difficult to understand than their conventional counterparts [Meie05]. This is due to the fact that the additional modularization dimension introduces a new form of complexity.

Purpose (ii) emerges from the design of the aspect-oriented ADORA language. As sketched in Section 7.2, some aspect-oriented ADORA elements have no execution semantics. For example, aspect containers are just some sort of capsules for the crosscutting elements but have no meaning at runtime. Moreover, behavior and scenario chunks are not self-contained and can only be executed meaningfully in the context of the crosscut elements. In contrast, each conventional ADORA construct has an execution semantics. As a consequence, an aspect-oriented ADORA model must be transformed first to a conventional model before it can be executed, e.g., by a simulation (cf. [Seyb06a]).

In order to gain the advantages from a weaving transformation as stated for Purpose i, the weaving semantics has to satisfy given premises:

a. The meaning[1] of a model as intended by its creator has to be preserved when a weaving transformation is executed.

b. The number of changes in the model performed during the weaving has to be minimized.

c. The number of changes in the resulting visual representation of the woven model must preserve the secondary notation as far as possible.

The need for satisfying Premise (a) is obvious. Premise (b) is desirable, because the more changes are performed by a model transformation, the more difficult is it for the reader of the model to relate the aspect-oriented elements with the conventional counterparts in the woven model. If Premise (b) is ignored by the transformation rules, the resulting advantages for intelligibility may get lost.

Finally, Premise (c) demands that the visual representation of the mental map, known as the *secondary notation* [Petr95] of a model (cf. Section 5.1.4), should be altered as little as possible when performing an automatic change in the layout of the model. When transforming or weaving a model, changes may impact the visual representation of the model. Ignoring Premise (c) in this case leads to difficulties for the reader of the model in finding his bearings and therefore in comprehending the model after the transformation.

A weaving semantics consisting of a set of transformation rules that follow the above premises is needed for performing the actual set of transformation steps. This chapter will elaborate on the corresponding weaving semantics rules for all model elements defined in Chapter 7. The chapter also discusses related problems, such as the changes in the visualization. The content of the chapter is based on [Meie06, Meie07], where the corresponding ideas are roughly outlined.

The remainder of the chapter deals with the weaving semantics. In Section 9.1, an overview of the actual weaving transformation process is given. As it works fundamentally differently depending on whether it deals with partial or non-partial model elements, the discussion of the weaving semantics is split into two separate parts. In Section 9.2, the transformation semantics for non-partial aspect-oriented elements is elaborated, whereas Section 9.3 presents the weaving semantics for partial elements. Both chapters discuss the weaving semantics only informally. Section 9.4 sketches how the weaving semantics is formally described. The actual formal description of the whole weaving semantics is given in Appendix G. Furthermore, Section 9.5 sketches brie y how the layout information is handled by the weaving process. Finally, in Section 9.6 the content of this chapter is summarized and discussed.

9.1 Weaving Process Overview

The weaving semantics describes the transformation from an aspect-oriented to a conventional ADORA model. An aspect-oriented model is described by the set of grammar rules P defined

[1]The meaning of a model is defined by the execution semantics. Two different models with the same initial state have the same execution semantics if they respond to the same sequence of system-external stimuli with the same output.

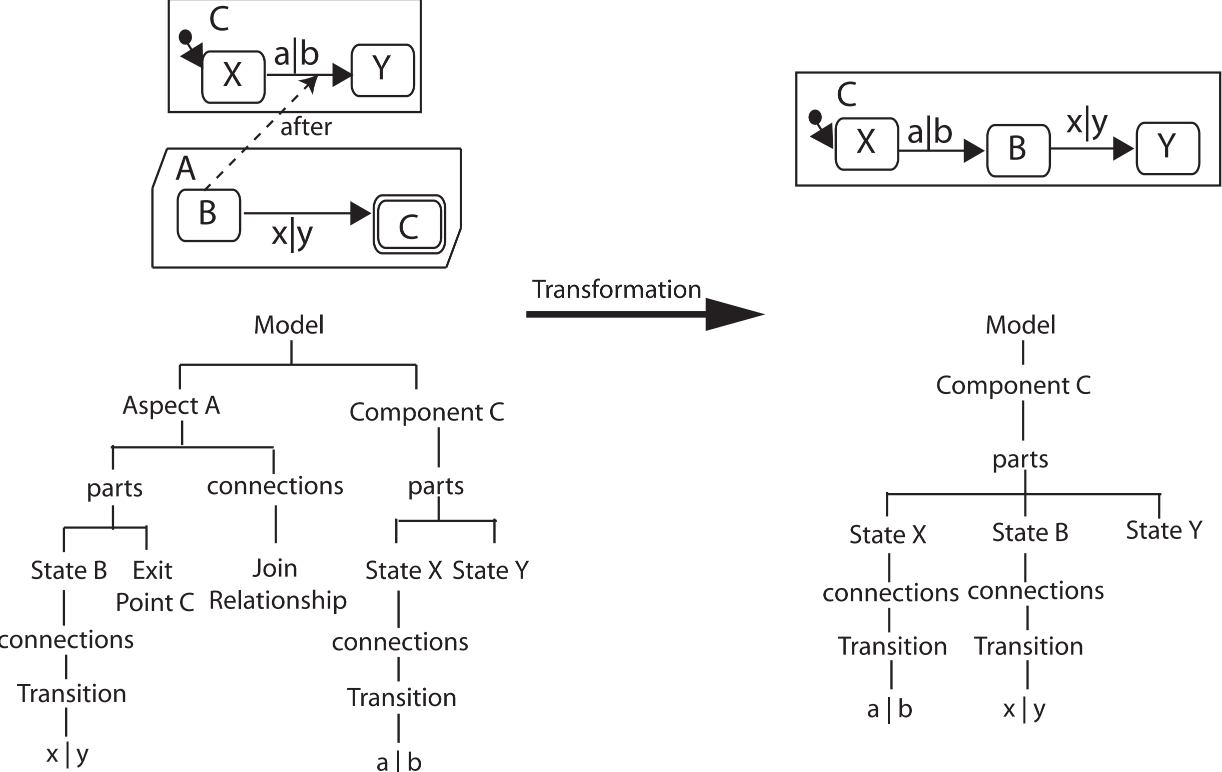


Figure 9.1: Illustration of the effects on a model/syntax tree when executing a weaving operation. On the left hand side, the visual representation of the aspect-oriented model and the corresponding (abstract) syntax tree are given. On the right hand side the woven model and corresponding syntax tree are shown.

in Appendix B. A conventional ADORA model is specified by a subset of rules $P_s \subset P$, where $P \setminus P_s$ contains the syntax rules which describe aspect-oriented elements.

A weaving transformation is the syntactical rewriting of a model which adheres to P to a model that can be described with the rules given by P_s. The syntactical transformation leads to a restructuring of the corresponding model syntax tree. Figure 9.1 illustrates such a rewriting. On the left hand side, the aspect-oriented model and the equivalent abstract syntax tree are given.[2] On the right hand side, the woven model and the corresponding syntax tree are shown.

The weaving process consists of a sequence of executed operations. The executed operations can be roughly divided into two distinct phases which are not necessarily executed in a sequential order:

1. **Preparation:** The model is checked and analyzed. This can be subdivided into the following sub-activities:

[2]Note that for the sake of simplicity, an abstract syntax tree is given here. However, the remainder of this work uses concrete syntax trees.

(a) Checking of constraints.

(b) Determining the order for the weaving of the join relationships.

(c) Determining the end target modules.

2. **Transformation:** The actual transformation operations gradually transform the aspect-oriented model into the conventional one. A transformation operation may be one of the following sub-activities:

   (a) Weaving of join relationships.

   (b) Weaving of so-called single instance elements: these are crosscutting statecharts, crosscutting scenariocharts, embedded components, crosscutting environment objects, functional specifications, and server components. They are called *single instance elements* because there is only one copy of them woven into a target module.

   (c) Removal of aspect modules in a post processing step.

9.1.1 Weaving Preparation

Checking constraints.

Before any weaving transformation can be performed, the leniently enforced language constraints for the aspect-oriented elements defined in Chapter 7 must be satisfied by the model.[3] These constraints must be satisfied as otherwise the resulting model is meaningless, or the transformation process may even break down.

Weaving order of join relationships.

The order in which the join relationships have to be woven must be determined. An aspect-oriented ADORA model may consists of a complex network of join relationships. The join relationships may either connect two elements, such as behavior chunks with transitions, scenario nodes with scenario nodes, or any other element specified in Section 7.6. A path through the network of join relationships starts with an aspect *A* and ends with an end target module *M*. *A* is said to transitively crosscut *C* if there is a path which consists of more than one join relationship.[4] The join relationships in such a network must be woven in a specific order, as otherwise the weaving process may result in a model with an unintended meaning.

The model in Fig. 9.2 is used to illustrate the problem. It shows the aspects *A1*, *A2*, *A3*, and the components *C1* and *C2*, and a network of join relationships that connects them. For purposes of referring to the join relationships in the following discussion, they are enumerated by Greek characters. The aspects *A3* and *A1* directly impact the component *C1* over the join relationships α and γ, respectively. *C1* is also transitively crosscut by *A1* and *A2*. Transitively crosscutting

[3]Remember that there are also strictly enforced constraints. However, they are already satisfied during the modeling (cf. Section 6.3).

[4]Remember that a network of join relationships may not contain any cycles (cf. Section 7.6.2).

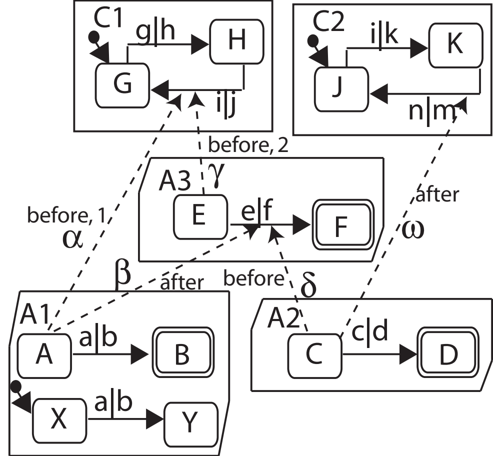


Figure 9.2: Illustration of a complex network of join relationships.

aspects in uence the weaving order of the join relationships: the weaving of γ depends on the β and δ, i.e., β and δ have to be woven first. Apart from the transitive crosscutting of aspects, there are other properties of a aspect-oriented ADORA model which in uence the weaving order of join relationships.

The ordering property determines the weaving order of join relationships impacting the same target. It is defined by one of the keywords *before*, *instead*, or *after* (cf. Section 7.4 and 7.5). For example, the order of β and δ is given by (δ,β). This is due to the fact that the chunk which is the origin of δ must be executed before the chunk which is the origin of β. Furthermore, the ordering of both chunks is with respect to the action f of the crosscut transition, i.e., the chunk δ is woven before f and β after f.

Another property which in uences the weaving order of the join relationships is the priority. It is used to determine which one of two *competing* join relationships is woven first. Two join relationships are competing, if they impact the same target with the same ordering. The higher priority indicates the precedence of the corresponding join relationship. If the two competing join relationships have the same ordering, one of them is non-deterministically chosen to be processed first. For example, α and γ in Fig. 9.2 impact the same transition and have the same ordering. However, transition γ is woven first, as it has the higher priority. Thus, the eventual weaving order that results in a correctly woven model is (δ, β, γ, α, ω).[5]

Topological sort. The weaving order is computed by executing a *topological sort* on the model's join relationships. The sort algorithm takes the *transitive weaving order*, the *ordering* with respect to a target, and the *priority* of the join relationships into account. However, not all join relationships are considered by the sorting process. Join relationships between crosscutting environment objects are handled differently because they are used to denote crosscutting roles

[5]Nevertheless, ω has no dependencies to the other join relationships, and therefore, it can be woven at any point in time in the weaving process.

rather than to show the impact of behavior/scenario chunks. Therefore, the join relationships between crosscutting environment objects are handled differently, which is discussed in detail in Section 7.7.

The topological sort algorithm first sorts the join relationships according to their target. Thus, the join relationships with the same target belong to the same *target group*. A target can either be a scenario node, a transition or, if the join relationship is partial, any of the elements discussed in Section 7.6.2, except an environment object.

In turn, the target groups are subgrouped according to their ordering property. Three subgroups result from this step: one containing all *before*, one containing all *instead*, and another one containing all *after* join relationships for the same target. In the next step, the join relationships of each ordering subgroup are sorted in a descending order by their priority.

In the final step of the topological sort, the list of target groups, is sorted according to the *predecessor relationship*, which is needed in order to get the correct weaving order of transitively crosscutting aspects. There are two cases which define that a join relationship j_p is the predecessor of j_i:

i. the target element t_p of j_p is contained in the same aspect A as the source element of j_i, or

ii. the join relationship j_P is connected to the aspect module A in which t_p is contained.

Note that case (ii) can only occur if j_p is partial, because partial join relationships can also be connected directly to aspect modules. Figure 9.2 illustrates the predecessor relationship, for example γ has the predecessors β and δ. The topological sort algorithm checks each group of targets g_i as to whether it contains a predecessor with respect to a join relationship contained in another group of targets g_k. If g_i contains a predecessor it is inserted before g_k in the resulting list. A formal description of the topological sort algorithm as pseudo-code can be found in Section E.6 of the Appendix.

Determining the End Target Modules of an Aspect

Apart from the weaving order of the join relationships in a model, the *end target modules* of an aspect need to be determined. This is necessary in order to weave the aspect's single instance elements, such as crosscutting statecharts. An aspect may have multiple end target modules. There may be multiple join relationship paths from an aspect *A* to an end target module *C*. However, a single instance element of *A* is just woven once into *C*.

An end target module can either be a component or an aspect. The latter is the case if a join relationship path ends in an aspect, but no join relationships originate from it. As all aspect modules are removed at the end of the weaving process, end target aspects are irrelevant and, therefore, aspects are not seen as end target modules in the following.

The model of Fig. 9.2 exemplifies these concepts. In total, there are four different paths of join relationships, namely (α), (β, γ), (δ, γ), and (ω). Consequently, the aspect *A2* has two end target modules, namely *C1* and *C2*, the former is crosscut transitively over the aspect *A3*. *C1* is the only end target module of the aspect *A1*. Even though there are the two paths (α) and (β, δ)

from aspect *A1* to the end target module *C1*, the crosscutting statechart contained in *A1* is woven only once into *C1*.

The end target modules of a model can be determined by the following procedure. A given aspect denotes the beginning of a join relationship path. The join relationships which are outgoing from it or from any of its children are determined. A join relationship belongs to a join relationship path if it connects (i) either an entry point of a behavior chunk with a transition, or (ii) a scenario chunk root node with a scenario node. It does not matter if the join relationship is partial or not. However, partial join relationships which do not fulfill either condition (i) or (ii), e.g., when connecting an aspect with a component, do not belong to a join relationship path, as these kinds of join relationships are handled differently (cf. Section 7.6).

For finding the end target modules, each found join relationship j is followed. If the target of j is part of an aspect, all outgoing join relationships are determined as described above. Each found join relationship which fulfills condition (i) or (ii) builds a new join relation path in combination with the path to which the incoming join relationship belongs, and so on. If no outgoing join relationships are found, the target of j is an end target module and is stored in a result list. Even though the given start aspect may have various join relationship paths to the same end target module, the end target module is only added once to the result list.

A formal description of the algorithm which determines the end target modules of an aspect can be found in Section E.5 of the Appendix.

9.1.2 Weaving Transformation

The actual weaving transformation consists of n weaving steps. Each step has an intermediate model as outcome, i.e., $n-1$ intermediate models are created during the transformation process. The model after the kth step is denoted in the following by t_k. Furthermore, t_0 denotes the initial aspect-oriented model and t_n the resulting conventional model. The model t_{k-1} is changed to the model t_k by the transformation operation ϕ_k. Summarized, the process can be depicted as follows:

$$t_0 \xmapsto{\phi_1} t_1 \ldots \xmapsto{\phi_{k-1}} t_{k-1} \xmapsto{\phi_k} t_k \ldots \xmapsto{\phi_n} t_n \tag{9.1}$$

Note that ϕ_0 denotes the constraints checking operation which is executed before any model transformations are done. The weaving operation ϕ_k is either an operation that weaves a join relationship or one that weaves a single instance element.

There are various dependencies between the weaving operations, which require them to be executed in a certain order. As a consequence the weaving process can be divided into two major phases. In the first phase, all join relationships must be woven, e.g., weaving the behavior and scenario chunks. In the second phase, the single instance elements such as the crosscutting statecharts and the environment objects are injected into the end target elements. The actual weaving semantics is described in the two subsequent sections.

The weaving operations are distinguished according to whether they work on non-partial or partial elements, because the corresponding weaving semantics can differ. Therefore the corresponding transformation semantics is discussed in two different sections. Section 9.2 deals with

the weaving semantics of non-partial aspect-oriented elements, whereas the next Section 9.3 discusses the weaving of partial elements. The order in which the weaving semantics is discussed follows the order that needs to be followed in order to get the intended woven models. The weaving semantics is delineated informally in both sections. The formal weaving semantics is brie y discussed in Section 9.4 and the full formal description can be found in the Appendix G.

9.2 Weaving Semantics of Non-Partial Aspect-Oriented Model Elements

This section delineates the transformation of non-partial aspect-oriented model elements to conventional model elements. The weaving semantics of behavior and scenario chunks is discussed in Section 9.2.1 and 9.2.2, respectively. Three are several single instance elements that are handled in the following: crosscutting statecharts (Section 9.2.3), crosscutting scenariocharts (Section 9.2.4), embedded components (Section 9.2.5), crosscutting environment objects (Section 9.2.6), functional specification (Section 9.2.7), and server components (Section 9.2.8). Section 9.2.9 sketches the weaving of context elements as well as the handling of naming con icts that may occur during the weaving. Finally, Section 9.2.10 discusses brie y the post-processing of the weaving transformation.

Note that for the sake of completeness, the weaving semantics of all non-partial elements are presented in the following. However, for the general understanding of the concepts, it is recommended that the first three subsections 9.2.1, 9.2.2, and 9.2.3 should be read. Of course, an interested reader may read the other sections too.

The presented weaving semantics is exemplified by means of the aspect-oriented library system model in Fig. 9.3 and its woven version in Fig. 9.4. The model shows a partial view of the same model as given in Fig. 7.2 — it visualizes only the aspect *Authorization* and the component *BookAdministration*.

9.2.1 Weaving Semantics of Behavior Chunks

An aspect can contain a behavior description which consists of behavior chunks and crosscutting statecharts (cf. Section 7.4). Behavior chunks are fragmentary statecharts which are used to augment another behavior description. The impact location is indicated by the join relationship (cf. Section 7.6) which connects the entry point of the behavior chunk with the transition that is crosscut. The exit point of the behavior chunk, visualized by a rounded rectangle with a double outline, denotes where the *crosscutting behavior* is left and the *crosscut behavior* is entered again.

Weaving Semantics

The weaving semantics of behavior chunks is controlled by the attributes of the join relationship. During the weaving, the crosscut transition is split and the ordering keywords *before*, *instead* and *after* guide how the action part of the crosscut transition is arranged with respect to the

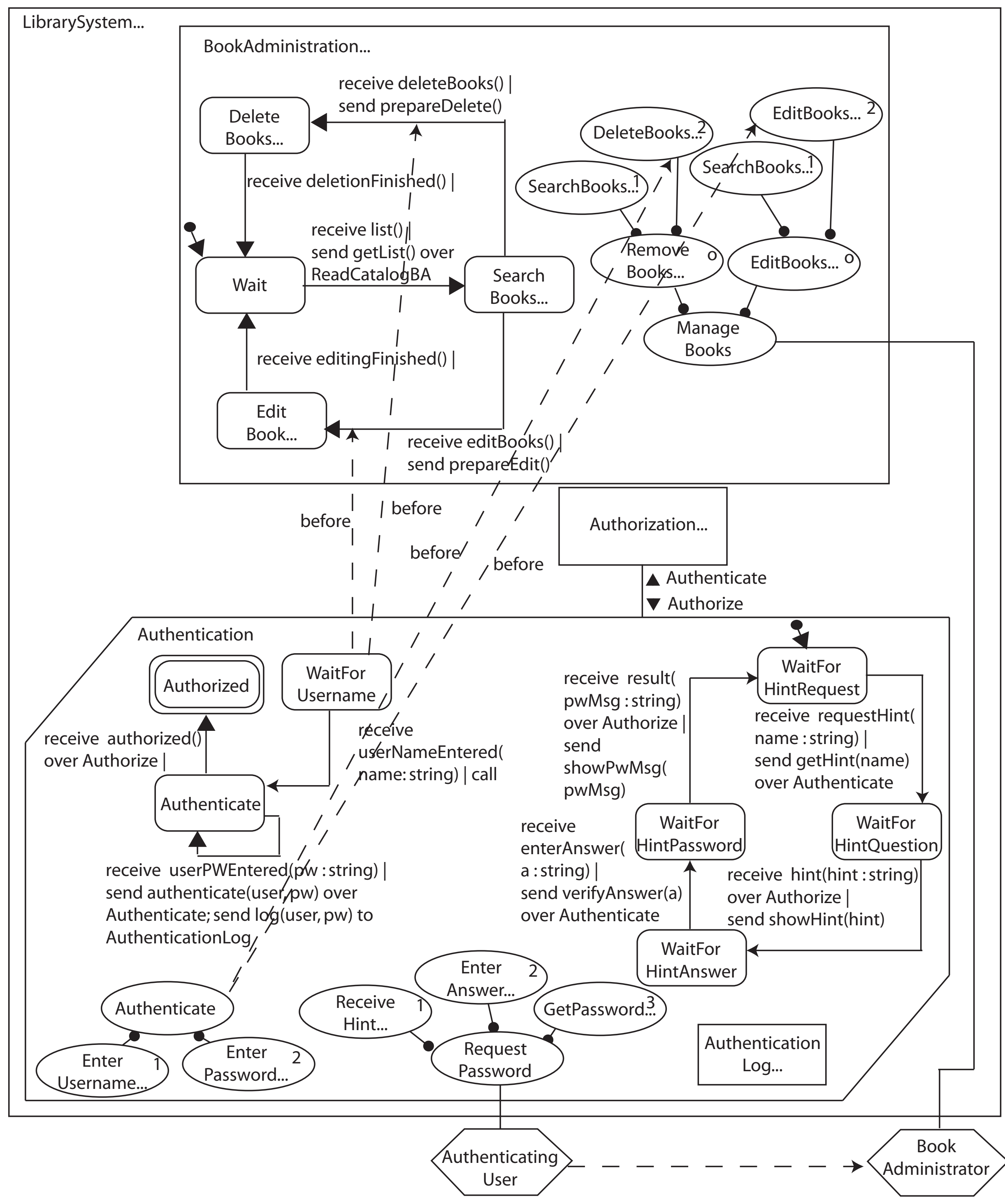


Figure 9.3: A partial view of the aspect-oriented library system model. It shows the authentication and password retrieval mechanism which has already been presented in Fig. 7.2.

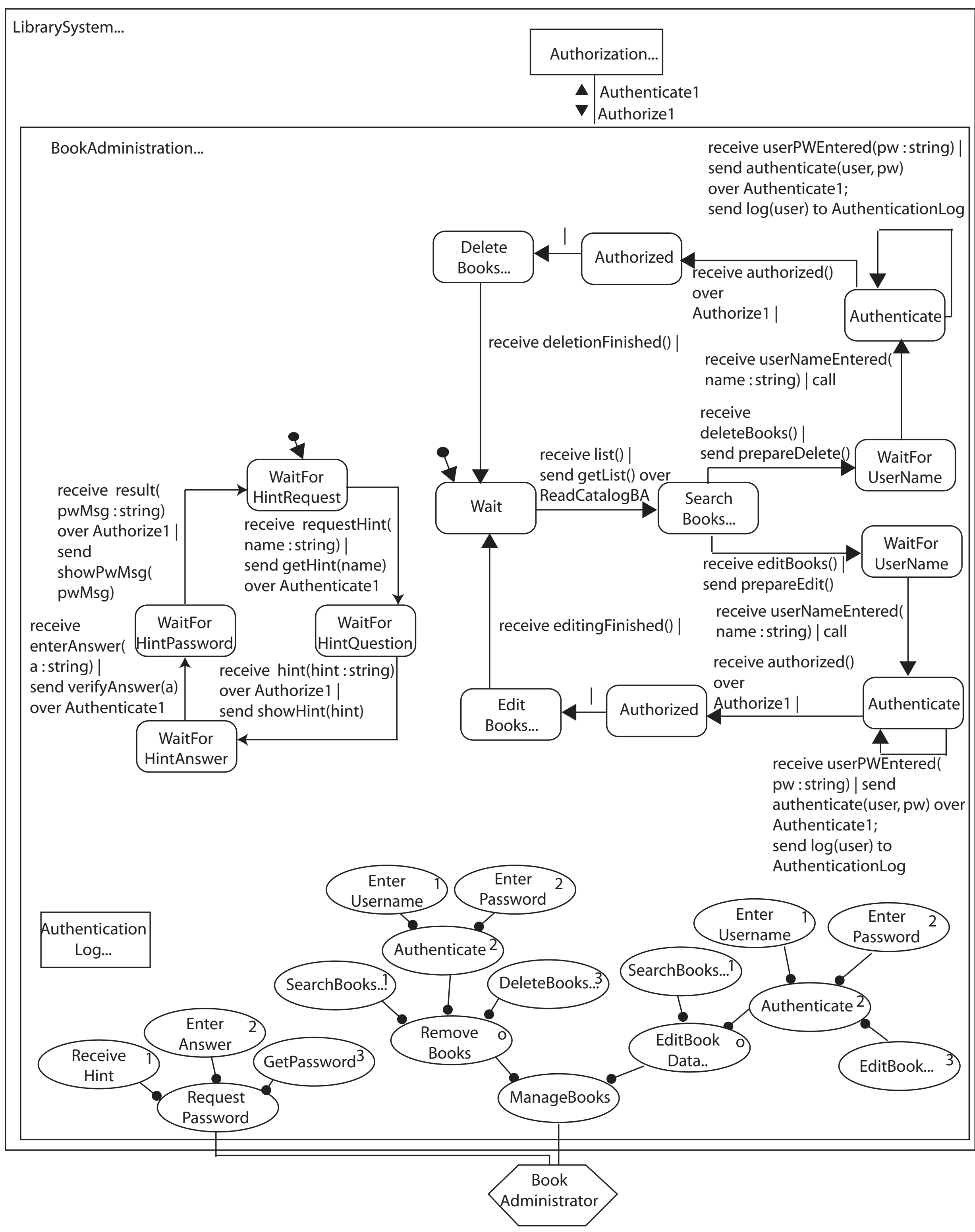


Figure 9.4: The woven model of the library system from Fig. 9.3

behavior chunk: the behavior chunk is either inserted *before*, *instead*, or *after* the action part of the crosscut transition.

There may be two or more join relationships that may crosscut the *same transition*. This kind of situation may lead to a race between join relationships with the *same ordering keyword*. The competition between them is resolved by the priority, specified by a number between 1 (lowest) and 10 (highest), which defines the precedence of competing join relationships. If nothing is specified, the lowest priority is taken as default value. The join relationship with the higher priority is woven first. If more than one join relationship has the highest priority, one of them is chosen nondeterministically for weaving.

Effectively, both the ordering keyword as well as the priority are used to sort the join relationships in the topological order in which they are woven (cf. Section 9.1).

Weaving Semantics Illustration

Fig. 9.5 explains the weaving semantics of behavior chunks more precisely for the *simple case* where just one join relationship crosscuts a transition. On the left hand side, there is the aspect-oriented model t_0, whereas on the right hand side, the woven version t_n is given.

Figure 9.5 (a) describes the weaving semantics for a *before* join relationship. The transition between *A* and *B* is crosscut by the behavior chunk contained in the aspect. It has a label $a|b$, where *a* denotes the guard and the message reception part. The letter *b* represents the action part. When weaving, the crosscut transition is reconnected to the woven behavior chunk. Furthermore, the action *b* is separated from the transition $a|b$ and then added to the outgoing transition of the state *Z*. The state *Z* is additionally introduced and is named according to the exit point *Z*.[6] The state *Z* facilitates a clearer model as it visually separates the operation *z* from the action part *b* of the crosscut transition. An alternative way of weaving would be to add the operation *b* at the action part *z* of the exit transition. However, such a solution would lead to an unclear model, as the reader has to search for the action part *b*. In contrast, the proposed solution is clearer, as it can be easily perceived that the inserted crosscutting behavior is executed before the action part *b*.

Figure 9.5 (b) describes the weaving semantics of an *instead* join relationship. It indicates that the action *b* of the transition between *A* and *B* in the crosscut behavior is removed and the entry point, i.e., state *Y*, of the behavior chunk is connected to the crosscut transition. The transition exiting the behavior chunk is connected to the state *B*.

Figure 9.5 (c) describes an *after* join relationship which declares that the entry point of the behavior chunk is inserted as the new target state of the crosscut transition $a|b$. The exit transition $y|z$ of the crosscutting behavior is connected to the state *B* of the crosscut statechart.

In all three cases (a)–(c), the join relationship is in the end removed from the model as a final step.

A concrete example for the weaving of behavior chunks is given in the library system model of Fig. 9.3 and 9.4. The behavior chunk of the authentication is woven in the behavior of the component *BookAdministration* at two different locations with a *before* semantics. Figure 9.4

[6] If the exit point is anonymous, an artificial name is given.

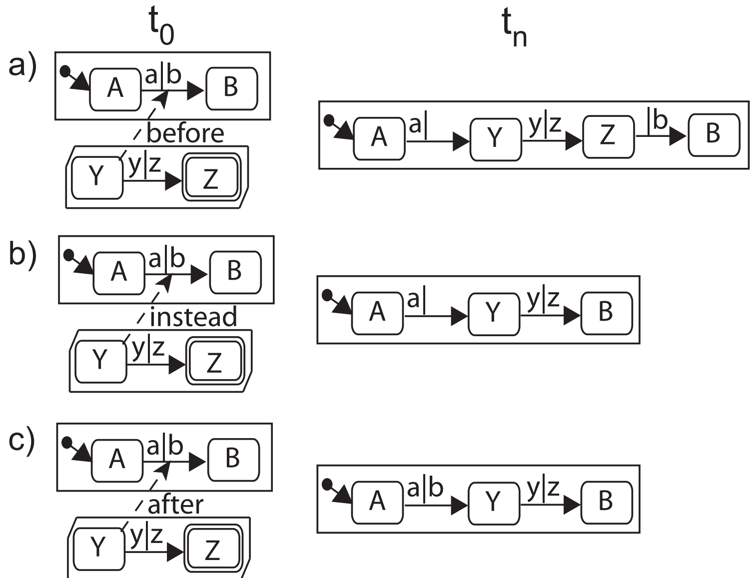


Figure 9.5: Illustration of the weaving semantics for behavior chunks. Part (a) - (c) show the weaving semantics for the situations where only one *before*, *instead*, or *after* join relationship impacts a transition.

shows the result of the weaving.

Apart from the simple case with only one join relationship, there can be the more complex case, where more than one join relationship targets the same transition. The topological sort algorithm (cf. Section 9.1) determines the weaving order by the keyword and the priority. After weaving the first join relationship with the target transition, some properties of the remaining join relationships targeting the same transition must be altered. This is necessary in order to get a logically correct model at the end of the weaving.

After the weaving of one join relationship, the necessary changes to the remaining join relationships depend on their ordering. As one join relationship is woven after the other, the possible cases that have to be distinguished can be listed as pairs of join relationships which are represented by their ordering keywords *before*, *instead*, and *after*. The following combinations may occur: (before,before), (before,instead), (before,after), (instead,instead), (instead,after), and (after,after).[7] These cases are shown by Fig. 9.6 and 9.7. The columns t_0, t_1, and t_n in the figures denote the transformation states of the model during the weaving. Hence, model t_0 denotes the (unprocessed) aspect-oriented model, t_1 an intermediate state of the transformation and t_n the resulting woven model.

Figure 9.6 (a) illustrates the weaving semantics for two *before* join relationships impacting the same transition. As the ordering keyword is the same, their precedence is determined by

[7]Other combinations, such as (instead,before) or (after,instead), end also in the cases mentioned above, because of the topological sorting of the join relationships (cf. Section 9.1). For example, the case (*instead*, *before*), results in (*before*, *instead*).

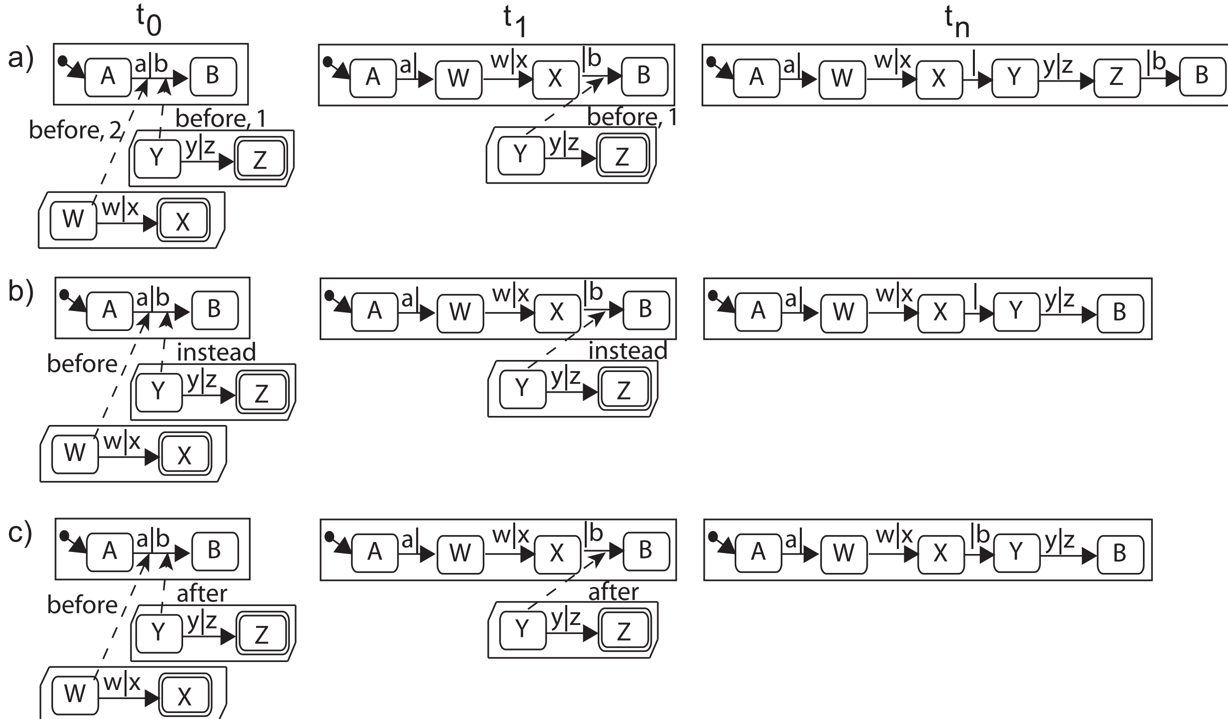


Figure 9.6: Illustration of the weaving semantics for multiple behavior chunks impacting the same target transition. The model in (a) shows the semantics for two competing *before* join relationships. Situation (b) illustrates the case of a *before* and an *instead* join relationship, and (c) the case for a *before* and an *after* join relationship.

their priority. Therefore, the join relationship with the higher priority of 2 between the state *W* and the transition $a|b$ is woven first. The result is shown in the model t_1. In order to get an intuitively correct model in the end of the weaving process, the target of the remaining *before* join relationship must be relocated to the transition $|b$ which exits the crosscutting behavior. The other combinations of join relationships, shown in Fig. 9.6 (b)–(c) and Fig. 9.7 (a)–(c), are handled similarly.

The case where two *instead* join relationships are crosscutting the same transition in Fig. 9.7 (a) needs special attention. After weaving the first join relationship, i.e., in the intermediate model t_1, the remaining *instead* join relationship needs to change the ordering keyword. This is due to the fact that the action part *b* has already been replaced. Therefore, any remaining *instead* join relationships must be interpreted with an *after* ordering. This ensures that the further weaving process does not accidentally remove the *x* action part from the crosscutting behavior.

The case with two join relationships can be generalized for the case where more than two join relationships crosscut the same transition. Suppose, the tuple J represents an ordered list that contains all join relationships impacting the same transition j_i. After weaving the $j_0 = \pi_0(J)$, all remaining join relationships $\pi_i(J)$, where $i \in \mathbb{N} \wedge 1 \leq i < \mathit{arity}(J)$, have to be relocated

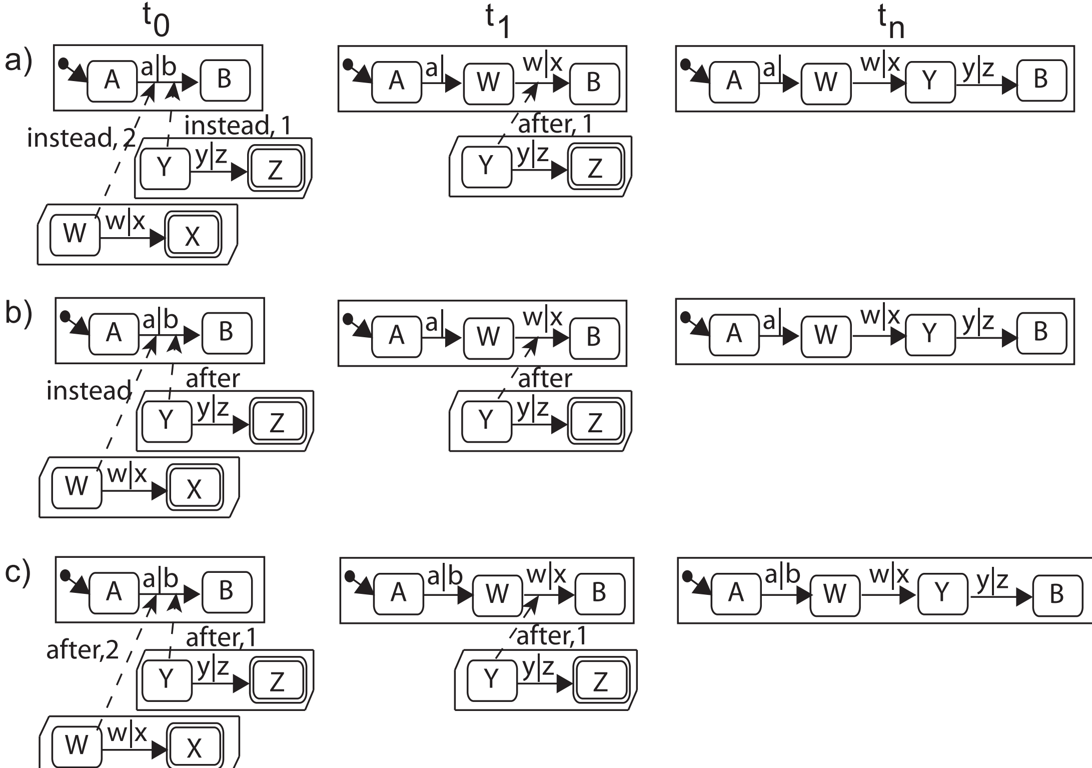


Figure 9.7: Illustration of the weaving semantics for multiple behavior chunks impacting the same target transition. The model in (a) shows the semantics for two competing *instead* join relationships, (b) a competing *instead* and an *after* join relationship. Finally in (c), the weaving of two competing *after* join relationships is illustrated.

to the transition which exits the crosscutting behavior.[8] Furthermore, if $\pi_0(J)$ is an instead join relationship, the ordering of all remaining *instead* join relationships must be changed to *after*.

9.2.2 Weaving Semantics of Scenario Chunks

An aspect can contain crosscutting use case descriptions which are defined in terms of scenario chunks and scenariocharts (cf. Section 7.5). The weaving semantics of scenario chunks is discussed in the following.

Weaving Semantics

A root node of a scenario chunk can be connected by a join relationship to a node of another scenario tree. The join relationship can be attributed with the ordering keywords *before*, *instead*

[8]The function π denotes the projection, and the function $arity$ returns the number of elements in a tuple (cf. Section E.3 of the Appendix).

or *after*. In the case that the target node and its siblings are of type *sequence*[9], the keywords *before* and *after* influence the weaving order. Moreover, in the case of a target of the type *sequence*, the root node of the woven scenario chunk gets a sequence number and the sequence numbers of the sibling nodes must be adapted accordingly.

In contrast to nodes of type *sequence*, sibling scenario nodes of the type *parallel*, *alternative*, and *root* have no order in the scenario tree, as they are executed either alternatively, concurrently, or solely, respectively. The keywords *before* and *after* can be specified for the target nodes of the types *parallel* and *alternative*, but they do not influence the weaving order. In contrast, the type *root* is not allowed to be targeted by a *before* or *after* join relationship (see below). However, all scenario types, including *alternative*, *parallel* and *root*, can be crosscut by an *instead* join relationship. The *instead* ordering directs the weaving process to replace the target node and *all* of its children with the given scenario chunk.

For the weaving of scenario chunks, the priority of the join relationship is used to solve conflicts. It defines the precedence of competing[10] scenario chunks targeting the same scenario node with the type *sequence*. Thus, the priority is used similarly as for behavior chunks (cf. Section 9.2.1). The join relationship with the highest priority is woven in first, followed by the one with the second highest priority and so on. If join relationships have the same priority, they are woven in a nondeterministic order. Actually, the ordering keyword and the priority are used by the topological sort algorithm (cf. Section 9.1) to create an ordered list of join relationships. This list specifies the weaving order and thereby resolves any conflicts.

Root nodes of a scenario tree can also be crosscut by a scenario chunk but this situation is treated as a special case. Root nodes can only be the target of a join relationship with an *instead* ordering. Attributing them with *before* and *after* is not allowed because root nodes may not be part of a sequence (cf. constraints definitions in Section 7.5). Furthermore, there may be a situation where more than one *instead* join relationship is targeting the root node of a scenario tree. In this case, only the join relationship with the highest priority is woven, the others are omitted. If there are several join relationships with the same priority, one of them is chosen nondeterministically.

Note that a node of a scenario chunk may be contained in an embedded component. In this case, the nodes of the crosscutting scenariochart are woven in two steps. First, the node is woven as a direct child into the target module. Second, the scenario node is moved into the embedded component, as soon as it is also woven (cf. Section 9.2.5).

Finally, before the next weaving step is taken, the join relationship between the woven scenario chunk and the target scenario is removed from the model.

Weaving Semantics Illustration

Figure 9.8 illustrates the case where just one join relationship crosscuts a sequentially executed target scenario node. On the left hand side, there is the aspect-oriented model t_0, while the woven

[9] As discussed in Section 7.5, there are the types *root*, *alternative*, *sequence*, and *parallel*. As discussed in Section 5.2.4, all siblings of a scenario node must have the same type.

[10] Two scenario chunks are competing if their join relationship target the same scenario node with the same ordering.

version can be found in model t_n on the right hand side. The weaving semantics depends on the ordering and the type of the target scenario node:

Figure 9.8 (a) describes the *before* case for a scenario target node of the type *sequence*. The weaving process inserts the root node of scenario chunk *X* before the target scenario node *B*. The inserted node gets the sequence number of the target node. The sequence numbers of the target node and its subsequent siblings are incremented by 1.

Figure 9.8 (b) shows a situation where the scenario chunk crosscuts the *root node A* of the target scenario tree. In this case, the join relationship must specify an *instead* ordering. When weaving this constellation, the target scenario node and all its children are replaced by the scenario chunk.

Figure 9.8 (c) exemplifies the weaving semantics for an *instead* join relationship in the case that the target is a *non-root* scenario node of type *sequence*. In this case, the target node and its direct and indirect children are replaced by the scenario chunk and the sequence number of the target node is taken over. This *instead* semantics applies also to nodes which are not of the type *sequence*. However, for these nodes, no sequence number is taken over from the inserted chunk root node.

Figure 9.8 (d) depicts an *after* join relationship in the case of a target node with the type *sequence*. The root node of the scenario chunk is inserted after the specified target node *B*. The sequence number of node *B* is preserved whereas the root node of the inserted scenario chunk gets the subsequent number. The indices of all following nodes are increased by 1.

Figure 9.8 (e) shows the situation where the target scenario node *B* has the type *alternative*.[11] In this case, there is no ordering between the target node and its siblings. Therefore, any *after* or *before* keyword of the join relationship is meaningless, but not forbidden.[12] In the example, the join relationship does not specify any ordering keyword. The root node *X* of the scenario chunk is inserted as an alternative to *B* and *C*.

Another example of the weaving semantics for scenario chunks is given by Fig. 9.3 and 9.4. In Fig. 9.3, the communication between the authentication mechanism and the authenticating user is described as a scenario chunk in the aspect. Correspondingly, the woven chunk can be found in Fig. 9.4.

There may be more than one join relationship targeting the same node. This case is handled similarly to the case of behavior chunks. The ordering keyword, as well as the priority are used to decide in which order the scenario chunks are woven into the target scenario tree. The topological sort algorithm (cf. Section 9.1) determines the weaving order by the ordering keyword and the priority. After weaving the first join relationship into the target scenario tree, some properties of the remaining join relationships may have to be altered. The necessary changes depend on the ordering of the remaining join relationships.

The different cases of join relationship combinations can be distinguished as pairs of the ordering keywords *before*, *instead*, and *after*, which results in the following combinations: (before, before), (before, instead), (before, after), (instead, instead), (instead, after), and (after, after).[13]

[11]The weaving semantics for target nodes of the type *parallel* is equivalent to the case shown.

[12]The only meaningful ordering keyword that in uences the weaving is *instead*.

[13]Cases such as (after, before) are also handled by the cases above, because they are ordered by the topological sort algorithm. For example (after, before) results in (before, after).

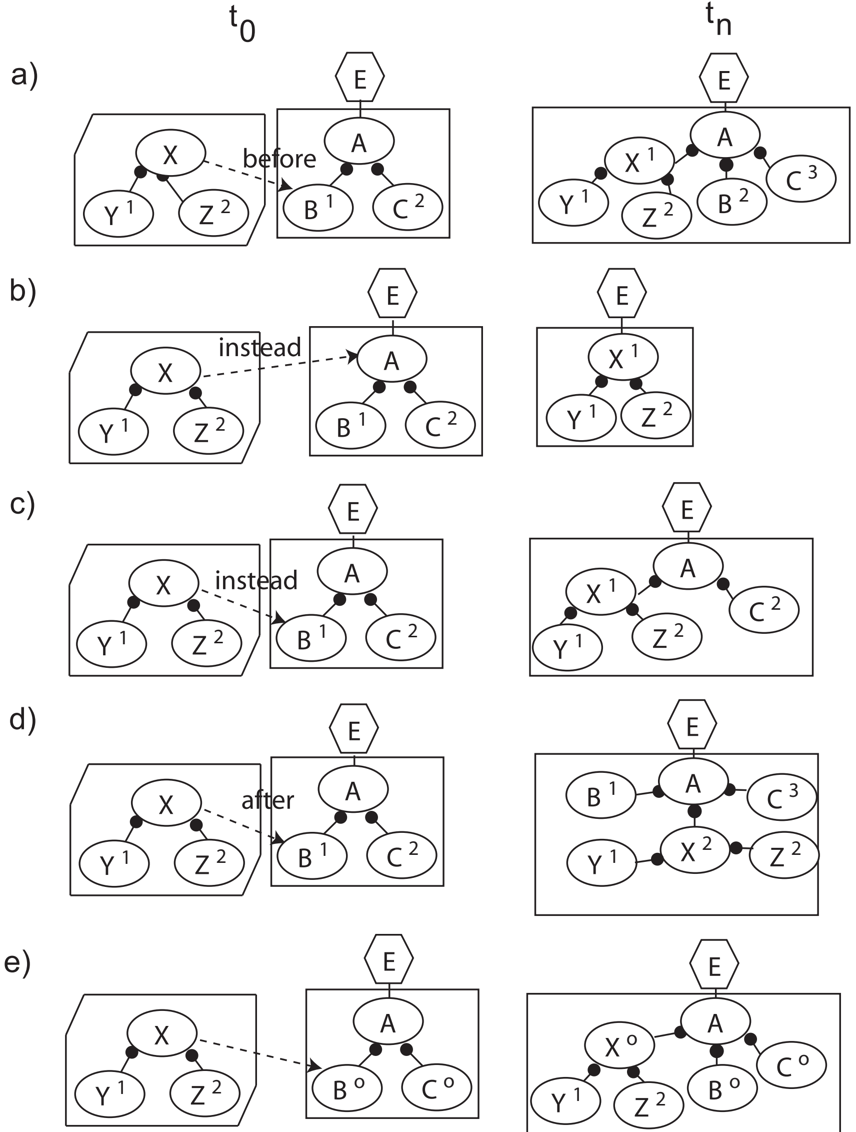


Figure 9.8: Weaving semantics for scenario chunks crosscutting a target scenario node. Fig. (a), (c), (d) illustrate the weaving semantics for sequential scenario targets with a *before* (a), *instead* (b)–(c), and *after* (d) ordering. Moreover, in (b), the semantics for an *instead* join relationship targeting a scenario root node is shown. In (e) the weaving semantics for a target node of the type *alternative* is illustrated.

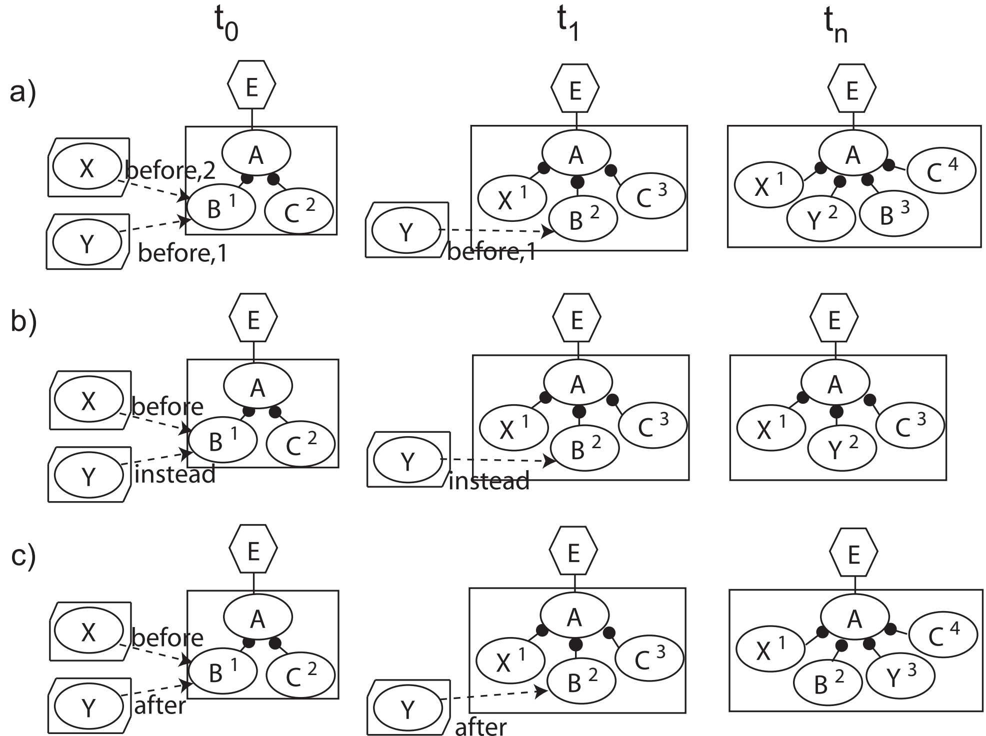


Figure 9.9: Weaving semantics for more than one join relationship targeting the same scenario node of type *sequence*. Fig. (a) shows the case where two *before* join relationships crosscut the same scenario node. In (b), a *before* and an *instead* join relationship, and in (c) a *before* and an *after* join relationship target the same scenario node.

Figure 9.9 (a) shows the case where two *before* join relationships impact the same scenario node *B*. Hence, the order in which they are woven is determined by their priority. The scenario node *X* is woven first, as its join relationship has the higher priority 2. The first weaving step results in the model t_1. In the next step, the second node *Y* is woven. The final model is shown as t_n.

The situations shown in Fig. 9.9 (b) and (c) illustrate the combinations of a *before/instead* and a *before/after* join relationship. The are handled similarly to the case shown in (a).

Figure 9.10 (a) illustrates the weaving semantics for two *instead* join relationships impacting a root scenario node. The join relationship which is woven first is the one between *X* and *A*, since it has the higher priority 2. The remaining *instead* join relationship is deleted, as the corresponding scenario chunk cannot be injected meaningfully into the target module. This leads to the situation described by model t_1. The final model resulting from the weaving process is shown in column t_n.

Figure 9.10 (b) shows the weaving semantics for two *instead* join relationships impacting the same *non-root* target scenario node. In a first step, the *instead* join relationship with the higher priority is woven first. Thus, the scenario node *X* replaces the node *B* in the component.

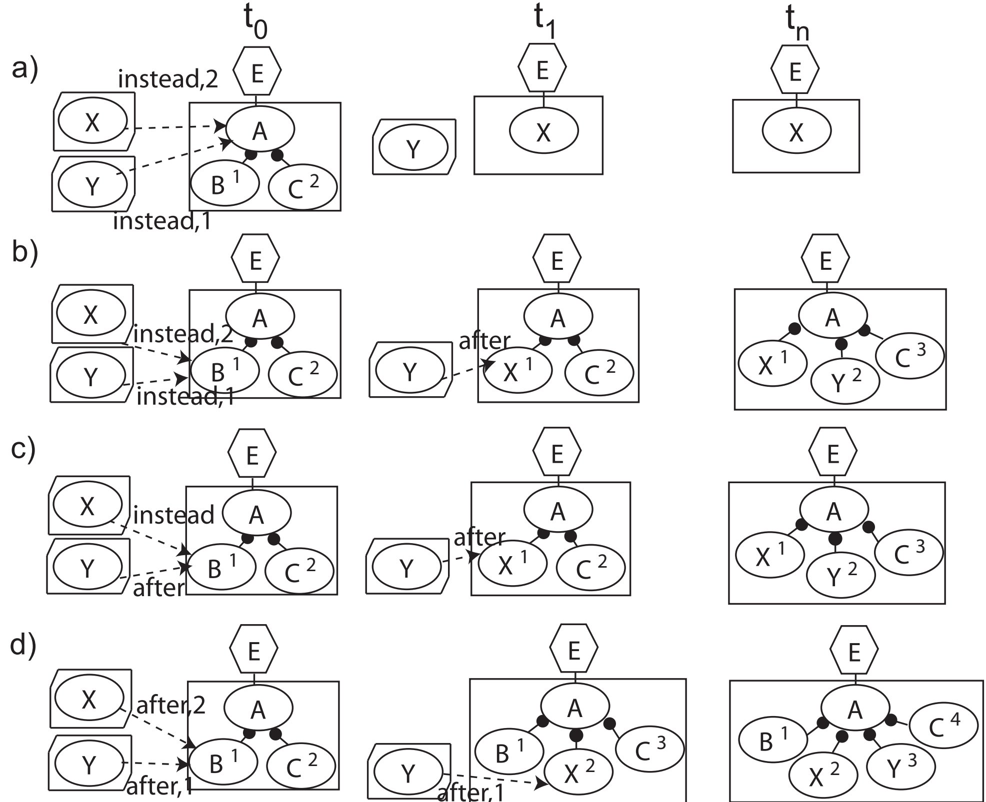


Figure 9.10: Further cases with more than one join relationship targeting the same scenario node. The model in (a) shows the case where two *instead* join relationships crosscut the same *root* scenario node. In (b), the same situation is given for a *non-root* scenario node. In (c) and (d), the combinations with *instead/after* and *after/after* join relationships are illustrated.

Furthermore, the weaving mechanism changes the ordering of the remaining join relationship between *Y* and *X* to *after*. This is necessary in order that the woven node *X* is not replaced by *Y*. The resulting model is shown in the intermediate model t_1. Subsequently, the remaining *after* join relationship is woven. The final model is shown in column t_n.

Figure 9.10 (c) and (d) show the weaving semantics for an *instead/after* and *after/after* join relationship targeting the same node. They are handled similarly to the case of Fig. 9.10 (b). However, in case (d), a change of the target for the remaining join relationship is needed (cf. t_1). This is necessary in oder to get a logically correct woven model at the end of the weaving process.

9.2.3 Weaving of Crosscutting Statecharts

Apart from behavior chunks, aspect containers may include crosscutting behavior that is concurrently executed with the behavior chunks. This kind of behavior is modeled by conventional

ADORA statecharts which are called crosscutting statecharts (cf. Section 7.4). Crosscutting statecharts are single instance elements (cf. Section 9.1), i.e., they are woven just once per end target module.

Weaving Semantics

A crosscutting statechart *S* contained in aspect module *A* is woven into the module *E*, if *E* is crosscut, either transitively or directly, one or more times by the join relationships originating in *A*. After the weaving, only one clone copy[14] of *S* is contained in *E*, no matter if there are one or more join relationships between *A* and *E*. Note that it does not matter whether the join relationship path between *A* and *E* weaves a behavior chunk or a scenario chunk.

Weaving Semantics Illustration

Figure 9.11 exemplifies the weaving semantics for crosscutting scenariocharts; on the left hand side the aspect-oriented model t_0 is given, whereas the right hand side depicts the woven model t_n. The dark printed model parts show the elements which are at the focus of interest, i.e., the crosscutting statecharts.

Figure 9.11 (a) shows the situation where a crosscutting statechart is injected from an aspect which contains a behavior chunk. The resulting model is shown by t_n. Figure (b) shows a similar situation to (a). However, in (b) the injection of the crosscutting statechart is caused by the join relationship originating from a scenario chunk. The result is shown by model t_n.

Finally, Fig. 9.11 (c) shows a situation where an aspect crosscuts a target at two different locations. Even though there are more than one join relationship paths (cf. Section 9.1) between the aspect and the target, *only one* copy of the crosscutting statechart is injected into the target, as shown by model t_n.

Another example of the weaving semantics of a crosscutting statechart is given in Fig. 9.3 and 9.4. In Fig. 9.3, the password retrieval mechanism for forgotten passwords is described by a crosscutting statechart. The woven model in Fig. 9.4 contains correspondingly only one instance of the statechart, although there are four join relationships targeting the *BookAdministration* component.

9.2.4 Crosscutting Scenariocharts

Apart from the scenario chunks, aspects can also contain *crosscutting scenariocharts*. A crosscutting scenariochart (cf. see Section 7.5) specifies the interaction between an environment object (or an external component) and a crosscutting statechart (i.e., crosscutting functionality that is concurrently executed to other crosscutting functionality in a system). They are described in terms of conventional scenariocharts.

[14] All parts of the clone copy and the original crosscutting statechart match, except the unique model element identifiers.

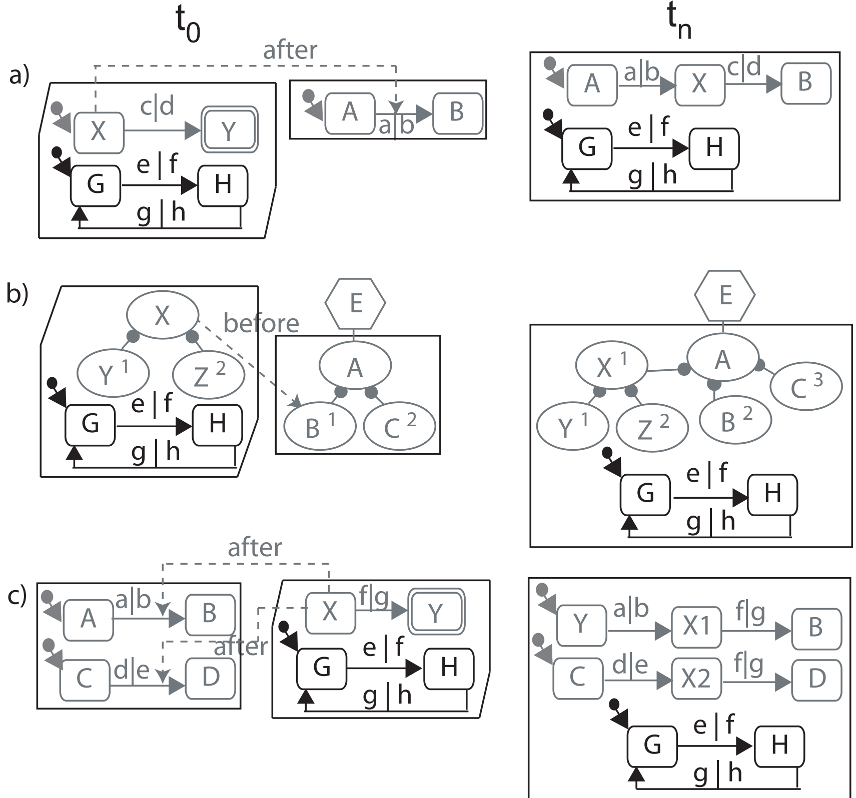


Figure 9.11: Weaving semantics for crosscutting statecharts. Situations (a) and (b) illustrate that the weaving of a crosscutting statechart can result from a join relationship that originates either from a scenario or a behavior chunk. The situation in (c) shows that a crosscutting statechart is just woven once into the target module, no matter if there is more than one join relationship path between the aspect and the target.

Weaving Semantics

Crosscutting scenariocharts belong to the single instance elements (cf. Section 9.1). Thus, a crosscutting scenariochart *S* contained in aspect *A* is woven just once into a target module *M*, no matter how many join relationship paths exist between *A* and *M*. Moreover, it does not matter whether the join relationship path has its origin in a scenario chunk, or a behavior chunk.

In order to be well-formed, the root node of a scenario needs to be connected by at least one association to an environment object (or an external component). When weaving the crosscutting scenariochart, the association connecting it to an environment object must be cloned. Hence, the cloned association connects the original environment object and the cloned crosscutting scenariochart in the woven model.

A node of a crosscutting scenariochart can either be contained as a direct child of in the aspect module itself or in one of its directly or indirectly embedded components. In the case, where a node is contained in an embedded component, it is woven in two steps. In a first step, the clone

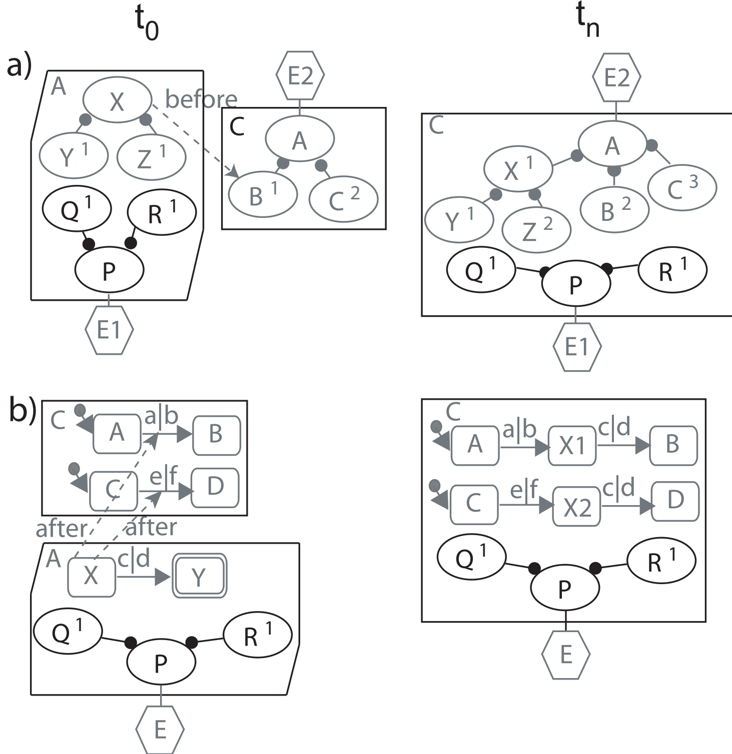


Figure 9.12: Example for the weaving semantics of crosscutting scenariocharts. Fig. (a) and (b) show the weaving semantics for a crosscutting scenariochart. In (a) the weaving is triggered by the join relationship between a *scenario chunk* and a *target scenariochart*. In (b), a situation is illustrated where more than one join relationship crosscuts a target module. Moreover, the situation exemplifies also that the weaving of a crosscutting scenariochart can also be caused by join relationships between behavior chunks and behavior descriptions.

of the node is just added as a direct child in the target module. Then, after weaving the corresponding embedded component, the woven scenario node is moved into it (cf. Section 9.2.5).

Weaving Semantics Illustration

Figure 9.12 illustrates the weaving semantics for crosscutting scenariocharts. Model t_0 on the left hand side is the aspect-oriented model. The woven model t_n is given on the right hand side. Figure 9.12 (a) shows the situation where one join relationship crosscuts the target module. A clone of the crosscutting scenariochart is injected into the target module, as shown by t_n. Figure 9.12 (b) illustrates the case where more than two join relationships crosscut the same target module. Nonetheless, only one clone of the crosscutting scenariochart is injected into the target module. Moreover, the figure also exemplifies that it does not matter that the join relationship causing the injection connects behavioral elements.

Figure 9.13 illustrates the weaving semantics for crosscutting scenariocharts in more detail.

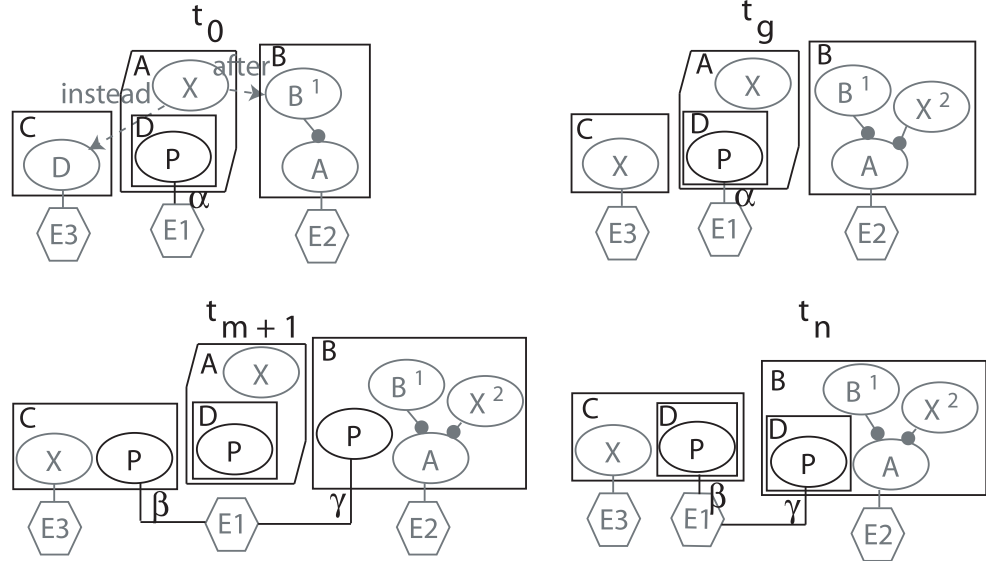


Figure 9.13: Illustration of the weaving process for crosscutting scenariocharts. The figure shows different steps during the weaving. Model t_0 is the initial aspect-oriented model. Model t_g shows the model after weaving all join relationships. In t_{m+1}, all crosscutting scenariocharts are woven into the target module. The final model is shown in t_n.

Model t_0 is the initial aspect-oriented model. After all join relationships are woven in the intermediate model t_g, the crosscutting scenariochart, which consists only of one scenario node *P* contained in an embedded component *D*, is woven into both target components *C* and *B*. The intermediate model t_{m+1} shows the resulting model. At this point in time, the node *P* is not contained in the clone of the embedded component *D*, as *D* is not woven yet. The clone of *P* is moved later into the corresponding embedded component, which is illustrated by the model of t_n.

Figure 9.13 also illustrates the cloning of the associations connected to a crosscutting scenariochart. The original association[15] α between the environment object *E1* and the scenario node *P*, given in t_0 is cloned two times. The association clones β and γ are connected to the woven crosscutting scenariocharts contained in the target modules *C* and *B*. This situation is shown by model t_{m+1}. The original association α is deleted at the end, as shown in t_n.

A further example of the weaving semantics for crosscutting scenariocharts is given in the library system model of Fig. 9.3 and 9.4. The interaction protocol between the *Authenticating-User* and the password retrieval mechanism is specified as a crosscutting scenariochart. The woven model in Fig. 9.4 shows the component *BookAdministration* which comprises the woven crosscutting scenariochart.

[15]The Greek letters are used to refer to the associations but they are not part of the model.

9.2.5 Weaving of Embedded Components

An aspect module can contain embedded components (cf. Section 7.9).[16] An embedded component is a single instance element (cf. Section 9.4.2). Therefore, an embedded component is just woven once per end target module, no matter how many join relationship paths lead from the aspect to the end target module.

When an embedded component *C* is woven, any scenario nodes contained in *C* or in the children of *C* are removed. This is necessary due to the fact that scenario nodes contained in embedded components have already been replicated by the operation which weaves scenario chunks or crosscutting scenariocharts (cf. Section 9.2.2 and 9.2.4), respectively. The woven scenario nodes have been placed as direct children in end target module *M*, and they are relocated into the corresponding embedded components after the embedded components have been woven into *M*.

Illustration of the Weaving Semantics

Figure 9.14 illustrates this weaving semantics. The aspect-oriented model t_0 is shown on the left hand side, the woven model t_n on the right hand side. In situation (a), an embedded component *C* is contained together with a scenario chunk in an aspect. When weaving this model, a clone of *C* is woven into the target component, as shown in model t_n. The model in Fig. 9.14 (b) shows a similar situation, but the weaving of the embedded component is caused by a behavior chunk. Moreover, (b) also illustrates that even though the same target component is crosscut twice by the same aspect, the target module in the final model t_n contains only one copy of the embedded component *C*.

Figure 9.15 shows the process for the weaving of embedded components and how scenario nodes are handled by the weaving operation.[17] The initial model is given in t_0. After weaving all join relationships and the crosscutting scenariocharts, model t_{m+1} results. The nodes of the crosscutting scenariocharts are located in the target module itself. After the weaving of the embedded components, the nodes of the crosscutting scenariochart are moved into the corresponding embedded components.[18]

The library system given in Fig. 9.3 also exemplifies the weaving semantics of embedded components. The component *AuthenticationLog*, which is used to log security-related events, is contained in the aspect *Authentication*. In Fig. 9.4, the woven view of this system model is given and the *AuthenticationLog* is woven into the corresponding targets. Note that although the component *BookAdministration* is crosscut by four different join relationships, there is only one instance of the component *AuthenticationLog* in the target after the weaving.

[16] Remember the difference between embedded components and components which are part of a behavior chunk. The latter are *not* handled as embedded components but as states of the behavior description (cf. Section 9.2.1).

[17] It shows the same example as in Fig. 9.13.

[18] The moving of scenario nodes belonging to a *scenario chunk* is not illustrated here, but works in the same way.

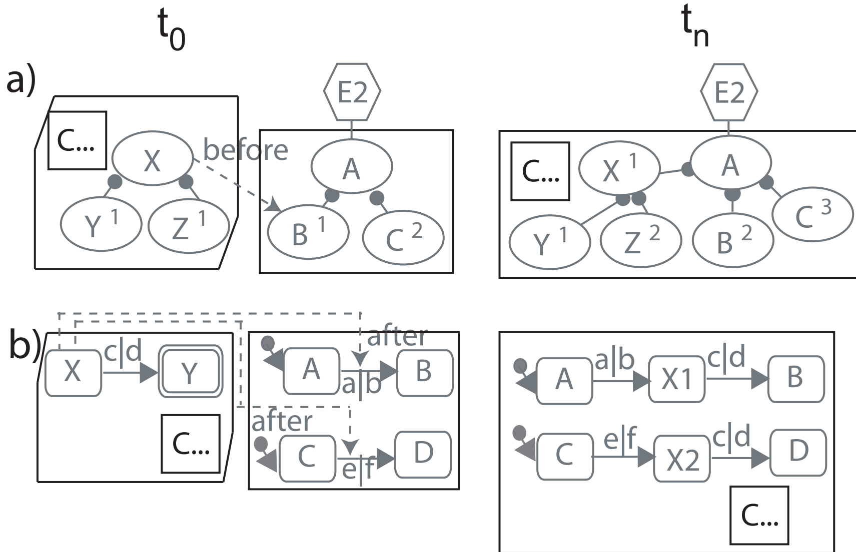


Figure 9.14: Example of the weaving semantics for embedded components. The aspect-oriented model t_0 is given on the left hand side, the woven version t_n on the right hand side. Figure (a) shows the weaving of the embedded component caused by a join relationship between a scenario chunk and a target scenario. In (b), multiple join relationships from a behavior chunk crosscut the target behavior. However, the embedded component *C* is just woven once into the target component.

9.2.6 Weaving Environment Objects

As discussed in Section 7.5, well-formed crosscutting scenariocharts must be connected by an association to an environment object (cf. Section 7.7). Such an environment object can crosscut another environment object, which is expressed by a join relationship connecting to the crosscut environment object. Join relationships between crosscutting environment objects are not handled like other join relationships, such as the ones used for weaving behavior or scenario chunks.

Weaving Semantics

When weaving a model containing an environment object *E1* that crosscuts another environment object *E2*, all associations connected to *E1* are cloned and connected to *E2*. As multiple environment objects can be crosscut, the associations may be replicated as often as there are join relationships outgoing from *E1*. After weaving, the environment object *E1*, the associations connected to *E1*, and the join relationships are removed from the model.

In contrast to crosscutting environment objects, non-crosscutting environment objects are neither removed from the resulting model, nor are their associations cloned and assigned to another environment object.

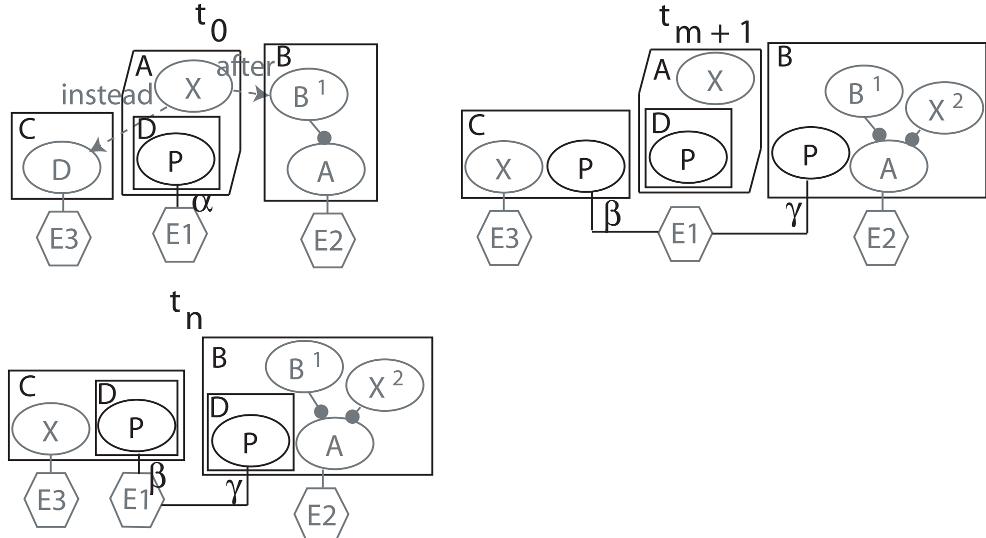


Figure 9.15: Illustration of the weaving process for embedded components. Model t_0 shows the initial model, t_{m+1} the model after the weaving of the crosscutting scenariocharts and t_n the model with the woven embedded components.

Weaving Semantics Illustration

Figure 9.16 exemplifies this weaving semantics. The left hand side shows the aspect-oriented model t_0, the right hand side the conventional model t_n.

Figure 9.16 (a) shows the situation where a *non-crosscutting* environment object *E1* is connected to a crosscutting scenariochart. When weaving, *E1* is still extant in the resulting model, being connected to the woven crosscutting scenariocharts. In contrast, Fig. 9.16 (b) shows the situation where an environment object *E1* crosscuts another environment object *E2*. The associations of *E1* are cloned and connected to *E2*, which is shown in the woven model t_n. The association between *E1* and *P*, the join relationship, as well as the crosscutting environment object *E1* are deleted from the resulting model.

Another example for the weaving of crosscutting environment objects can be found in the aspect-oriented library system model in Fig. 9.3, where the environment object role *AuthenticatingUser* crosscuts the environment object *BookAdministrator*. In the woven model shown by Fig. 9.4, the association originally connected to the environment object *Authenticating User* is incorporated in the crosscut environment object *BookAdministrator*.

Figure 9.17 illustrates how the weaving of crosscutting environment objects interplays with the other steps of the weaving process. Model t_0 shows the situation before the scenario chunk is woven. The intermediate model t_1 shows the woven scenario chunk. The model t_{u-1} contains the woven crosscutting scenariocharts of the aspect. The woven scenario node is connected by two associations γ and δ that are clones of α and β, respectively.[19] The original associations

[19]The Greek characters are only used to refer to the associations. They are not part of the model.

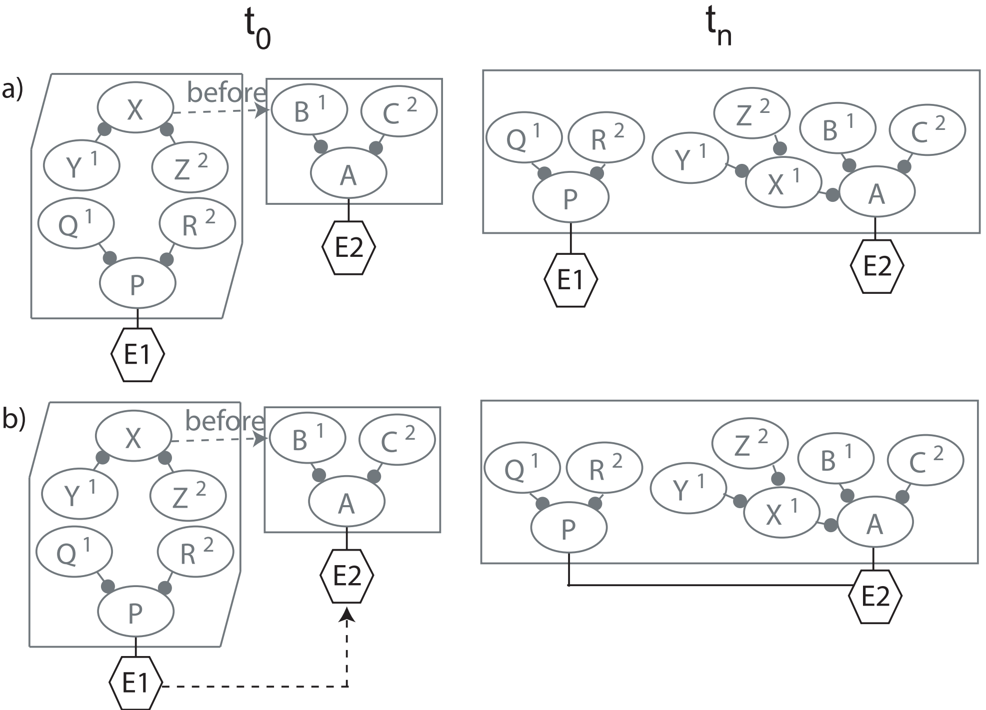


Figure 9.16: Example of the weaving semantics for a crosscutting environment object. Figure (a) exemplifies the weaving semantics for a non-crosscutting environment object *E1*. The model in (b) illustrates the weaving semantics for an environment object *E1* crosscutting *E2*.

α and β have been removed from the model when weaving the crosscutting scenariocharts (cf. Section 9.2.4).

The actual weaving of crosscutting environment objects is carried out in several substeps. The crosscutting environment objects belonging to one aspect are processed step by step for each target module. In the example, the join relationship between *E1* and *E3*, as well as between *E1* and *E2* are woven in the same step. The associations γ and δ are cloned as many times as there are join relationships outgoing from *E1*. The associations ω and σ, which are clones of γ and δ, are created and connected with the root nodes *P* and *Q* and *E2*. Similarly this is done for η and φ, *P*, *Q* and *E3*. Finally, the cloned associations γ and δ, introduced when weaving the crosscutting scenariocharts (cf. Section 9.2.4), as well as the join relationships, and the crosscutting environment object *E1*, are removed from the model t_u.

9.2.7 Weaving the Functional View of an Aspect

An aspect is a module, and therefore, it can also contain a functional specification (cf. Section 7.8). It defines the properties of an aspect, such as attributes and operations, on which the behavior or the scenario descriptions may rely. The functional specification must also be woven in the crosscut module.

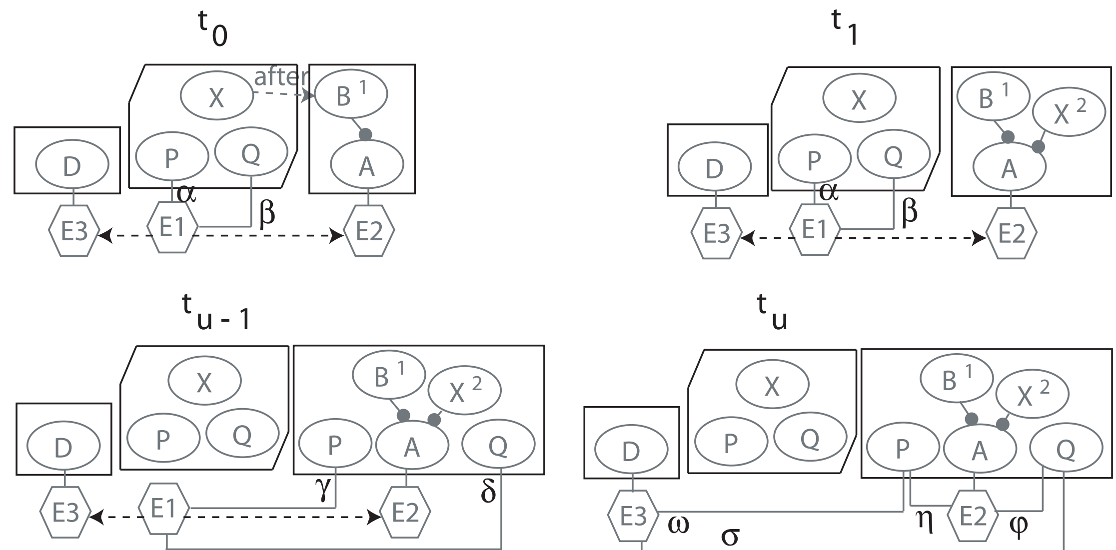


Figure 9.17: Illustration of the weaving steps for crosscutting environment objects. The figure shows four subsequent steps, including the stages before and after the weaving of the crosscutting environment object *E1*.

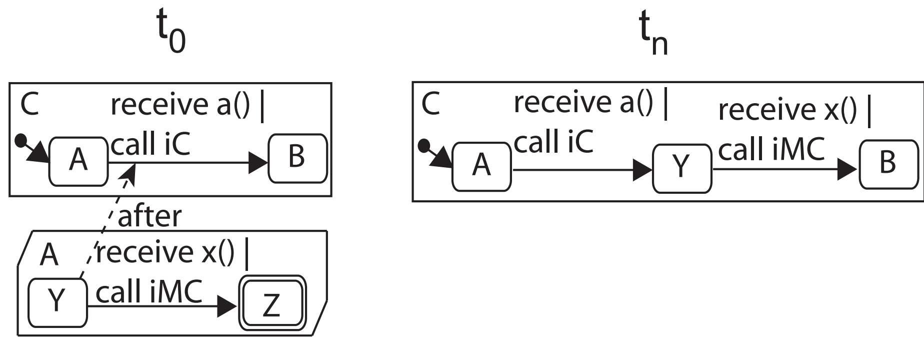


Figure 9.18: Model illustrating the weaving semantics of the functional specification.

Weaving Semantics Illustration

The corresponding weaving semantics is demonstrated by means of the model in Fig. 9.18. It contains an aspect-oriented model t_0 and the woven version t_n. Listing 9.1 shows the functional specification of component *C* and Listing 9.2 that of aspect *A*.

The elements of an aspect's functional specification are woven by simply appending them to the elements in the functional specification of each end target module of an aspect. In the case of naming con icts, the weaving operation has to rename the con icting element, which is discussed in Section 9.2.9. Listing 9.3 shows the resulting woven functional specification of the two listings 9.1 and 9.2.

Listing 9.1: The functional specification of the component *C* of model t_0 in Fig 9.18.

```
functional speci cation
  provides c;
  requires B.d;
  data type c : constrained x : integer (x >= 0);
  property req_source 1 "W. Miller";
  attributes counter : c = 0;
  inv c < d;
  operation iC()
    statements
      counter = counter + 1
  end operation counter
end functional speci cation
```

Listing 9.2: The functional specification of the aspect *A* of model t_0 in Fig 9.18.

```
functional speci cation
  provides m;
  requires B.k;
  data type m : constrained x : integer (x >= −1 and b <=1);
  property author "D. Cunningham";
  attributes modcount : m = −1;
  inv modcount < k;
  operation iMC()
    statements
     /*
      if modcount over ows,
      it starts again with −1
     */
    modcount = modcount + 1
  end operation counter
end functional speci cation
```

Listing 9.3: The functional specification of the woven component *C* of model t_n in Fig 9.18.

```
1  functional speci  cation
2    provides c, m;
3    requires B.d, B.k;
4    data type
5      c : constrained x : integer (x >= 0);
6      m : constrained x : integer (x >= −1 and b <=1);
7    property req_source 1 "W. Miller";
8    property author "D. Cunningham";
9    attributes counter : c = 0;
10     modcount : m = −1;
11   inv c < d;
12     modcount < k;
13
14   operation iC()
15     statements
16       counter = counter + 1
17   end operation counter
18
19   operation iMC()
20     statements
21       /*
22        if modcount over  ows,
23        it starts again with −1
24       */
25       modcount = modcount + 1
26   end operation counter
27 end functional speci  cation
```

9.2.8 Weaving Server Components Connected to Aspects

Similarly to embedded components, server components (cf. Section 7.9) allow the decomposition of aspects and the delegation of responsibilities. A server component *C* is connected by an association *R* to the aspect *A* and makes it possible to share a common state and common data amongst all crosscut modules.

Weaving Semantics

In order to get a correct model at the end of the weaving process, the aspect-oriented model containing server components must fulfill several preconditions before a weaving can be performed. The weaving of server components consists of three steps. First, each association *R* between *A* and *C* is cloned for each end target module. Thus, each end target module is connected by a clone of *R* with *C* after the weaving. Second, the role names of each cloned association are adapted in order to be unambiguous. Third, the behavior description and the scenario nodes contained

in the target modules and in the server component may refer to the role names of the cloned associations. These references must be adapted accordingly.

Illustration of the Weaving Semantics

The weaving semantics for server components is illustrated by Figure 9.19. Model t_0 shows the aspect-oriented model where the aspect *A* is connected to a server component *C*. *A* crosscuts two components *M1* and *M2*. The right hand side of the figure shows the woven model t_n.

Preconditions. Before executing the weaving, a model containing server components must satisfy two preconditions:

a. The behavior of the server component must use the concurrency control mechanism sketched in Section 5.2.3 and discussed in detail in [Meie09a].

b. The behavior description of the server component must use the redundancy reduction mechanism also sketched in Section 5.2.3 and elaborated in [Meie09a].

Precondition (a) must be fulfilled because after the weaving, several end target modules M_i $(0 < i \leq n)$, crosscut by the aspect *A*, access *C* concurrently. Without a concurrency control, this may cause a race condition. Thus, all behavioral elements of *C* providing a service for the crosscutting behavior in *A* need to be part of a *critical section*. In the woven model, a message m_c initiates the provision of a service and is sent from one of the crosscut modules M_i. To indicate the start of a critical section, this message m_c has to be marked as *queued* in the corresponding receive part in the aspect. The end of the critical section has to be indicated by a transition leading back to the state where the transition receiving m_c is originated.

The satisfaction of Precondition (a) is illustrated by the server component *C* in model t_0 of Fig. 9.19. The statechart in *C* provides a service for the aspect *A*. The message initiating the provision of the service is *x()*, therefore it has to be marked as *queued*. The critical section consists of the state *X* and the corresponding in- and outgoing transitions.

Precondition (b) deals with the avoidance of redundancy in behavior descriptions. It is strongly intertwined with the problem of the concurrency control and emerges also when using server components [Meie09a]. If the model were woven without the redundancy reduction mechanism, the behavior describing the service in *C* would have to be replicated for each communication channel between a service-consuming component M_i and the server component *C* . This is due to the fact that the communication channels have to be specified statically in ADORA, which in turn causes redundancy. However, introducing redundancy for handling concurrent access can affect the understandability of the model [Meie09a]. Therefore precondition (b) must be fulfilled by using the proposed mechanism. This mechanism allows the message properties of m_c to be stored, and in turn, allows them to be used to dynamically determine the answer channel of the message when sending the result back to the service consumer.

The satisfaction of precondition (b) by Fig. 9.19 is illustrated by model t_0. The behavior of the server component *C* receives one message *x()* initiating the provision of a service. This message

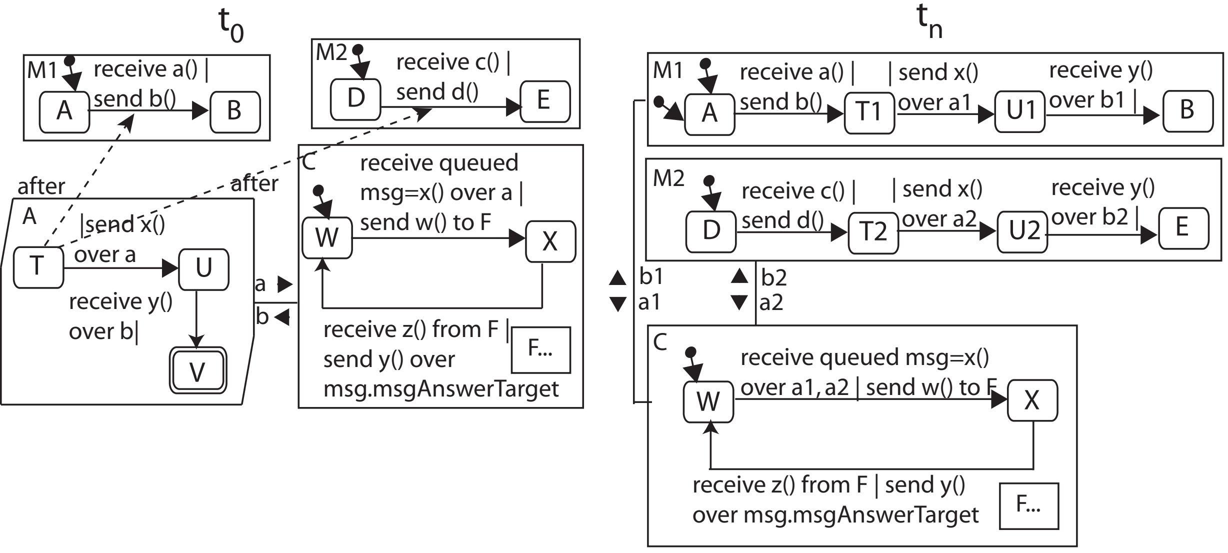


Figure 9.19: Illustration of the weaving semantics for server components.

is received by the transition between the state *W* and *X*. The message *x()* is sent from the behavior chunk contained in *A*, and therefore, it is sent from any crosscut component M_i in the woven model. The answer message is *z()* which is sent back to *A* by firing the transition between *X* and *W*. For using the redundancy reduction mechanism, the incoming message *x()* must be stored in the variable *msg*.[20] Furthermore, the answering channel must be determined dynamically by the expression *msg.msgAnswerTarget* in the *send* statement of the second transition.

Violating Precondition (a) or (b) can lead to a woven model which has an unpredictable behavior when being executed by a simulation. Nevertheless, there is no automatic check of these preconditions, thus, the modeler is responsible for ensuring that they are satisfied.

Weaving associations. Before the associations between an aspect *A* and a server component *C* can be woven with the target module M_i, all join relationships between *A* and M_i, the crosscutting statecharts, the crosscutting scenariocharts, and the embedded components must have been woven.

Then, each association R_l (where $0 \leq l < d$, where $d \in \mathbb{N}_0$) connecting *A* and *C* is replicated for each end target module M_i. The source-to-target role $r_{l,s}$ and target-to-source role $r_{l,t}$ attached to the cloned association are also replicated.

Associations between aspects and server components are handled as single instance elements when being woven. Thus, each of them is only replicated once, no matter if there are multiple join relationship paths to the target module.

The resulting cloned association R_l^i connects the target modules M_i with the server component *C*. After the weaving, the original association R_l between *A* and *C* is deleted.

[20]This variable is declared in the functional specification of *C*.

Figure 9.19 illustrates the weaving of the associations. The association between the server component *C* and the aspect *A* in t_0 is replicated two times. In the woven model, the first association clone connects *C* with *M1*, the second clone connects *C* and *M2*, as shown in model t_n.

Changes in the behavior of the model Each cloned association R_l^i between M_i and *C* has a source-to-target role $r_{l,s}$ and a target-to-source role $r_{l,t}$. Just using these role names in the model results in ambiguities. For example in Fig. 9.19, if the associations between *A* and *C* were just cloned when the model is woven, the two resulting associations between *M1* and *C*, and *M2* and *C* would have two roles *a* and *b* each. These two roles would be referenced by the send and the receive statements in the behavior description of *A* and *C*, to send and receive messages over the corresponding communication channels. However, just using references to *a* and *b* in the behavior description of *C* would be ambiguous and may lead to side effects. Therefore, apart from the cloning of the associations, the role names of each association R_l^i and the corresponding references to them must be adapted:

i. The role names $r_{l,s}$ and $r_{l,t}$ have to be renamed uniquely for each cloned association R_l^i.

ii. The corresponding role names referred to in the behavior description of *C* have to be adapted.

iii. The corresponding role names referred to in the woven crosscutting behavior of each M_i have to be adapted.

The adaptation stipulated in (i) is achieved by renaming the roles $r_{l,s}$ and $r_{l,t}$ uniquely for each R_i^l to the roles $r_{l,s}^i$ and $r_{l,t}^i$, where i may be, for instance, a unique index that is prepended or appended to the labels of role name.

The result of this step is illustrated in Fig. 9.19. The role labels are renamed uniquely. In the example, the disambiguation is done by adding a unique index to the original role names *a* and *b*. The resulting association role names are *a1*, *b1*, *a2*, and *b2* shown in model t_n of Fig. 9.19.

For the adaptation of the server component's behavior[21], each occurrence of the role label $r_{l,s}$ in the *receive* statements of the behavior has to be replaced by a comma-separated list $r_{l,s}^x, \ldots, r_{l,s}^y, \ldots, r_{l,s}^z$ comprising all receiving roles contained in $r_{l,s}^i (0 \leq i < n)$ of all cloned associations R_l^i. Each occurrence of $r_{l,t}$ in the receive statements of the behavior has to be adapted accordingly.

In Fig. 9.19, the result of this weaving step is also exemplified. Each occurrence of the association role *a* or *b* in the reception part of a transition in *C* is replaced by a list of roles comprising all corresponding role names of the cloned associations between *C* and *M1* and *C* and *M2*, respectively. There is only one reception part in the model t_0 of Fig. 9.2.1 which comprises the reference to *a*. In t_n, this role is replaced by the list of roles *a1, a2*. This list denotes the channels over which the messages can be received, either from *M1* or from *M2*. The

[21]The adaptations do not only affect the behavior but also the transform expressions in scenario nodes. However, for the sake of simplicity the example model demonstrates the problem with the behavior description only.

answer channel *b* for the answering message *z()* is dynamically determined by the expression *msg.msgAnswerTarget*. However, this does not need any adaptations when weaving.

Furthermore, in the last step of weaving a server component, each occurrence of the role labels $r_{l,s}$ and $r_{l,t}$ in a target module M_i has to be substituted by the corresponding role name $r^i_{l,s}$ and $r^i_{l,t}$, respectively. This is illustrated by Fig. 9.19. All occurrences of *a* and *b* in the *send* and *receive* statements are replaced by the corresponding roles. In the component *M1*, each occurrence of role *a* is replaced by *a1* and each occurrence of role *b* by *b1*. The roles in *M2* are replaced correspondingly by *a2* and *b2*.

The weaving semantics of server components is also exemplified by the library system example in Fig. 9.3 and Fig. 9.4. Figure 9.3 comprises the server component *Authorization* connected to the aspect *Authentication* with the two association roles *Authenticate* and *Authorize*. In Fig. 9.4, the roles are replicated and renamed accordingly. The references to the role contained in the crosscutting behavior as well as the behavior in the *Authorization* component are adapted correspondingly.[22]

9.2.9 Solving Naming Con icts, Handling Context Mappings, and Adjusting Scope

The description of the weaving semantics in the previous sections does not cover three issues: *naming con icts* caused by the weaving, the *mapping of context variables*, and adjusting the *scope of message arguments*. All three problems as well as how they are handled during the weaving are sketched in the following.

Naming con icts Naming con icts occur if the woven elements, such as variables, states, components, etc. are named like elements that already occur in the target module. The transformation must avoid naming con icts, because they may result in ambiguities and malformed models, which may result in a decreased understandability. The problem is illustrated in Fig. 9.20 (a). The left hand side shows the aspect-oriented model t_0, the right hand side the woven model t_n. A naming con ict is caused when weaving by the state *A* of the aspect. When weaving this state, its name con icts with the name of an existing state in the target component. The problem is solved by renaming the con icting state names in a unique way. Any references to the changed names must also be adapted accordingly.

Context mapping and scope extension A join relationship of an aspect can define a context map (cf. Section 7.6) which consists of entries that are composed of a left and a right hand side. The left hand side contains a variable name occurring in the aspect and the right hand side an expression that is composed of element names appearing in the target module of the aspect. The weaving process must substitute every occurrence of such a left hand side variable with the right hand side expression.

[22]Note that there are also other end target modules of the aspects the *BookAdministration*. However, they are not shown in Fig. 9.3 and Fig. 9.4. Their Behavior is adapted correspondingly.

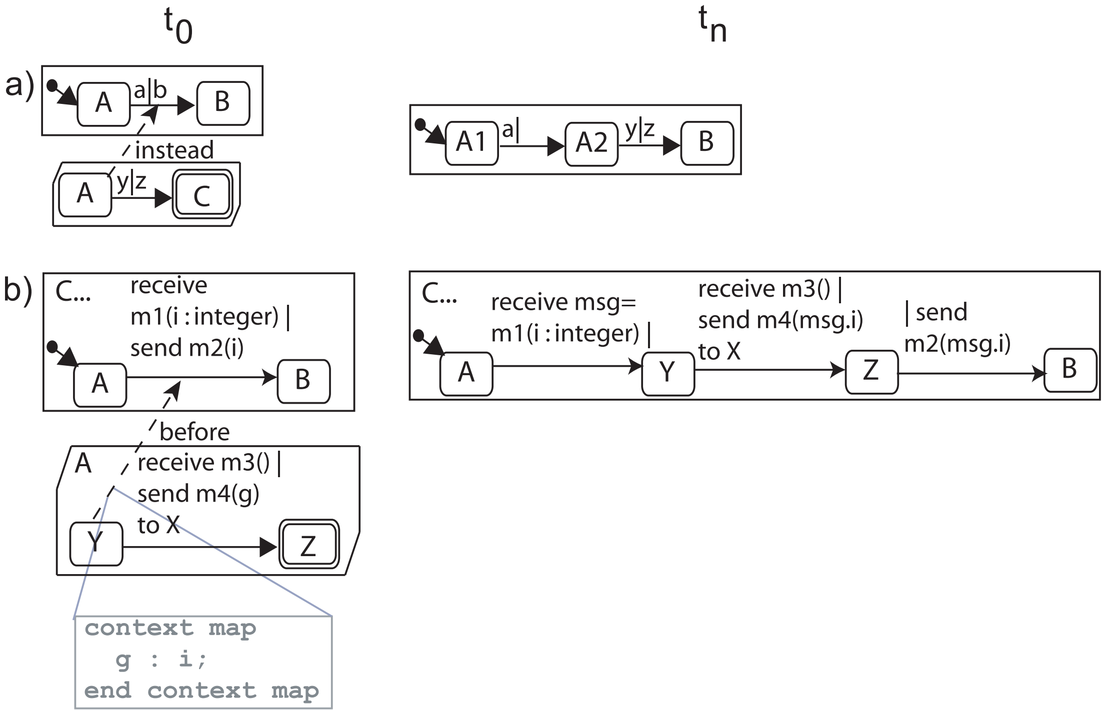


Figure 9.20: Example for the handling of naming con icts, the weaving semantics of a context map, and the scope extension of message arguments. Fig. (a) shows the problem of a naming con ict and how to handle it. The model in (b) illustrates the weaving semantics of a context mapping and the scope extension of a message parameter.

Apart from the context mapping, the weaving process must take care of the message arguments which are accessed by the action part of the transition. They must still be accessible after the weaving.

Figure 9.20 (b) illustrates the weaving semantics for the context mapping as well as for the scope extension of the message argument. In model t_0, an aspect *A* crosscuts a target component *C*. The join relationship defines a context map (shown in the gray box) which maps the term *g* to *i*. Element *i* is a parameter of the message *m1* and has therefore a scope which is restricted to the transition. Parameter *i* is also referred to by the action part of the transition.

The transition is split during the weaving, because the join relationship has a *before* semantics, and therefore, the receive part is separated from the action part. The weaving process must ensure that the variable *i* stays visible for the action part. Moreover, it must take care of the substitution of the proxy variable *g*. Therefore, the weaving process has to extend the scope of *i*, which is achieved by employing the message storing mechanism presented in [Meie09a] and Section 5.2.3. The message *m1* is stored in the variable *msg*,[23] as the stored properties of a message allow reception of the parameters of a message [Meie09a]. Thus, the references to *i* are

[23]This variable is defined in the functional specification of *C*.

substituted by the term *msg.i*. Correspondingly, the context term *g* used in the aspect *A*, as well as the reference *i* in the action part of *C* are substituted by *msg.i*.

Handling the issues The weaving process must employ a symbol table which allows existing names of elements to be looked up. It must also provide a way to resolve the left hand side and the right hand side of a context mapping. As the solution to these problems is rather implementation specific, they are not described in detail here.

9.2.10 Post processing

After the weaving, the aspect modules must be removed to complete the weaving. This must be done in the final step of the weaving transformation. Figure 9.3 and Fig. 9.4 exemplify this post-processing. The aspect module of model Fig. 9.3 is removed in the woven model of Fig. 9.4.

9.3 Weaving Semantics Involving Partial Aspect-Oriented Elements

The weaving semantics discussed in Section 9.2 only deals with highly evolved aspect-oriented elements. However, the weaving of partial aspect-oriented elements may have a different outcome. Actually, the weaving semantics depends on the type of aspect-oriented element and whether the element is partial. The following cases can be distinguished:

i. A *partial* join relationship between a state of a behavior chunk and a transition extends the weaving semantics for the corresponding non-partial case presented in Section 9.2.1.

ii. A *partial* join relationship between a scenario node and another scenario node extends the weaving semantics of the corresponding non-partial case in Section 9.2.2.

iii. Any other case[24] of *partial* join relationships, except for cases (i) and (ii), have a weaving semantics which works on the informal model description of the elements.

iv. The weaving semantics of a partial join relationship between environment objects extends the weaving semantics presented in Section 9.2.6.

v. The weaving semantics of a partial aspect module extends the weaving semantics presented in Section 9.2.

Note that other partial elements, such as partial states of behavior chunks or partial scenario nodes of scenario chunks do not in uence the weaving semantics: they are woven as they are. The same happens with partial embedded components contained in an aspect. The weaving semantics for the cases (i)–(v) is discussed in the following sections.

[24]See also Fig. 7.6 in Section 7.6.

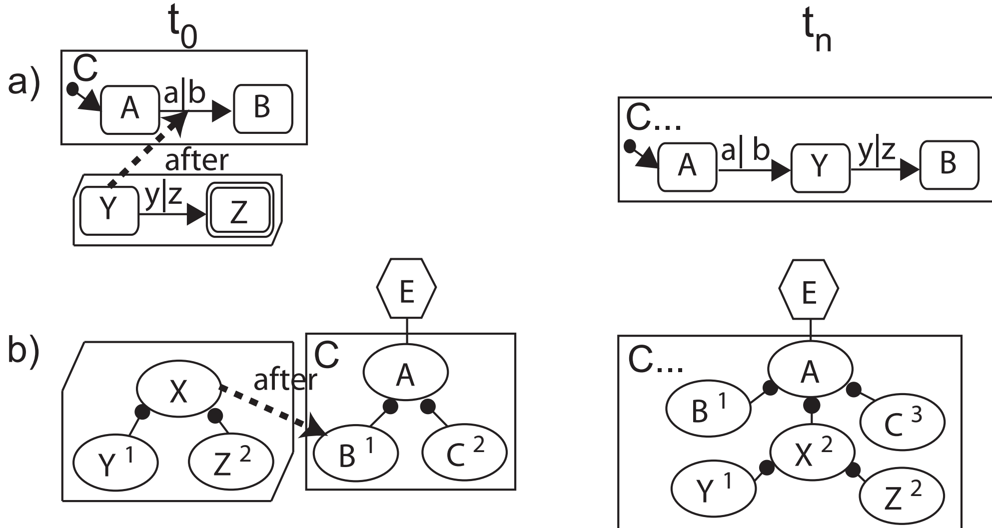


Figure 9.21: Illustration of the weaving semantics for an abstract join relationship connecting a behavior chunk with a transition and a scenario chunk with a scenario node, respectively.

9.3.1 Partial Join Relationship Connecting Scenario/Behavior Chunks with a Scenario/Transition

A join relationship between a *behavior chunk* and a *transition*, or between a *scenario chunk* and *scenario node* can be partial.

Weaving Semantics

The weaving semantics for non-partial join relationships, as presented in Section 9.2.1 and Section 9.2.2, also applies for partial join relationships connecting behavior/scenario chunks with transitions/scenarios. However, this semantics needs to be extended due to the fact that the abstract join relationship is not finally evolved. When weaving the partial join relationship, it becomes a part of the target module, and therefore, its partiality is propagated to the target. Consequently, the partial property of the target module is set. Note that partial join relationships connecting behavior/scenario chunks with transitions/scenarios may also form part of a join relationship path (cf. Section 9.1). Consequently, they have an in uence on the weaving of single instance elements.

Illustration of the Weaving Semantics

Figure 9.21 illustrates this weaving semantics. The left hand side shows the aspect-oriented model t_0 and the right hand side the woven model t_n. Fig. 9.21 (a) shows the situation where an abstract join relationship connects a behavior chunk with a transition, Fig. (b) exemplifies a similar case for a scenario chunk. The target module in the resulting model t_n is partial.

9.3.2 Partial Join Relationship Connecting Other Elements

Apart from partial join relationships between behavior/scenario chunks and transitions/scenarios, there may be a partial join relationship between other elements, such as aspects and associations, behavior chunks and components, or aspects and components, etc. (cf. Section 7.6).

Weaving Semantics

This kind of abstract join relationship may occur at earlier stages of an aspect and describes its unfinished evolution and the corresponding partial join relationship. The partial join relationship denotes in this case a fuzzy relationship between the aspect and the crosscut element, which cannot be handled by the weaving semantics delineated in Section 9.3.1, Section 9.2.1 and Section 9.2.2. Due to its fuzziness, such a partial join relationship cannot be a part of a join relationship path. As a consequence, single instance elements, such as crosscutting statecharts, cannot not be woven due to the occurrence of this kind of join relationship.

However, the information contained in the aspect from which the join relationship originates is highly valuable for the reader of the woven model. It consists of crosscutting model parts that are already modeled, as well as informal descriptions (cf. Section 6.1.3) which may contain information about the future evolution of the aspect. This information must not be lost after the weaving.

To avoid such a loss of information, the weaving process creates a textual representation of the aspect module and its parts, which also contains the informal description of the elements. Both, the textual model representation, as well as the informal description are injected into the informal description of the target element. The ordering keyword (*before*,*instead*,*after*) of the abstract join relationship is used to determine the precedence of inserted textual description when more than one aspect impacts the same target. This allows the reader of the informal comment in the woven model to determine the precedence of the aspect-oriented requirements. However, the priority is not taken into account, i.e., competing textual description with the same ordering can have an arbitrary order. Apart from the woven textual representation of the aspect, the partial property of the target element is set to indicate the incompleteness injected by the crosscutting concern.

Illustration of the Weaving Semantics

Figure 9.22, 9.23 and 9.24 exemplify the weaving semantics for three different situations where a partial join relationship is involved. The left hand side contains the aspect-oriented model t_0, the right hand side the woven model t_n.

Fig. 9.22 shows a situation where a partial join relationship between an aspect and a component is given. The gray boxes denote the informal description contained in the aspect *A* and the component *C*, respectively.[25] The component *C* in the woven model t_n shows the informal description and the textual model representation of the aspect inserted *before* the informal de-

[25] Note that the informal description is usually not visualized in the graphical representation of the model.

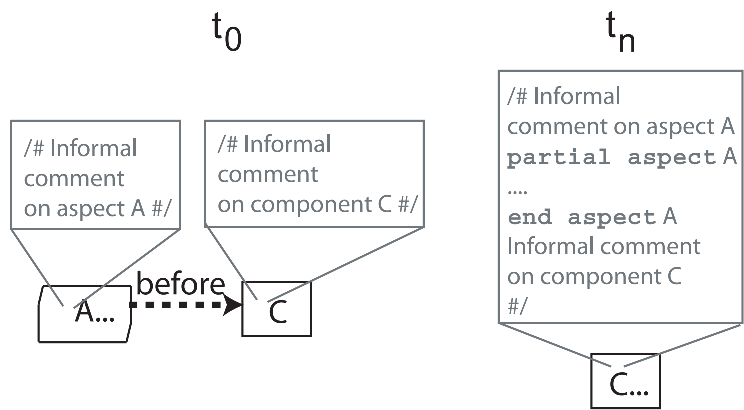


Figure 9.22: Illustration of the weaving semantics for a join relationship between an aspect *A* and a component *C*. The left hand side shows the aspect-oriented model t_0, the right hand side the woven model t_n. The gray boxes denote the informal description of the corresponding elements. The informal description of the aspect and its textual representation are woven *before* the informal description of *C*.

scription of the component. Moreover, the component is set to partial (indicated by the ellipsis trailing the name of *C*).

Fig. 9.23 shows a similar situation where an aspect is the source and an association the target of the join relationship. The *instead* ordering of the join relationship causes the replacement of the association's informal description by the textual representation of the aspect and its informal description. Moreover, the association is set to partial. Correspondingly, in Fig. 9.24 a situation is shown, where a partial *after* join relationship connects a behavior chunk with a component.

Partial join relationships impacting the same target are handled according to the ordering keyword which is used by the topological sort algorithm (cf. Section 9.1) for getting the correct weaving order of the formal comments. However, the priority is not taken into account for the resolution of the competition between join relationships having the same ordering. Furthermore, the handling of multiple partial join relationships with an *instead* ordering needs special attention. Only the first one is handled with an *instead* ordering, the others are woven with an *after* ordering, as otherwise only the last *instead* would be injected into the informal description of the target element.

9.3.3 Weaving Semantics of Partial Join Relationships Between Environment Objects

A join relationship between two *environment objects* denotes the crosscutting of a role in the environment of a system with another role. Such a join relationship can be partial and describes in this case an uncertainty about whether the join relationship is finally evolved or not.

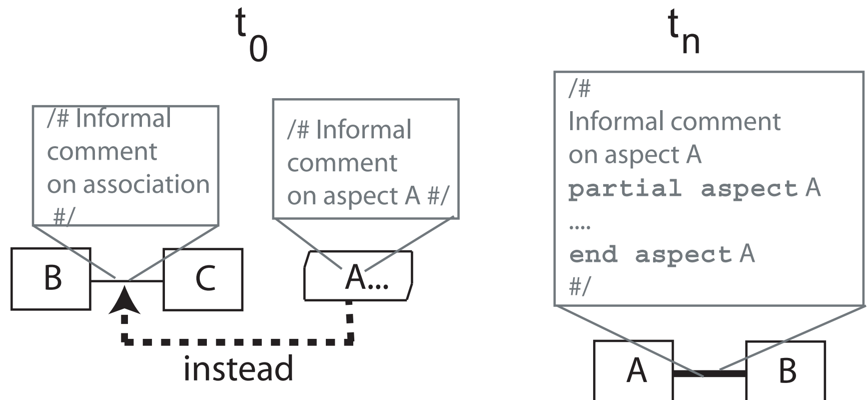


Figure 9.23: Illustration of the weaving semantics for an abstract join relationship between an aspect *A* and an association. The informal description of *A* and its textual representation are woven *instead* the informal description of the association.

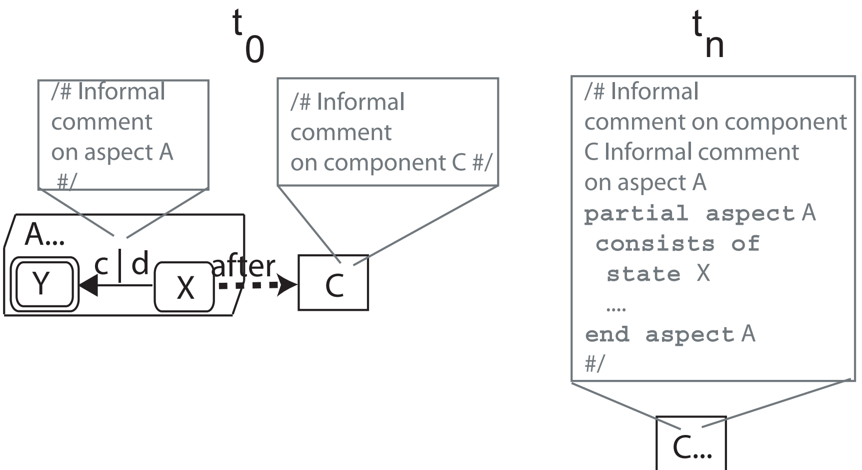


Figure 9.24: Illustration of the weaving semantics for the abstract join relationship between a behavior chunk in aspect *A* and a component *C*. The informal description of *A* as well as its textual representation are appended to the informal description of *C* in the woven model.

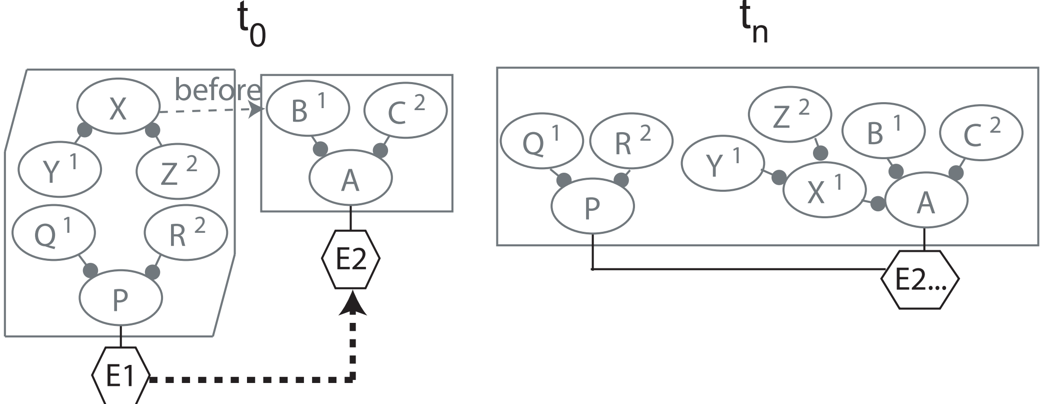


Figure 9.25: Illustration of the weaving semantics for an abstract join relationship connecting two environment objects. The target environment object must be partial after the weaving.

Weaving Semantics

The weaving semantics for environment objects presented in Section 9.2.6 applies also in the case where the join relationship is partial. However, it has to be extended. The partial indicator of a join relationship between the environment objects denotes its unfinished evolution. When weaving, the corresponding uncertainty has to be propagated to the target environment object. Thus, after the weaving, the target environment object has to be partial, which means that it is not yet certain whether the crosscut environment object also comprises the injected crosscutting role or not.

Illustration of the Weaving Semantics

Figure 9.25 illustrates this weaving semantics. The left hand side shows the aspect-oriented model t_0 and the right hand side the woven model t_n. After the weaving, the target environment object *E2* is set to partial, because the corresponding join relationship is partial.

9.3.4 Weaving Partial Aspects

An aspect module can be partial, which indicates its non-finished evolution. The corresponding fuzziness must also be propagated to the crosscut modules. As a consequence, the target module of the aspect is set to partial. Furthermore, the aspect's informal description is appended to the informal description of the target module, as it may contain valuable additional information about the future evolution of the aspect.[26] This weaving semantics only applies in the case where the aspect is the origin of a join relationship between a behavior/scenario chunk and a transition/scenario. However, it does not matter if the join relationship is partial. Thus, the present weaving

[26]Note that the ordering whereby the informal description is injected into the target module is *neither* in uenced by the *ordering keyword nor* by the *priority* of the join relationship.

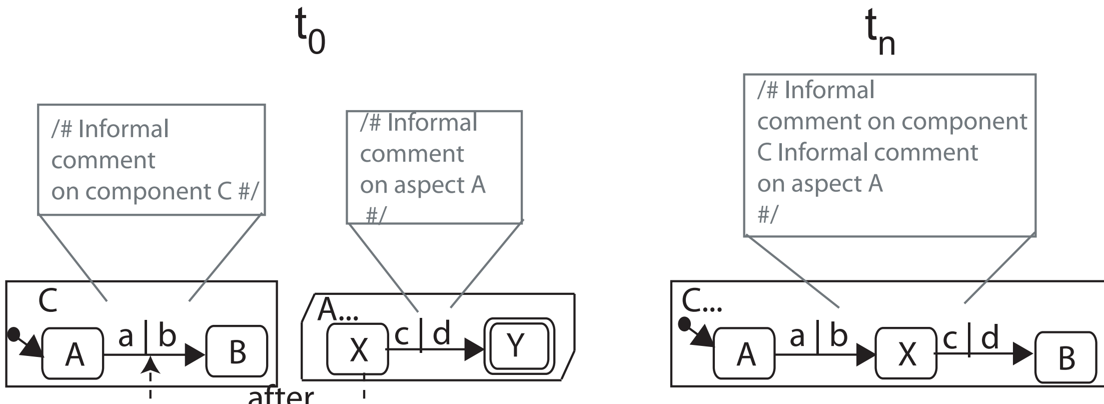


Figure 9.26: Weaving semantics of a partial aspect. When weaving a partial aspect module, the target module is set to partial and the aspect's informal description is appended to the informal description of the target module.

rule only applies when the aspect is set to partial in the cases described in Sections 9.2.1, 9.2.2, and 9.3.1. In contrast, the present weaving rule does not apply when the aspect is partial and the join relationship connects a behavior/scenario chunk with a transition/scenario, which is already covered by Section 9.3.2.

Illustration of the Weaving Semantics

Figure 9.26 illustrates the weaving semantics for a partial aspect module. The left hand side shows the aspect-oriented model and the right hand side the woven model. The gray boxes denote the informal descriptions of the corresponding elements. After the weaving, the target module is partial and the informal description of the aspect is appended to the informal description of the target module.

9.4 Formal Weaving Semantics

This section covers how the previously presented informal weaving semantics can be formally described. For this reason, the most obvious techniques to specify the weaving semantics are evaluated brie y in Section 9.4.1. In Section 9.4.2, the weaving process is revisited and the specification schema which is used to define the transformation rules is presented. Finally, in Section 9.4.2, an example for the formal description of a small part of the weaving semantics is presented.

9.4.1 A Description Schema for the ADORA Weaving Semantics

The detailed description of implementation of the weaving transformation is not at the focus of the present work. Nevertheless, a concise but abstract specification of it should be given. Techniques which can be potentially used to describe the weaving semantics in a precise and formal way are brie y assessed in this section. The following criteria are used to select the most appropriate method:

- The method must support a precise and unambiguous description of the weaving semantics.
- It should be simple to use and easy understandable.
- The resulting weaving transformation specification should be compact.
- It should abstract from unimportant facts and implementation details.
- The technique should be easily applicable to the EBNF-based model description of the ADORA language.

There are various means to describe the semantics of the operations executed during a weaving transformation. The methods either originate from the field of the formal description of programming languages or from the field of model transformations.

Graph Transformations

Graph transformations [Roze97, Heck06][27] deal with the representation of computation states as graphs. Further steps in the computation of a system are represented as transformation rules on these graphs. A transformation rule consists of an *original graph* L and a *replacement graph* R which are instance graphs describing a particular pattern *before* and *after* a transformation, respectively. L is also called left hand and R the right hand side of the transformation rule.

Formally, a graph transformation is a homomorphism defining a rule $p : L \rightarrow R$. In order to transform a given graph G with the rule p, the left hand side L has to match the graph G. R describes a pattern, what L looks like *after* executing the transformation operation. Thus, applying $G \mapsto (G \backslash (L \backslash R) \cup (R \backslash L))$ results in H which is the transformed graph G. Furthermore, additional constraints on the described transformation need to be defined to ensure that the resulting graph is well-formed [Heck06]. Several rules can be executed in sequence, which allows the creation of complex transformations.

Graph transformations especially became of interest in the field of model-driven software development [Schm06, OMaG03a], where user-specified transformations in the application domain play a key role in the development process of a software. Graph transformation rules are especially well suited in this case, as they can be expressed by visually modeled instance graphs that are easier for users to understand than more abstract concepts such as the *Axiomatic Semantics* which is described below.

[27] Graph transformations are sometimes called graph rewriting systems [Schu98].

There are two reasons why a graph transformation approach is not considered for describing the ADORA weaving semantics formally: first, the pattern descriptions with example instances may end up in a large number of rules which are hard to handle and which may result in complex transformation systems, especially for semantically rich graph languages. Thus, formulating the ADORA weaving semantics with graph transformations may be tedious. Second, for describing the left hand and the right hand side of a transformation rule, a particular pattern language is needed. It must be based on the language that is transformed and must introduce some kind of wildcards instead of concrete language elements. Therefore the realization of such a language is beyond the scope of the present work.

However, a graph transformation approach may be of interest for ADORA when allowing the user to define their own transformations. These may either be *domain-speci c* weaving transformations, or *other types of model transformations in the application domain*. Therefore, graph transformations in ADORA may be a future research topic.

Denotational and Structured Operational Semantics

Denotational semantics as well as structured operational semantics (SOS) are approaches which originate from the formalization of the semantics of programming languages.

The *denotational semantics* was originally introduced by [Scot71] (cf. also [Tenn76]). It concerns the construction of mathematical objects (called denotations) which describe the meanings of expressions in the specified languages. For this purpose, the syntactical elements of a language are strictly separated from the mathematical structures representing the meaning of the language. The syntax elements are then mapped to the semantics domain. Examples can be found in [Tenn76].

The *structured operational semantics* approach was originally mentioned in [Plot81]. The approach is based on a transition system $(S, A, \rightarrow)$, where S denotes a set of states, A a set of actions, and a relation $\rightarrow: S \times A \times S$ representing the transition in the system. It describes the transition from an original state $s \in S$ to a subsequent state $s' \in S$ if the action $a \in A$ occurs. S consists of the set of statements of a language L. For each statement s, there exists an action which reduces the statement s to s'. In turn s' may be reduced again, resulting in a concrete value at the end of all the steps.

Both denotational semantics as well as the SOS approach are designed for describing the semantics of computer languages. Using one of them for specifying the ADORA weaving semantics formally, would require the specification of a formal syntax for describing the transformation of the ADORA models. However, as stated in the section on graph transformations, specifying such a language is beyond the scope of this work.

Algorithmic Description

An obvious way to describe the semantics of a transformation is the use of an algorithmic description. However, pure algorithmic descriptions can anticipate an implementation and, thus, they can result in suboptimal solutions. Furthermore, they can get rather detailed, complex and unclear. Hence, pure algorithmic descriptions are rejected for the present work.

Axiomatic semantics

Axiomatic semantics is an approach based on mathematical logic that can be used to prove the correctness of computer programs for a given specification [Hoar69]. Several other approaches, such as specification of abstract datatypes [Gutt77] and design by contract [Meye92] are based on the axiomatic semantics approach. The specification consists of two assertions given by a precondition P which has to be satisfied in order that the given program C executes and results in a satisfied postcondition Q. P and Q are predicates usually expressed in a first-order predicate logic.

Pre- and postconditions allow the semantics to be formally specified without using an imperative algorithmic description, i.e., the operation is specified as a black-box. This allows an implementation-independent specification of the weaving operations. Furthermore, it is compact and more understandable than bloated algorithms. However, there are parts which are difficult to express in a first-order logic.

9.4.2 Description Schema of the ADORA Weaving Semantics

As discussed above, denotational semantics, SOS and graph transformations are inadequate, and therefore, they are not considered for the formal description of the weaving operations. The formal description employs rather an axiomatic semantics which is complemented with algorithmic specifications. It is in some ways similar to the refinement calculus used in [Xia04]; however, it does not use the same syntax for the description of the rules. The schema and the types of the operations used are discussed in the following.

Weaving Operations Described by Pre- and Postconditions

An axiomatic semantics uses pre- and postconditions to describe the meaning of an operation ϕ_k. Pre- and postconditions are described for each transformation operation in a fixed schema. First, the given elements are delineated. Second, the precondition is defined. It is a predicate that specifies what must be satisfied in order that ϕ_k executes properly. Third the postcondition is specified. It describes the changes in the transformed syntax tree, i.e., the semantics of the operation ϕ_k. The predicates of the pre- and postconditions use functions, which are usually specified as algorithms, returning specific properties of the syntax tree (see also Appendix G).

Weaving Process Revisited

As discussed in Section 9.1, a weaving transformation can be distinguished into two major phases. In the first phase, each join relationship, except the ones between environment objects, are processed by weaving their constituting elements. A join relationship is removed after it has been woven. In the second phase, the single instance elements, such as crosscutting statecharts, etc., are woven.

For referring to the major steps in the weaving process, the letters g, m, p, q, u, v, w, and n are reserved as the indices. Step g denotes the weaving of a join relationship, step m the weaving

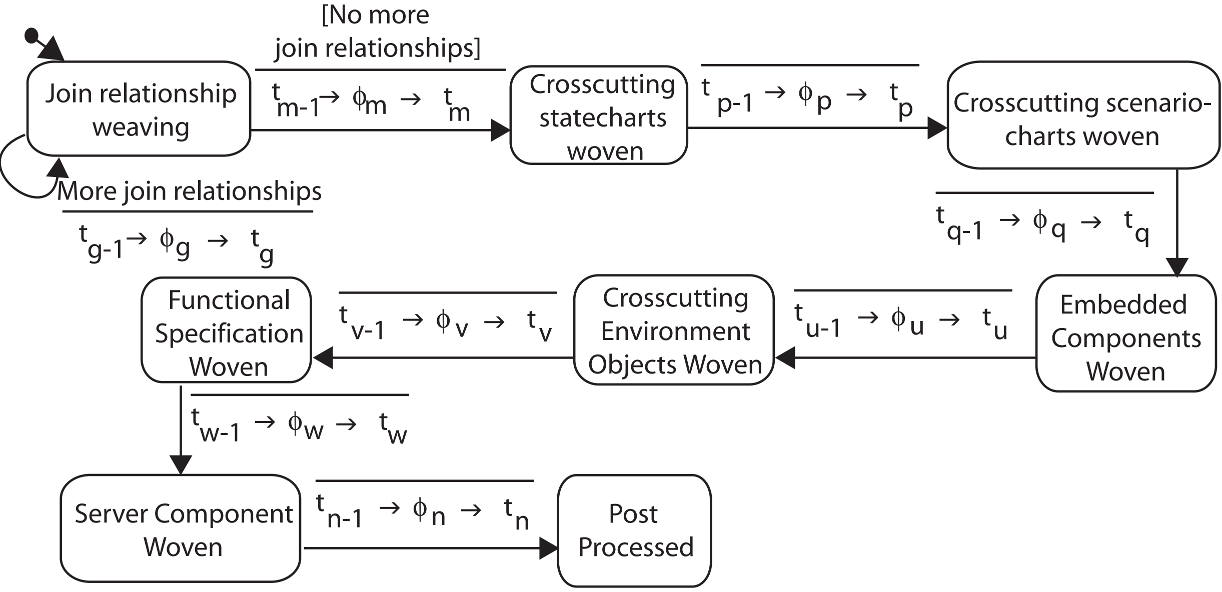


Figure 9.27: Illustration of the weaving process. The weaving process is described as statechart-like diagram.

of the statecharts into the end target modules, step p the weaving of crosscutting scenariocharts, step q the weaving of embedded components, step u the weaving of crosscutting environment objects, step v the weaving of the functional specification, step w the weaving of the server components, and step n the post processing of the model. Moreover, there may be intermediate steps between major steps.

For example, ϕ_g denotes the operation which weaves a join relationship, ϕ_m an operation which weaves the crosscutting statecharts of the model, and so on. The model before the execution of ϕ_g is given by t_{g-1}, the model after its execution by t_g.

Figure 9.27 depicts the actual weaving process as a statechart. For example, the operation ϕ_g takes the model t_{g-1} as input and creates a new model t_g. The operation converts the original model t_{g-1} to the resulting model t_g, which is expressed by the notation $t_{g-1} \rightarrow \phi_g \rightarrow t_g$ in the action part of the transition.

Weaving operations may depend on an intermediate state and the corresponding data structures used in the previous transformation steps. Therefore, it is sometimes necessary that a weaving operation is able to access the data structures used by another weaving operation. An element from another step can be accessed by referencing it with the unique index of the weaving step. For example, a term called $rootAssociations_h$, where h denotes the unique index, allows the accessing of the data structures represented by the term $rootAssociation$ at the weaving step h.

Note also that the presented formal description abstracts from the handling of naming con- icts, the handling of the context mappings, and the adjusting of the name space of message arguments. They are not included, as this would result in a more complex formal description,

which is not helpful in the understanding of the weaving rules. These issues can be dealt with at the implementation stage.

9.4.3 Illustration of the Formal Weaving Semantics

The formal definition of the weaving semantics is exemplified in the following by a small excerpt showing a part of the axiomatic specification for the weaving of behavior chunks (cf. Section 9.2.1). The formal description uses functions that allow the retrieval of properties of ADORA syntax trees (cf. Chapter E of the Appendix).

Given elements

The formal description usually first introduces a set of given elements, i.e., terms that are used to define the pre- and postconditions of the operation described. The following example illustrates the given elements. They are used for formally describing the weaving semantics of behavior chunks.

The operation which weaves behavior chunks is ϕ_g, the model t_{g-1} is the input, and t_g the output model.[28] The term $jrlist$ defined in Formula (9.2) denotes the topologically sorted list of join relationships contained in t_{g-1}. The current join relationship j is defined by the formula given in (9.3). Furthermore, the entry point of the behavior chunk is given by the term $entryState$ specified in Formula (9.4). A short hand $targetTransition$ for the target of j, i.e., the transition which is crosscut by the behavior chunk, is defined in Formula (9.5).

$$jrlist = topologicalJrSort(t_{g-1}) \tag{9.2}$$

$$j = firstJr(jrlist) \tag{9.3}$$

$$entryState = source(t_{g-1}, j) \tag{9.4}$$

$$targetTransition = target(t_{g-1}, j) \tag{9.5}$$

Precondition of ϕ_g.

In the next step, the precondition of the operation is defined. It describes what conditions have to be satisfied in order that the weaving operation can be executed properly. The following example illustrates this by means of the precondition for the weaving of a behavior chunk.

For the correct working of the operation ϕ_g, the precondition in (9.6) must be satisfied. As described in this predicate, the join relationship must be connected from a state or a component to a transition.

[28] The model ϕ_{g-1} is either an intermediate model resulting from a previous weaving of a join relationship, or it is the initial aspect-oriented model.

$$\begin{aligned}&\big(type(entryState) = StateDefinition \vee \\ &type(entryState) = ComponentDefinition\big) \wedge \\ &type(targetTransition) = TransitionDefinition\end{aligned} \tag{9.6}$$

Postcondition of ϕ_g.

The postcondition usually consists of several predicates which are implicitly connected by a logical *And* operation. Apart from these predicates several helper terms may be defined. They are used to simplify the predicates. The following example illustrates this by means of a part of the postcondition for the weaving operation of a behavior chunk.

The terms defined in the formulae given in (9.7)–(9.11) are used to simplify the predicates in the following: $stateGroup$ is a set of syntax trees describing the nodes and the connections of the behavior chunk which will be woven by ϕ_g into the target module.[29] The expression $exitPoints$ specifies the set of exit points of the behavior chunk.[30] The target module $tModule$ denotes the aspect or component which contains the target transition $targetTransition$ after the weaving of the behavior chunk. The term $exitState$ denotes the target state of the crosscut transition.

$$stateGroup = findStateGroupMembers(t_{g-1}, \emptyset, entryState) \tag{9.7}$$

$$exitPoints = exitPoints(stateGroup) \tag{9.8}$$

$$tModule = identicalElement(t_g, targetModule(t_{g-1}, j)) \tag{9.9}$$

$$exitState = target(t_{g-1}, targetTransition) \tag{9.10}$$

The term $cloneMap$ defined by Formula (9.11) denotes a tuple which contains the mapping between the original states, components, and transitions of the behavior chunk and the corresponding clone copies. Two different cases have to be distinguished. In the case where the join relationship j has a *before* ordering, an additional state and transition have to be introduced apart from the clones of the other behavior chunk elements. This is done by the function $createAdditionalExitStateClone$. The resulting map contains a mapping between the exit point of the behavior chunk and an extra state. In the *instead* and *after* case, only the states, components, and the transitions without the exit point are cloned. Thus, in this case it does not contain a mapping between the exit point and an extra state.

$$cloneMap = \begin{cases} createCloneMap(t_{g-1}, stateGroup \backslash exitPoints, & \textbf{if } ordering(j) = \\ \quad createAdditionalExitStateClone(t_{g-1}, exitPoints)) & \quad \texttt{before} \\ createCloneMap(t_{g-1}, stateGroup \backslash exitPoints, \emptyset) & \textbf{if } ordering(j) \neq \\ & \quad \texttt{before} \end{cases} \tag{9.11}$$

[29] The (source) nodes also contain the transitions which form the state group.

[30] Even though it is a *set*, there can only be one exit point, as defined by the language constraint in Section 7.4.2.

The actual postcondition of ϕ_g consists of several predicates. The first predicate in (9.12) specifies that copies of the states and the corresponding transitions of the behavior chunk are found in the target module after weaving.

$$
\begin{aligned}
&\big(\forall x \in stateGroup \backslash exitPoints : \\
&\quad identicalElement(t_g, findClone(x, cloneMap)) \in parts(tModule)\big) \wedge \\
&\big(ordering(j) = \mathbf{before} \Rightarrow \forall x \in exitPoints : \\
&\qquad\qquad identicalElement(t_g, findClone(x, cloneMap)) \in parts(tModule)\big)
\end{aligned}
\tag{9.12}
$$

The predicate in (9.13) specifies how the cloned behavior chunk is connected to the crosscut behavior in the case that j is a *before* join relationship (cf. Fig 9.5 (a)). The extra state created in the clone map is part of the target module. It has an outgoing exit transition that has no condition part but the same action part as the crosscut transition. The exit transition is connected to the target state ($exitState$) of the crosscut transition. Furthermore, the crosscut transition is reassigned to the entry point of the cloned behavior chunk and its action part is deleted.

$$
\begin{aligned}
&ordering(j) = \mathbf{before} \Rightarrow \\
&\quad \big(\forall x \in exitPoints : \\
&\qquad \big(\forall y \in connections(findClone(x, cloneMap)) : \\
&\qquad\quad \exists z = identicalElement(t_g, y) \ : \\
&\qquad\qquad target(t_g, z) = identicalElement(t_g, exitState) \wedge \\
&\qquad\qquad actionPart(z) = actionPart(targetTransition) \wedge \\
&\qquad\qquad conditionPart(z) = \varepsilon \\
&\qquad \big) \\
&\quad \big) \wedge \big(\\
&\qquad \exists reassignedTransition = identicalElement(t_g, targetTransition) : \\
&\qquad\quad conditionPart(reassignedTransition) = \\
&\qquad\qquad conditionPart(targetTransition) \wedge \\
&\qquad\quad actionPart(reassignedTransition) = \varepsilon \wedge \\
&\qquad\quad target(t_g, reassignedTransition) = findClone(entryState, cloneMap) \\
&\quad \big)
\end{aligned}
\tag{9.13}
$$

Apart from the *before* case, there are the *instead* and the *after* that need to be described. Moreover, the case where multiple join relationships impact the same transition must also be specified. The full description for the weaving semantics of a behavior chunk is delineated in Section G.1.1.

Complete Formal Description of the Weaving Semantics

The full formal weaving semantics is specified in Appendix G. The weaving semantics of non partial elements can be found in Section G.1 and the one of the partial elements in Section G.2.4.

9.5 Weaving the Layout Information

The sections before discuss the weaving of the actual model information, but they do not delineate how the layout of a model is handled by the weaving process. The layout information of the crosscutting elements and the crosscut elements must somehow be merged when performing the weaving. One possible way to achieve this is to create a new layout using an automatic mechanism, such as the one described in [Eigl04]. However, this usually leads to a total rearrangement of all model elements, which ends in the loss of the model reader's bearings. As a consequence, the reader has problems in finding the link between the aspect-oriented and the corresponding woven elements in the resulting model, which in turn decreases the advantages gained from the weaving. Thus, premise (c) for the weaving, which is discussed at the beginning of Chapter 9, is violated.

The actual problem is caused by the destruction of the secondary notation, which is also known as visual layout of the user's mental map (cf. Section 5.1.4). At the time of writing, there were no model transformation or weaving approaches which satisfactorily handle the visual information of models.[31] In order to circumvent the problems that arise from using an automatic layout algorithm, the weaving process must try to create a fused visual representation that preserves the secondary notation as far as possible. A rudimentary, straightforward solution is presented in the following.

Approach. The approach is based on the following idea. Crosscutting elements such as behavior chunks or crosscutting scenariocharts, must be inserted at a position where the reader of the model expects those elements after the weaving. Consequently, the nodes of the inserted state groups and scenario groups, such as the states and scenarios nodes, should keep their absolute distance to each other.

Figure 9.28 (a) illustrates how this is achieved for behavior chunks. Before weaving a behavior chunk, the bounding box of the elements to weave is computed. It is indicated by a gray box in the figure and has the dimensions Δx and Δy. In an intermediate step, shown in model $t'g-1$, extra space of the size $(\Delta x, \Delta y)$ is allocated between the end point of the transition and the state *B*. The crosscut transition is the point where the reader of the model probably expects the behavior chunk to be inserted. For the allocation of the required space in the target module, the layout algorithm from [Rein07] (cf. Section 5.1.4) is employed.[32] When weaving a chunk into the target, it is inserted in the allocated space. The elements of the chunk preserve their absolute distance to each other, which is shown by the woven model t_g. This ensures that the

[31] Note that the weaving of the layout is also of interest in the field of model-driven software development, where model transformations play a key role.

[32] This algorithm is also used to implement the abstraction mechanisms of ADORA.

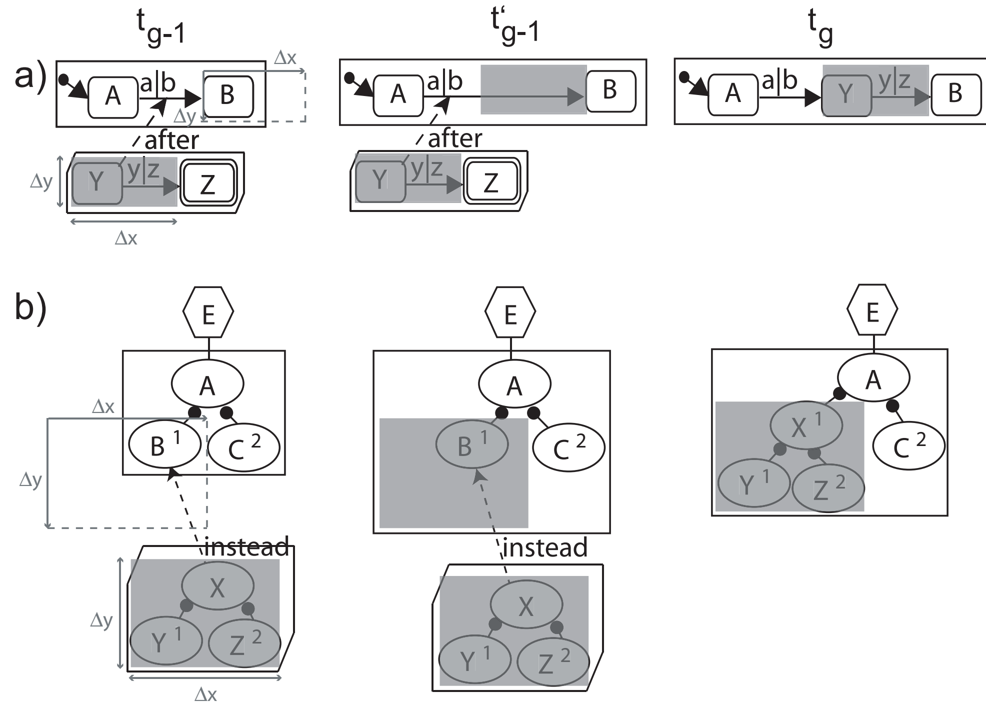


Figure 9.28: Model illustrating the visual weaving of behavior and scenario chunks.

inserted chunk keeps its size. Figure 9.28 (b) shows the corresponding steps for the weaving of a scenario chunk.

Apart from chunks, the layout for single instance elements such as crosscutting scenariocharts and embedded components must be computed appropriately. Figure 9.29 shows an example which illustrates how this is done for a crosscutting statechart.[33] As for chunks, the bounding box for the crosscutting statechart is calculated, which is shown by Δx and Δy in model t_0. Furthermore, the distance x and y of the bounding box from the left upper corner and the $height$ and $width$ of the aspect module are determined. The position where the crosscutting statechart is inserted in the target module is given by its relative position (x', y') from the upper left corner of the aspect, which is shown by the intermediate model t_1. Thus, the equations $\frac{x'}{width'} = \frac{x}{width}$ and $\frac{y'}{height'} = \frac{y}{height}$ apply.[34]

The required space is allocated with the dimension $(\Delta x, \Delta y)$ at the position (x', y') by the algorithm described in [Rein07]. Thereby, any occlusions between the allocated space and existing nodes are resolved. That means that the inserted allocated space in the model, where *G* and *B* are inserted later, is shifted away from *A*, *X*, and *B*. Then, the crosscutting statechart is inserted in the reserved area, which results in the model t_n. Correspondingly to this example, crosscutting scenariocharts and embedded components are visually woven.

[33] For crosscutting scenariocharts and embedded components, it is done analogously.
[34] The coordinate (x',y') is calculated by $x' = \frac{x}{width} width'$ and $y' = \frac{y}{height} height'$.

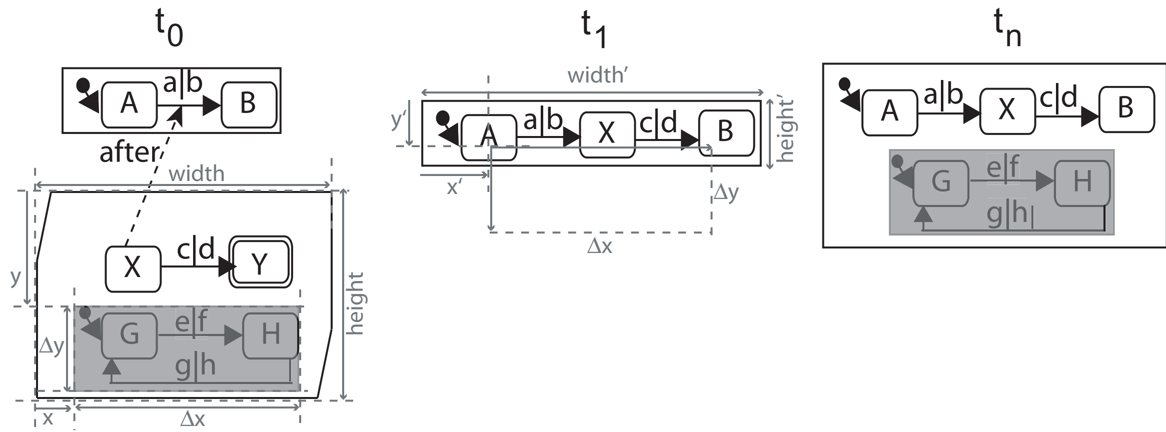


Figure 9.29: Model illustrating the visual weaving of crosscutting statecharts, crosscutting scenariocharts, and embedded components.

However, a counterintuitive layout may result when having multiple single instance elements, such as crosscutting statecharts, that are subsequently inserted into the same target module. This is due to the allocation of space and the resolution of occlusions. Elements which are occluded by the reserved space are shifted away. Hence, previously inserted single instance elements may also be shifted away and may finally be located at another position than expected by the reader of the model. To avoid this problem, the aggregated space of all single instance elements is allocated at once in the module and the single instance elements are inserted in this space together.

Apart from chunks and single instance elements, such as crosscutting scenariocharts, associations may also be woven, e.g. when weaving server components or environment objects. In this case, the ratio between left and top corner and the point where the association crosses the border of the shape is preserved.

Drawbacks of the approach. The presented visual weaving approach does not cover all problems. There may be situations which still lead to unclear models. The main issue emerges from the fact that elements, such as statecharts and scenario trees have a user-specified orientation. Suppose the model t_0 in Fig. 9.30 (a). The transition between state *A* and *B* in the component of the model t_0 has an orientation which is from right to left. In contrast, the transition of the behavior chunk in the aspect has a reverse orientation. Thus, when weaving t_0 with the presented approach, the resulting model t_n has an unclear visualization, as two different orientations are mixed in the same diagram. The effect is amplified when having various crosscutting elements for the same target with a various directions. The problem can also be observed for other elements, such as crosscutting scenario chunks, as for instance shown in the example of Fig. 9.30 (b), where a scenariochart with a particular orientation is crosscut by a scenario chunk with another orientation.

The presented visual weaving approach is only partially satisfactory. Especially models with a higher number of woven elements having a different orientation can become unclear. A proper

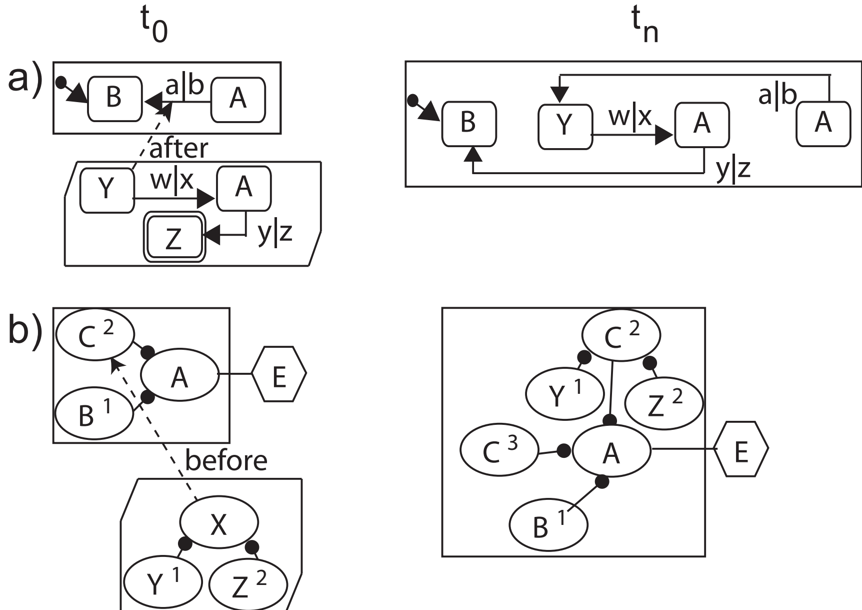


Figure 9.30: Illustration of the open issues in the visual weaving of ADORA models. The algorithm does not take into account the orientation of the chunks and the targets.

visual weaving must consider the handling of different orientations. The corresponding heuristics to handle these situations are subjective and have to be addressed by a future investigation of the topic.

9.6 Summary and Discussion

This chapter has presented the weaving semantics for the transformation of aspect-oriented to conventional ADORA models. The semantics is described formally and informally. The formal description employs an axiomatic semantics for the specification of the weaving steps. Even though the semantics specification is compact and quite well readable, it has the disadvantage that it cannot be directly interpreted by a software. To obtain an executable transformation in a tool, it has to be implemented manually according to the axiomatic specification (See also the Section 11.4). As long as the weaving transformation is language specific, the axiomatic semantics is adequate. However, if there is a need for a user to be able to create domain-specific weaving transformations, the axiomatic description is inadequate. In this case, a graph transformation approach is probably better suited, as the specified semantics can be defined by abstract models and then directly executed on the models.

The presented weaving transformation allows switching between the aspect-oriented and the conventional view of a model. Nevertheless, the weaving is a unidirectional transformation, i.e., an aspect-oriented model can be transformed to a conventional model, but the transformation can-

not be reverted, because it is a one-to-many mapping of the crosscutting elements to conventional model elements. Consequently, changes in the woven model cannot easily be back-propagated to the corresponding aspect-oriented model.

Currently, an implementation (cf. Chapter 11) of the presented weaving transformation needs to save the original aspect-oriented model. When the user switches back, the saved copy has to replace the woven model again. Therefore, changes in the woven model are not allowed, as they would otherwise get lost.

The problem originates in the fact that a software tool cannot decide automatically whether a change in the woven model belongs either to an aspect or to a conventional model part. As an example, suppose that you connect an additional transition and state to a (woven) behavior chunk in a woven model. A software tool cannot decide if the added behavior belongs to the chunk or to the core behavior. Moreover, if the change belongs to a crosscutting element, it would have to be propagated to the locations crosscut by the aspect, which may in turn cause inconsistencies in the model. The problem of back-propagating changes from the woven to the aspect-oriented model is also strongly related to the traceability issue and the round trip engineering problem and needs the attention of the future research.

This chapter also presented also a straightforward approach for the weaving of the layout information in the model. Even though the approach works quite well for simple models, it may produce a rather confusing layout for more complex ones. As discussed above, an improvement of the visual weaving mechanism may also become a focus for future research.

Chapter 10

Applying the Aspect-Oriented ADORA Approach

The foregoing chapters presented the syntax as well as the weaving semantics of an aspect-oriented extension for the requirements modeling language ADORA. The approach supports the elicitation and evolution of requirements by way of several language features, such as the support for a variable degree of formality and the syntactical annotation of partial and semi-formal elements. However, a proper use of them demands some guidance in the form of a process.

This chapter sketches and illustrates a possible process framework and describes the use of the syntactical elements. Section 10.1 gives an overview of the process. In Section 10.2, the processing of a functional requirements increment is delineated. Section 10.3 discusses how to identify and extract functional crosscutting concerns from the evolved requirements. In Section 10.4, the elicitation and evolution of non-functional requirements is depicted. Finally, Section 10.5 summarizes and discusses the present chapter.

10.1 ADORA Modeling Process Overview

A process for the elicitation and evolution of requirements is presented in [Seyb04a, Seyb06a]. It is based on a semi-formal simulation technique which uses the ADORA language. The present chapter sketches an extension to this process in order to enable it to handle crosscutting non-functional and functional concerns. As shown in Fig. 10.1, the process is iterative and consists of two increments in one cycle. The first increment deals with the elicitation and the refinement of the functional requirements. Once the functional requirement increment is considered complete, the non-functional requirements which refer to the previously elaborated functional requirements are identified and then refined.[1] The whole process ends when the functional and non-functional requirements are considered complete and at sufficiently risk-free.

[1] The non-functional increment follows the elicitation of the functional requirements because non-functional requirements are easier to identify when the functional requirements are already identified.

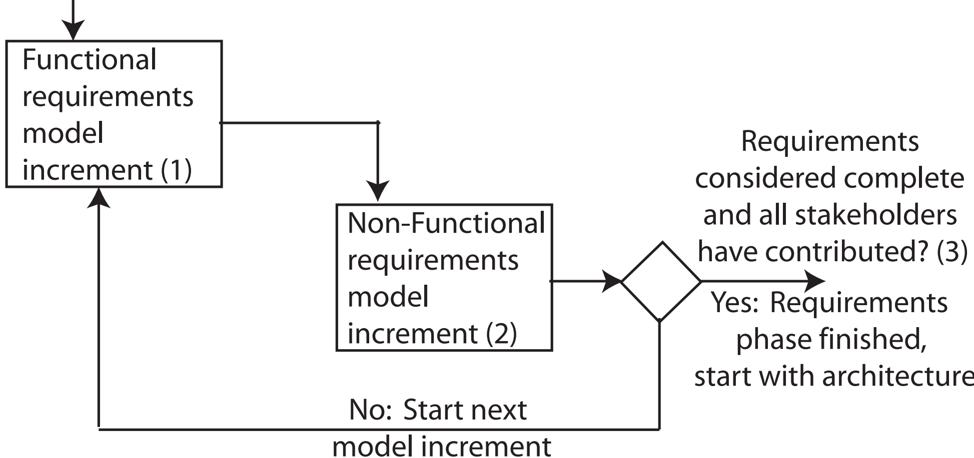


Figure 10.1: Overview of the requirements process

Note that the size of an increment as well as the scope of the increment has to be defined by the requirements engineer. Moreover, a requirements cycle may be intertwined with other phases of the software process depending on the software process model that has been chosen.

The process is exemplified in the following by means of the library system employed as an illustrative model in the preceding chapters.

10.2 Functional Requirements Model Increment

The detailed process for the elicitation/refinement of the functional requirements is presented in Fig. 10.2. It starts with the identification and elicitation of the coarse functional requirements, which can be done by any one of a number of state-of-the-art techniques. In the following, a use-case-driven approach is employed. The process is usually driven by one or more requirements engineers who iteratively elicit and refine, and simulate a model.[2]

All direct stakeholders[3] are involved in the process in a round robin manner. In step (1.1), a stakeholder describes functional requirements in terms of use cases which are modeled as an ADORA scenariochart at the top level of the system. Furthermore, additional requirements may be captured as high-level comments in the existing components. In the following step (1.2), these informal descriptions are analyzed, identifiers of system parts are extracted, and the corresponding system parts, such as components and behavior descriptions are created. The detailed process is described in [Seyb06a, p. 76f]. The nodes of the elicited use cases are associated with the components to which they belong. This is done by placing them as parts into the components.

During the elicitation/refinement and the modeling phase, the stakeholder can amend already

[2] The concurrent working of several requirements engineers working on the same model is enabled by the multi-user capabilities of ADORA [Moda03].

[3] There are also indirect stakeholders who are not involved directly in the project.

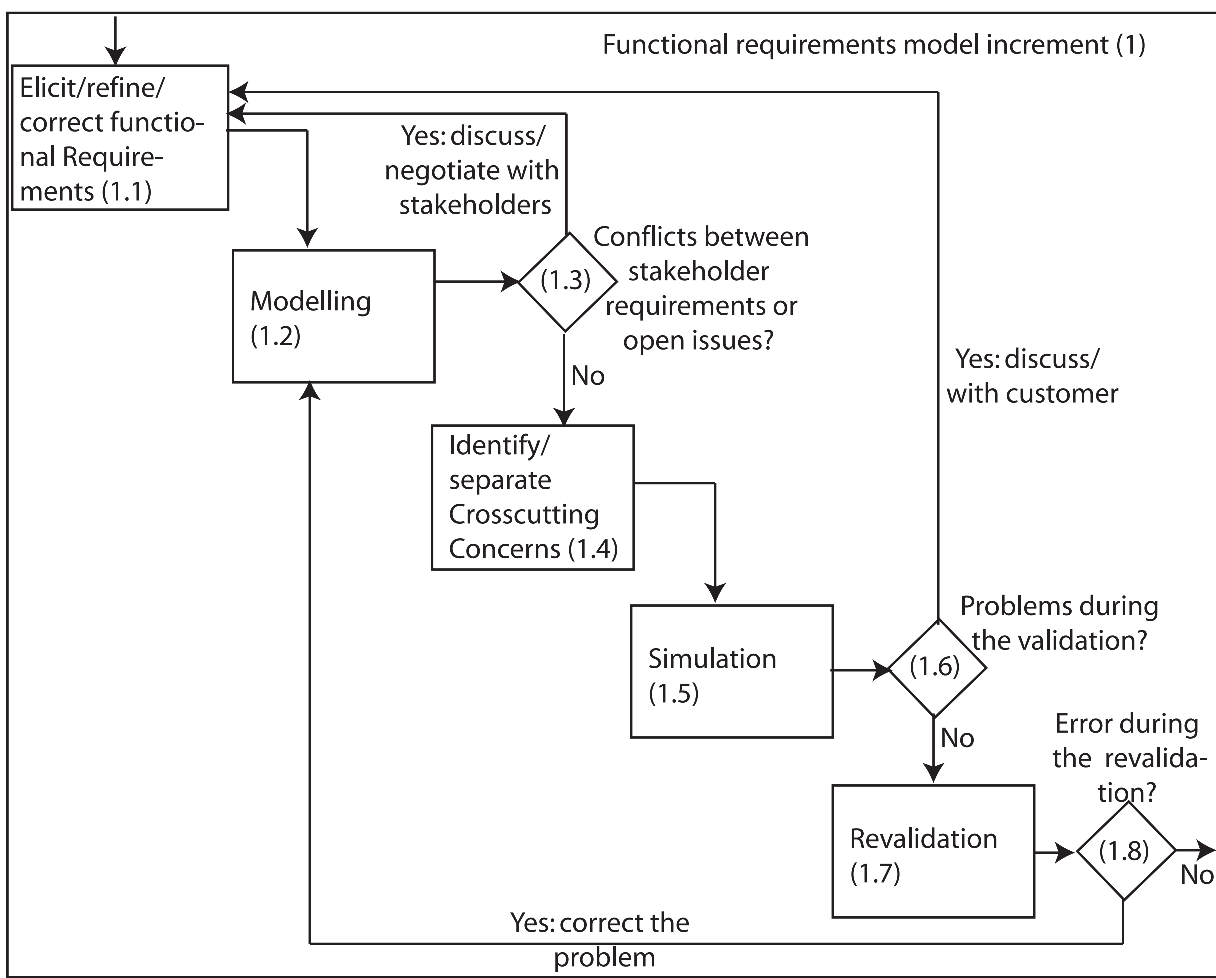


Figure 10.2: Elicitation and refinement process for functional requirements.

existing use cases which have been raised by another stakeholder previously. There may be con icts (1.3) between the currently elicited functional requirements and the prior requirements introduced by another stakeholder. In this case, the con icts are resolved by negotiating between all the stakeholders involved. After a successful remodeling of the requirements increment, *functional crosscutting concerns* (cf. Section 3.1.9) are identified and separated (1.4), which is done by applying a set of heuristic rules. The details on identifying functional crosscutting concerns are discussed in the following Section 10.3.

In the subsequent step, the newly elicited/model parts are executed by semi-formal simulation (1.5).[4] The semi-formal simulation enables a dynamic validation of model parts which are either not fully formalized, inconsistent, or which contain errors. All these problems can be discovered

[4]In order to execute the model by the semi-formal simulation, the aspect-oriented parts have to be woven first, as motivated in Chapter 9.

and resolved by various means.

A so-called driver-and-stub simulation can be used to mock up non-existing parts [Seyb04a]. For this purpose, the person who drives the simulation has to manually complete the action and reaction of the missing parts. The recorded protocol of the mocked up behavior can subsequently be used for semi-automatically completing the missing model parts [Seyb06a]. Furthermore, inconsistencies and errors in the model can be revealed by the simulation process, which also suggests ways of handling those problems. Consequently, the semi-formal simulation allows the engineers to refine, complete the model, correct it, and make it consistent.

When changing a model, it is necessary to show that the already fixed requirements have not been accidentally changed and do not cause a con ict with newly introduced model parts. Therefore, a revalidation (1.7) has to be done. It can be accomplished by a so-called regression simulation [Seyb04a, Seyb06a]. For this purpose, the simulation sessions of already fixed requirements can be recorded: that means that all input stimuli of the executed use cases and the corresponding outputs are logged. The recorded input stimuli can be injected during the revalidation phase. If the fixed model parts are correct, the resulting output must be equal to the recorded output (1.8). Models which do not pass the revalidation must be scrutinized and corrected, if necessary, with the help of the stakeholders involved.

Example (10.1). Figure 10.3 shows the first elicitation steps of the library system. The basic use cases of the system are identified (1.1) in collaboration with the stakeholder and are then elaborated in the following steps. During the modeling (1.2) of the system there may be several steps. In Fig. 10.3 (a), the basic system and the identified use cases are modeled in an intermediate step. After a further refinement, the basic components in the system have been identified as shown in Fig 10.3 (b). As the system is at that time not highly evolved, there are not yet any con icts (1.3) and no functional crosscutting concerns can be identified (1.4). Furthermore, the system has not yet been simulated (1.5), nor has it been revalidated.

10.3 Detection of Functional Crosscutting Requirements

Crosscutting concerns are hard to discover in functional requirements, because crosscutting relationships are not initially apparent. This is due to the fact that functional requirements are usually not complete from the beginning. As a consequence, crosscutting relationships are concealed, and therefore they cannot be identified before the corresponding parts of the requirements are elaborated fully. Therefore, crosscutting functional requirements cannot be identified and separated from the beginning but only during the course of the elicitation.

Crosscutting concerns are initially documented by conventional means, i.e., by using conventional relationships between dependent concerns, as discussed in Section 3.1.3. Consequently, they are scattered over the whole model, and tangled with the modules of the core concerns.

Nevertheless, at a later stage, when all functional requirements involved have been elicited, the crosscutting concerns can be identified and separated into aspects. In this case, the requirements engineer has to decide whether to separate identified crosscutting concerns. However, not every separation of a crosscutting concern accrues benefits. For example, separating a concern

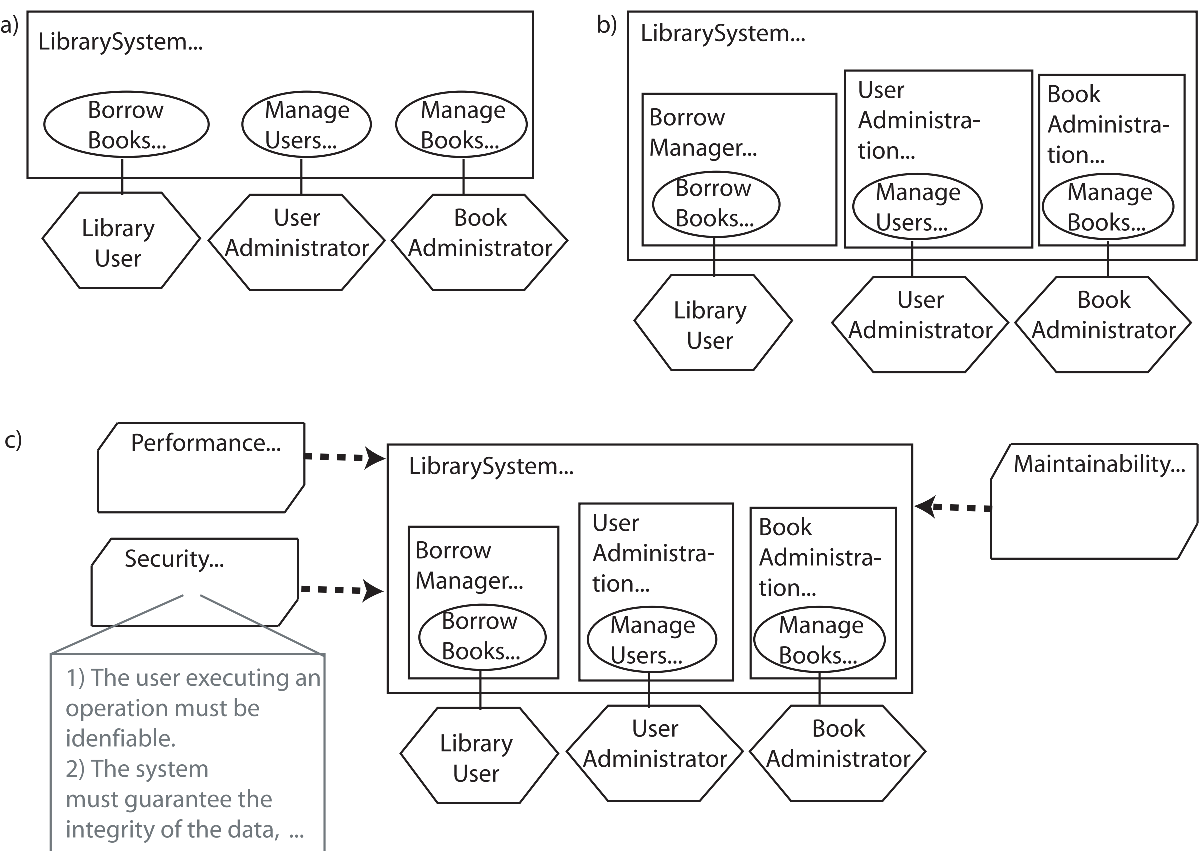


Figure 10.3: Example of the first few steps in an evolutionary process creating the model of a library system.

which has only one crosscutting relationship to another concern will probably neither improve the understandability nor help to simplify the handling of the crosscutting requirements.

In the approach presented, the identification and separation of functional crosscutting concerns has to be done manually. There are several heuristics to identify a crosscutting concern, two of which are listed in the following:

1. Components which provide a service for other components may be an indicator for functional crosscutting concerns. A service is usually provided by the same protocol (of events), which is the same for all consumers. In accessing the service, each consuming component must implement the same behavior. This in turn causes redundancy, which can be seen as a functional crosscutting concern.

2. There may be parts of statecharts or scenariocharts located in different components which have the same purpose/semantics. These cases may also indicate a scattered crosscutting concern. Moreover, this kind of redundant scenario/behavior descriptions may refer to

child components. In this case, the refereed components may also be part of the crosscutting concern.

Currently, the patterns mentioned above must be searched for in person in the model; the present approach does not provide a way to identify such patterns automatically or at least semi-automatically.

The automated discovery of potential crosscutting structures would require checking for similar structures within ADORA models, which is also known as the subgraph isomorphism problem. Although the problem is NP complete, there are algorithms which allow the processing of at least small graphs, such as [Corn70] and [Ullm76]. Furthermore, there are efficient algorithms for special cases of graphs, such as trees [Kell57]. As ADORA models can be represented as trees, the graph isomorphism problem can be solved with an adequate performance.

However, the subgraph isomorphism problem is not the only problem that must be tackled if functional crosscutting concerns in models are to be found automatically. When several people are working on the same requirements models, they may employ different ways to represent semantically equal model parts. They may also have different ontologies of the modeled problem in mind. Both these factors may result in models where a search for isomorphic sub-model parts does not necessarily lead to success, although there are sub-parts in a model which are semantically equivalent. The automatic discovery of crosscutting model parts may be addressed in future research on this approach.

10.4 Eliciting/Re ning Non-functional Requirements

High-level non-functional requirements are represented in this approach by aspect modules, as delineated in Section 3.1.9. Compared with functional crosscutting concerns, non-functional crosscutting concerns can be kept separate right from the time when they are elicited at the beginning of the process. They subsequently co-evolve with the functional requirements. Fig. 10.4 shows how the process for the elicitation of non-functional requirements is carried out.

Non-functional requirements are identified by using categorizations, such as proposed in [ISO 01], or catalogs, such as used in [Chun00]. They are discovered by going through these catalogs together with the stakeholders involved (step 2.1). Usually, the elicited non-functional requirements relate to the functional requirements determined in the preceding steps.

After finding the non-functional requirements, they are modeled as aspects (step 2.2). Each non-functional requirement is represented as a partial aspect that may be amended by an informal description which specifies its purpose in more detail. Join relationships connect the non-functional aspect with the impacted elements and therefore indicate what requirements are in uenced by the non-functional requirement.

During the modeling, con icts between non-functional requirements may be discovered. Con icts can either be caused by the different viewpoints of the stakeholders regarding the meaning, the purpose, or the impact location of an aspect. Furthermore, there may be a competition between different aspects crosscutting the same target, which is indicated by join relationships in the ADORA model pointing at the same target. The competition between different aspects

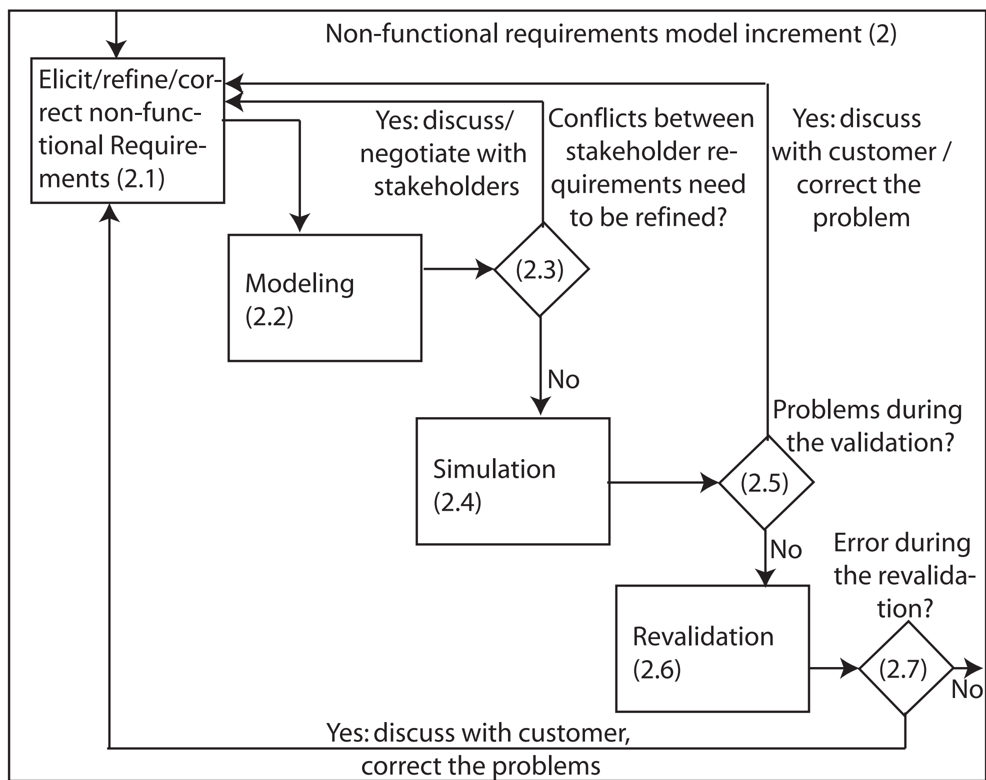


Figure 10.4: Elicitation and refinement process for non-functional requirements.

has to be resolved by defining a priority for the corresponding join relationships. Con icts between the viewpoints of the stakeholders as well as competing impact locations of the aspect (decision 2.3) have to be negotiated by involving all the stakeholders who have contributed to the non-functional requirement currently being refined.

Some of the non-functional crosscutting concerns may need a further refinement. A refinement step may either concern the aspect itself or one of its outgoing join relationships. An aspect may be refined by splitting it into sub-aspects that describe the non-functional requirement on a more detailed level. For example, a high-level non-functional security requirement may be split into a requirement which demands that the transmission of data must be done securely and another requirement which states that a user must be authenticated in order to access particular functions in the software. Moreover, the hierarchical decomposition of aspects also allows for a simple backward and forward traceability of non-functional requirements.

The actual target of the aspect may become clearer during the course of the refinement. Therefore, the join relationships also have to be refined. A join relationship may be relocated or even split into several join relationships.

For the refinement of aspects, three different types of non-functional requirements can be distinguished, as delineated in Section 2.1.2 and in [Meie07]. A non-functional requirement of type (i) can be operationalized to a functional requirement, i.e., it becomes manifest in a piece

of code. For example, an authentication mechanism may be a result of type (i). Non-functional requirements of type (ii) do not result in additional functionality, but in restrictions on how the system performs. An example may be the demand for a minimal throughput. Non-functional requirements of type (iii) neither contribute to the functional requirements of a system, nor do they in uence how the system performs. They rather result in a decision which in uences the software process or the form of the resulting software artifacts. An instance of this type is the decision to use a particular schema to document the code artifacts.

The refinement of non-functional requirements is continued until they are fully concretized. The point in time when the refinement stops depends on the type. It may be the case that a non-functional requirement cannot be refined any more during the requirements phase, because it is a decision which is not realized until a later phase in the software process, e.g., the architectural or the implementation phase. Aspects describing non-functional requirements of type (ii) and (iii) usually remain partial during the requirements phase. A non-functional requirement is fully evolved when it is not marked as partial and the corresponding aspect module neither has any more partial elements nor outgoing partial join relationships.

After the refinement steps of the non-functional increment have been carried out, the semi-formal simulation must be executed (step 2.4), which is preceded by a weaving of the aspect-oriented model. The simulation must execute the model parts that have been added recently. It may help to concretize non-functional requirements of type (i) which result in functionality. As the simulation aims at the refinement of the *functional system description*, non-functional requirements of type (ii) and (iii) are usually not refined by the semi-formal simulation and must be elaborated manually.

The refined non-functional crosscutting elements of type (i) must be manually propagated back to the aspect-oriented model because the proposed weaving transformation, presented in Chapter 9, does not allow this to be done automatically.

After the simulation, previously recorded simulation runs have to be executed in order to revalidate the model by a regression simulation (step 2.6). Any failure during the revalidation is a hint of a problem in the newly introduced non-functional requirements or a con ict between them and other requirements. In this case, the problems have to be corrected in another elicitation and requirements cycle (decision 2.7).

Example (10.2).
Figure 10.3 (c) shows the first step of the elicitation in the non-functional requirements for the library system. There are three non-functional requirements represented by aspects. Each of them may be described in more detail by an informal description, as alluded to by the gray box for the security concern. The abstract join relationships between the aspect modules and the system indicate the impact location of the non-functional requirements. Initially, the non-functional requirements relate to the library system; however, in the course of the further system elicitation they may be refined and then point to parts of the library.
Figure 10.5 shows a later stage of the further elicited model, where the security and the maintainability aspect have been evolved further. Moreover, the join relationships have been refined, too.
The refinement of *Security* and *Maintainability* is indicated by its decomposition into sub-aspects.

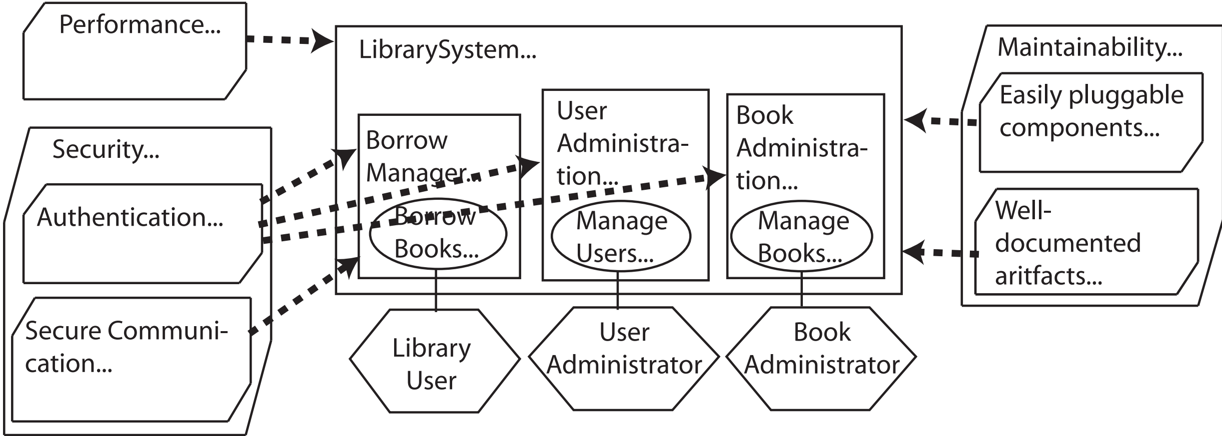


Figure 10.5: Some further steps in the evolution of the example.

These sub-aspects describe a specific part of the parent aspect in more detail. For example, the *Security* and its sub-aspect may be refined to the level of functionality, i.e., a type (i) outcome, in the course of the requirements stage. Correspondingly, the join relationships originating in the sub-aspects may be refined. In contrast, *Performance* and *Maintainability* result in decisions, for example, at the architectural stage. Thus, their further refinement stops at a certain point of the requirements and is continued at a later stage.
During the further refinement of the model, the functionality as well as the non-functional elements are refined. At the end of the requirements phase, a detailed requirements model as shown in Fig. 7.2 may result. Note that the hierarchical decomposition of the aspects *Security* and *Maintainability* as shown in Fig. 10.5 may be dissolved, as was done in Fig. 7.2. However, in doing so, the backward and forward traceability of the non-functional aspects gets lost and so the source at least from which the aspect parts of the non-functional requirements originate, i.e., particular stakeholder, must be documented.

10.5 Discussion

This chapter has sketched a process which shows how to use the syntactical elements and the weaving mechanism presented in the previous chapter. However, the process serves rather as an example of how to use them and, correspondingly, there are several open issues that are not covered by the present work: the outlined process has to be elaborated in more detail and to be validated through deployment in real world projects.

Another open issue is the identification and separation of *functional* crosscutting concerns that currently has to be done by a person. Furthermore, the interplay between the semi-formal simulation and the aspect-oriented elements must be investigated thoroughly in future research. To find a mechanism that allows changes to be propagated back (cf. Section 9.6) to the aspect-oriented model is a primary desideratum.

Part III

ADORA Tool Implementation and Validation

This part of the work delineates some facets of the practical realization of the concepts presented in a Software-Tool (Chapter 11). Moreover, it presents the results of an experimental validation (Chapter 12) of some of the approach that has been proposed in this work.

Chapter 11

Tool Implementation

An implementation of the concepts presented is desirable for various reasons. First, an implementation allows the feasibility of the proposed concepts to be verified and their weaknesses to be identified and corrected. Second, requirements engineering and modeling are practically motivated by real world software development. To be useful in practice, the proposed theoretical concepts have to be validated in the real world. However, practitioners are more likely to adopt new ideas if they are supported by a tool. Third, the concepts, such as the integrated modeling, the abstraction mechanism (cf. Section 5.1.3), and the weaving transformations (cf. Chapter 9) are only helpful if they are supported by a software tool. Using them without software support would be too tedious and result in an inefficient handling of ADORA models.

The work of [Seyb06a] presents a prototypical implementation of the introduced ADORA concepts. It is a proof-of-concept implementation, not optimized for a real world use and therefore rather complicated to handle and difficult to use. Consequently, the concepts of the present work have been embodied in a completely redesigned and reimplemented ADORA tool implementation. The resulting version 1.1 of the ADORA tool is based on the Eclipse platform [Ecli07a] and Java [Sun 07a].

The Eclipse platform provides a very stable and mature Mode-View-Controller framework [Reen79] for the implementation of graphical model editors, called Graphical Editing Framework (GEF) [Ecli07b]. In contrast to the self-made MVC framework used in the previous ADORA tool versions, GEF is very powerful and maintained by the Eclipse community. Using GEF for the ADORA tool relieves the development from the costly maintenance of an ADORA-specific MVC implementation.

Furthermore, the Eclipse platform uses third-party libraries and provides basic libraries for recurring implementation tasks, which reduces the maintenance work and improves the usability of the tool implementation. Usability is especially important, since the more user-friendly the tool, the more easily it can be applied in a real world project or be used in a tool-based validation procedure (Chapter 12).

Moreover, the plug-in architecture allows the Eclipse platform itself, as well as existing plug-ins to be extended in a simple way. Using the plug-in mechanism also facilitates a good modularization.

The remainder of this chapter presents several selected facets of the current ADORA tool

architecture and its implementation. In Section 11.1 the coarse architecture is introduced. Section 11.2 presents how the BNF language description is mapped to an object-oriented meta-model implementation, and Section 11.3 discusses the checking of the language constraints. Finally, Section 11.4 deals brie y with the implementation of the model weaver.

11.1 Tool Overview

The reimplementation of the ADORA tool was extended by several new features and is based on a completely new architecture. The current version 1.1 can be freely downloaded from [RERG07].

11.1.1 Features of the ADORA Tool

The tool provides modeling capabilities for the ADORA language as well as the corresponding navigation and abstraction mechanism presented in Chapter 5. Furthermore, the aspect-oriented concepts discussed in Part II of the present work are also realized in the prototype: the tool provides all the proposed aspect-oriented language elements (cf. Chapter 7) as well as the corresponding weaving mechanism (cf. Chapter 9).[1] It also implements the abstraction and navigation mechanism for the aspect-oriented modeling elements (cf. Chapter 8).

Fig. 11.1 shows a screen shot of the ADORA tool. The tool has its own perspective in Eclipse, i.e., the tool windows that are shown together with the main model editor window can be freely composed and the configuration of the last session is restored after the start of the Eclipse environment. The model editor is shown in the center of the Eclipse window. It provides a drawing canvas and a palette of tools for drawing the elements and navigating through the model. The tool windows, called views in Eclipse terminology, surround the main editor window.

There is a project explorer view in the upper left section of the window that allows the user to browse through project files and to open ADORA models. The property editor in the lower left, shows the properties of the currently selected element and provides editing capabilities. The middle lower section of the Eclipse window displays several tabs: the *Syntax Problems* tool window displays the syntax errors of the textual descriptions contained in the current model. The hidden tabs *Manage Constraints* and *Checked Constraints Tree* are used, respectively, for the administration of language constraints and displaying the model elements which violate them.

The tool bar below the menu of the Eclipse window provides buttons for changing the view of the model. There is a button for weaving an aspect-oriented model, and others for showing and hiding the different views of a model (cf. the horizontal abstraction mechanisms discussed in Section 5.1.4 and Section 8.2). The outline view on the right-hand side shows the tree of the hierarchical decomposition of the model. It can be employed to navigate through the model by choosing a corresponding node and also enables crosswise abstraction (cf. Section 5.1.4) by allowing the showing and hiding of each model node separately. Note that the vertical abstraction (zooming of nodes) can be performed directly on the model canvas using a zoom tool.

[1] Except for the weaving semantics for partial elements. The implementation of this part has been omitted for this work and will have to be implemented in a future version.

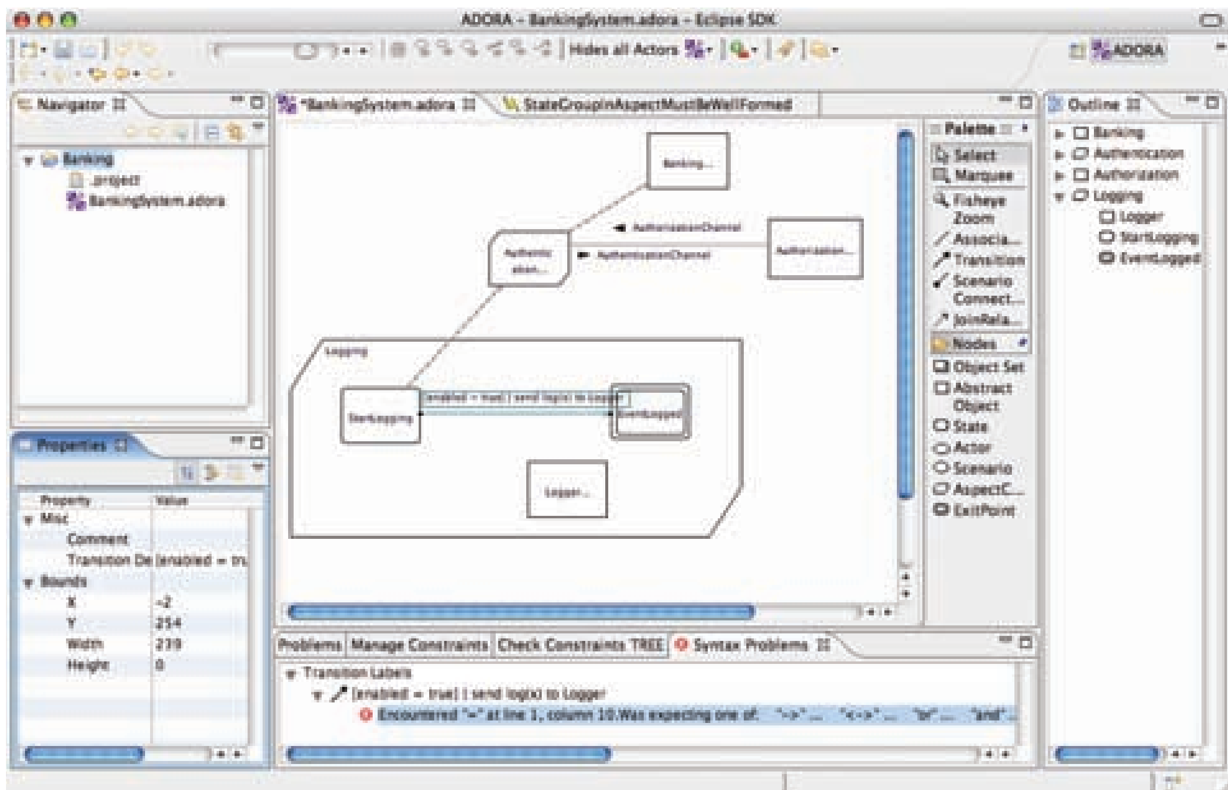


Figure 11.1: A screen shot of the ADORA tool.

11.1.2 Architecture of the ADORA Tool

The architecture of the ADORA tool version 1.1 is given as an ADORA model[2] in Fig. 11.2 which shows an instantiation of the tool. The minimal editor configuration consists of several required plug-ins and libraries:

- **Graphical Editing Framework.** The *Graphical Editing Framework* (GEF) is an implementation of the Model-View-Controller (MVC) pattern and comes with the Eclipse platform. The *GEF core* package provides the abstract implementation for controllers and *Draw 2D* contains the abstract elements for the graphical representation of a model element. Apart from the MVC pattern, GEF provides a set of standard functionalities for graphical editors which reasonably augment the usability of an implemented graphical editor.

- **Eclipse Platform.** The Eclipse Platform provides a lot of elementary libraries which simplify the recurring tasks occurring during the implementation of a piece of software. For example, Eclipse contains a platform independent implementation to build graphical user

[2]For the sake of better readability, the decomposition levels are alternately colored with a gray or white background.

interfaces (*JFace* and *SWT*); it has built-in *Team* collaboration functionality[3]; a framework for a context sensitive help; and many more features which are not mentioned here.

- **ADORA Meta-model Implementation.** The meta-model implementation of the ADORA tool is object-oriented and incorporates the EBNF syntax discussed in Chapter 6 and 7. The meta-model implementation is discussed in detail in Section 11.2. Each element in the model is represented by an object providing the model information.

- **Editor Core Plug-in.** The *Editor Core Plug-in* implements view and controller classes of the MVC pattern in order to provide the editing functionality for graphical ADORA models. Each *Model Element* object of the ADORA *Meta-model Implementation* has exactly one *View* and one *Controller* object as a counterpart. View objects are responsible for the graphical representation of a model element, whereas controller objects handle the inputs of the editor user and translate them into actions on the model element object. Controller and view elements in the ADORA editor use and extend the abstract controller and view elements of the GEF.

- **Zooming Plug-in.** The zooming plug-in provides the ADORA fisheye zoom algorithm. It is used to effect the abstraction mechanisms discussed in Section 5.1.4. The concepts of the zoom algorithm are presented in [Rein07].

- **Line Routing Plug-in.** Apart from the zooming mechanism, the proper routing of the connections between nodes, i.e., the routing of associations, transitions, etc., in ADORA models is crucial in order to obtain a higher comprehensibility of the model. The approach implemented in this plug-in is based on [Rein06].

- **XStream.** XStream is a third-party library which is used to store ADORA models in an XML format.

Apart from the required libraries and plug-ins to run the ADORA editor, it can be extended by the following plug-ins:

- **ADORA Help Plug-in.** The ADORA help plug-in provides manual as well as context sensitive help for the ADORA editor.

- **Constraint Checker Plug-in.** This plug-in is used to check leniently enforced constraints of the ADORA language, e.g., the ones defined in Chapter 7. A check of the constraints is executed by the *Aspect Weaver Plug-in* before a model is woven. A constraint check is also initiated by the *Simulation Plug-in* to check whether a formal simulation can be executed properly [Seyb06a]. A discussion of this plug-in can be found in Section 11.3.

- **Aspect Weaver Plug-in.** The aspect weaver plug-in performs a model transformation weaving the crosscutting elements of aspects into the crosscut concerns. Beside the weaving of the model, the plug-in performs also a weaving of the visual information. It is delineated in more detail in Section 11.4.

[3] Such as support for the Concurrent Versions System [Cede05].

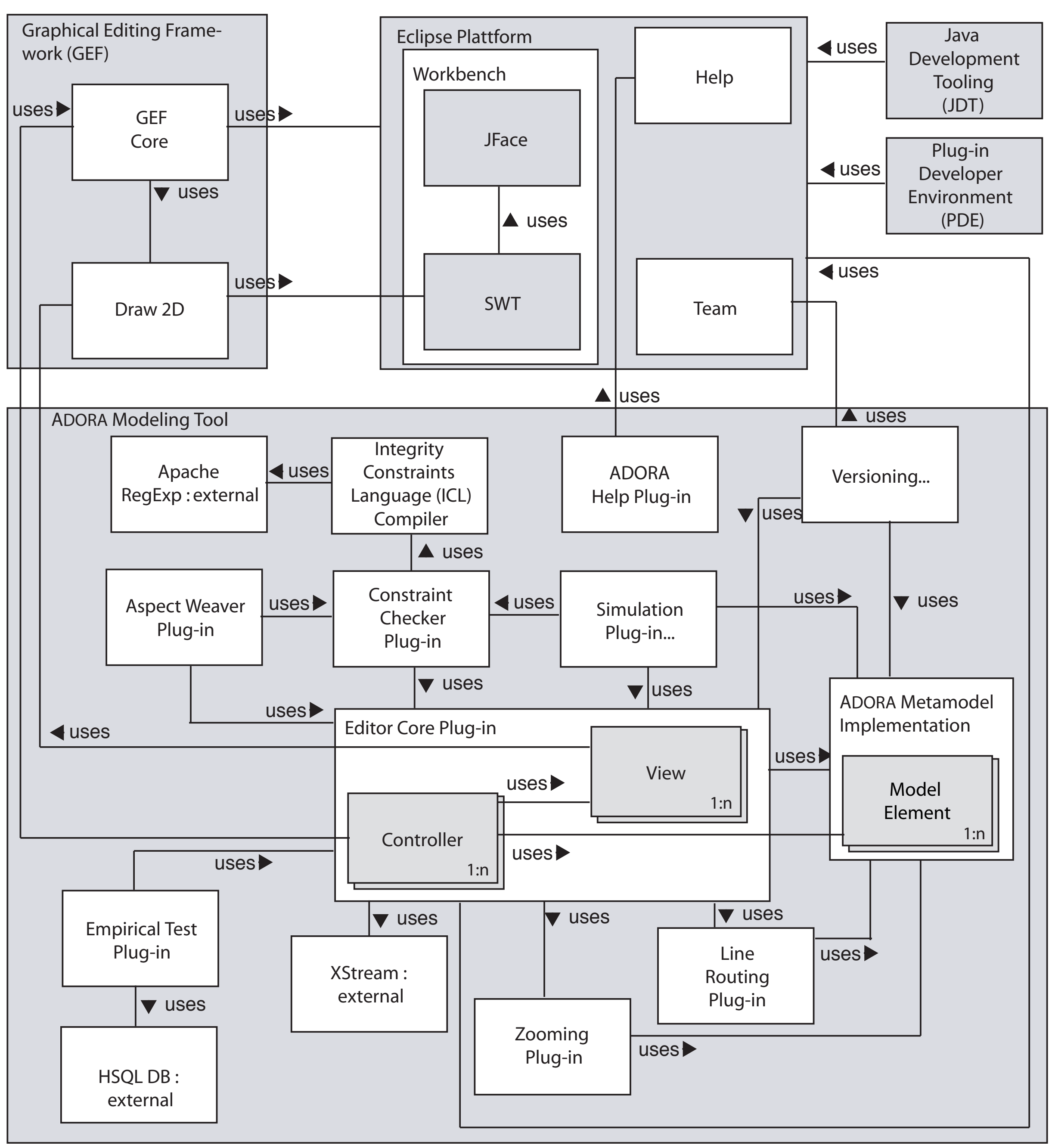


Figure 11.2: The architecture of version 1.1 of the ADORA tool.

- **Empirical Test Plug-in.** The empirical test plug-in helps when conducting experiments on ADORA models. It allows a questionnaire to be presented together with particular models. Answers to the questions presented, as well as the time used to respond to the answers are stored. This plug-in was used for the experiment presented in Chapter 12.
- **Simulation Plug-in.** The formal and semi-formal simulation approach [Seyb06a] is currently not implemented in the ADORA editor 1.1. However, an implementation is planned as a separate plug-in.[4]
- **Versioning.** The versioning concept proposed by [Moda03] has currently not been migrated from the old prototype. Nevertheless, it is planned that it also should be implemented as an Eclipse plug-in.

The ADORA language specification, consisting of an Extended Backus Naur Form (EBNF) grammar definition, the language constraints, and the weaving transformation are the central concepts of this work. For this reason, the remainder of this chapter discusses their practical realization in the ADORA tool.

11.2 Meta-Model Implementation

In Chapter 6, parts of ADORA meta-model are presented. The ADORA language is defined as EBNF that is complemented with language constraints. An ADORA model is actually a well-formed textual description which is mapped to a graphical representation. The following section discusses what the meta-model implementation of ADORA looks like.

11.2.1 Choosing an Appropriate Design for the Meta-model Implementation

An interactive tool must be high-performance, so that an operation on a model is executed with an adequate response time. There are three different design/implementation alternatives for a meta-model implementation of the ADORA tool:

i. ADORA models can be handled as textual representation only, i.e., the graphical representation is directly built from the textual specification of a model. In turn, every change is propagated to the textual description of the model. Even though this solution seems to be reasonable, direct manipulation of the textual model is complex and an implementation for such a mechanism is error prone and has a poor performance.

ii. Instead of manipulating the textual representation of a model directly as discussed in (i), it is possible to transform a textual model first into a syntax tree, which is then easier to manage than straight text. Thus, before a change on a model is performed, a syntax tree is created by parsing the text (cf. Chapter 6). The changes are performed on the elements

[4] Therefore the partial indicator, i.e., the ellipsis trailing the plug-in name, is set.

of the syntax tree. In the end, the syntax tree is transformed back to a textual representation. The graphical representation is still mapped on the syntax tree. This approach simplifies the handling of the textual representation. Nevertheless, it does not improve the performance of the operations on the model, as the parsing step as well as the back transformation are complex operations. This is especially problematic with graphical elements that are modified by a user-tool-interaction.

iii. It is possible to completely omit the textual representation of a model and to use only the corresponding syntax tree. This means that there are no intermediate transformations from the textual model to the tree and back. Instead, the syntax tree is directly mapped to graphical elements and changes in an ADORA model are directly executed on the syntax tree. However, all operations on the syntax tree have to ensure that the resulting syntax tree adheres to the ADORA grammar rules.

The design alternative (iii) has a better performance than (i) and (ii) and is therefore chosen for the implementation of the ADORA tool. However, the meta-model implementation approach (iii) is only used in the case of graphically visualized ADORA language elements. Textual parts, such as the functional specification of components or the transition labels are still handled by approach (ii), i.e., they are stored textually and only transformed to a syntax tree, if they need to be processed, e.g., for the simulating or weaving of a model. This is due to the fact that the user of the ADORA tool is supposed to change these textual representations directly.

11.2.2 Grammar Mapping

As with most contemporary graphical modeling tools, the ADORA tool is implemented within an object-oriented paradigm. Consequently, the syntax tree will be represented by an object structure in which each node in the tree is mapped to an object.

The syntax trees used for the discussion of the concepts in Chapter 6 and 7 are *concrete*: they are representations of a decomposed textual model which adheres one-to-one to the concrete ADORA grammar. However, working directly with concrete syntax trees is tedious because a lot of extra work has to be done when manipulating them. This is due to the fact that a semantical unit, such as a component or an aspect module, is scattered over several nodes of the syntax tree. Hence, for most model operations, it is necessary to involve multiple nodes of the concrete syntax tree which need to be handled when manipulating the semantical unit. Moreover, there are nodes in a concrete syntax tree which have a purely syntactic but not a semantic meaning, e.g., semicolons. Thus, they are unnecessary ballast in the syntax tree when interpreting its semantics.

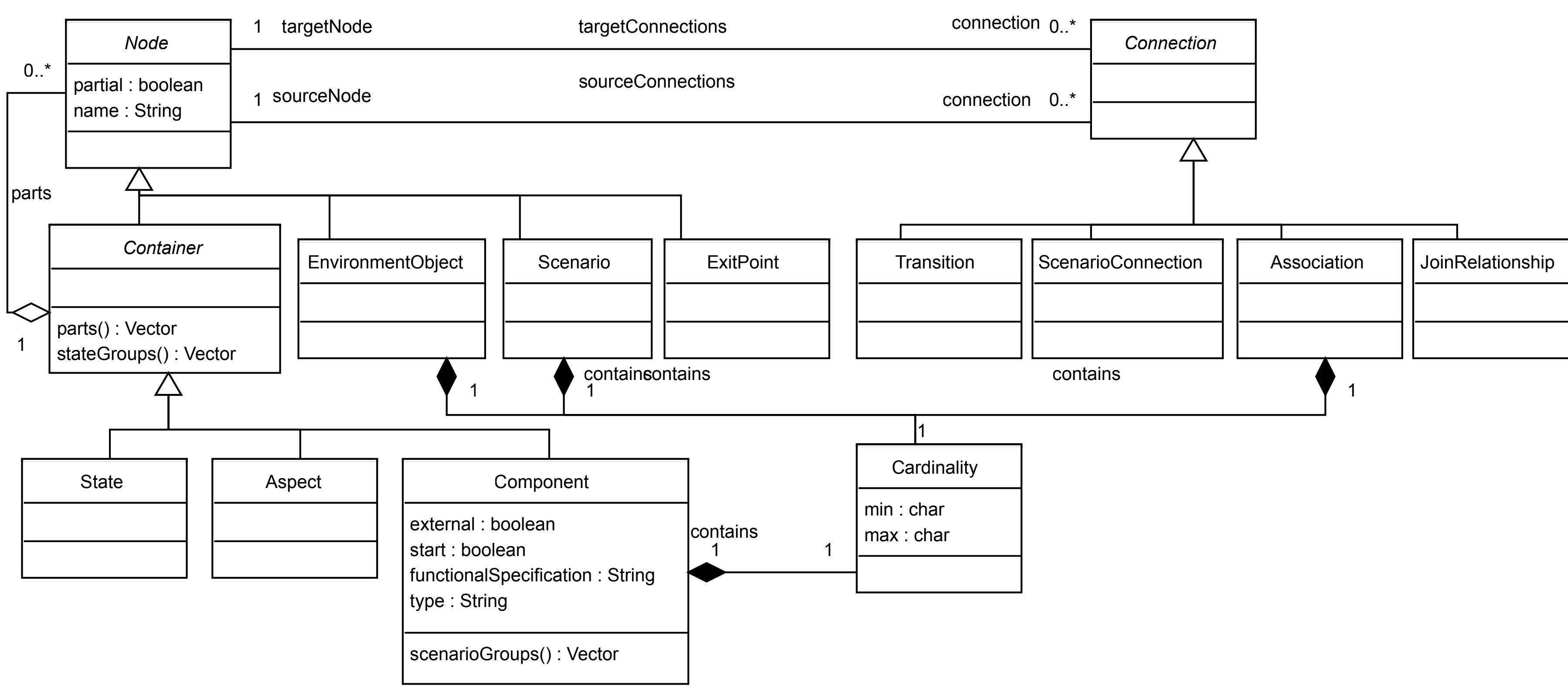


Figure 11.3: A coarse overview of the meta-model implementation showing a simplified excerpt of the meta-model.

Hence, it is better to simplify the concrete syntax tree by merging all nodes which belong to a semantic unit and to put them into one node that contains several attributes. Furthermore, nodes representing purely syntactical nodes can be omitted from the resulting data structure. Consequently, an abstract syntax tree results. In an object-oriented implementation, a node of an abstract syntax tree is represented by an object in the system. Note that the actual data structure for storing an ADORA model is not a tree in the strict sense because there are cross-references between the elements of different paths in the tree, e.g., an association node may refer to a component node.

Consequently, the ADORA EBNF rules belonging to the same semantic unit are mapped to the same class. The corresponding mapping is discussed in the following.

A Class Model Describing the ADORA EBNF

The mapping between the EBNF and the corresponding meta-model classes is exemplified by some of the EBNF rules given in Table 11.1 and the excerpt form the meta-model in Fig. 11.3. The table contains the ADORA grammar rule *ComponentDe nition* and related rules, whereas the figure shows a coarse and simplified extract of the structure of the meta-model implementation.

The class *Component* in Fig. 11.3 represents an abstract object or an object set, i.e., it encapsulates the information described by the EBNF rule *ComponentDe nition* in Table 11.1. For the sake of less redundancy in the implementation of the meta-model, properties which are common to more than one class are generalized in a superclass. For example, the classes *Node* and *Container* contain common relationships, attributes, and operations of the corresponding subclasses. The class *Node* is the superclass of all node elements in the ADORA language. Attributes may be generalized, too. For instance, the attributes *partial* and *name* are properties which are available in all types of nodes in an ADORA model.

For instance, the aggregation relationship *parts* is common to aspects, states, and components, since these elements can contain parts. For this reason, this relationship is specified by the superclass *Container*. Another example is given by the associations *targetConnections* and *sourceConnections* which are associations between the classes *Node* and *Connection*. Both associations represent the target and the source constituent of a connection, respectively.

Description of the Visual Representation in the Object-Oriented Meta-model

The part of the meta-model implementation presented above neither deals with the spatial layout nor with the visibility information of the ADORA language elements. However, the graphical visualization of a model element, as well as the abstraction mechanisms presented in Section 5.1.4 need this kind of information.

The visualization information is contained in a strongly separated class hierarchy. Each class of this hierarchy represents the visualization of a particular element, such as a component or an association, and is in turn associated with the corresponding model class. As the visualization of models is not a focal concern of this work, the details of how visual information is represented is not elaborated on here. An interested reader can find more details on how the visual information is represented in Section H.1 of the Appendix.

Constraints on the Class Model

As discussed above, the implementation of the syntax tree makes use of a generalization to reduce redundancy and simplify the implementation. However, the generalization of class properties results in an object-oriented meta-model which sometimes allows the creation of models that are not syntactically correct with respect to the ADORA EBNF grammar: there is a tradeoff between a simple, low-redundancy implementation, and the proper mapping of the language definition. For example, according to the class structure in Fig. 11.3, the aggregation *parts* implies that any *Container* object, can contain any node. For instance, this means that a component can contain an environment object, which is actually not correct. Another divergence between the ADORA EBNF and the meta-model can be exemplified for source and target connections with respect to exit points of behavior chunks. Exit points may actually have only incoming connections. Thus, exit points can only be the target but not the source of a connection, and consequently, the *sourceConnections* relationship in the class meta-model must be constrained to be valid only for non-exit points.[5]

All operations applied to a syntactically correct ADORA model must also produce again a syntactically correct syntax tree. Therefore, implementation-specific *syntax constraints* must be introduced in order to inhibit such syntax violations. These constraints are only part of the implementation, i.e., there is no corresponding language constraint for the EBNF grammar. Nevertheless, in contrast to *syntax constraints* that only apply to the class meta-model, there are language constraints which are needed for constraining the EBNF as well as the class meta-model definition. An example constraint that occurs in both is that an association cannot connect a component with a state.

Mapping of the EBNF Grammar and the Syntax Tree Operations to the Class Model

There are particular rules for mapping the parts of the EBNF grammar to an object-oriented meta-model implementation. These mapping rules are discussed in more detail in Section H.2 of the Appendix.

Furthermore, the functions working on ADORA syntax trees (cf. Section 6.2.2 and Appendix E), can be mapped to methods in classes of the meta-model implementation.[6] The implemented methods are assigned to the class definitions where they most belong. Some of those functions are shown in Fig. 11.3, for example *scenarioGroups()* in the class *Component*, as well as the methods *parts()* and *stateGroups()* in the class *Container*.

The mapped functions can be used by the ADORA tool to process the model instances, e.g., when executing a simulation or a weaving transformation on a model. Moreover, the implementation of the language constraints, which is discussed in Section 11.3, utilizes the mapped functions extensively.

[5]The corresponding constraint is not shown in the model.

[6]The functions are originally defined for working on a *concrete syntax tree* (cf. Section 6.2.2). However, the object-oriented implementation describes an *abstract syntax tree* of the ADORA meta-model. Thus the functions have to be adapted accordingly. Moreover, performance issues have to be taken into account.

Table 11.1: Excerpt of the ADORA EBNF grammar rules defining a component.

Production Name	Production Rule
ComponentDefinition ::=	("**partial**")? ("**external**")? ("**start**")? "**component**" ComponentName UniqueModelElementIdentifier (Cardinality)? ("**is**" InheritedType)? ComponentParts FunctionalSpecification (ComponentConnections)? "**end**" "**component**" ComponentName
ComponentName ::=	SpecialIdentifier
UniqueModelElementIdentifier ::=	SpecialIdentifier
InheritedType ::=	"**type**" QualifiedIdentifier
ComponentParts ::=	("**consists**" "**of**" (ComponentDefinition | StateDefinition | ScenarioDefinition | AspectDefinition)+ "**end**" "**consists**" "**of**")?
ComponentConnections ::=	"**connections**" (AssociationDefinition | AssociationRoleDefinition | TransitionDefinition)* "**end**" "**connections**"
FunctionalSpecification ::=	("**functional**" "**specification**" (Provides)? (Requires)? (Invariants | DataTypeDeclarations | AttributeDefinitions | OperationDefinition)* "**end**" "**functional**" "**specification**")?
Cardinality ::=	"**(**" (<INTEGER_LITERAL> | <IDENTIFIER>) "**,**" (<INTEGER_LITERAL> | <IDENTIFIER>) "**)**"

11.2.3 Tool Support for the Mapping of the Meta-model

The mapping between the ADORA language definition and the corresponding meta-model implementation can be performed semi-automatically by using tools. For example, JavaCC [Sun 07b] allows generating of a parser as well as the corresponding abstract syntax tree from a given grammar. The abstract syntax tree can be employed as object-oriented meta-model in the tool. The parser can be used to transform the textual parts of a model, e.g., the functional specification, to a syntax tree. The resulting syntax tree can be used for the weaving or the simulation of models.

11.2.4 Discussion of the ADORA Tool Implementation

In [Xia04], it is claimed that a language definition for a wide spectrum modeling language that is based on an EBNF, such as ADORA, may be easier to read and understand than a language definition based on a (object-oriented) class model. This is probably true to some extent. However,

most of these languages can only be handled by a tool, and therefore, the primary goal of the corresponding language definition is that it supports an easy implementation — the specification must be mappable as simply as possible to an implementation (cf. the general quality characteristics of software specifications in Section 2.2.1). Nowadays tools are mostly implemented by using an object-oriented[7] paradigm, and consequently, a language description in an object-oriented manner is better suited to facilitate its the implantation in a piece of software than an EBNF. However, the ADORA language is specified by a textual EBNF and additional constraints, i.e., the language specification is not object-oriented, although it is finally realized as an object-oriented implementation. The two different paradigms clash when implementing the support for the ADORA language, which is also known as impedance mismatch [Cono99, p. 736].

Summarized, the EBNF approach for specifying the ADORA language shows a weakness when trying to map the language definition to an object-oriented implementation: a notable effort is needed to find such a mapping. Moreover, the structure of the implementation also differs significantly from the structure of the language specification, which has a negative in uence on the maintainability of the language and the tool implementation. Hence, an object-oriented specification of the language would ease the corresponding implementation.

11.3 Constraints Checking

To obtain well-formed ADORA models, it is necessary that the language constraints are satisfied (cf. Section 6.3). However, a constraints checking implementation must not be too strict, as otherwise the modeling process would be hindered [Cram07, Section 3.6]. For example, enforcing all language constraints immediately after or before a modification is problematic as it restricts the user of the software too strongly, or it even prohibits the execution of certain modeling actions.

Therefore, it makes sense to categorize constraints according to the point in time when they need to be satisfied. As discussed in Section 6.3, there are two different categories of constraints. *Strictly enforced* constraints are checked *before* a modification is executed on a model. If the change ends in a malformed model, its execution is prohibited. In contrast, *leniently enforced* constraints are allowed to be temporarily violated. Nevertheless, at a specific point in time of the modeling process, they need also to be satisfied, more concretely, either *before weaving*, or *before the simulation of a model*, or *at a user speci ed time*.

Not all of the constraints specified in the language definition of ADORA are mapped to the implementation. Some of them are specific to the textual representation of a model.[8] Furthermore, there are constraints which occur only in the implementation of the tool but not in the language definition (cf. the syntax constraints in Section 11.2.2).

Constraints in the ADORA tool implementation are enforced by two different mechanisms:

[7]This includes also aspect-oriented approaches that are based on object-oriented systems.

[8]One example is the constraint given in Section 7.3.2 on page 116, which ensures that the name in the header and the footer of the textual description of an aspect module is consistent. This constraint is not needed in the meta-model implementation, since only the name in the header is represented in the object-oriented (abstract syntax) tree of the meta-model implementation.

There are hard-coded language constraints as well as constraints that are dynamically compiled, loaded and checked.

Dynamic Constraints vs. Hard-Coded Constraints As shown in the architectural model in Fig. 11.3, the constraint checking plug-in uses the Integrity Constraints Language (ICL) compiler presented in [Sche98]. The ICL language allows the formulation of first order logic expressions to declare predicates on object structures in the Java language. The compiler translates constraints written in ICL language [Sche98] to corresponding pieces of Java code, which are then dynamically loaded into the editor. The constraint checking plug-in allows its user to dynamically write, compile, and use language constraints for the ADORA language. Thus, they are not hard-coded and, therefore, they are decoupled from the rest of the implementation. This allows a user of the tool to modify the restrictions on the ADORA language, i.e., to introduce restrictive prototypes [Bern99a] to adapt the language.

This dynamic constraint checking in the ADORA tool is only applied to leniently enforced constraints. In contrast, strictly enforced constraints are statically defined and therefore hard-coded in the current version 1.1 of the ADORA tool. This is due to the fact that the current implementation of the ICL compiler is not optimized for performance.[9] Most of the strictly enforced constraints need to be satisfied in the context of interactive operations between the user and the tool, and they need to be checked very often. In contrast, leniently enforced constraints are not checked as often as strictly enforced constraints. Thus, using the constraint plug-in for strictly enforced constraints would severely slow down the editor and hinder the modeler.

Moreover, there are also syntax constraints (cf. Section 11.2.2) which need to be implemented by the ADORA tool, so that an abstract syntax tree describing a model corresponds with the EBNF syntax of the ADORA language definition. A syntax violation of the model cannot be tolerated at any point in time. Therefore, they need to be checked before the execution of an operation. Consequently, they are also hard-coded, like strictly enforced constraints.

Section H.3 of the Appendix, discusses in more detail what the leniently enforced constraints look like and how they are evaluated by the constraints checking plug-in of the ADORA tool.

11.4 Model Transformations

The weaving of aspect-oriented models is handled by the aspect weaver plug-in shown in the architectural model of Fig. 11.2. It implements the transformation process described in Section 9.1. Before the transformation is performed, the model is checked for violations of the leniently checked constraints presented in Chapter 7. They have to be satisfied in order that the weaving creates a syntactically and semantically correct resulting model. If one or more of them fail, the weaving process is not started.

The actual weaving process consists of a sequence of transformation operations which are

[9] However, there are optimization techniques for first order languages, which allow better performance results to be achieved. This performance optimization is subject to future development of the ICL language and compiler.

implemented according to the pre- and postconditions specifications defined in Chapter 9.[10] Furthermore, the layout of the model is woven according to the approach presented in Section 9.5.

[10]Note that the transformation description given in Chapter 9 uses concrete syntax trees. However, the implementation of the transformations is based on abstract syntax trees.

Chapter 12

Experimental Validation of the Aspect-Oriented Modeling Approach

In Section 3.1.7, it was hypothesized[1] that the ability to switch between the aspect-oriented and the corresponding conventional view is useful for improving the understanding of the model. This chapter aims at testing this hypothesis and finding a legitimation for aspect-oriented models and the weaving mechanism.

This mechanism allows the reader of the model to choose the view which is more adequate for performing a particular task on a model. The type of view which is more appropriate depends on the type of focus which is set on the crosscutting concerns in the model. There are two different types of focus:

1. An *isolated focus*[2] is a decoupled view of the concerns. An isolated focus does not deal with the interaction between crosscutting and core concerns.

2. In contrast, an *interrelated focus*[3] on a crosscutting concern deals with the interplay between the crosscutting concern and other concerns.

Depending on the type of focus, the use of an aspect-oriented model[4] affects the understandability of the model:

Having an isolated focus on the concerns in an aspect-oriented model improves the understandability. This is due to the fact that modularized crosscutting concerns are strongly decoupled from the other concerns. Therefore, the parts of crosscutting concerns are not scattered in the system but rather concentrated in one module and the reader of the model does not need to seek for the crosscutting model parts.

However, having an interrelated focus on crosscutting concerns in an aspect-oriented model does not improve the comprehensibility. This is due to the fact that the reader of the model has

[1] The basic idea for this hypothesis was formulated in [Meie05].
[2] In [Meie05] this type of focus is called *local*. However, in this work it is called more adequately *isolated* focus.
[3] In [Meie05] this type of focus is called *global*. In this work it is called more adequately *interrelated* focus.
[4] In the aspect-oriented view of a system, crosscutting concerns are modularized, i.e., described as aspects.

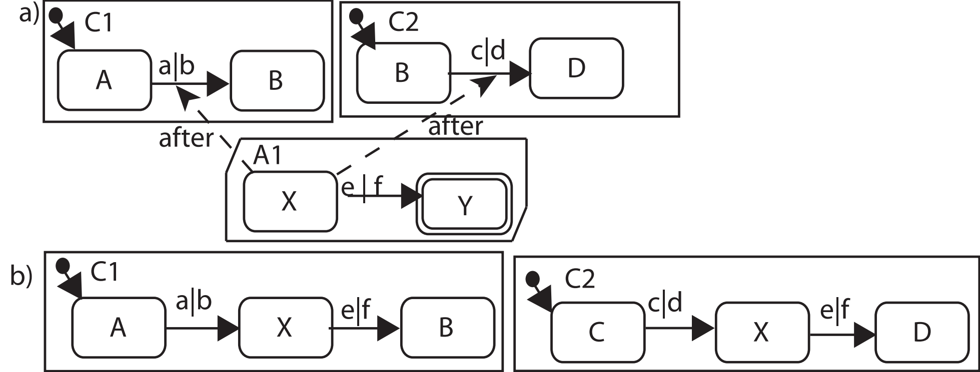


Figure 12.1: Example used for illustrating the isolated and the interrelated focus on a crosscutting concern. (a) shows an aspect-oriented view of a model, (b) the conventional view. For the sake of simplicity, the reception and the action part are simply denoted by letters and not by the full syntax as used in ADORA.

to *integrate* the crosscutting and the conventional modeling parts in his mind to understand how they interplay. In fact, it may even result in a worse understandability of the model.

Fig. 12.1 (a) and (b) is used to exemplify the concept of interrelated and isolated foci on a crosscutting concern. Situation (a) shows the aspect-oriented and (b) the conventional view of the same model. For example, suppose that the engineer who is maintaining the model needs to make a change in the requirements of the crosscutting concern. For instance the action part f of the crosscutting concern's behavior needs to be changed to h. This is a task which has an isolated focus on the crosscutting concern. In view (a), where the crosscutting concern is modularized as aspect *A1*, the engineer can simply locate the element which needs to be changed by searching for the action part in the aspect-container. In contrast, the engineer needs to locate in view (b) all occurrences first. Then he can perform the changes at each location. Thus, using (a) for performing the changes on the crosscutting concern makes it easier to perform the task.

In contrast, suppose the reader of the model needs to interpret the system behavior, e.g., what is the execution order of the action parts in the runtime instance of component *C1*. Using view (a) requires the reader first to integrate the model in mind and then to determine the order of the actions, i.e., b, f. However, model (b) allows the viewer to directly grasp the order of the action part. Hence, in this case, using model (b) is less complex than using view (a).

Thus, the discussion above results in the assumption that there exists a correlation between the type of focus, the type of view used, and the performance of an executed task. This correlation is summarized in Table 12.1. The cells with a gray background indicate tasks with a higher performance.

Table 12.1: Impact on the performance of tasks resulting from different combinations of focus type and the type of view.

	Aspect-Oriented Model	**Conventional Model**
Isolated Focus	High Performance	Low Performance
Interrelated Focus	Low Performance	High Performance

The type of focus depends on the executed task. There are various tasks which can be performed on a model containing a crosscutting concern. When performing a task, the model is either read/interpreted or modified. Each modification task also contains a read/interpret subtask. A few examples of tasks and their focus on the concerns are given in Table 12.2. Note that *locating the elements of a crosscutting concern* is a subtask which has to be performed for all tasks with an isolated focus.

Table 12.2: Some examples of tasks with an isolated or an interrelated focus on a crosscutting concern.

Focus	**Task**	**(m)odify (r)ead**
Isolated	Locating crosscutting behavior.	r
Isolated	Locating crosscutting scenarios.	r
Isolated	Locating components which are used by crosscutting behavior to delegate responsibilities.	r
Isolated	Editing crosscutting scenarios.	m
Isolated	Interpreting the responsibilities of the core concern.	r
Isolated	Interpreting the responsibilities of the crosscutting concern.	r
Isolated	Identifying the other components which provide a service to the crosscutting concern, e.g., an embedded or a server component.	r
Interrelated	Interpreting the interplay of the crosscutting behavior and the crosscut behavior.	r
Interrelated	Interpreting the interplay of the crosscutting scenarios and the crosscut scenarios.	r
⋮	⋮	⋮

As argued above, it is desirable to be able to switch between the aspect-oriented and the conventional model, and thereby to select the more adequate view to perform a given task. This claim can be formulated more formally by the following two hypotheses:

Hypothesis 1 *Tasks with an isolated focus on a crosscutting concern can be performed more ef ciently on an aspect-oriented model.*

Hypothesis 2 *Tasks with an interrelated focus on a crosscutting concern can be performed more ef ciently on a conventional model.*

Both hypotheses contain claims that have to be confirmed by an experiment. In the remainder of this chapter, such an experiment is presented together with the results. Section 12.1 introduces the experimental setup and how it was conducted. Section 12.2 presents the results and discusses the validity of the experiment.

12.1 Experiment

The claims of hypotheses 1 and 2 concern subjective matters. Therefore, the evidence for their correctness can only be provided by conducting experiments. In this section, the setup and the execution of such an experiment is presented. The experiment aims at measuring the performance of tasks executed on given models and testing both hypotheses against the data collected. The models are based on two different case studies which are both given as an aspect-oriented and a conventional view. For the sake of simplicity, just some of the tasks listed in Table 12.2 are tested, e.g., modification tasks are not included in the experiment. The tasks are given in terms of multiple-choice questions. In the experiment, there are objective and subjective questions.

Objective questions. Objective questions ask facts about the models shown which must be answered by the individuals tested. They allow the experimenter to deduce the performance of the answer. The performance depends mainly on two factors: the efficiency and the effectiveness.

The *ef ciency* can be measured indirectly by determining the *time used to respond* to the answer. Although the resulting values for single samples may vary enormously from subject to subject answering the question, the average time needed to answer the question will converge to the mean of the population for bigger samples. Therefore, the sample must be heterogenous and its size must be large enough to avoid a bias in the results.

The *effectiveness* of a task can be measured by determining the average accuracy of the answers. In the case of multiple choice answers this is the ratio[5] between the multiple choice items that are correctly answered and the items that are wrongly answered.

To determine the performance, the so-called *performance points* are calculated. They are calculated by the ratio $\frac{Correctness}{TimeUsed}$ and indicate the number of correct items per time unit, which

[5]It is expressed as a percentage value.

is an measure for the performance of the answering. The performance points can be used for testing hypotheses 1 and 2.

For the hypothesis test, two different groups of people are needed/required. One group is the comparison group which answers the objective multiple choice questions by using only the conventional model. The other group is the actual test group which answers the same questions for the aspect-oriented case. Due to the fact that the concept proposed in this work allows switching between the aspect-oriented model and the conventional model, the aspect-oriented model is always provided together with the conventional one. An intermediate question after each objective question allows the experimenter to determine to what extent the aspect-oriented and the conventional model are actually used in answering a question.

Subjective questions. Subjective questions ask about the test person's impression of the usefulness of the aspect-oriented model.

12.1.1 Planning and Preparation of the Experiment

There are four different elements in the empirical experiment: the case studies, the questionnaire, the individuals tested, and the test environment.

12.1.2 Case Studies

If questions are to be meaningful and objective, the quality of the case study models is crucial. A case study model must fulfill several requirements:

1. It must admit questions with both isolated and interrelated foci on crosscutting concerns.
2. It must admit questions of various complexity.
3. It must not be too complex, as the experiment has to be performed within a rather narrow time frame.

Furthermore, there are two sources at bias which need to be eliminated. First, a bias may be caused by a learning effect. Therefore, two different case studies are used. After a certain amount of time, the case studies are exchanged. Second, both case studies have never been used before, neither for previous research on ADORA nor for educational purposes. Therefore, no participant of the experiment has ever seen the models of the case study before.

The case studies chosen for the experiment are a banking system and a badge system. The banking system consists of an automated teller machine and bank counters for withdrawal of money. The badge system specifies the access control system for the security doors of a building and the cashless payment of food at vending machines.

The models of the case studies, as well as their description, can be found in the Appendix of [Meie09a]. Both case studies are specified as an aspect-oriented, as well as a semantically

equivalent conventional model.[6] To become feasible within the time frame of the experiment, some parts of the case studies are partial.

The details about the case studies as well as the corresponding figures can be found in [Meie09b].

Questionnaire

The questionnaire used can be found in the appendix of [Meie09b]. It consists of four types of questions:

- **Personal Information:** The first type of question asks for personal information about the test person, e.g., the age and the gender of the test person.
- **Objective Questions:** The second type of question asks about objective facts in respect of the model. The questions are used to measure the performance of the executed tasks using the aspect-oriented and the conventional models, respectively. Furthermore, the corresponding questions can be subdivided into two subsets. The first subset has a local and the other an interrelated focus.
- **Use of Model Types:** The third type of question inquires about the extent to which the aspect-oriented and the conventional model were used by a test person to answer an objective question. This type of question is shown only to the group which answers the questions presented together with an aspect-oriented model.
- **Subjective Questions:** The fourth type of question is subjective and asks the subject about the usefulness of having an aspect-oriented model beside the conventional model.

There are two requirements for the questionnaire: first, the objective questions must be fair, i.e., questions need to be correct and answerable within an adequate amount of time. Second, the experiment is conducted anonymously. Thus, the questionnaire must not allow the identification of the test persons participating in the experiment as otherwise the results may be biased.

Participants

The range of people participating in the experiment must re ect the potential users of the ADORA language and its aspect-oriented extension. For a realistic representation, the individuals tested should have different and broadly ranging modeling skills. Moreover, the number of people with the same modeling skills should be equally distributed in the two test groups of the experiment.

[6] At the time of the experiment, the weaving had not been implemented. Therefore, the weaving of the aspect-oriented model was simulated by providing the conventional model, too.

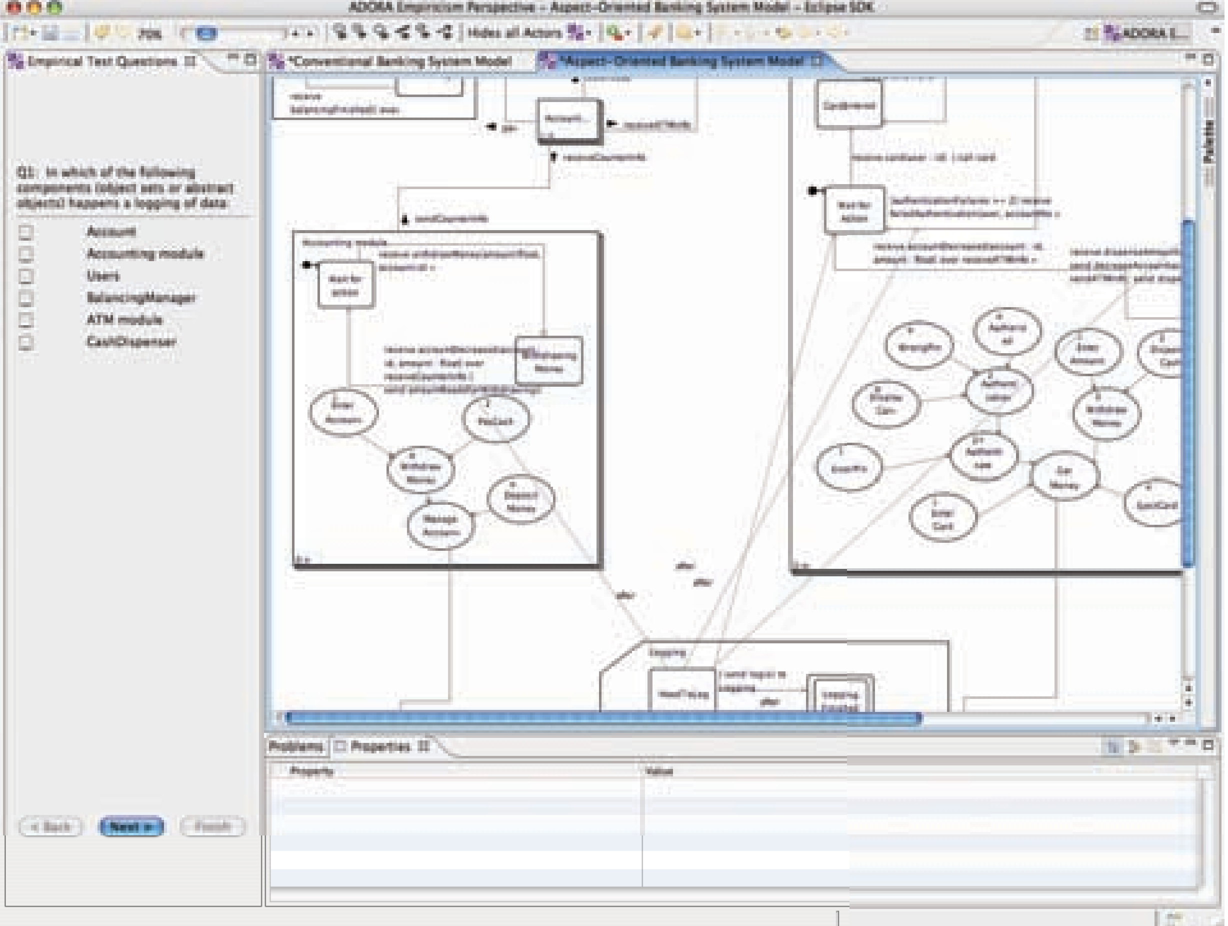


Figure 12.2: Screen shot of the empirical testing environment of the ADORA tool.

Test Environment

To calculate the performance points, it is necessary to measure the time used for answering a question. However, conducting the experiment by manually measuring manually the time needed for answering each question is not feasible. Therefore, the experiment has to be performed with tool support.

For this purpose, the ADORA modeling tool [RERG07] allows inquiries to be made by means of an electronic questionnaire. The questionnaire shows a sequence of models and for each model a couple of questions are asked. The answers given to a question as well as the time used to respond are stored. After the test, the database can be used to retrieve and analyze this data. Figure 12.2 shows a screen shot of the questionnaire environment of the ADORA tool. In the left-most part of the application window, the questions are displayed, whereas the corresponding models are shown in the center.

12.1.3 Realization of the Experiment

The experiment was conducted within a restricted time frame of about three and a half hours. This time span also included an introduction to the ADORA language and its aspect-oriented extension. To assure that the results gathered during the testing are not biased by surrounding conditions, the following measures are taken:

- For conducting the experiment a standard hardware and software combination was used:
  - Computer: Apple iMac computer with a 24 inch wide screen.
  - Eclipse 3.2.1 with the ADORA editor version 1.0.
- The different use of the abstraction mechanisms, i.e. the hiding and showing of modeling elements, might have caused a bias. Therefore, the test persons were not allowed to use them.
- Each model was shown several times. For each question, the models displayed were reloaded, showing the same initial situation and positioning of the model.
- To avoid a bias due to the exhaustion of the subjects, the time-taking was paused between the answering of the objective questions. During this intervals the test subjects had the opportunity to take breaks.

Test subjects. The experiment was conducted with a total of 13 participants on two different dates. On the first date of the experiment, eight subjects took part; on the second date five people. They were recruited from MSc and doctoral computer science students of the University of Zurich, which re ects well the population of the potential users of modeling techniques in real world projects. However, all the test subjects had no or just a little experience with the ADORA language. Therefore, all individuals tested underwent training in ADORA and the ADORA tool (see below) so as to be able to read ADORA models using the tool.

The first eight subjects were MA/MSc students in computer science who were participating in the experiment as part of a requirements engineering class. As preliminary homework, they had to read the papers [Glin02b], [Glin07c] and [Meie06]. Moreover, they had to model the requirements of a small aspect-oriented case study of an electronic tourist guide which was based on [Davi01]. To assure that the students had approximately the same experience with ADORA and the aspect-oriented extension, both were discussed in class again for about an hour just before the experiment. Furthermore, they were given a short introduction and some training in the use of the ADORA tool.

The subjects of the second session consisted of MSc and PhD students from the department of informatics at the University of Zurich. They were given an intensive 3-hour introduction to ADORA, its aspect-oriented extension, and the editor tool.

On both dates, the participants were randomly divided into two groups. However, it was ensured that each group contained approximately the same number of subjects. Furthermore,

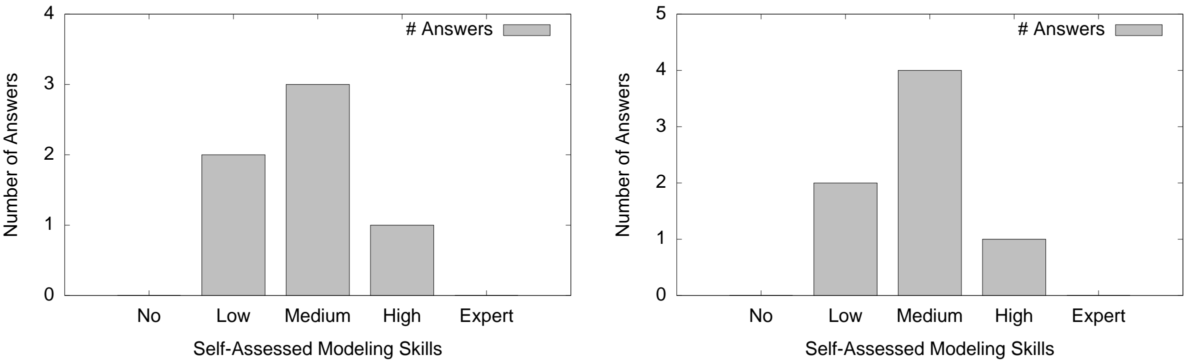


Figure 12.3: Modeling skills of the test persons

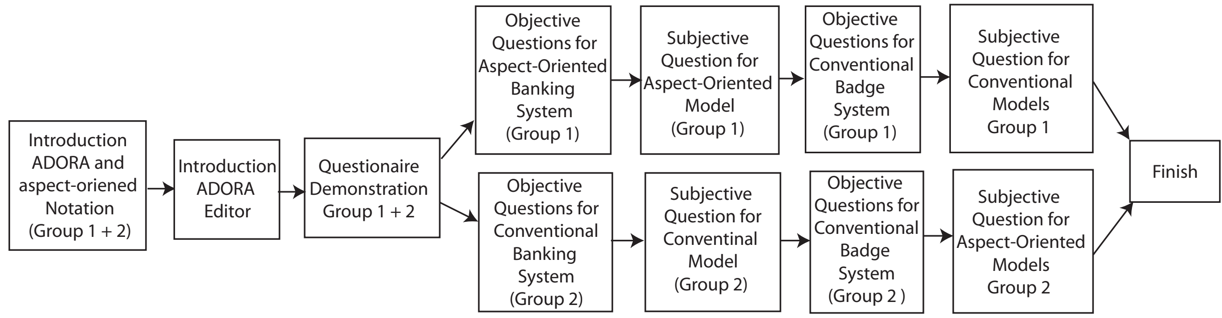


Figure 12.4: Process of the validation experiment.

care was taken that the female and male probands in both groups were approximately equally distributed.

The subjects self-assessed their modeling skills at the beginning of the experiment. The results are shown in Fig. 12.3. The detailed figures fort the test subjects can be found in the appendix of [Meie09b]. All test persons rated their modeling skills in the self-assessment between a value of 2 and 4, where 1 denotes low and 5 high modeling skills. The resulting distributions for the modeling skills of group 1 and 2 are approximately equal.

Conducting the experiment. The experiment was conducted according to the process given in Fig. 12.4. The experiment started with a training phase during which the subjects were introduced to the ADORA language, its aspect-oriented extension, the ADORA editor, and the course of the experiment. After the training phase, the test subjects started answering a set of the introductory questions. They were asked for their age, their gender and their self-assessed modeling skills.

Subsequently, the first block of objective questions was posed. These questions dealt with the banking system case study. The first group answered the questions while having both the aspect-oriented and the conventional models, whereas the second group was the comparison group answering the same questions only with the conventional model. After the first block of

objective questions, group 1 and 2 were asked about their subjective impression of the helpfulness of the aspect-oriented model for answering the questions about the banking system.

A block of objective questions for the badge system followed. The roles of both groups were now exchanged, the first group was the comparison group having only the conventional model. The second group was now the test group having both the aspect-oriented and the conventional models. This block of objective questions was followed by another subjective question about the helpfulness of the aspect-oriented model when answering the preceding questions about the badge system.

For each person tested, all their answers and their response times were automatically logged by the test environment. The assembled data can be found in the appendix of [Meie09b].

12.2 Analysis of the Results

Objective answers. The performance values gathered from the answers to the objective questions can be used to test the hypotheses formulated above. As mentioned in Section 12.1.1, the questions can be divided into two sets. The first set of questions has an isolated focus on crosscutting concerns, whereas the second set has an interrelated focus. Correspondingly, the first set is used to test Hypothesis 1, and the second set is used to test Hypothesis 2. The detailed performance data for the experiment can be found in the appendix of [Meie09b].

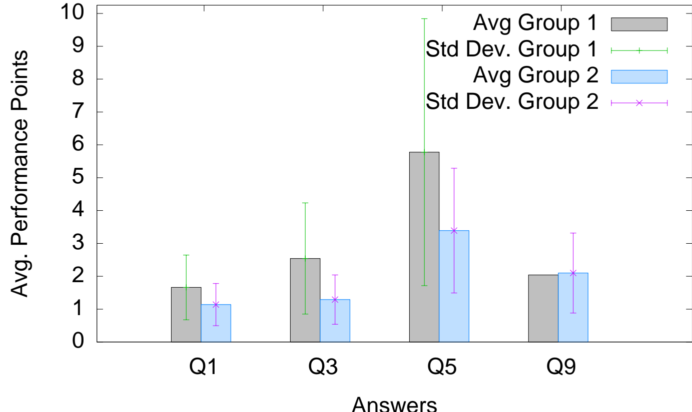
Figures 12.5 (a) and (b) show the average of the performance points achieved by each group for each question. Figure (a) presents the results of the questions about the case study 1, where group 1 used the aspect-oriented model and group 2 had only the conventional model. In (b) the results for case study 2 are given, where group 2 used the aspect-oriented model and group 1 used only the conventional model.

The results in (a) are as expected — the performance of the people in group 1 using the aspect-oriented model is better than the performance of the people in group 2. However, it must be shown that the differences in the performance of group 1 and group 2 are significant, which is subject to the hypothesis test presented below.

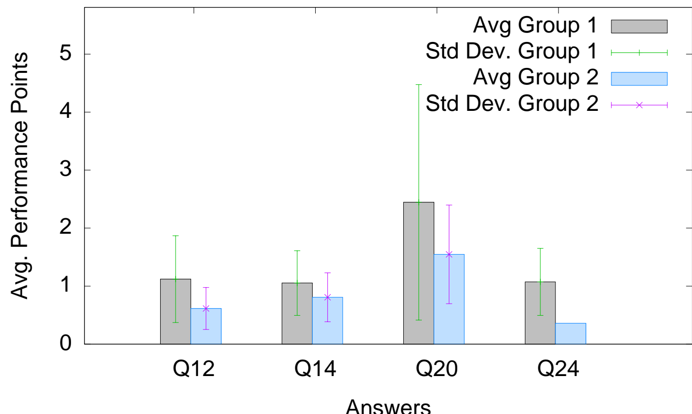
The averages shown in (b) are contrary to expectations: group 2, which had used the aspect-oriented model for case study 2, performed worse than group 1. As discussed in [Meie09b], there may have been various reasons for this. There is evidence that the test may have been biased by a too short training phase which may have in uenced the performance on the more complex badge system model.

In Fig. 12.6, the average performance points achieved by each group for the questions are shown. In (a) the only relevant question of case study 1 is Q7. According to Hypothesis 2, tasks with an interrelated focus should perform worse on an aspect-oriented model. However, the people of group 1 using the aspect-oriented model performed slightly better, which is contrary to expectations. The results shown in (b) are as expected: the people of group 2, who used the aspect-oriented model, performed worse or approximately equal to the people of group 1.

Hypotheses 1 and 2 can be statistically tested with the performance data resulting from the experiment. The corresponding hypothesis tests for the objective questions can be found in the appendix of [Meie09b]. Both Hypotheses 1 and 2 were tested against the performance data with

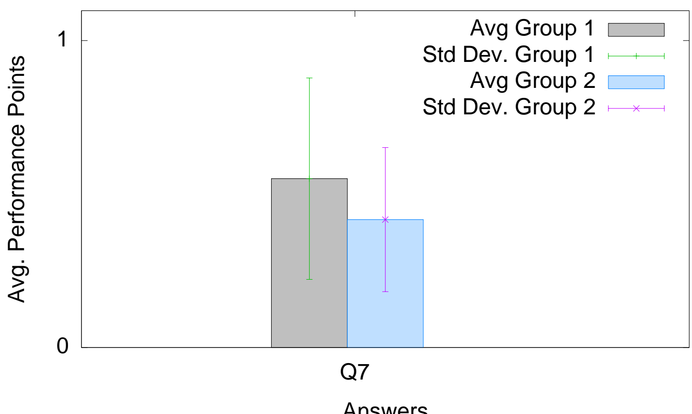


(a) Tasks performed on the banking system, group 1 used the aspect-oriented model

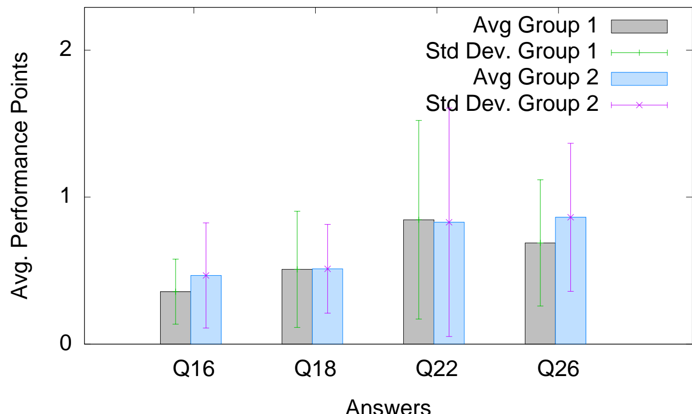


(b) Tasks performed on the badge system, group 2 used the aspect-oriented model

Figure 12.5: Average performance results per question for the objective questions with an isolated focus on the crosscutting concerns. The dark gray bars show the results of the first group, the light gray bars the results of group 2. The error bars denote the standard deviation around the mean, i.e., longer bars show a higher variance in the results.

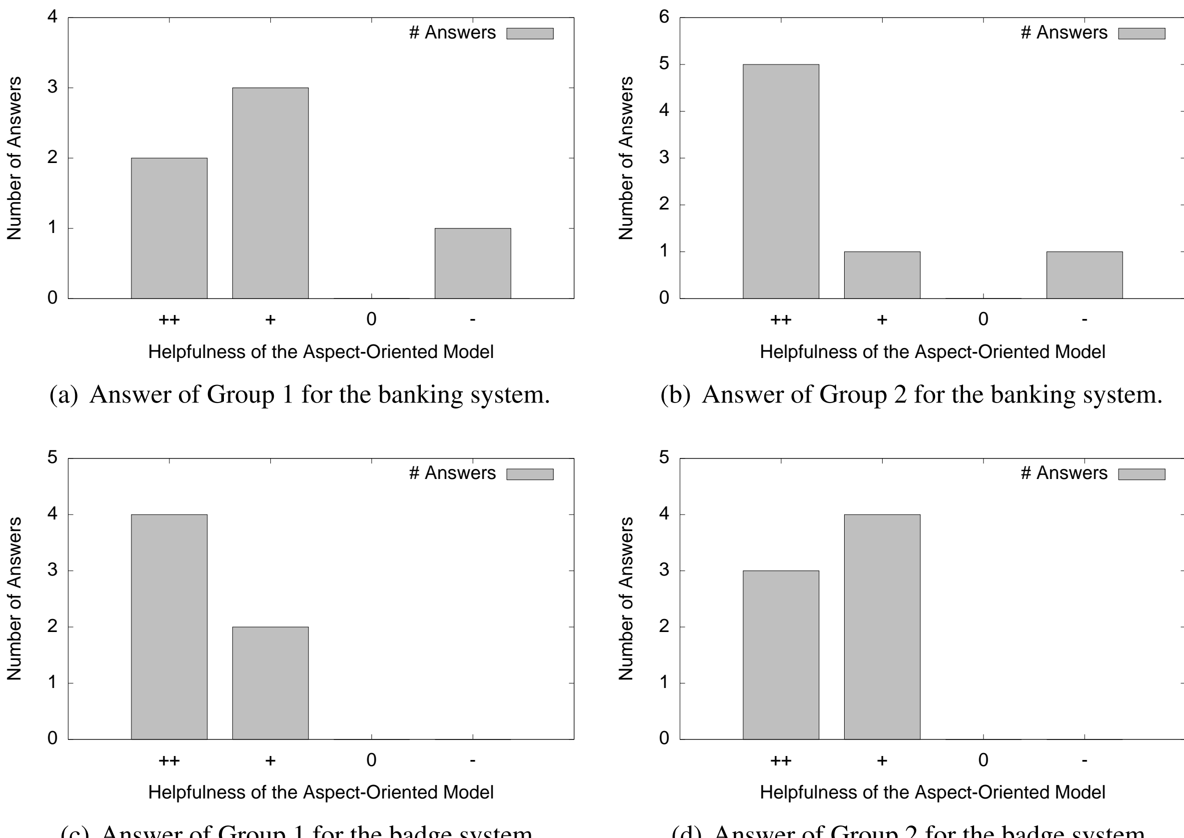


(a) Tasks performed on the banking system, group 1 used the aspect-oriented model



(b) Tasks performed on the badge system, group 2 used the aspect-oriented model

Figure 12.6: Average performance results per question for the objective questions having an interrelated focus on the crosscutting concerns. The dark gray bars show the results of the first group, the light gray bars the results of group 2. The error bars denote the standard deviation around the mean, i.e., longer bars show a higher variance in the results.



(a) Answer of Group 1 for the banking system.
(b) Answer of Group 2 for the banking system.
(c) Answer of Group 1 for the badge system.
(d) Answer of Group 2 for the badge system.

Figure 12.7: Subjective answers about the usefulness of the aspect-oriented modeling view. (++) means the aspect-oriented model is helpful, (+) means that the aspect-oriented model is partially helpful, (0) means that the aspect-oriented model is not helpful but it also does not hinder the work, and (-) means that the aspect-oriented hinders the work with the model.

an error probability of 5%. According to the hypothesis test, Hypothesis 1 is confirmed for simple models, such as the one in Fig. 12.5. However, for more complex models, e.g., models which consist of transitively crosscutting concerns, Hypothesis 1 cannot be confirmed. As discussed in the appendix of [Meie09b], this may be the result of the rather short training phase for the ADORA language and its aspect-oriented extension, which may have biased the results of the aspect-oriented model. Therefore, Hypothesis 1 cannot be rejected totally for more complex models. In contrast, Hypothesis 2 is fully confirmed by the data gathered from the experiment.

Subjective answers. The detailed results of the subjective questions can be found in Fig. 12.7 (a)–(d). The majority in both groups of test persons appreciated having an additional aspect-oriented model, which is shown by the high agreement that the aspect-oriented model is helpful in answering the given questions.

12.2.1 Validity of the experiment

Summarizing, the following guidelines were followed during the experiment so that it would yield valid results:

- The case studies should adequately reflect the occurrence of crosscutting concerns in models.
- The models used should be of good quality, i.e., the aspect-oriented models and the conventional models must be consistent and the models must be meaningful.
- Every effort was made to ensure that the questionnaire should consist of fair questions with various levels of difficulty.
- All participants should have (approximately) equal training in ADORA and the aspect-oriented extension. Thus, their skills in reading ADORA models were comparable.
- The subjects should be randomly divided into groups with an approximately equal number of members and the number of male and female subjects each group was guided to be equal.[7]
- The results of the experiment should be collected anonymously.
- Each subject should answer the questionnaire under the same conditions.

However, two factors may have influenced the results: first, the results of the case study may have been biased by a rather short training phase for the ADORA language and its aspect-oriented extension, as discussed in [Meie09b]. Despite a possible bias, the results confirm Hypothesis 1 partially, and Hypothesis 2 fully.

Second, for a good reflection of the statistical population, a large sample is desirable. However, the experiment was conducted with rather small groups of test persons which could possibly lead to statistical outliers in the data, as the sample of subjects might only reflect a specific cluster of the population.

12.3 Summary

In this chapter, an experiment and its results for testing Hypotheses 1 and 2 were presented. The results of the experiment confirm Hypothesis 1 partially and Hypothesis 2 fully. Moreover, both hypotheses are also supported subjectively by the participants.

For the sake of simplicity, the presented experiment focused only on the reading and interpreting of models containing crosscutting concerns. Reading and interpreting are the basic tasks (cf. Table 12.2) on models containing crosscutting concerns which must be performed for any more complex task, such as *manipulating the behavior of crosscutting concerns* or *adding a new*

[7] Note that the number of female participants was rather small.

impact location in the model. Therefore, the experiment covered a major part of the effort spent on performing tasks on models containing crosscutting concerns. However, it would be desirable to test Hypotheses 1 and 2 also on manipulation tasks, since the resulting performance might differ from the tested cases. Moreover, it would be desirable to conduct more experiments in order to reproduce and confirm the above results.

Two further issues must be taken into account for future experiments on aspect-oriented requirements modeling. First, the models used for the experiment presented in this chapter are rather small. Thus, it would be desirable to test the hypotheses on real world models. Second, in future experiments, a more extensive training phase for the ADORA language and the aspect-oriented extension should be considered so as to eliminate possible biases in the results.

Part IV

Conclusions

Chapter 13

Conclusions

The majority of contemporary requirements methods suffer from the communication gap caused by problem-exogenous complexity (cf. Section 1.1) arising from a suboptimal way of describing the requirements. The communication gap contained in software requirements specifications is especially critical because it has negative impacts on the software project, such as poor intelligibility or higher maintenance costs. Furthermore, there are various types of software project stakeholders, such as customers, who are usually non-experts in the field but who are indispensably involved. Thus, software requirements specifications must be as simple as possible but should not omit any important information. Consequently, they must minimize unnecessary complexity. Aspect-oriented requirements approaches help to avoid issues of tangling and scattering by describing crosscutting concerns modularly. Therefore they reduce the problem-exogenous complexity and overcome the resulting problems.

However, there are not only advantages that accrue from the use of aspect-oriented techniques at the requirements phase. Aspect-orientation introduces a new form of complexity, as a system is more highly modularized than with conventional techniques, which introduces more relationships between modules. Trying to follow the complex interplay between core and crosscutting concerns may prove to be as much or even more of an obstacle to understanding the system, as compared to using a conventional modularization of the system. Consequently, there is a need for a composition mechanism so as to switch between the aspect-oriented and the conventional description, which allows the user to choose the more adequate view of a system.

Some of the existing approaches, such as the the aspect-oriented development with use cases [Jaco03, Jaco05], do not posses an explicit composition mechanism. Moreover, current aspect-oriented requirements approaches suffer from several other problems (cf. Chapter 4): some of them use a representation which is rather difficult to read, understand, and communicate, such as the aspect-oriented requirements engineering with Arcade [Rash03] or the Concern-Oriented Requirements Engineering approach [More05a, More05b]. Other approaches, such as ARGM [Yu04] or Theme/Doc [Bani04c, Bani04a, Clar05] have representations that do not scale well for larger requirements specifications. Scenario modeling with aspects (SMA) [Arau04, Whit04] or Aspect-oriented software development with use cases (AOSD/UC) [Jaco03, Jaco05] do not adequately address non-functional requirements which are inherently crosscutting.

13.1 Discussion and Contribution of the Present Work

The approach presented here overcomes the problems delineated above. It is based on the requirements modeling language ADORA, which innately possesses several characteristics that are desirable for an aspect-oriented approach.

ADORA is a modeling approach which is based on abstract objects. It is well suited as a means for describing and communicating software requirements. Furthermore, it allows the recording of requirements on various levels of formality by a mixture of formal and informal descriptions. The use of abstract objects is a good means for communicating system requirements. Moreover, it enables their hierarchical decomposition which in turn allows the requirements to be displayed in an integrated visualization.

Abstraction mechanisms, such as a zoom mechanism and the view management, help to manage the hiding of model elements that are not at the focus of interest. The hierarchical decomposition and the abstraction mechanisms allow reduction of some of the problem-exogenous complexity in the requirements, as it simplifies the way a modeler can integrate single model parts in his mind (cf. Section 4.1). Furthermore, the formal and semi-formal simulation techniques presented in [Seyb06a] allow an engineer to dynamically validate, evolve and revalidate the behavior of a system.

The approach presented extends the ADORA language to an *asymmetric*[1] *general-purpose* aspect-oriented approach (cf. Section 3.2.1). The weaving semantics allows switching between the aspect-oriented view and the conventional view of a software model.[2]

13.1.1 Summary, Discussion, and Contribution

To allow a comparison with the other presented state-of-the-art aspect-oriented requirements approaches discussed in Section 4.1, the assessment of the proposed approach is based upon the evaluation schema presented in Table 4.1.

Concern handling. The aspect modules introduced in Chapter 7 allow the modeler to properly separate and modularly describe functional as well as non-functional crosscutting concerns in ADORA. Aspect modules are self-contained containers that allow the encapsulation of parts of a crosscutting concern, such as behavior chunks or crosscutting statecharts, and scenariocharts. A join relationship expresses that an aspect crosscuts another aspect module or a component.

The aspect-oriented ADORA approach overcomes the problem of fragile join points as it does not allow the formulation of quantifications which are used to dynamically determine the the target of an aspect. Fragile join points are problematic in most of the approaches discussed in Section 4.1. Instead, the impact locations are explicitly defined by join relationships. Critics may object that this is not useful as there are too many of them. However, the resulting number is still

[1] However, as the approach describes crosscutting relationships separately from the aspects, it can also partially be seen as a symmetric approach.

[2] The weaving mechanism is currently implemented as a static composition mechanism but may be extended to a dynamic composition in future.

manageable, as requirements are on a quite abstract level. Moreover, the abstraction mechanisms presented in Chapter 8 help to reduce unnecessary information.

The crosscut elements are oblivious (cf. Section 3.1.8) of the crosscutting elements, which is also achieved by the join relationships and the mapping of any accessed elements in the context to a independent variable. However, the approach does allow a breaking of the information principle: the crosscut elements can be manipulated by the crosscutting modules so that the software contracts are no longer valid. However, an approach which might mitigate the problem is briefly sketched in Section 13.2.

Composability. The proposed approach specifies a weaving semantics which allows an automated composition of aspect-oriented requirements models. Therefore, it allows switching between the aspect-oriented and the corresponding conventional view, depending on which view is better suited for the reader/writer of the model. Hence, this mechanism facilitates better understandability and an easier validation/verification of the requirements.

In contrast to other state-of-the-art methods, the present approach also employs a *visual composition* for graphical models which tries to preserve the secondary notation of the model as far as possible. This is especially important for the easy comprehensibility of automatically transformed models. It allows the reader of the model to grasp the interrelationships between the elements of the aspect-oriented model and their counterparts in the conventional model more easily. This is necessary if any advantages are to be gained from a composition mechanism.

Trade-off and decision support. There are two kinds of conflicts that may occur when using crosscutting concerns. The first conflict occurs if a crosscutting concerns has a (cyclic) impact on itself, which must not be allowed because this is not meaningful. The present approach does not allow such cyclic crosscutting, and avoids this by employing a strictly enforced language constraint. The violation of the constraint is indicated to the modeler immediately before the offending operation is executed. A modeler recognizes this type of conflict during the modeling of complex crosscutting concern structures and may be able to find a resolution at an early point in time.

The second conflict occurs if a target is impacted several times by different join relationships having the same weaving order on the target. Even though the conflict resolution must be done manually, the solving of these conflicts is supported by the explicit visualization of the join relationships. In contrast, conflicts are more difficult to find in presented state-of-the-art approaches which describe crosscutting relationships with implicit quantification mechanisms. The conflict is resolved by using the priority of the join relationship which defines the weaving order precedence of the crosscutting elements.

Mapping. Aspect-oriented ADORA is well suited for being mapped to approaches at later stages of the software process. As it is based on object-oriented modeling techniques, it is predestined for being mapped at later stages to aspect-oriented approaches that are also based on object-oriented techniques. However, due to its clear separation of concerns, a mapping to other

aspect-oriented technologies is also possible. Furthermore, the resulting artifacts can be mapped easily to a symmetric approach at a later stage although the ADORA approach is asymmetric.

Modi ability and Evolvability. The hierarchical decomposition and the aspect-oriented elements of the present approach facilitates good modularization which, in turn, fosters easy modifiability. Easy modifiability is also supported by the use of a consequently employed decomposition, the integrated visualization, and the abstraction mechanisms. The latter allows concentrating on the facts that are currently at the focus of interest, without the danger of being overwhelmed by a huge amount of information. Furthermore, the checking of language and user-defined constraints allows inconsistencies in the model to be found more easily. The abstraction mechanism and the language constraints checking are tool-supported, which is necessary for an efficient handling of the models.

A simple evolvability is fostered by the partial indicator which helps to plan the evolution. Setting the partial indicator of components and aspect modules (cf. Chapter 7 and 10) specifies that the elements are intentionally incomplete and need some further elaboration. Similarly, associations or join relationships may be partial (called abstract) and indicate an incompleteness. The partial indicator for aspects and join relationships facilitates the early separation and evolution of crosscutting concerns. Moreover, the decomposition of aspect modules can be used to evolve non-functional crosscutting concerns. Moreover, aspects can be hierarchically decomposed into sub-aspects and can therefore be used to express the operationalization of non-functional requirements. This feature allows the relation of the more evolved non-functional requirements to the corresponding high-level non-functional requirement. Furthermore, the approach is also accompanied by a very coarse-grained process framework described in Chapter 10 that can be seen as a rough guideline on how to modify and evolve models. The process also employs the semi-formal simulation techniques presented in [Seyb06a].

The language elements presented together with the outlined process support the separation of non-functional crosscutting concerns at the first round of the software process. In contrast, the identification of functional crosscutting concerns is not supported explicitly: crosscutting concerns are evolved by the simulation process presented in [Seyb06a] and therefore at first tangled with the core concerns. After that, they need to be identified, e.g., by some sort of aspect mining, and separated manually. However, the improvement of the identification and separation of functional crosscutting concerns will be at the focus of future research on the topic (cf. Section 13.2).

Scalability. Scalability is supported by the hierarchical decomposition, the integrated visualization, and the abstraction mechanisms of the ADORA approach. These three mechanisms have been extended in order to be able to handle aspect-oriented elements discussed in Chapter 7.

The hierarchical decomposition allows a modeler to separate and modularize the requirements of a complete system into small and manageable system parts. The integrated visualization allows a reader to comprehend a model more easily than with a non-integrated visualization of a model, which is due to the fact that there are no separated parts that have to be integrated in mind. Finally, the vertical, horizontal, and crosswise abstraction mechanisms (cf. Section 5.1.4)

allow parts of the model that are currently not at the focus of interest to be hidden, but without losing the context of the element that is at the focus.

Understandability. Just as for scalability, the intelligibility is fostered by the hierarchical decomposition, the integrated visualization and the abstraction mechanisms.

Aspect-oriented elements (cf. Section 7) and the composition mechanism (cf. Chapter 9) allow improved understandability when crosscutting concerns are involved. The aspect-oriented elements improve the understandability of the model when the reader of the model has a local focus on the concern. In the case that the reader of the model has an interrelated focus, the conventional view is better suited. This can be generated by the composition mechanism.

The formal and semi-formal simulation techniques presented in [Seyb06a] also support the understandability of the model by allowing models to be executed and the semantics of the modeled content to be apprehended by dynamic means. This is also a means for non-expert stakeholders to grasp the meaning of the software.

Validation. The manual validation of aspect-oriented ADORA models is supported by the same means as discussed for their understandability. The hierarchical decomposition, the integrated visualization, and the abstraction mechanisms allow the easy comprehension and therefore the easy validation of an aspect-oriented ADORA model. Moreover, the strict separation of crosscutting concerns allows an easier understanding of a model when having a local focus on the crosscutting concerns. The model woven by the composition mechanism presented in Chapter 9 helps to validate the understanding of the interrelationships between crosscutting concerns. Furthermore, the simulation of formal and semi-formal models support the automatic or semi-automatic validation and revalidation of models [Seyb06a].

Traceability. There is no explicit support for traceability in aspect-oriented ADORA. Nevertheless, some of the existing concepts facilitate traceability in a straight-forward way. Multi-user editing [Moda03] also enables forward and backward traceability of ADORA models in general. Moreover, the ability to decompose aspects allows in particular a forward and backward tracing of non-functional concerns. Standardized properties (cf. Section 5.2.5, p. 80) allow the stakeholders who are responsible for a requirement to be recorded. Cross-referencing of elements is inherently provided by the ADORA modeling language. Nevertheless, automatic mechanisms which exploit the traceability data to improve the modeling process are currently not part of the approach.

13.2 Outlook

The present work covers a very broad topic, and consequently, it has not been able to analyze all the details that may be of interest. The corresponding open issues that may be part of the future research on the present approach are outlined in the following.

Composition. There are several issues with respect to the composition of aspect-oriented models which may be improved in the future. First, the composition mechanism presented for aspect-oriented models is unidirectional, which means that aspect-oriented models can be transformed to conventional models but not vice-versa. However, a back transformation would be desirable as it would allow changes made in the conventional model to be propagated back to the aspect-oriented model.

Second, the visual weaving mechanism presented works satisfactorily for smaller models. Nevertheless, models with a more complex layout may end up requiring major changes in the secondary notation (cf. Section 5.1.4). One of the major causes for these problems is the missing inclusion of the model elements' orientation (cf. Section 9.5) during the weaving. A future investigation must research certain heuristics which could cope with this problem.

Third, when executing a model, it has to be woven first. However, it would be useful to weave dynamically on the y, i.e., during the execution of the model. This would also allow a quasi-direct execution of an aspect-oriented ADORA model. This kind of dynamic weaving may become the topic of the future research.

Visualization. The visualization and abstraction mechanisms presented could be improved in several ways. First, the current mechanisms are only designed to hide or show *no or all* join relationships together (cf. Chapter 8). However, it would also be desirable to hide and show join relationships individually, or to show and hide any other connection in the model.

Second, the aspect-oriented visualization of the present approach only distinguishes between the modules of crosscutting concerns and core concerns. However, readers of the model would benefit from a mechanism which allows them to classify to which concern a module belongs. This classification would allow them to hide and to show only the modules which belong to particularly selected concerns. This kind of mechanism would help to achieve a multidimensional visualization/separation of the concern space, as proposed by HyperJ [Tarr99] and CORE [More05a, More05b] (cf. Section 3.2.5), which would help to improve the understanding of a model, too.

Improving the aspect-oriented concepts presented. There is an investigation of the usefulness of the presented aspect-oriented visualization concepts in order to improve the understanding of the models (cf. Chapter 8). Even though the results show that the concepts presented tend to be useful, it is necessary to do additional empirical tests to adduce evidence for the usefulness of the approach. First, the results presented have to be reproduced again by similar experiments. Second, there are several tasks where the usefulness of aspect-oriented constructs still needs still to be shown, such as when editing models.[3]

Preserving the information hiding principle. The introduction of aspect-oriented constructs can lead to the breaking of the information hiding principle and consequently result in problems

[3] In contrast, the experiments presented concentrated only on the reading of the models.

for a design by contract (cf. Section 3.3.3). Both issues also become evident for the aspect-oriented ADORA approach. However, these problems are not explicitly covered by the present work.

Having a look at the problem reveals that a contract of a crosscut module may only be broken by a crosscutting aspect if the aspect alters the state of the crosscut target module. A resolution or at least a mitigation of the problem may be an extension of the current concept which introduces abstract annotations for the crosscut targets (cf. Section 3.3.4). Abstract annotations denote particular points in the crosscut module where an aspect may cut across. Aspects that *modify* the state of the target are only allowed to crosscut at these specified points. In contrast, aspects that only read the state of the target may still crosscut anywhere. Furthermore, the annotations may specify a contract stating under which conditions a modification of the target module state is allowed by an aspect. An aspect must not violate the specified crosscutting contract. The contract may be defined in a similar fashion to contracts as given by the design-by-contract principle using pre- and postconditions.

Using this kind of mechanism is a compromise between losing obliviousness and exibility (see also Section 3.3.4). However, future research on the topic must fully work out this idea.

Other open issues. There are some more research issues that have not been covered by the present work. First, the process sketched in Section 10.1 must be refined and validated to be useful for practical employment, and the interplay between the process and the semi-formal simulation mechanism must be elaborated in more detail. Furthermore, the present approach assumes that functional crosscutting concerns are manually discovered and separated during the evolution of crosscutting concerns. However, an automatic discovery by an aspect mining approach could support the modeler with this task.

The present approach implicitly supports traceability. Nevertheless, it does not provide any mechanisms to follow the traceability links. A mechanism which allows the automatic tracing of requirements and concerns in a forward, backward, cross-referencing manner would be desirable.

Furthermore, it has been shown that the present approach can be also extended [Stoi07, Stoi08] in order to be used for handling variable software requirements.

13.3 Conclusion

From the author's point of view the approach presented has the potential to greatly simplify models that contain complex crosscutting concern structures on the requirements level. However, aspect-oriented techniques at the requirements stage are not necessarily useful in every case. In simple cases where just a few crosscutting concerns are part of a described system or where the system itself is rather simple[4], using aspects does not lower the complexity, or may even introduce a higher degree of complexity. However, in other cases, such as the use for software variability, the presented approaches may bring advantages over the use of conventional methods.

[4] Most of the examples in this work are rather simple.

Part V

Appendix

Appendix A

Discussion of Aspect-Oriented Requirements Approaches

In the past, plenty of conventional and aspect-oriented requirements approaches have been proposed. Based on the surveys of [Chit05, p. 19–113], [Arau05], and [Bono04], several aspect-oriented and conventional requirements approaches are evaluated in the following. First, a set of evaluation criteria is derived from the characteristics of a good specification in Section 2.2.1 and the criteria given in [Chit05]. Second, the approaches are presented brie y and evaluated according to those criteria.

A.1 Evaluation Criteria

The criteria presented in Table A.1 are used to judge the quality of the requirements approaches under evaluation. The evaluation criteria are derived from the characteristics in [Chit05, p. 19–113] and the general quality characteristics of requirements documents in Section 2.2.1.

Table A.1: Criteria for the evaluation of aspect-oriented requirements approaches.

Criteria	Description
Traceability	How well can requirements be traced through the software process? The support for traceability can consist of the recording of the *source of requirements* as well as the sources for changes in the requirements. *Forward traceability* allows an interested party to trace the refinement of requirements to the resulting architectural, design and implementation artifacts. *Backward traceability* allows the artifacts created in later stages of the software process to be traced back to the originating requirements. *Cross referencing* allows the tracing of the dependencies of artifacts between the requirements on the same level of abstraction/refinement.

Continued on the next page . . .

Criteria	**Description**
Composability	Is there explicit support for *decomposing crosscutting concerns* and for *composing* them to a set of integrated artifacts? Is there a mechanism which allows automatic composition?
Modifiability and Evolvability	How well can *changes* be introduced in the requirements artifacts? Are there features which support the change of the requirements document, such as *integrity checking* mechanisms?
Scalability	How well does the approach scale? Is the provided *process* capable of handling the requirements of a small scale as well as large scale project? Are the corresponding *artifacts* describing the requirements able to describe small scale as well as large scale software?
Understandability	How difficult is the approach to understand for any stakeholder? Are the artifacts produced by the approach *easy to understand*, can they easily be used to *communicate with a stakeholder*, e.g., during validation? Usually, an approach which uses concepts that are related to the domain, such as objects or workflows, is easier to understand than an approach that uses more abstract concepts.
Trade-off	How well is the identification and resolution of trade-offs between different overlapping concerns (non-orthogonal aspects) supported by the approach?
Mapping	How well can requirements artifacts be mapped to following stages? Does the approach provide such a *mapping*? Does the paradigm used at the requirements stage support a simple mapping?
Verification and Validation	How easy is it to conduct a validation or verification with the artifacts resulting from the requirements process? Usually, *dynamic means of validation or verification* are more convenient and more efficient in finding problems, than *static means of validation and verification*.
Concern handling	How well is the *identification and separation of functional and non-functional crosscutting concerns* supported? Are all concerns *treated* in the same way or are there differences between non-functional and functional concerns? Can both types be represented adequately?

A.2 Conventional Approaches

A.2.1 PREView

Viewpoint-oriented approaches [Fink92, Fink96] take up a stakeholder-centric perspective on the system to be developed. Such a perspective is called a viewpoint. As an example, PREView [Sawy96] is discussed briefly.

There are two main artifacts in PREView: concerns and viewpoints. *Concerns* denote an overall matter of interest in the software and cut across all viewpoints. Nevertheless, concerns are usually of a non-functional nature.[1] A concern can imply external requirements, constraints or questions. External

[1] In PREView, the term *concern* does not have exactly the same meaning as the term *concern* in the aspect-oriented approaches.

requirements are directly derived from a concern and cannot be in uenced by a stakeholder and may even override the requirements of stakeholders. Questions are used to check the consistency of requirements. Constraints can be formulated for each concern to reveal inconsistencies during the (usually) iterative process.

In contrast to a concern, a *viewpoint* is a stakeholder-centric perspective on the system, which is used as a starting point to find the user requirements. Thus, after elicitation, a viewpoint may be assigned requirements, i.e., usually functional requirements imposed by the stakeholder.

PREView is supported by its own requirements process. It starts by finding the concerns, which are then refined to a set of external requirements, constraints and questions. Subsequently, the stakeholders and their viewpoints are identified. In the following discovery phase, new requirements for the viewpoints are found. The interactions between the viewpoints are then analyzed and inconsistencies resolved, which may result in feedback cycles. The feedback may in uence the already found concerns and viewpoints. The evaluation of PREView can be found in Table A.2.

Table A.2: Evaluation of the PREView approach.

Criteria	**Description**
Traceability	A viewpoint template has a focus and a source, which allows its origin to be identified. Concern and viewpoint templates have a history section which allows backward traceability. The origins of external requirements and viewpoint requirements are traceable to the corresponding concerns and viewpoint, respectively.[2]
Composability	Even though PREView decomposes some of the crosscutting concerns of a system,[3] there is no composition mechanism. The (crosscutting) requirements of concerns and the requirements of the user viewpoints are handled separately. Thus, if the composition of the artifacts is needed, it has to be done manually.
Modifiability and Evolvability	PREView does not actively support a simple change management and evolution of artifacts. Thus, broader changes in the concerns can lead to severe changes to the requirements derived from the viewpoints, especially when there are many concerns and viewpoints (cf. Scalability). Nevertheless, the approach supports constraints which can help in finding inconsistencies during the process.
Scalability	The more viewpoints and concerns there are that need to be handled by the approach, the more con icts occur between crosscutting and base requirements. Since the approach does not provide sufficient means to resolve con icts, PREView can be used only with a small number of concerns and viewpoints (cf. Modifiability).[4]

Continued on the next page ...

[2] This is achieved by unique identifiers.

[3] Due to the non-functional nature of the term *concern* in the approach, it is especially the non-functional requirements that are handled.

[4] In [Chit05, p. 100], PREView is said to encounter problems with more than six concerns.

Criteria	Description
Understandability	For a small number of artifacts, the approach is unproblematic. However, for systems with a larger number of concern and viewpoint artifacts the relationships between the artifacts can become unclear. This is due to the fact that the contextual linkages between the requirements originating in concerns and the requirements originating in viewpoints as well as their relationships cannot be determined easily.
Trade-off	Organizational concerns and external requirements take precedence over viewpoint requirements, which may result in suboptimal solutions.
Mapping	In PREView, there is no predefined way for mapping the requirements artifacts to later stages.
Verification and Validation	The approach does not directly support the validation or the verification of the software. Nevertheless, this may be important to the approach, as there may be a lot of con icts between external requirements and user requirements.
Concern handling	The approach considers the non-functional concerns (organizational concerns) as primary artifacts. Functional concerns are secondary artifacts. Crosscutting functional concerns are not considered in the approach. The approach partially supports the discovery of crosscutting concerns. The concern concept of PREView deals with non-functional requirements which are inherently crosscutting. Nevertheless, the elements belonging to the NFR can scatter over different artifacts. Moreover, functional crosscutting concerns are not detected by the approach.

A.2.2 NFR Framework

Goal-oriented approaches generate requirements from objectives that must be achieved by a system. The goal-oriented approaches KAOS, I* and the NFR Framework are brie y sketched and evaluated in the following.

The *Non-Functional Requirements Framework* (NFR) [Chun00] concentrates on the representation and analysis of non-functional requirements. The framework uses *soft goals* to represent non-functional requirements and provides a catalog for a refinement of soft goals. Refined soft goals are satisfied by *operationalizing soft goals* which represent a set of design or implementation solutions. Furthermore, *claim soft goals* can be used to represent domain characteristics in the decision process. Correlations denote relationships between NFRs which either express the support or the con ict of a NFR by another NFR.

Soft goals, *re ned soft goals*, operationalizations of *soft goals*, and *claim soft goals* together form the soft goal interdependency graph (SIG). The approach aims at finding the configuration in the SIG which satisfies best a given goal. Operationalization links allow the best solution for an NFR to be connected with the functional requirements. The NFR process defines how a SIG is created and the best variant of the SIG is found. Refinement catalogs support the and guide the process. The evaluation for the NFR approach is given in Table A.3.

Table A.3: Evaluation of the NFR approach.

Criteria	Description
Traceability	The SIG allows the tracing of the dependencies of the artifacts resulting from the refinement, the operationalization, and relationships to *claim soft goals*. The links between the FR and the NFR allows the dependencies to be traced between both. However, the origin of the high-level *soft goals* is not recorded.
Composability	The approach does not allow the composition of the artifacts. However, indirectly the links between the artifacts can be created and used for a manual composition.
Modifiability and Evolvability	The approach allows a refinement/evolution of the SIG elements by means of catalogs which guide the refinement and operationalization of elements. Nevertheless, changing a *soft goal* can have a major impact on the SIG which in turn may cause major changes in the artifacts.
Scalability	The approach works only for small systems, as the SIG can become unclear rather quickly and therefore unmanageable for larger systems. Nevertheless, the catalogs provided by the approach help in handling larger systems.
Understandability	The concepts behind the NFR approach are goals which are rather abstract concepts that are far removed from the actual domain concepts. This may lead to problems in the communication with stakeholders.
Trade-off	There are correlation catalogs which help in understanding the effects on impacted non-functional requirements. Furthermore, the SIG helps interested parties to visualize and understand the relationships between non-functional requirements and alternative operationalizations. Thus, the trade-off analysis and the corresponding decision taking is well supported.
Mapping	There are no specific guidelines for the mapping of requirements artifacts to artifacts in later stages. However, the catalogs support systematic decomposition and linking to the functional requirements. Indirectly, this linking also simplify indirectly simplifies the mapping to artifacts in later software stages.
Verification and Validation	There is no specific means for validating/verifying the artifacts of the NFR approach. However, it is possible to visually inspect and assess the SIG and the other artifacts.
Concern handling	The framework mainly supports non-functional concerns, which are inherently crosscutting. However, there is no clear-cut way to identify non-functional requirements.

A.2.3 KAOS Approach

The Knowledge Acquisition in Automated Specification (KAOS) approach [Dard93, Dari97] deals with acquisition of requirements and handling formal specifications. The approach provides a conceptual model for acquiring requirements and supporting languages, as well as a set of elaboration strategies.[5] One of the conceptual meta-models that can be chosen is goal-directed. It comes with the corresponding requirements

[5] The choice of the appropriate strategy is supported by an automated assistant.

acquisition strategy. A knowledge base comes with predefined heuristics and tactics for the requirements acquisition and also provides domain specific knowledge.

Goal-decomposition trees are artifacts resulting from a goal-directed strategy. The goal acquisition is driven by a process which first identifies goals and the objects concerned, the potential agents and their capabilities. In the subsequent process the goals are operationalized, actions and objects refined. There may be feedback cycles which strengthen other objects and actions as a consequence of the refinement. Subsequently, the responsibilities of agents are determined and the corresponding actions are assigned to the responsible agents. The evaluation criteria for KAOS are discussed in Table A.4.

Table A.4: Evaluation of the KAOS approach.

Criteria	Description
Traceability	The decomposition graph allows the artifacts to be linked on different abstraction levels. It is possible to create links to interview transcripts, so that the source of a goal can be traced. The traceability to artifacts later in the software process is not directly supported. Nevertheless, tasks are assigned to agents, which may facilitate the traceability of artifacts at later stages.
Composability	There is no direct support for artifact composition in KAOS. However, the structure of the decomposition tree allows the parts which belong together to be collected manually.
Modifiability and Evolvability	KAOS allows the reuse of knowledge, which facilitates the evolution. The approach also allows analysis of the dependencies resulting from the relationship of agents, which in turn allows analysis and understanding of the impact of changing, removing, or adding requirements.
Scalability	In KAOS, scalability is reduced since this approach can result in a huge effort invested in formalizing a task (pre-, postcondition, etc.). This is not practical for the majority of large systems. Moreover, the huge database of domain data collected may be a hindrance to better scalability.
Understandability	The approach uses a rather abstract goal concept which is far from most application domains it describes. Therefore, the approach may feel less tangible for stakeholders. Moreover, there are many formal elements, which may lead to difficulties in understanding for non-experts.
Trade-off	Trade-offs between goals are solved by priorities assigned to the goals. Furthermore, trade-off decisions are supported by a broad knowledge base, tactics and heuristics.
Mapping	The decomposition catalogs help to map non-functional as well as functional goals to artifacts in later stages of the software process. Tasks could possibly be directly mapped to procedures and functions in the system, which facilitates the mapping.
Verification and Validation	The informal decomposition graph can be used for reviews. Moreover, formal artifacts such as pre- and postconditions can be subject to an automatic verification.

Continued on the next page . . .

Criteria	Description
Concern handling	NFRs as well as FRs are equally important in KAOS. Thus, NFRs which are inherently crosscutting concerns are handled automatically by KAOS. However, crosscutting functional concerns are not treated adequately and in general there are no specific tactics for handling them.

A.2.4 I* Approach

I* [Yu01] is another goal-oriented approach which is based on the modeling of relationships between agents, goals, tasks, and resources. The *strategic dependencies* (SD) part analyzes actors and their dependencies. It aims at the understanding of the business process of a domain, and how it is affected if a dependency fails. The *strategic rational* (SR) part aims at finding different configurations of agents and their dependencies.

Both the SD as well as the SR part make use of different model types. The strategic dependency diagram describes the dependencies of goals, resources, soft goals, tasks and the corresponding agents. Goals represent objectives that can be achieved in various ways. Soft goals represent either non-functional properties or functional properties which are not clear-cut at the time of representation. Tasks are actions that must be achieved in a fixed manner. The SR diagram is used to study the dependencies between agents and other elements in a range of alternative situations.

The I* process starts with the identification of agents involved in a given system. Subsequently, the SD diagrams are constructed and further elaborated into SR diagrams. The alternatives for achieving goals identified in the SR and SD diagrams need to be analyzed for vulnerabilities and problems. For the decomposition of the goals, the process provides catalogs similar to the NFR approach. The evaluation of I* is given in Table A.5.

Table A.5: Evaluation of I*

Criteria	Description
Traceability	The source traceability is not covered explicitly in the approach. The traceability to artifacts of the later software process is also not explicitly supported but can be implicitly achieved by following the dependencies between functional and non-functional goals. The functional goals can give a hint as to where the corresponding functionality can be found in later stages.
Composability	The approach does not provide means for composing the artifacts.
Modifiability and Evolvability	The approach aims at being used in volatile environments and therefore supports modifiability quite well. It is especially supported through the framework which primarily allows analysis of different situations that may have an impact on a system.
Scalability	The approach is suited for smaller problems. Working with bigger I* graphs is rather cumbersome, even with tool support.

Continued on the next page . . .

Criteria	Description
Understandability	I*, as is the case with the other similar goal-oriented approaches, operates on a rather abstract level. The artifacts of the system to be build, i.e., the goals, are usually remote from the domain and therefore less tangible. This can result in difficulties when communicating with non-expert stakeholders.
Trade-off	The trade-off analysis is well supported by the strategic dependency and strategic rational graphs. I* allows the use of weights and priorities that help to decide which solution is better suited.
Mapping	The approach is used to identify major parts of a software architecture by agents and top-level goals and thus suited for identifying a high-level mapping. However, a detailed mapping of the I* artifacts to artifacts at later stages is not provided.
Verification and Validation	The approach allows a step by step verification or validation by walking through the SD and SR graphs.
Concern handling	Goals represent either functional or non-functional properties of a system; thus both are addressed equally. Apart from that, the approach allows the use of soft goals for non-functional properties. Thus, I* does handle non-functional concerns which are inherently crosscutting. Nevertheless, it does not clearly distinguish between crosscutting and core functional concerns.

A.2.5 Use Case Method

Use cases are defined as a sequence of actions *"performed by a system, which yields an observable result that is typically valuable for one or more actors or other stakeholders of the system"* [Jaco05, p. 404]. The sequence of actions is usually initiated by an actor. It can contain branches and iterations. Instantiated use cases are called *scenarios* and describe one single path through a use case. Use cases follow the "natural" work ow of a task in an application domain. They are intelligible for stakeholders and can therefore be easily understood. In the following the *Use Case Method* is discussed as a representative of all use case based approaches.[6]

The *Use Case Method* [Jaco92] employs use cases as its main artifacts. They are presented in textual form describing the stimulus initiating the sequence of actions as well as the actors involved.[7] Alternate ows and extension points (see below) are separately described. There are three different types of relationship between use cases: specialization, inclusion, and extension. *Specialization* allows functionality to be added to the inherited use case. *Inclusions* calls another use case at a particular point in the action sequence. *Extension* use case allows extending a use case A by any other use case B, where B is not known to A. However, the extension of A is allowed only at predefined extension points defined by A. Use cases and their relationships to other use cases as well as actors can be visualized by a UML use case diagram [Rumb05, Chapter 18].

The process of the use case method initially identifies all the actors in a system. Subsequently, the high-level system functionality needed by each actor is identified and decomposed up to an adequate level of detail.[8] Finally, the use cases are validated, e.g., through reviews in collaboration with the stakeholders

[6] Another use case approach is the method of misuse cases [Alex03], which is not discussed here.
[7] Example (3.2) sketches such a textual use case.
[8] A use case should not result in a functional decomposition which reveals the details of system functionality.

to ensure completeness and consistency. The evaluation of the use cases method is given in Table A.6.

Table A.6: Evaluation of the Use Cases Method approach.

Criteria	**Description**
Traceability	There is no explicit means for managing the source of use cases or the source of changes. However, as use cases are usually handled in structured text, it is conceivable for the source as well as the source of changes to be handled in a history associated with the textual use case. Furthermore, use cases usually cannot be properly traced to artifacts of the later stages in the architecture as they disperse over several units when they are mapped to artifacts of later stages, e.g., classes. Moreover, there is no direct support for tracking changes to the use case document.
Composability	The inclusion and extension relationships are a means for decomposing (functional) crosscutting concerns. There is usually no automatic composition. The composition is rather done manually at the architectural or design stage, when the use cases are mapped to a class structure.
Modifiability and Evolvability	Use cases can be enriched simply by new use cases with include or exclude relationships. Changes to the functionality described by a use case can be made by locating the use case and by simply changing the corresponding content. However, to maintain the consistency of the use cases, which are usually formulated in natural language, may require a validation even after smaller changes.
Scalability	For systems with a lot of use cases, it is easy to lose track of the use cases. The use case diagram provides only limited help in this regard, as it lacks expressiveness and decomposition.
Understandability	Use cases provide a good tool for communication with stakeholders, as they are easy to understand. However, the overview given by a use case diagram can result in difficulties in understanding, as crucial information, such as the order in which included or extended use cases are executed, is not visualized.
Trade-off	The use case approach does not consider con ict analysis. This has to be done at a later stage of the software process.
Mapping	There are some guidelines for mapping use cases to later stages in the software process. Furthermore, the UML collaboration construct can be used.
Verification and Validation	Text-based use cases are well suited for manual validation with stakeholders. However, natural language can be imprecise and ambiguous. Such problems are not necessarily found by an interpersonal validation process.
Concern handling	Non-functional concerns as well as functional crosscutting concerns are not identified by the process. However, the extension relationship provides the ability to decouple crosscutting (functional) concerns from the other concerns. Non-functional concerns cannot be represented adequately in use cases. They are either scattered as annotations over the use cases or not described at all.

Thus, each sequential step of a use case should stay on an abstract level.

A.3 Aspect-Oriented Approaches

A.3.1 AORE with Arcade

Aspect-Oriented Requirements Engineering (AORE) [Rash03] with Arcade[9] is an approach which can be used with any requirements technique. In [Rash03], the approach is used together with the PREView approach. AORE aims at the modularization of crosscutting concerns and the creation of a consistent requirements specification document. *Aspectual requirements* are similar to external requirements in PREView. They crosscut user requirements derived from various viewpoints.

The approach uses XML-based templates to represent the viewpoint requirements as well as aspectual requirements. Composition descriptions allow crosscutting relationships to be described, i.e., they specify how requirements and aspectual requirements are composed into an integrated specification. The composition description is also specified in XML.

The AORE process starts with the identification of the viewpoints and the collection of the associated user requirements. Subsequently, the PREView-like concerns are identified from the user requirements.[10] As soon as the concerns are identified, coarse-grained relationships between concerns and viewpoints are established. If a concern affects several viewpoints, the concern is called a candidate aspect. The aspectual requirements are then derived from the aspect candidates. Furthermore, the aspectual requirements are identified. Concerns, viewpoints and requirements are described with XML elements.

Subsequently, the engineer identifies the composition rules between aspectual requirements and the user requirements. When composing, conflicts can be identified when two or more concerns affect the user requirements of the same viewpoint. In this case, the conflicting aspectual requirements are assigned priorities. Finally, the PROBE framework [Katz04] allows the generation of proof obligations that must hold in the implementation of the aspect-oriented system. The approach is evaluated in Table A.7.

Table A.7: Evaluation of the AORE with Arcade.

Criteria	**Description**
Traceability	Viewpoints identify the sources of a requirement (cf. PREView approach) which in turn allow the sources to be related to the corresponding XML elements. However, the approach does not elaborate on how the sources of changes are tracked. AORE records the resulting type of architectural artifact (decision, function, etc.) to which a requirement evolves. Thus, it allows tracing to artifacts in later stages. The PROBE framework allows the tracing of the trade-offs to artifacts in later stages.
Composability	The approach has a clear composition semantics that is provided through composition rules and operators. Rules and operators can be adapted to a specific problem.

Continued on the next page ...

[9]Arcade is the tool supporting the AORE approach. It is mentioned to distinguish the aspect-oriented requirements engineering approach from the general field of aspect-oriented requirements engineering.

[10]Note that this is the reverse of the process in PREView, where first the concerns are identified and then the viewpoints.

Criteria	Description
Modifiability and Evolvability	AORE has a clear separation of concerns. Changes have therefore a local impact on the corresponding concern and perhaps the composition description. Thus, the changes can be handled easily.
Scalability	The XML artifacts produced by the approach are scalable. However, the composition tables can grow very large, which can affect the readability.
Understandability	Despite the claim that XML is human-legible, it is not necessarily easy to understand. Especially many cross references, a large number of documents, or large XML documents affect the understandability, especially when used to communicate with stakeholders.
Trade-off	AORE supports trade-off analysis. Con icts between crosscutting concerns are detected by the composition process and the PROBE framework. The approach supports the resolution of con icts.
Mapping	AORE provides guidelines for mapping requirements to later stages in the software process. This is done by using aspect specification dimensions which specify to what kind of artifacts, e.g., functions or decisions, a requirement is mapped.
Verification and Validation	The validation of AORE artifacts is done manually, and can be difficult (cf. Understandability). Validation and verification are supported by the generated proof obligations of PROBE, either for a formal validation or as a basis for test cases.
Concern handling	Both crosscutting as well as core concerns are handled by the approach, but, the identification of non-functional crosscutting concerns is not clearly guided. Furthermore, the identification of functional crosscutting concerns is not clearly covered by the approach. Nevertheless, the approach can be extended with aspect-mining techniques, such as [Samp05], which allow the identification of potential crosscutting concerns in the natural language description of the requirements.

A.3.2 ARGM

Aspects in Requirements Goal Models (ARGM) [Yu04] is a goal-oriented approach which is based on the non-functional requirements framework [Chun00]. Goals and soft-goals can be decomposed into subgoals and sub-soft-goals, respectively. This is done iteratively until the decomposition is reduced to a task. (Sub)soft-goals can be related to operationalizations. Correlations can represent in uences between (sub)soft-goals and goals. Together they form a goal/soft-goal interdependency graph, which is represented as a so-called *V-graph*. The approach aims at identifying aspects during goal-oriented requirements analysis. Aspects are identified as tasks with many links satisfying different goals.

The approach is supported by a process that starts with the gathering of goals and soft goals. These are then iteratively broken down into sub-goals and sub-soft-goals. During the decomposition, soft goals are correlated with goals, thereby solving con icts. Soft goals contribute either positively or negatively to a goal. The magnitude of the contribution is specified by a value between 1 and 0, which is propagated to the parent goal. If any subgoals contribute negatively to their parent, the con ict is resolved by removing the link between goal and subgoal. Thus, a goal/soft goal must not be less satisfied due to decomposition.

Finally, aspect identification gathers the tasks (operationalizations) contributing to soft goals which, in turn, contribute to a goal.

Table A.8: Evaluation of ARGM.

Criteria	**Description**
Traceability	The ARGM approach does not record the sources of a goal or a soft goal. The V-graph relates the refinement and operationalizations of functional and non-functional elements, so their origin can be traced. The artifacts are traceable into the design and implementation phase by using *operationalization links* similar to the NFR framework.
Composability	In [Yu04], there is no clear description of a composition mechanism. However, the approach allows the crosscutting concerns to be separated.
Modifiability and Evolvability	Similarly to the SIG in the NFR framework, major changes in the requirements can cause major changes in the V-graph. However, in the majority of cases, changes stay local in the graph and the effects on the graph can be easily identified by the relationships in the graph.
Scalability	The approach suffers from poor scalability. Even small models may be hard to handle (cf. for example Fig. 12 in [Yu04]), despite tool support. A partial view of the graph does not solve the problem, as some of the relationships between the nodes of a different view are then hidden. Therefore, the whole model becomes harder to understand.
Understandability	Goal/soft goal V-graphs are hard to interpret manually, due to the lack of scalability. Furthermore, as with all goal concepts, the approach is abstract. Thus, there can be problems in understanding when V-graphs are used for communication with non-expert stakeholders.
Trade-off	Trade-offs are solved by removing negatively contributing links. However, this simple trade-off solution has major draw-backs. For example, the resulting solutions can be suboptimal as better compromise solutions for the problem may get discarded.
Mapping	ARGM is based on the NFR framework, which does not specify guidelines for the mapping of requirements artifacts to artifacts in the later stages of the software process. However, the NFR provides catalogs which support systematic decomposition and linking to the functional requirements indirectly. This also facilitates the mapping to artifacts in later software stages.
Verification and Validation	The approach does not propose special means for validating/verifying the resulting artifacts. However, a manual validation/verification can be done by simply walking through the resulting graph, e.g., in a formal review with the customer.
Concern handling	The approach treats non-functional and functional crosscutting concerns equally.

A.3.3 AOSD/UC

Aspect-oriented software development with use cases (AOSD/UC) [Jaco03, Jaco05] proposes an extension to the use case method [Jaco92]. The approach can be employed throughout all phases in the software process and introduces new aspect-oriented symmetric and asymmetric constructs. The two new main elements are pointcuts and use case slices.

A *pointcut* specifies *where* a crosscutting concern cuts across other use cases, i.e., it represents a group of use case join points. *Use case slices* are UML packages containing all artifacts concerning the use cases at a particular level, e.g., the requirements stage. Furthermore, *use case modules* are packages which contain all use cases and the related artifacts which accumulate throughout the software process. Between the subpackages of a use case module, dependencies with the stereotype «trace» can be used to describe upstream and downstream dependencies in the development process.

Peer use cases are independent of each other and cut across the classes of the design. *Extension use cases* contain crosscutting functionality and *infrastructure use cases* contain functionality originating in non-functional requirements. In [Jaco05], use case templates are employed for realizing infrastructure use cases.

Extension as well as infrastructure use cases use the extension relationship to express the cutting across of other use cases. An extension use case augments one or more base use cases at a specific extension point identified by the extension pointcut. The extension flow is injected at the specified location. The meaning of the *extend* relationship of the original approach is enriched by certain elements. For example, the extending use case can define whether the extension flow is inserted *before*, *after*, or *around* the extension point. Due to the fact that the standard UML use case diagram is not powerful enough to represent these extensions, a UML classifier construct is used in [Jaco05] to describe the dependencies between the use cases.

The approach is supported by a process throughout the whole software life-cycle. The requirements process is taken from the traditional use case approach [Jaco92]. The most significant change is the ability to handle non-functional requirements. They are identified and documented as infrastructure use cases along with functional use cases. An evaluation of the approach can be found in Table A.9.

Table A.9: Evaluation of AOSD/UC.

Criteria	**Description**
Traceability	It is not clear how source traceability is achieved in AOSD/UC. However, as use cases are usually specified in a free form, one can just textually add the source of the information. Collaborations provide traceability for the use cases. Furthermore, the «trace» relationship in a use case module provides a degree of forward and backwards traceability between the artifacts of different stages associated with use cases.
Composability	In contrast to the plain use case approach, composability has been extended. For example, the extension construct defines the order of the extension use case with respect to the extended use case.
Modifiability and Evolvability	The approach has the same characteristics as the plain use case approach.

Continued on the next page . . .

Criteria	Description
Scalability	The scalability issues of the conventional use approach also apply to AOSD/UC. However, the additional structures of use case slice and use case module help to organize the elements better and to describe the relationships between the various constructs.
Understandability	The characteristics of the conventional use case approach also apply to AOSD/UC.
Trade-off	A con ict of aspects can be handled in a limited way by the ordering of the extension use cases with the *before*, *after* and *around* keyword. However, there is no mechanism for con ict detection and resolution at the requirements level in AOSD/UC.
Mapping	As in the traditional use case approach, the mapping of the requirements artifacts is supported through collaborations. Furthermore, the approach outlines a rough process by which the aspect-oriented use case artifacts can be mapped to the software design.
Verification and Validation	The approach has the same validation and verification characteristics as the traditional use case approach.
Concern handling	The approach handles crosscutting and core concerns equally. However, it is debatable whether the approach is suited to all kinds of non-functional concerns. There may be non-functional requirements which do not manifest in code, and it is not clear how NFRs are identified. Furthermore, employing use cases for handling NFRs contradicts the definition of the term *use case*. A use case is usually initiated by an actor. However, there are NFRs which are not necessarily user-centric, i.e., initiated by an actor. For example, the maintainability NFR of a system cannot be properly expressed by a use case, as this would make no sense.

A.3.4 SMA

Scenario modeling with aspects (SMA) [Arau04, Whit04] is an approach for creating more consistent and complete use cases by modeling base and crosscutting scenarios separately from each other. A composition mechanism generates state machines from the elicited scenarios. These state machines help to validate the modeled use cases by simulation.

The approach is supported by a process which starts with the identification of the use cases. Then the coarse-grained use cases are refined. Non-functional requirements can be identified with the help of templates and usually result in aspectual scenarios. Base scenarios are modeled as simple UML sequence diagrams. Sequence diagrams can be transformed to finite state machines (FSM) by the semi-manual algorithm presented in [Whit00].

Aspectual scenarios are modeled as interaction pattern specifications (IPS) which are UML sequence diagrams containing *roles* instead of concrete message and lifeline identifiers. A set of IPS can be transformed to a state machine pattern specification (SMPS) which represents a state machine containing generic roles instead of messages and states.

For the composition of the base and crosscutting behavior the FSM and SMPS have to be merged. However, before this can be done, the roles of the SMPS must be instantiated with concrete values which

are provided by a state machine binding specification. The concrete values in the binding specification are state and message names of the FSM. After the instantiation of the values, the FSM and SMPS are merged, resulting in a unified state machine, which can be used, for example, to check or simulate the model. The evaluation of SMA is given in Table A.10

Table A.10: Evaluation of SMA.

Criteria	Description
Traceability	The approach discusses neither how the source of a requirement nor how the resulting artifacts at later stages of the software process are tracked. There is also no discussion of tracking the sources of a change.
Composability	The approach provides the composition of behavior from the modeled scenarios. However, the composition process (i.e., the mapping of the roles to concrete identifiers), as well as the synthesis of statecharts from the scenarios need manual intervention.
Modifiability and Evolvability	The approach is based on scenarios for defining use cases. Changes to requirements may cause changes in one or more scenarios. Even though these changes are local, they may cause the additional effort of checking the consistency between the different scenarios and the corresponding bindings.
Scalability	The state machine generation algorithm scales well. Nevertheless, scalability in general is compromised by the need to provide individual binding specifications for each composition.
Understandability	Intelligibility may be hampered by the numerous scenarios that are needed to specify FSMs and SMPSs. Thus, it is easy to lose track of the many artifacts.
Trade-off	There is no support for detecting and handling trade-offs between different concerns.
Mapping	There is no mechanism for mapping artifacts to later stages.
Verification and Validation	The validation of specifications is well supported, as the approach aims at generating executable statecharts which can be used for simulating the requirements. Apart from the simulation, the approach also allows the validation of the message sequence charts and IPS manually together with the stakeholder.
Concern handling	The approach is well-suited for handling crosscutting as well as core concerns. However, it neither discusses in detail how the functional crosscutting scenarios are found nor clearly addresses how to handle non-functional requirements by scenarios. Furthermore, the approach cannot cope with non-functional requirements that do not end up in functionality.

A.3.5 AUCDA

Aspectual use case driven approach (AUCDA) [Arau03] is another use-case-based aspect-oriented approach. It is similar to AOSD/UC [Jaco05] and aims at identifying and describing crosscutting non-

functional (called quality attributes) and functional concerns at the requirements stage. However, in contrast to AOSD/UC, non-functional requirements are not handled by infrastructure use cases. A template, proposed in [More02], is introduced, which facilitates the identification of non-functional requirements.

Crosscutting use cases resulting from non-functional requirements are split into three different categories: overlapping, overriding and wrapping. An *overlapping* use case crosscuts another use case either *before* or *after* a given execution point. *Overriding* means that a non-functional use case replaces functionality in the crosscut use case. Finally, a *wrapping* non-functional use case encloses the functionality of the crosscut use case.

The approach introduces new relationships between use cases. Apart from extension, inclusion, and generalization, use cases are also allowed to be in a collaboration, damage, or constrain relationship. A *collaboration* relationship between two use cases means that one use case contributes positively to the other, *damage* means a negative correlation, and *constrain* indicates a restriction. Furthermore, the approach introduces a *use case pattern speci cation* mechanism which allows a use case to be represented in a abstract way as a template that can be instantiated later for a concrete situation.

The AUCDA process starts with the identification of actors and high-level use cases. The high-level use cases are subsequently refined, then redundant behavior is identified, removed and described by separate use cases. The separated use cases are then connected to the other use cases by include or extend relationships. After these steps the non-functional requirements are identified with the help of quality attribute templates. The non-functional requirements found are mapped to use cases in the use case diagram and connected with the appropriate relationship to the use cases they in uence. Finally, the candidate aspects are identified, i.e., the use cases that probably map to an aspect implemented at the coding phase. Candidate aspects have more than one include, extend or constrain relationship to other use cases. The approach is evaluated in Table A.11.

Table A.11: Evaluation of the AUCDA approach.

Criteria	Description
Traceability	Non-functional requirements can be traced through the information captured when identifying the non-functional requirements with the help of the proposed template. However, the approach does not mention how the sources of the functional requirements are recorded. As with the original use case approach this can be done by simply noting the source in the corresponding functional use case. The traceability to artifacts in later stages is not explicitly mentioned but can be supported by UML collaborations for functional use cases or NFRs resulting in functionality. However, there is no discussion about the traceability of NFRs which do not end as functionality in the final system.
Composability	The approach extends the decomposition facilities of conventional use case approaches by introducing several new relationships, such as *collaboration* and *damage*. However, the approach does not discuss a mechanism for an automatic composition. Thus, the composition has to be done manually if needed.
Modifiability and Evolvability	As with other use case approaches, changes to artifacts are rather local to use cases. However, major changes to requirements may require changes to other use cases too. As with the other use-case-based approaches, there is no possibility for an automatic checking of the use case consistency.

Continued on the next page . . .

Criteria	Description
Scalability	The use case diagram is the main artifact. However, it becomes unreadable rather quickly, when the number of use cases increases.
Understandability	The use case diagram can become unreadable when many use cases are found.
Trade-off	There is no trade-off analysis support. However, the relationships between the use case diagrams, such as collaboration, damage and constrain provide information about con icts which can be used when negotiating the requirements with stakeholders.
Mapping	Functional use cases and NFRs which can be operationalized to functional use cases can be mapped to UML collaborations.
Verification and Validation	The approach is well suited for manual validation techniques.
Concern handling	Functional as well as non-functional concerns can be handled equally by the approach. However, the same problems as with AOSD/UC also apply to this approach.

A.3.6 Cosmos

The *Cosmos* approach [Stan02, Sutt03, Sutt04] is based on the principle of the multi-dimensional separation of concerns [Tarr99], where the concern space consists of a set of peer concerns that may overlap. Cosmos proposes an artifact-independent way of modeling concerns. It provides a general concern-space modeling schema for classifying concerns, their relationships, etc. After a concern has been identified, it is assigned to one or more *concern categories*. Furthermore, the relationships to other concerns are identified and classified [Sutt04]. A concern is either logical or physical. A *logical* concern is of a conceptual nature whereas a *physical* concern deals with real world artifacts.

There is no clear-cut process in cosmos. However, there are several simple guideline rules which help to create a good categorization of the concerns discovered, such as for components-off-the-shelf development, or development from scratch. The basic approach to finding concerns is to carefully analyze the document and artifacts of interest. After a list of concerns has been found, they have to be assigned to Cosmos categories.

Table A.12: Evaluation of the Cosmos approach.

Criteria	Description
Traceability	The Cosmos approach discusses neither how the sources of a concern nor how the sources of changes are recorded. The traceability to artifacts at a later stage is not discussed.
Composability	The approach proposes different types of relationships for the decomposition of concerns. A composition has to be done manually.

Continued on the next page . . .

Criteria	**Description**
Modifiability and Evolvability	Cosmos categorizes concerns. Usually changes are made in the artifacts that are assigned to the concern. Thus changes generally have a low impact on the concern structure.
Scalability	The concern schema of Cosmos is quite simple and models the concerns on a meta-level. Therefore, the approach is very scalable. However, concerns may have a lot of dependencies resulting in greater efforts being required.
Understandability	The approach is rather abstract. Therefore, the communication with stakeholders may be difficult.
Trade-off	The approach does not propose any trade-off analysis, as it does not deal with the elicitation and negotiation of requirements. However, constraints allow the capture of any trade-off decisions taken during the analysis.
Modifiability	Cosmos does not consider the mapping of concerns to artifacts at a later stage in the software process. Nevertheless, it allows the mapping between physical concerns and artifacts to be recorded.
Verification and Validation	There is no support for dynamic validation/verification. However, the Cosmos model may assist the static validation of the physical requirements artifacts.
Concern handling	The approach treats all concerns as peers, no matter if they have a functional or non-functional origin.

A.3.7 CORE

Concern-oriented requirements engineering (CORE) [More05a, More05b] extends the AORE with Arcade approach [Rash03] by the concepts of the multidimensional separation of concerns [Tarr99]. CORE decomposes requirements using different concerns as decomposition criteria. It therefore introduces, similarly to the Cosmos approach [Stan02], a meta-concern space which models the concerns of a software system and their relationships. An instantiation of the corresponding concerns allows the assignment of the requirements belonging to the concern. A set of composition rules describes the impact of a concern on the requirements of other concerns. Similarly to AORE, CORE uses XML for the representation of its artifacts, i.e., the meta-concern structure, the concern artifacts, and the composition descriptions.

The CORE process starts with the identification of the concerns of the system to be developed. The process can be carried out with any requirements engineering approach, such as a viewpoint- or use-case-based approach. After the concerns have been identified, the coarse-grained relationships between them are identified. Subsequently, the relationships are described in more detail by composition rules.

A trade-off analysis results the identification of the conflicts between different concerns impacting the same requirements. This is done by folding the concern relationships. In [More05a], folding employs a *compositional intersection operation*. Conflicting concerns are discussed with the stakeholders of the project. Priority weights can be used to reason about different impact situations on the same requirements, which can help in reaching a decision for resolving the conflicts. Finally, the mapping of the concerns to artifacts of the following stage in the software process is done by the same means as used in the AORE approach. The characteristics of CORE are evaluated in Table A.13.

Table A.13: Evaluation of the CORE approach.

Criteria	Description
Traceability	Even though CORE is based on AORE, it no longer uses the viewpoint templates. Therefore the sources of the concerns are not recorded. The same applies to the sources of changes in the requirements. In [More05a, More05b], there is no mention of how to trace the source of a concern or a requirement. However, this can be achieved by extending the XML schema with the needed elements. The traceability to artifacts in later stages of the software process is done similarly to AORE.
Composability	The approach has the same composability capabilities as AORE. However, the composition is more exible, since all concerns are uniquely represented in a meta-concern space.
Modifiability and Evolvability	The approach has a high separation of concerns, therefore changes to the requirements have low impact.
Scalability	As with AORE, the XML artifacts used in the CORE approach are very scalable. However, the composition tables can grow very large, which can affect the readability.
Understandability	As with the AORE approach, the XML representation is not necessarily easy to understand, which may cause problems, especially when used for communication with non-expert stakeholders.
Trade-off	The approach provides a well suited trade-off analysis, which trade-offs to be detected, analyzed and resolved.
Mapping	CORE provides the same support for mapping artifacts to later stages as AORE.
Verification and Validation	Artifacts of the CORE approach have to be validated manually.
Concern handling	Functional as well as non-functional concerns are treated as peers. Overlapping concerns are documented separately.

A.3.8 AOREC

Aspect-oriented requirements engineering for component-based software systems (AOREC) [Grun99, Grun00] provides a classification schema for categorizing the systemic aspects of components. In AOREC, an aspect is a characteristic of a system which consists of components providing or requiring services.

The main artifacts for documenting aspects are diagrams. The diagrams describe the components, the related aspects and the corresponding relationships. Furthermore, there are textual descriptions that additionally describe the functional and non-functional requirements of the system. A so-called aggregated aspect describes a group of interrelated components.

The approach is supported by a process. First, a set of requirements is elicited and a set of candidate components is identified. Then, the requirements for the components are elaborated. In the subsequent steps, the aspects of each component are identified and refined and the requirements are assigned to aspects. In the following, the aspect can be used to reason about the associated components, their compo-

sition and configuration. In a next step, the refined aspects are analyzed in order to see if they form an aggregate aspect. Aggregate aspects form new components. The creation of new components may cause further iterations through the process. After all requirements have been allocated, the components are verified against the system requirements. If all are met adequately, the design phase can start. The evaluation of AOREC can be found in Table A.14.

Table A.14: Evaluation of AOREC.

Criteria	Description
Traceability	AOREC does not discuss tracking the source of requirements or source of changes. The aspects found at the requirements level are directly propagated to the design level, which allows them to be traced.
Composability	The requirements of a component are composed from aspects. In turn, components can provide requirements to aspects. A composed view of the artifacts has to be generated manually if needed.
Modifiability and Evolvability	Changes are usually local. However, there may be changes which cause major changes in the artifacts. In this case, the relationships between aspects and components can help to estimate the costs of a change of requirements.
Scalability	The approach allows the separation of requirements which are common to more than one component in an aspect. Therefore, AOREC helps to avoid the scattering of the same requirements at different locations, which facilitates scalability. However, the graphical model consisting of aspects, components and relationships does not scale. It easily becomes very complex.
Understandability	The graphical and textual representations are well suited for communication with all types of stakeholders. However, for larger systems, the lack of scalability may cause problems.
Trade-off	AOREC does not provide any analysis of, or decision support for trade-offs. Nevertheless, it is possible to locate the impacts of decision.
Mapping	The artifacts from the requirements phase are directly transferred to the design phase.
Verification and Validation	There is no explicit validation/verification support. However, a validation of the artifacts can be done manually.
Concern handling	AOREC artifacts can support non-functional as well as functional crosscutting concerns by aspects and components. However, it is not clear how functional crosscutting concerns are handled by the AOREC process. Furthermore, there is no support for the identification of crosscutting concerns. However, some example decomposition patterns are suggested for the initial breakdown of aspects.

A.3.9 Theme/Doc

Theme/Doc [Bani04c, Bani04a, Clar05] is the requirements part of the Theme approach. In contrast to other approaches, such as ARGM, the Theme approach has been designed from scratch as an aspect-oriented approach and originates in subjective programming [Harr93]. It aims at the identification and further handling of crosscutting concerns extracted from natural language requirements.

Theme/Doc analyzes the requirements statements of a piece of software, thus it is applied at a later stage in the requirements process, when the requirements have already been elicited. Theme/Doc has a focus on functional requirements, and so it is less suitable for detecting non-functional crosscutting concerns. The analysis relies on the use of a tool. The artifacts produced by the tool are based on a manually compiled list of *action words* and *entities*. Action words are derived from the verbs in the natural language requirements specification and are called *themes*. A theme can be seen as the "encapsulation of a concern" [Clar05]. *Entities* are nouns related to the action words. Entities can be seen as objects on which the actions rely.

The artifact generated from this input list is the *action view graph* [Bani04c], which shows the relationships between the themes and the requirements. The *clipped action view* is produced by separating the *crosscutting themes* from the base themes. They are then associated with a crosscut relationship. A theme is crosscutting if it belongs to a requirements statement which is already associated to another requirements statement.[11]

The Theme/Doc process starts with the identification of *action words* from a set of given requirements statements. Furthermore, the requirements, *the themes*, as well as the *entities* contained in the requirements are used as input to the further process. In the following step, the *action view* and the *clipped action view* is created. From the *clipped action view*, a *theme view* is created. It shows the requirements and the *entities* associated with the *themes*. The *theme view* is a basis for the design with Theme/UML [Bani04c, Clar05] (cf. Section 3.2.3). The characteristics of Theme/Doc are summarized in Table A.15.

Table A.15: Evaluation of the Theme/Doc.

Criteria	**Description**
Traceability	Theme/Doc does not consider the source traceability of the requirements, as it is based on the elicited and negotiated requirements at a later stage of the requirements process. Theme/Doc links to the design artifacts which are created by Theme/UML.
Composability	Theme/Doc provides an order for the composition of crosscutting themes in the *clipped action view*. However, a composition is not done until the design with Theme/UML where a composition semantics is defined.
Modifiability and Evolvability	Changes in the requirements are handled to a large extent by the tool. However, the changes in the requirements have to be analyzed and the corresponding *action words*, *entities*, and requirements statements in the input list of the tool have to be added or modified, respectively. The tool then regenerates all views.

Continued on the next page . . .

[11]Thus, when two themes share the same requirements, one of the themes is crosscutting.

Criteria	**Description**
Scalability	The graphical representation of Theme/Doc very quickly becomes unmanageable, and, therefore, it does not scale well. In [Bani04b] possible approaches to improve the scalability are explored.
Understandability	The understandability of the Theme/Doc artifacts is rather poor, as the resulting graph structure tends to be overloaded.
Trade-off	The approach neither explicitly supports the identification of conflicting themes nor how these trade-offs are to be resolved. Therefore, the user of the approach has to manage trade-offs manually, based on his experience and intuition.
Mapping	The requirements models can be mapped directly to Theme/UML design models.
Verification and Validation	There are no means for automatic validation, thus, a validation has to be done manually by walking through the *(clipped) action view* resulting from the analysis.
Concern handling	The approach allows all kinds of crosscutting concerns to be described. However, since the approach is based on natural language analysis of action words, it is suitable for finding functional crosscutting concerns rather than non-functional crosscutting concerns. This is due to the fact that non-functional requirements are often not described by action words.

Appendix B

EBNF of the Aspect-Oriented ADORA language

In the following, the grammar production rules for the ADORA language and its aspect-oriented extension are presented. An introduction to the ADORA language can be found in Sections 5 and 6. Since the release of the last version in [Seyb06a], the grammar presented has undergone some changes to support the aspect-oriented extensions presented in this work.

B.1 Extended Backus Naur Form

In the subsequent section, the Extended Backus Naur Form (EBNF) is used for describing the production rules of the ADORA languages. The EBNF rules are explained in Table B.1.

Table B.1: Informal Description of the EBNF grammar.

EBNF Element Example	Description
"**`partial`**"	A terminal symbol of the grammar is enclosed between two quotes and written in bold typewriter font. In the given example, the terminal symbol for the keyword **`partial`** is given.
ComponentDefinition	An element of the grammar starting with an upper case character denotes a non-terminal symbol, i.e., the name of a production rule. In the example, the name of the syntax rule for the definition of a component is given.
<INTEGER_LITERAL>	Denotes a set of terminal elements in the grammar given by a regular expression. In this example, the set of all positive and negative integer numbers is described (cf. Section B.2).

Continued on the next page ...

EBNF Element Example	Description
(. . .)	Denotes a grouping of the elements between the two parentheses. Any operation immediately following the closing parenthesis relates to the group of elements.
(. . .) ?	Anything between the parentheses may occur optionally.
(. . . | . . .)	Denotes an alternative, i.e., *either* the elements before *or* the elements after the vertical line are given.
(. . .) +	The elements grouped by the parentheses may occur *one* or *more* times.
(. . .) *	The elements between the parentheses may occur *zero* or *more* times.
"**`state`**" StateName UniqueModelElementIdentifier	An element following another one indicates a sequence of these elements. In the given example, the keyword *state* is followed by the non-terminal *StateName* which in turn is followed by the non-terminal *UniqueModelElementIdenti er*.

B.2 Regular Expressions in the Grammar

In the ADORA grammar, several regular expressions are given which describe a set of terminal elements. They are denoted with a name in uppercase letters enclosed by angle brackets. All terminals symbols in the grammar are given in the following Table B.2.

Table B.2: Regular Expressions in the ADORA grammar.

Token	Description
<EOF>	Denotes the (non-printable) end-of-file character.
<INTEGER_LITERAL>	Describes the set of negative or positive integer literals.
<IDENTIFIER>	Denotes the set of identifiers which are formed, as it is usually done in programming languages, by a sequence of underscores, upper-, lowercase letters and digits. The start character must be a letter or an underscore.
<APOSTROPHIZED_IDENTIFIER>	Describes the set of identifiers composed of any character. This type of identifier has to be enclosed by two apostrophes.
<BOOLEAN_LITERAL>	Denotes the set of the boolean literals *true* or *false*.

Continued on the next page . . .

Token	Description
<FLOATING_POINT_LITERAL>	Describes the set of positive or negative numeric oating point literals.
<DATE_LITERAL>	Denotes a date of the form *yyyy/mm/dd*, where *yyyy* is the year (any positive four-digit number), *mm* the number of the month (01 – 12) and *dd* the number of the day in the month (01 - 31).
<TIME_LITERAL>	Describes the set of time literals of the form *hh:mm:ss*, where *hh* is the number of the hour (00 – 23), *mm* the number of the minute (00 – 59) and *ss* the number of the second (00 – 59).
<INFORMAL_DESCRIPTION>	Describes an informal description which is introduced by /# and closed by #/. Between these delimiters, any character may be placed.

B.3 EBNF Grammar of the ADORA Language

The ADORA grammar comprises of a set of production rules, as shown in the following Table B.3. The last rule in this list defines the so-called informal description, which can occur at any place of a textual ADORA model description (cf. Chapter 6.1.2). This fact is not expressed explicitly as this would bloat the grammar.

Table B.3: EBNF grammar of the aspect-oriented ADORA language.

Left Hand Side (Rule Name)	Right Hand Side (Production)
SpecificationDefinition ::=	(“**partial**”)? “**specification**” (SpecialIdentifier)? Model (Representation)? “**end**” “**specification**” (SpecialIdentifier)? <EOF>
Model ::=	“**model**” ((ComponentDefinition | EnvironmentObjectDefinition | AspectDefinition))* TypeDefinitions PropertyDefinitions StereotypeDefinitions “**end**” “**model**”
ComponentDefinition ::=	(“**partial**”)? (“**external**”)? (“**start**”)? “**component**” ComponentName UniqueModelElementIdentifier (Cardinality)? (“**is**” InheritedType)? ComponentParts FunctionalSpecification (ComponentConnections)? “**end**” “**component**” ComponentName
ComponentName ::=	SpecialIdentifier
UniqueModelElementIdentifier ::=	SpecialIdentifier

Continued on the next page . . .

Left Hand Side (Rule Name)	Right Hand Side (Production)
EnvironmentObjectDefinition ::=	(“**partial**”)? “**environment**” “**object**” EnvironmentObjectName UniqueModelElementIdentifier (Cardinality)? (EnvironmentObjectConnections)? “**end**” “**environment**” “**object**” EnvironmentObjectName
EnvironmentObjectName ::=	SpecialIdentifier
InheritedType ::=	“**type**” QualifiedIdentifier
ComponentParts ::=	(“**consists**” “**of**” (ComponentDefinition | StateDefinition | ScenarioDefinition | AspectDefinition)+ “**end**” “**consists**” “**of**”)?
StateDefinition ::=	(“**partial**”)? (“**start**”)? “**state**” StateName UniqueModelElementIdentifier StateParts (StateConnections)? “**end**” “**state**” StateName
StateName ::=	SpecialIdentifier
StateParts ::=	(“**consists**” “**of**” (StateDefinition)+ “**end**” “**consists**” “**of**”)?
AspectDefinition ::=	(“**partial**”)? “**aspect**” AspectName UniqueModelElementIdentifier AspectParts FunctionalSpecification AspectConnections “**end**” “**aspect**” AspectName
AspectParts ::=	(“**consists**” “**of**” (ComponentDefinition | StateDefinition | ScenarioDefinition | AspectDefinition | ExitPointDefinition)+ “**end**” “**consists**” “**of**”)?
ExitPointDefinition ::=	“**exit**” StateName UniqueModelElementIdentifier “**end**” “**exit**” StateName
AspectName ::=	SpecialIdentifier
AspectConnections ::=	(“**connections**” (AssociationDefinition | AssociationRoleDefinition | JoinRelationshipDefinition)* “**end**” “**connections**”)?
JoinRelationshipDefinition ::=	(“**partial**”)? “**joinrelationship**” UniqueModelElementIdentifier “**from**” ElementReference “**to**” ElementReference (“**before**” | “**instead**” | “**after**”)? (Priority)? (ContextMap)?
Priority ::=	<INTEGER_LITERAL>
ContextMap ::=	“**context**” “**map**” ContextMapping (ContextMapping)* “**end**” “**context**” “**map**”
ContextMapping ::=	Identifier “**:**” Expression “**;**”

Continued on the next page . . .

<table>
<tr><th>Left Hand Side (Rule Name)</th><th>Right Hand Side (Production)</th></tr>
<tr><td>ScenarioDefinition ::=</td><td>(“partial”)? ScenarioType “scenario” ScenarioName UniqueModelElementIdentifier (“on” GuardPart)? (“iteration” Expression)? (ScenarioConnections)? (TransformationElements)? “end” “scenario” ScenarioName</td></tr>
<tr><td>ScenarioName ::=</td><td>SpecialIdentifier</td></tr>
<tr><td>ScenarioType ::=</td><td>(“alternative” | “sequence” (<INTEGER_LITERAL>)? | “parallel” | “root” (Cardinality)?)</td></tr>
<tr><td>StateConnections ::=</td><td>“connections” (TransitionDefinition)* “end” “connections”</td></tr>
<tr><td>TransitionDefinition ::=</td><td>(“partial”)? “transition” UniqueModelElementIdentifier “to” ElementReference (SimpleTransition | DecisionTableTransition)?</td></tr>
<tr><td>SimpleTransition ::=</td><td>ConditionPart “|” ActionPart</td></tr>
<tr><td>DecisionTableTransition ::=</td><td>“table” ConditionBlock ActionBlock “end” “table”</td></tr>
<tr><td>ConditionBlock ::=</td><td>IdentifiableConditionPart (“;” IdentifiableConditionPart)*</td></tr>
<tr><td>IdentifiableConditionPart ::=</td><td>ConditionID “:” ConditionPart</td></tr>
<tr><td>ConditionPart ::=</td><td>(TimeCondition)? (GuardPart)? (MessageReceive)?</td></tr>
<tr><td>GuardPart ::=</td><td>“[” Expression “]”</td></tr>
<tr><td>ActionBlock ::=</td><td>“do” ActionPart (ActionConditions (“or” ActionConditions)*)?</td></tr>
<tr><td>ActionConditions ::=</td><td>“on” NegatableConditionId (“,” NegatableConditionId)*</td></tr>
<tr><td>NegatableConditionId ::=</td><td>(“not”)? ConditionID</td></tr>
<tr><td>ConditionID ::=</td><td><INTEGER_LITERAL></td></tr>
<tr><td>ComponentConnections ::=</td><td>“connections” (AssociationDefinition | AssociationRoleDefinition | TransitionDefinition)* “end” “connections”</td></tr>
<tr><td>EnvironmentObjectConnections ::=</td><td>“connections” (AssociationDefinition | AssociationRoleDefinition | JoinRelationshipDefinition)* “end” “connections”</td></tr>
<tr><td>AssociationRoleDefinition ::=</td><td>Role “of” “association” ElementReference</td></tr>
<tr><td>Role ::=</td><td>“role” RoleName (Cardinality)?</td></tr>
<tr><td>AssociationDefinition ::=</td><td>(“partial”)? “association” UniqueModelElementIdentifier “to” ElementReference Role</td></tr>
<tr><td>RoleName ::=</td><td>SpecialIdentifier</td></tr>
<tr><td>ScenarioConnections ::=</td><td>“connections” (ScenarioConnectionDefinition | AssociationDefinition | AssociationRoleDefinition)* “end” “connections”</td></tr>
</table>

Continued on the next page . . .

<table>
<tr><th>Left Hand Side (Rule Name)</th><th>Right Hand Side (Production)</th></tr>
<tr><td>ScenarioConnectionDefinition ::=</td><td>“scenarioconnection” UniqueModelElementIdentifier “to” ElementReference</td></tr>
<tr><td>TypeDefinitions ::=</td><td>(TypeDefinition)*</td></tr>
<tr><td>TypeDefinition ::=</td><td>(“partial”)? “type” TypeName (“inherits” InheritedType)? ComponentParts FunctionalSpecification “end” “type” TypeName</td></tr>
<tr><td>TypeName ::=</td><td>SpecialIdentifier</td></tr>
<tr><td>PropertyDefinitions ::=</td><td>(PropertyDefinition)*</td></tr>
<tr><td>PropertyDefinition ::=</td><td>“propertydef” SpecialIdentifier (“numbered”)? DataTypeReference (“constraints unique”)? “end” “propertydef” SpecialIdentifier</td></tr>
<tr><td>StereotypeDefinitions ::=</td><td>(StereotypeDefinition)*</td></tr>
<tr><td>StereotypeDefinition ::=</td><td>“stereotype” SpecialIdentifier “for” QualifiedIdentifier (“,” QualifiedIdentifier)* <STRING_LITERAL> LocalVariableDefinitions (“condition” Expression)? “end” “stereotype” SpecialIdentifier</td></tr>
<tr><td>FunctionalSpecification ::=</td><td>(“functional” “specification” (Provides)? (Requires)? (Invariants | DataTypeDeclarations | AttributeDefinitions | Property | OperationDefinition)* “end” “functional” “specification”)?</td></tr>
<tr><td>Provides ::=</td><td>“provides” Identifier (“,” Identifier)* “;”</td></tr>
<tr><td>Requires ::=</td><td>“requires” QualifiedIdentifier (“,” QualifiedIdentifier)* “;”</td></tr>
<tr><td>Invariants ::=</td><td>“inv” (Expression “;”)+</td></tr>
<tr><td>DataTypeDeclarations ::=</td><td>“data” “type” (DataTypeDeclaration)+</td></tr>
<tr><td>DataTypeDeclaration ::=</td><td>DataTypeName “:” DataTypeDefinition “;”</td></tr>
<tr><td>DataTypeDefinition ::=</td><td>(PrimitiveType | EnumerationTypeDefinition | DesignedTypeDefinition)</td></tr>
<tr><td>DataTypeName ::=</td><td>Identifier</td></tr>
<tr><td>DataTypeReference ::=</td><td>(PrimitiveType | Identifier)</td></tr>
<tr><td>AttributeDefinitions ::=</td><td>“attributes” (VariableDefinition “;”)+</td></tr>
<tr><td>VariableDefinition ::=</td><td>VariableNameList “:” DataTypeReference (“=” ExtendedTypeLiterals)?</td></tr>
<tr><td>VariableName ::=</td><td>Identifier</td></tr>
<tr><td>VariableNameList ::=</td><td>VariableName (“,” VariableName)*</td></tr>
<tr><td>Property ::=</td><td>“property” Identifier (<INTEGER_LITERAL>)? TypeLiteral “;”</td></tr>
</table>

Continued on the next page …

Left Hand Side (Rule Name)	Right Hand Side (Production)
OperationDefinition ::=	(AsyncOperationSignature | SyncOperationSignature) ("**is**" "**set**" "**operation**")? (LocalVariableDefinitions)? (PreConditions)? (PostConditions)? (Statements)? "**end**" "**operation**" OperationName
OperationName ::=	Identifier
AsyncOperationSignature ::=	"**operation**" OperationName "**(**" (Parameters)? "**)**"
SyncOperationSignature ::=	"**syncoperation**" OperationName "**(**" (SyncParameterList)? "**)**" ("**:**" DataTypeReference)?
LocalVariableDefinitions ::=	"**var**" (VariableDefinition "**;**")+
PreConditions ::=	"**pre**" (Expression "**;**")+
PostConditions ::=	"**post**" (Expression "**;**")+
Statements ::=	"**statements**" Statement ("**;**" Statement)*
Statement ::=	(GuardPart)? (Assignment | MessageSend | MetaFunction)
Assignment ::=	ExtendedQualifiedIdentifier "**=**" Expression
MetaFunction ::=	(StructureMetaFunction | QueryMetaFunction | ObjectSetMetaFunction)
StructureMetaFunction ::=	("**compositionOf**" | "**componentsOf**" | "**isActiveState**") "**(**" (ActualParamList)? "**)**"
QueryMetaFunction ::=	("**findObjects**" | "**systemtime**") "**(**" (Expression)? "**)**"
ObjectSetMetaFunction ::=	("**maxObjectNumber**" | "**minObjectNumber**" | "**actObjectNumber**" | "**create**" | "**dispose**") "**(**" (ActualParamList)? "**)**"
SyncParameterList ::=	(SyncParamListOut | SyncParamListIn) ("**;**" (SyncParamListOut | SyncParamListIn))*
SyncParamListIn ::=	("**in**")? Parameters
SyncParamListOut ::=	"**out**" Parameters
Parameters ::=	ParameterDefinition ("**,**" ParameterDefinition)*
ParameterDefinition ::=	VariableName "**:**" DataTypeReference
Expression ::=	(ConditionalOrExpression ("**or**" ConditionalOrExpression)* | QuantifiedExpression)
QuantifiedExpression ::=	("**for**" "**all**" | "**exists**") VariableName "**:**" DataTypeReference "**insetof**" ExtendedQualifiedIdentifier "**(**" Expression "**)**"
ConditionalOrExpression ::=	ConditionalAndExpression ("**and**" ConditionalAndExpression)*
ConditionalAndExpression ::=	ImplEqualExpression (("**->**" | "**<->**") ImplEqualExpression)*

Continued on the next page ...

<table>
<tr><th>Left Hand Side (Rule Name)</th><th>Right Hand Side (Production)</th></tr>
<tr><td>ImplEqualExpression ::=</td><td>EqualityExpression ((“==” | “!=”) EqualityExpression)*</td></tr>
<tr><td>EqualityExpression ::=</td><td>RelationalExpression ((“<” | “>” | “<=” | “>=”) RelationalExpression)*</td></tr>
<tr><td>RelationalExpression ::=</td><td>AdditiveExpression ((“+” | “–”) AdditiveExpression)*</td></tr>
<tr><td>AdditiveExpression ::=</td><td>MultiplicativeExpression ((“*” | “/”) MultiplicativeExpression)*</td></tr>
<tr><td>MultiplicativeExpression ::=</td><td>(“not” | “–”)* UnaryExpression</td></tr>
<tr><td>UnaryExpression ::=</td><td>(ActualParam | “(” Expression “)”)</td></tr>
<tr><td>PrimitiveType ::=</td><td>(“string” | “integer” | “time” | “float” | “boolean” | “id”)</td></tr>
<tr><td>EnumerationTypeDefinition ::=</td><td>“(” VariableName (“,” VariableName)* “)”</td></tr>
<tr><td>DesignedTypeDefinition ::=</td><td>(TypeConstraintDefinition | ListTypeDefinition | StructureTypeDefinition)</td></tr>
<tr><td>TypeConstraintDefinition ::=</td><td>“constrained” VariableName “:” PrimitiveType “(” Expression “)”</td></tr>
<tr><td>ListTypeDefinition ::=</td><td>“list” “of” DataTypeReference</td></tr>
<tr><td>StructureTypeDefinition ::=</td><td>“structure” “of” “(” VariableDefinition (“;” VariableDefinition)* “)”</td></tr>
<tr><td>TimeCondition ::=</td><td>(“steps” | “secs”) <INTEGER_LITERAL> “after” (“simulation start” | “state entered” | “object initialized”)</td></tr>
<tr><td>MessageReceivePart ::=</td><td>MessageReceive (“;” MessageReceive)*</td></tr>
<tr><td>MessageReceive ::=</td><td>“receive” (“low” | “high” | “default”)? (“queued”)? (ExtendedQualifiedIdentifier “=”)? EventName “(” (SyncParameterList)? “)” (“over” AssociationRoleNameList)? (“from” ComponentNameList)?</td></tr>
<tr><td>AssociationRoleNameList ::=</td><td>SpecialIdentifier (“,” SpecialIdentifier)*</td></tr>
<tr><td>ComponentNameList ::=</td><td>QualifiedIdentifier (“,” QualifiedIdentifier)*</td></tr>
<tr><td>ActionPart ::=</td><td>(EmbeddedOperation | OperationCall)?</td></tr>
<tr><td>EmbeddedOperation ::=</td><td>(“syncexec”)? (LocalVariableDefinitions)? (PreConditions)? (PostConditions)? Statement (“;” Statement)*</td></tr>
<tr><td>OperationCall ::=</td><td>“call” (OperationName (“(” (ActualParamList)? “)”)?)?</td></tr>
<tr><td>MessageSendPart ::=</td><td>MessageSend (“;” MessageSend)*</td></tr>
<tr><td>MessageSend ::=</td><td>“send” (“multicast”)? EventName “(” (ActualParamList)? “)” (“over” AssociationRoleNameList)? (“to” ComponentNameList)? ObjectSetAddress</td></tr>
<tr><td>ObjectSetAddress ::=</td><td>(Index)? (AnonymousList | QualifiedIdentifier)?</td></tr>
</table>

Continued on the next page ...

<table>
<tr><th>Left Hand Side (Rule Name)</th><th>Right Hand Side (Production)</th></tr>
<tr><td>TransformationElements ::=</td><td>(TransformInput | TransformOutput) (“;” (TransformInput | TransformOutput))*</td></tr>
<tr><td>TransformInput ::=</td><td>“transform” “input” EventName “(” (SyncParameterList)? “)” (“over” RoleName | “to” QualifiedIdentifier)?</td></tr>
<tr><td>TransformOutput ::=</td><td>“transform” “output” EventName “(” (SyncParameterList)? “)” (“over” RoleName | “from” QualifiedIdentifier)?</td></tr>
<tr><td>ActualParamList ::=</td><td>ActualParam (“,” ActualParam)*</td></tr>
<tr><td>ActualParam ::=</td><td>(TypeLiteral | ExtendedQualifiedIdentifier | MetaFunction)</td></tr>
<tr><td>Identifier ::=</td><td><IDENTIFIER></td></tr>
<tr><td>SpecialIdentifier ::=</td><td>(Identifier | <APOSTROPHIZED_IDENTIFIER>)</td></tr>
<tr><td>ElementReference ::=</td><td>SpecialIdentifier</td></tr>
<tr><td>QualifiedIdentifier ::=</td><td>SpecialIdentifier (“.” SpecialIdentifier)*</td></tr>
<tr><td>ExtendedQualifiedIdentifier ::=</td><td>QualifiedIdentifier (“@pre”)? (Index)?</td></tr>
<tr><td>EventName ::=</td><td>Identifier</td></tr>
<tr><td>TypeLiteral ::=</td><td>(<STRING_LITERAL> | <BOOLEAN_LITERAL> | <INTEGER_LITERAL> | <FLOATING_POINT_LITERAL> | TimeStamp)</td></tr>
<tr><td>TimeStamp ::=</td><td>(<DATE_LITERAL> (“,” <TIME_LITERAL>)? | <TIME_LITERAL>)</td></tr>
<tr><td>ExtendedTypeLiterals ::=</td><td>TypeLiteral | QualifiedIdentifier | AnonymousList</td></tr>
<tr><td>AnonymousList ::=</td><td>“{” ExtendedTypeLiterals (“,” ExtendedTypeLiterals)* “}”</td></tr>
<tr><td>Index ::=</td><td>“[” IndexDimension (“,” IndexDimension)* “]”</td></tr>
<tr><td>IndexDimension ::=</td><td><INTEGER_LITERAL> | ExtendedQualifiedIdentifier</td></tr>
<tr><td>Cardinality ::=</td><td>“(” (<INTEGER_LITERAL> | <IDENTIFIER>) “,” (<INTEGER_LITERAL> | <IDENTIFIER>) “)”</td></tr>
<tr><td>InformalDescription ::=</td><td><INFORMAL_DESCRIPTION></td></tr>
</table>

B.4 Production Rules of Identifiers and References

There are four different basic grammar rules (cf. Section B.3) on which all identifiers and references are based. The *Identifier* rule specifies identifiers as they are known from common programming languages. It consists of digits, upper and lower case letters and underscores, where the first character is always a letter or an underscore. Another type of identifier is a *SpecialIdentifier* which is either an *Identifier* or any character string, including the empty string, enclosed in apostrophes.

A *QualifiedIdentifier* is a sequence of *SpecialIdentifier*s connected by dots. *QualifiedIdentifier*s are used to reference elements which have a *name* [Seyb06a, p. 14]. The segments in a reference consist of two parts: a path and the name of the referred element. The path is composed of the names of the ancestors

of the model decomposition hierarchy. For example, if a component with the name *C* is referred and it is embedded in component *B* which in turn lies in *A*, then the qualified name is *A.B.C*, where *A.B* is the path.

Furthermore, if the first segment of a *Quali edIdenti er* is `this`, then the following path is relative to the component where the *Quali edIdenti er* occurs. For instance with respect to the example `A.B.C`, a qualified identifier `this.C` occurring in the component `B` is equal to the qualified name `A.B.C`. If the first segment of a qualified identifier is **`parent`**, then the following path relates to the parent of the component where the *Quali edIdenti er* occurs. With respect to the example `A.B.C`, `parent.X` occurring in the component `C` is equal to the qualified identifier `A.B.X`.

Another identifier is the *ExtendedQuali edIdenti er* which is a *Quali edIdenti er* that might optionally have an index for referring to the elements of a field and a `pre` directive which is employed in the description of pre-, postconditions and invariants (cf. Section 5.2.6). Furthermore, the *ExtendedQuali-edIdenti er* may be used in *Expression*s and the actual parameters of operations.

The rule *UniqueModelElementIdenti er* defines the generated identifiers in the ADORA grammar of Appendix B, whereas *ElementReference* gives a definition of the corresponding references. Grammar rules ending with the suffix *Name* define what a name looks like, whereas the rule *Quali edIdenti er* or *ExtendedQuali edIdenti er* describe the corresponding references. A list of all the production rules that define generated identifiers or names and the corresponding references are given in Table B.4.

The four basic identifier rules are used to define specific identifier rules which are referred by the rules of other elements of the language. In Table B.4 a list of the rules which define generated identifiers and names (cf. Section 6.1.5) is given. The first column shows the name of the particular rule. The second colum indicates the type of the identifier, where *G* means generated, *N* means Name, *R* reference and *I* identifier. The third column describes in detail, where the rule is used.

Table B.4: Grammar rules of identifiers and references

Grammar Rule Name	Type	**Description**
ComponentName	N I	Describes the name of a component.
UniqueModelElementIdentifier	G I	Is used for identifying main elements in a textual ADORA model (*Speci cationDe nition*, *ComponentDe nition*, *EnvironmentOb-jectDe nition*, *StateDe nition*, *ScenarioDe nition*, *Transition-De nition*, *AssociationRoleDe nition*, *ScenarioConnectionDe -nition*).
EnvironmentObjectName	N I	Describes the name of an environment object.
TypeQualifiedName	N R	Describes which type is used for an object. A type may consist of a qualified name. The qualified name may denote the inheritance path to disambiguate which type is used.
StateName	N I	Describes the name of a state.
ScenarioName	N I	Describes the name of a scenario node.
RoleName	N I	Describes the role name of an association role.

Continued on the next page ...

Grammar Rule Name	Type	**Description**
TypeName	N I	Describes the name of a Type.
DataTypeName	N I	Describes the name of a data type.
VariableName	N I	Describe the name of a variable. This production rule is used for variable declarations.
OperationName	N I	Describes the name of an operation.
ElementReference	G R	Gives a reference on an arbitrary modeling element. *ElementReference* refers to the element which is identified by a *UniqueModelElementIdenti er*.
EventName	N I	Defines the names of events. This production rule is used in the *MessageSend*, *MessageReceive*, *TransformInput* and *TransformOutput* elements.
Identifier	N R	Describes the reference of a name.
SpecialIdentifier	N R	Describes the reference of a name.

Appendix C

Mapping of Graphical ADORA Model Elements

In the ADORA language, each main language element is mapped to a graphical element [Xia04]. The following table presents a version of the graphical mapping which was adapted in [Seyb06a] to fit a revised grammar version of ADORA. The current mapping is extended to the newest version of the ADORA grammar in Appendix B.

In the first column of the following table, the language element which is mapped is named. The second column indicates the grammar rule. The third column gives an instance of the grammar rule and, finally, the fourth column shows the corresponding graphical representation.

Table C.1: Graphical Mapping for the aspect-oriented ADORA Language

Name	Grammar Rule	Textual Instance	Graphical Mapping
Abstract Object	Component-Definition	/# Informal Description #/ component *Name Identi er (1, 1)* ... end component *Name*	Name
Object Set	Component-Definition	/# Informal Description #/ component *Name Identi er (n,m)* ... end component *Name*	Name (n,m)
Abstract Start Object	Component-Definition	/# Informal Description #/ start component *Name Identi er (1,1)* ... end component *Name*	Name

Continued on the next page ...

Name	Grammar Rule	Textual Instance	Graphical Mapping
Partial Abstract Object	Component-Definition	/# Informal Description #/ partial component *Name Identi er (1,1)* . . . end component *Name*	*Name...*
State	State-Definition	/# Informal Description #/ state *Name Identi er* . . . end state *Name*	*Name*
Start State	State-Definition	/# Informal Description #/ start state *Name Identi er* . . . end state *Name*	*Name*
Root Scenario	Scenario-Definition	/# Informal Description #/ root *(n,m)* scenario *Name Identi er* . . . end scenario *Name*	*Name* *(n,m)*
Parallel Scenario	Scenario-Definition	/# Informal Description #/ parallel scenario *Name Identi er* on [Entry Condition] iteration *IterationCondition* . . . end scenario *Name*	*EntryCondition* || *Name* * *IterationCondition*
Sequential Scenario	Scenario-Definition	/# Informal Description #/ sequence *no* scenario *Name Identi er* . . . end scenario *Identi er*	*no.* *Name*
Alternative Scenario	Scenario-Definition	/# Informal Description #/ alternative no scenario *Name Identi er* on [*Entry Condition*] . . . end scenario *Name Identi er*	**O** *Name*
Environment Object (Actor)	Environment-Object-Definition	/# Informal Description #/ environment object **Name** *Identi er (m,n)* . . . environment object *Name*	*Name*
Environment Object Set	Environment-Object-Definition	/# Informal Description #/ environment object *Name Identi er (m,n)* . . . end environment object *Name*	*Name* *(n,m)*

Continued on the next page . . .

Name	Grammar Rule	Textual Instance	Graphical Mapping
Aspect	Aspect-Definition	/# Informal Description #/ aspect *Name Identi er* end aspect *Name*	*Name*
Association	Association-Definition	/# Informal Description #/ association *Identi er* to *target_identi er* role *Role (m,n)*	Name *Role (m,n)* target_ identifier
Transition	Transition-Definition	/# Informal Description #/ transition *Identi er* to *target_identi er* *Condition* | *Action*	Name *Condition* | *Action* target_ identifier
Join Relationship	Join Relationship-Definition	/# Informal Description #/ joinrelationship *Identi er* from *orig_identi er* to *target_identi er* before	A X *before* target_ identifier B

Appendix D

Textual ADORA Example Model

D.1 Example of a Conventional ADORA Text Model

The following Listing D.1 contains the textual representation of the model given by Fig. 5.2. In the following, it is assumed that the components and scenarios containing an ellipsis are partial.

Listing D.1: An example of a textual ADORA model which shows a partial version of the library system.

```
 1 speci cation LibrarySystem
 2   model
 3     component LibrarySystem ’1.1’ (1,1) is type Root
 4       consists of
 5         partial component User ’1.1.1’ (0,m)
 6           connections
 7             role ReadUserInfo (1,1) of association ’1.1.4.a.2’
 8             association ’1.1.1.a.1’ to ’1.1.6’ role GetUserId (1,1)
 9             role borrows (0,p) of association ’1.1.2.a.2’
10           end connections
11         end component User
12
13         partial component Book ’1.1.2’ (0,n)
14           connections
15             role ReadCatalogBA (1,1) of association ’1.1.5.a.1’
16             association ’1.1.2.a.1’ to ’1.1.3’ role ReadCatalogBM (1,1)
17             association ’1.1.2.a.2’ to ’1.1.1’ role borrowedBy (0,1)
18           end connections
19         end component Book
20
22         partial component BorrowManager ’1.1.3’ (1,1)
23           consists of
24             root scenario BorrowBooks ’1.1.3.1’
25               connections
26                 scenarioconnection ’1.1.3.1.s.1’ to ’1.1.3.2’
```

```
27                  scenarioconnection '1.1.3.1.s.2' to '1.1.3.3'
28                  scenarioconnection '1.1.3.1.s.3' to '1.1.3.4'
29                  association '1.2.a.1' to '1.2' role '' (1,1)
30               end connections
31            end scenario BorrowBooks
32
33            partial sequence 1 scenario SelectBooks '1.1.3.2'
34            end scenario SelectBooks
35
36            partial sequence 2 scenario Authenticate '1.1.3.3'
37            end scenario Authenticate
38
39            partial sequence 3 scenario RegisterBorrowing '1.1.3.4'
40            end scenario RegisterBorrowing
41         end consists of
42         connections
43            role BorrowBooks (1,1) of association '1.1.2.a.1'
44            association '1.1.3.a.1' to '1.1.6' role AuthenticateBorrowing (1,1)
45         end connections
46      end component BorrowManager
47
48      partial component UserAdministration '1.1.4' (1,1)
49         connections
50            role GetFeedback (1,1) of association '1.4.a.1'
51            association '1.1.4.a.1' to '1.1.6' role AuthenticateUserAdm (1,1)
52            association '1.1.4.a.2' to '1.1.1' role ManageUserInfo (1,1)
53         end connections
54      end component UserAdministration
55
56      partial component BookAdministration '1.1.5' (1,1)
57         connections
58            role GetFeedback (1,1) of association '1.3.a.1'
59            association '1.1.5.a.1' to '1.1.2' role ManageCatalog (1,1)
60            association '1.1.5.a.2' to '1.1.6' role AuthenticateBookAdm (1,1)
61         end connections
62      end component BookAdministration
63
64      partial component Authorization '1.1.6' (1,1)
65         consists of
66            start state Wait '1.1.6.1'
67               connections
68                  transition '1.1.6.1.t.1' to '1.1.6.2'
69                     [not authenticated]
70                     receive queued msg=authenticate(user : string, pw : string) over
71                        AuthenticateBorrowing, AuthenticateBookAdm,
```

```
 72                      AuthenticateUserAdm | send getUserInfo(user, pw) over SendUserInfo
 73
 74                  transition '1.1.6.1.t.2' to '1.1.6.1'
 75                    [authenticated] receive queued msg=authorized(user : string, pw : string) over
 76                      AuthenticateBorrowing, AuthenticateBookAdm,
 77                      AuthenticateUserAdm | call
 78                end connections
 79              end state Wait
 80              state UserInfo '1.1.6.2'
 81                connections
 82                  transition '1.1.6.2.t.1' to '1.1.6.3'
 83                    receive userInfo(ui : userinfo) over GetUserId |
 84                    syncexec send composeUserCredentials() to UserCredentials
 85                end connections
 86              end state UserInfo
 87              state Authorizing '1.1.6.3'
 88                connections
 89                  transition '1.1.6.3.t.1' to '1.1.6.1'
 90                    receive userCredentials() from UserCredentials |
 91                    send authorized() to msg.msgAnswerTarget
 92                end connections
 93              end state Authorizing
 94              partial component '1.1.6.4' UserCredentials
 95              end component UserCredentials
 96            end consists of
 97            functional speci  cation
 98              requires LibrarySystem.User.userinfo;
 99              attributes msg : message;
100              operation authenticate(user : string, pw : string)
101                 statements
102                   send authorized() to msg.msgAnswerTarget
103              end operation authenticate
104            end functional speci  cation
105            connections
106              role SendUserInfo (1,1) of association '1.1.1.a.1'
107              role AuthorizeUserAdm (1,1) of association '1.1.4.a.1'
108              role AuthorizeBookAdm (1,1) of association '1.1.5.a.2'
109              role AuthorizeBorrowing (1,1) of association '1.1.3.a.1'
110            end connections
111          end component Authorization
112        end consists of
113      end component LibrarySystem
114
115      environment object LibraryUser '1.2'
116        connections
```

```
117         association '1.2.a.1' to '1.1.3.1' role '' (1,1)
118       end connections
119     end environment object LibraryUser
120
121     environment object BookAdministrator '1.3'
122       connections
123         association '1.3.a.1' to '1.1.1.5' role ManageBooks (1,1)
124       end connections
125     end environment object BookAdministrator
126
127     environment object UserAdministrator '1.4'
128       connections
129         partial association '1.4.a.1' to '1.1.4' role ManageUsers (1,1)
130       end connections
131     end environment object UserAdministrator
132   end model
133 end speci cation LibrarySystem
```

D.2 Example of Textual Description of an Aspect Module

Listing D.2 contains the textual model description of the aspect *Authentication* shown in Fig. 7.2. It is assumed that the elements containing an ellipsis in their name are partial.

Listing D.2: The *Authentication* aspect from the model of Fig. 7.2 as textual specification.

```
 1 aspect Authentication '1.1.7'
 2   consists of
 3     state WaitForUserName '1.1.7.1'
 4       connections
 5         transition '1.1.7.t.1' to '1.1.7.2'
 6           receive userNameEntered(name : string) | call
 7       end connections
 8     end state WaitForUserName
 9     state Authenticate '1.1.7.2'
10       connections
11         transition '1.1.7.t.2' to '1.1.7.3'
12           receive authorized() over Authorized |
13         transition '1.1.7.t.3' to '1.1.7.2'
14           receive userPWEntered(pw : string) |
15           send authenticate(user, pw) over Authenticate;
16           send log(user, pw) to AuthenticationLog
17       end connections
18     end state Authenticate
19     start state WaitForHintRequest '1.1.7.3'
20       connections
```

```
21         transition '1.1.7.3.t.1' to '1.1.7.4'
22           receive requestHint(name : String) | send getHint(name) over Authenticate
23       end connections
24     end state WaitForHintRequest
25     state WaitForHintQuestion '1.1.7.4'
26       connections
27         transition '1.1.7.4.t.1' to '1.1.7.5'
28           receive hint(hint : string) over Authorize | send showHint(hint)
29       end connections
30     end state WaitForHintQuestion
31     state WaitForAnswer '1.1.7.5'
32       connections
33         transition '1.1.7.5.t.1' to '1.1.7.6'
34           receive enterAnswer(a:string) | send verifyAnswer(a) over Authenticate
35       end connections
36     end state WaitForAnswer
37     state WaitForHintPassword '1.1.7.6'
38       connections
39         transition '1.1.7.6.t.1' to '1.1.7.3'
40           receive result(pwMsg : string) over Authorize | send showPwMsg(pwMsg)
41       end connections
42     end state WaitForHintPassword
42     exit Authorized '1.1.7.7'
43     end exit Authorized
44     root scenario Authenticate '1.1.7.8'
45       connections
46         scenarioconnection '1.1.7.8.s.1' to '1.1.7.9'
47         scenarioconnection '1.1.7.8.s.2' to '1.1.7.10'
48       end connections
49     end scenario Authenticate
50     sequence 1 scenario EnterUsername '1.1.7.9'
51       transform input userNameEntered(name : string)
52     end scenario EnterUsername
53     sequence 2 scenario EnterPassword '1.1.7.10'
54       transform input userPWEntered(pw : string)
55     end scenario EnterPassword
56     partial component AuthenticationLog '1.1.7.11'
57     end component AuthenticationLog
58     root scenario RequestPassword '1.1.7.12'
59       connections
60         scenarioconnection '1.1.7.12.s.1' to '1.1.7.13'
61         scenarioconnection '1.1.7.12.s.2' to '1.1.7.14'
62         scenarioconnection '1.1.7.12.s.3' to '1.1.7.15'
63         association '1.1.7.12.a.1' to '2' role ''
64       end connections
```

```
66        end scenario RequestPassword
67        partial sequence 1 scenario ReceiveHint '1.1.7.13'
68        end scenario ReceiveHint
69        partial sequence 2 scenario EnterAnswer '1.1.7.14'
70        end scenario EnterAnswer
71        partial sequence 3 scenario GetPassword '1.1.7.15'
72        end scenario GetPassword
73     end consists of
74     functional speci cation
75        attributes
76           user : string;
77        operation userNameEntered(name : string)
78           statements
79              user = name
80        end operation userNameEntered
81     end functional speci cation
82     connections
83        association '1.1.7.a.1' to '1.1.6' role 'Authenticate'
84        joinrelationship '1.1.7.j.1' from '1.1.7.1' to '1.1.3.t.2' before
85        joinrelationship '1.1.7.j.2' from '1.1.7.1' to '1.1.5.t.2' before
86        joinrelationship '1.1.7.j.3' from '1.1.7.1' to '1.1.5.t.3' before
87        joinrelationship '1.1.7.j.4' from '1.1.7.4' to '1.1.3.3' before
88        joinrelationship '1.1.7.j.5' from '1.1.7.4' to '1.1.5.8' before
89        joinrelationship '1.1.7.j.6' from '1.1.7.4' to '1.1.5.11' before
90     end connections
91  end aspect Authentication
```

Appendix E

Functions on Syntax Trees

In the this Section of the appendix, a catalog of ADORA syntax tree functions is presented. They are used for defining the constraints discussed in Chapter 7 and the model transformations in Chapter 9. The catalog is divided into three subsections. The first Section E.1 gives a formal description of the set of syntax trees Δ (cf. Section 6.2). Section E.3 contains primitive functions, such as projection, insertion and deletion of syntax tree elements. Section E.4 presents functions which are used for retrieving properties of a given model tree, such as the parts of a component or the sequence number of a scenario. Section E.2 contains an alphabetical register of all ADORA syntax tree functions presented in the following. Section E.5 presents functions which are used to define the language constraints of the aspect-oriented language elements. Section E.6 introduces functions that are employed for the weaving of model transformations.

E.1 Formalized Data Structure for Syntax Tree

There is a parsing function[1] $\rho : (\mathcal{G} \times \mathcal{L}) \rightarrow \Delta$ which takes the rules in grammar $\mathcal{G}$ (cf. Definition 6.1) and an arbitrary word $w \in \mathcal{L}$, where $\mathcal{L} \subseteq T^*$ is the language generated by $\mathcal{G}$. ρ produces a *syntactically correct* syntax tree $\delta \in \Delta$ from the given input, where Δ is the set of syntax trees (cf. Section E.1).

Let N be the set of non-terminals and T the set of terminals of the ADORA grammar definition (cf. Section 6.1.1). In that case, the set Δ of ADORA syntax trees is defined inductively as follows:

i. The set of syntax trees Δ contains all terminal elements, i.e., $T \subset \Delta$.

ii. Furthermore, the set of syntax trees Δ contains pairs of the form $N \times \Delta^k$ where $k \in \mathbb{N}$ and Δ^k denotes a tuple of k elements from the set Δ. (E.1)

iii. Δ contains only elements obtained from the clauses i and ii.

Hence, the set Δ consists of all syntax trees adhering to the grammar G as well as all possible subtrees also including terminal elements.[2] Note that Δ will be used in the appendix of this work to refer to ADORA syntax trees when defining parts of the language formally.

[1] The parsing function is not at the focus of interest and therefore, it is not elaborated here.

[2] Thus, the parsing function ρ is not surjective as it does not return all elements of its codomain Δ.

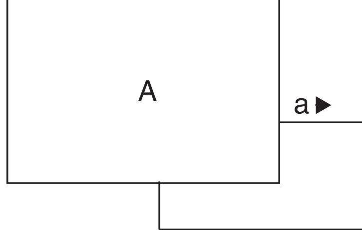


Figure E.1: A simple example model for the illustration of functions working on syntax trees. The model shows a component *A* with a re exive transition.

E.2 Alphabetical Catalog of Functions

The functions use logical functions and set operations as basic constructs. They are either defined in a mathematical notation or, where it is more convenient, in pseudocode. Note that the definition of a function in pseudocode aims at presenting the function's semantics, and not at an efficient performance.

For discussing some of the functions defined in the following, the simple ADORA model in Fig. E.1 will be used. Its concrete syntax tree is given in Fig. E.2, whereas its tuple structure can be found in Fig. E.3.

The following Table E.1 contains an alphabetical list of all syntax tree functions presented in the following.

Table E.1: Alphabetical list of syntax tree function.

Function name	Page	Type of function
π $\longrightarrow$ see also projection	309	primitive
ρ	311	primitive
ρ^{-1}	311	primitive
actionPart	312	basic
adaptCloneReferences	352	weaving
arity	309	primitive
attributes	313	basic
distance	319	basic
calcdistance	313	basic
child	312	primitive
childOfType	313	basic
children	312	primitive
childrenSet	314	basic
childrenSetOfType	314	basic
cloneElement	352	weaving

Continued on the next page . . .

Function name	Page	Type of function
conditionPart	314	basic
connections	316	basic
containedAspects	343	aspect
containsElement	316	basic
createAdditionalExitStateClone	354	weaving
createCloneId	354	weaving
createCloneMap	355	weaving
createElementReferenceTree	316	basic
createUniqueElementIdentTree	316	basic
dataTypeDeclarations	319	basic
decompositionParent	319	basic
delete	310	primitive
descendants	319	basic
distance	319	basic
elementReference	321	basic
enclosingAspects	343	aspect
endTargetModules	345	aspect
equals	321	basic
exitPoints	345	aspect
lterFsElement	321	basic
lterOperationOrProperties	321	basic
lterProvidesRequires	323	basic
lterSet	323	basic
nd	323	basic
ndClone	356	weaving
ndEoJrs	346	aspect
ndJoinRelationshipCycle	346	aspect
ndOrderingGroups	356	weaving
ndParentAndChildrenScenarios	325	basic
ndScenarioGroupMembers	325	basic
ndScenarioSubtreeMembers	327	basic
ndStateGroupMembers	327	basic
rstJr	357	weaving
atenTuple	327	basic
functionalSpec	327	basic

Continued on the next page ...

Function name	Page	Type of function
gatherAspects	346	aspect
gatherJrs	357	weaving
generateUniqueElementIdenti er	357	weaving
generateUniqueName	359	weaving
generateUniqueRole	359	weaving
hasJoinRelationshipCycles	348	aspect
identicalElement	359	weaving
insert	310	primitive
invariants	329	basic
isPredecessorGroup	360	weaving
jrHostingAspect	348	aspect
mappedReference	360	weaving
operations	329	basic
orderedChildrenOfType	329	basic
ordering	348	aspect
partial	330	basic
parts	330	basic
priority	351	aspect
prioritySort	360	weaving
projection	309	primitive
provides	331	basic
requires	331	basic
removeScenarios	362	weaving
roleChannelNames	332	basic
roleName	332	basic
rootScenarios	332	basic
scenarioGroups	332	basic
scenarioParent	335	basic
scenarioSiblings	335	basic
scenarioType	335	basic
seekTargetConnections	335	basic
serverComponentAssociations	351	aspect
seqNo	335	basic
sortTargetGroups	362	weaving
source	337	basic

Continued on the next page . . .

Function name	Page	Type of function
sourceRole	337	basic
specialIdenti er	338	basic
standardizedProperties	338	basic
startStates	338	basic
stateGroups	338	basic
target	339	basic
targetConnections	341	basic
targetModule	351	aspect
targetRole	341	basic
topologicJrSort	362	weaving
treeAncestor	343	basic
type	311	primitive
uniqueElementIdenti er	343	basic

E.3 Primitive Functions

ADORA models are syntax trees that are represented as tuple structures (cf. Section 6.2.1). Several primitive functions are introduced in the following. The first function allows retrieval of the *arity* and doing a projection on a specific element in a syntax tree. The function *delete* and *insert* provide a way to manipulate a syntax tree. The function ρ allows creation of a syntax tree from a textual model, and the function ρ^{-1} performs the reverse operation. The functions *child* and *children* are introduced for reasons of convenience. They return one specific child, or all children of a given root node of a syntax tree, respectively. Finally, the function $type$ allows retrieval of the type of a given syntax (sub)tree.

All primitive functions are employed to build more complex functions. These complex functions are in turn either used to read the basic properties, to retrieve aspect-oriented elements, or to perform the weaving transformations of ADORA model.

Arity of a tuple. In Definition E.2, the function $arity : M_0 \times M_1 \times \ldots \times M_k \to \mathbb{N}$ is given, where $M_0, M_1, \ldots, M_k$ denote arbitrary sets. The function takes a tuple $m \in M_0 \times M_1 \times \ldots \times M_k$ and returns the number of elements contained in the tuple. Note that tuples with no elements are denoted as $\emptyset$.

$$arity(m) = \begin{cases} k+1 & \text{if } m = (m_0, \ldots, m_n, \ldots, m_k) \\ 0 & \text{if } m = \emptyset \end{cases} \tag{E.2}$$

Projection of tuple. Definition E.3 specifies the projection function $\pi_n : \mathbb{N}_0 \times M_0 \times M_1 \times \ldots \times M_n \times \ldots \times M_k \to M_n$ for a given tuple $m = (m_0, \ldots, m_n, \ldots, m_k)$, where $n \in \mathbb{N}_0$. The function returns the empty word ε if $n \geq arity(m)$.

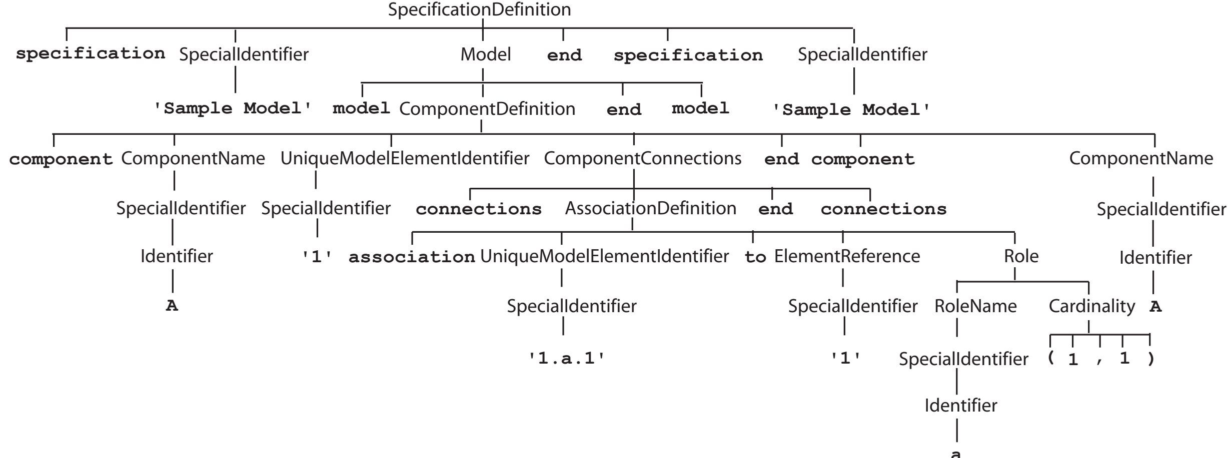


Figure E.2: A concrete syntax tree example. It shows the concrete syntax tree for the model of Fig. E.1.

$$\pi_n(m) = \begin{cases} m_n & \text{if } n < arity(m) \\ \varepsilon & \text{if } n \geq arity(m) \\ \varepsilon & \text{if } arity(m) = 0 \end{cases} \tag{E.3}$$

Insert an element into a tuple. Definition E.4 defines the function $insert_n$ which inserts an element $r \in R$, where R is an arbitrary set, in a tuple $m \in (M_0 \times M_1 \times \ldots \times M_k)$ at the position of the element m_n. The elements m_i with $i \geq n$ are positioned at the index $i + 1$. If the position of n is equal or greater than the arity of the tuple, the value is inserted at the end.

$$insert_n(m,r) = \begin{cases} \big(\emptyset, r\big) \mapsto (r) & \text{if } arity(m) = 0 \\ \big((m_0, \ldots, m_k), r\big) \mapsto (m_0, \ldots m_k, r) & \text{if } arity(m) \leq n \\ \big((m_0, \ldots, m_n, \ldots m_k), r\big) \mapsto (q_0, \ldots, q_n, \ldots q_{k+1}), & \\ \quad \text{where } \big(\forall i \in \mathbb{N}_0 : i < n \Rightarrow (q_i = m_i)\big) \wedge & \text{if } arity(m) > n \\ \qquad \big(\forall i \in \mathbb{N}_0 : (n < i < k+2) \Rightarrow (q_i = m_{i-1})\big) \wedge q_n = r & \end{cases} \tag{E.4}$$

Delete an element from a tuple. Definition E.5 describes the $delete$ operation for an element of a tuple. The function removes the element at position n from a tuple $m \in (M_0 \times M_1 \times \ldots \times M_k)$.

(SpecificationDefinition, (**specification**, (SpecialIdentifier, (**'Sample Model'**)), (**specification**, (Model, (**model**, (ComponentDefinition, (**component**, (ComponentName, ((SpecialIdentifier, ((Identifier, (**A**)))))), (UniqueModelElementIdentifier, ((SpecialIdentifier, (**'1'**)))), (ComponentConnections, (**connections**, (AssociationDefinition, (**association**, (UniqueModelElementIdentifier, ((SpecialIdentifier,(**'1.a.1'**)))), **to**, (ElementReference, ((SpecialIdentifier,(**'1'**)))), (Role, (**role**, (RoleName, ((SpecialIdentifier, ((Identifier,(**a**)))))), (Cardinality, (**(, 1, ,, 1,)**)))))), **end**, **connections**)), **end**, **component**), (ComponentName, ((SpecialIdentifier, ((Identifier, (**A**))))))), **end**, **model**)), **end**, **specification**), (SpecialIdentifier, (**'Sample Model'**))))

Figure E.3: A tuple representation of the ADORA model of Fig. E.1.

$$delete_n(m) = \begin{cases} \emptyset \mapsto \emptyset & \text{if } arity(m) = 0 \\ (m_0, m_1, \dots m_k) \mapsto (m_0, m_1, \dots m_k) & \text{if } arity(m) \leq n \\ (m_0, \dots, m_n, \dots m_k) \mapsto (q_0, \dots, q_{n-1}, q_n, \dots q_{k-1}), & \\ \quad \text{where } \big(\forall i \in \mathbb{N}_0 : i < n \Rightarrow (q_i = m_i)\big) \wedge & \text{if } arity(m) > n \\ \quad\quad \big(\forall i \in \mathbb{N}_0 : (n \leq i < k) \Rightarrow q_i = m_{i+1}\big) & \end{cases} \tag{E.5}$$

Creating a syntax tree from a text model. The function $\rho : (\mathcal{G} \times \mathcal{L}) \rightarrow \Delta$ takes the grammar $\mathcal{G}$ (cf. Definition 6.1) and an arbitrary word $w \in \mathcal{L}$, where $\mathcal{L} \subseteq T^*$ is the language generated by $\mathcal{G}$. ρ produces a *syntactically correct* syntax tree $\delta \in \Delta$ from the given input, where Δ is the set of syntax (sub)trees. See also Section 6.2.1 for further explanations. This function is not algorithmically described in the appendix.

Creating a text model from a syntax tree. The function $\rho^{-1} : \Delta \rightarrow T^*$ is the inverse of the parsing function ρ presented above. It takes the syntax tree of any ADORA model element and creates a textual description of it. This function is not algorithmically described in the appendix.

Retrieving the type of a syntax tree. A particular ADORA syntax tree or subtree has a specific type. Its type is defined by the label of the tree's root node. For example, in Fig. E.2, the subtree defining the component *A* is of the type *ComponentDe nition*. The type information can contain additional information which might be useful to relate a particular semantics to a given element of a subtree. For instance in the model of Fig. E.2, the unique identifier of the component *A* is *1* is defined by the corresponding *UniqueModelElementIdenti er* branch (cf. the third branch from the left of the component subtree). Furthermore, the component *A* is referenced by a re exive association. Thus, the corresponding *AssociationDe nition* contains a reference *1* (cf. the second-last branch in the association's subtree). Both, the identifier of component *A* as well as the reference adhere to the same rules.[3] However, for distinguishing identifiers from references, the parent trees containing them are of different types. Identifiers are defined

[3]Both are defined by a *SpecialIdenti er* rule.

by a tree with the type *UniqueModelElementIdenti er* whereas references are defined by the trees with the type *ElementReference*.

The function $type : \Delta \rightarrow (N \cup T)$, given in Definition E.6, returns either the label of the root node of a given syntax tree $t \in \Delta$ or the label of a given terminal node.[4] Hence, the function returns an element of the set $N \cup T$.

$$type(t) = \begin{cases} t \mapsto t & \text{if } t \in T \\ \pi_0(t), \text{ where } t = \big(r, (c_0, c_1, \ldots, c_k)\big) & \text{if } t \in \Delta \end{cases} \tag{E.6}$$

Retrieving children. For reasons of convenience, the functions $children$ and $child$ are defined. The function $children_k : \Delta \rightarrow \bigcup_k \Delta^k \cup \{\varepsilon\}$, where $k \in \mathbb{N}_0$, is given in Definition E.7. It returns a tuple $(c_0, c_1, \ldots, c_k)$ containing the children of $t \in \Delta$, where t is $\big(r, (c_0, c_1, \ldots, c_k)\big)$.

$$children(t) = \begin{cases} t \mapsto \varepsilon & \text{if } t \in T \\ \pi_1(t) & \text{else} \end{cases} \tag{E.7}$$

In contrast, the function $child$ allows retrieval of *one* child of a given tree root node. The n_{th} child of a syntax tree node $t \in \Delta$ is obtained by applying the projection function (see Definition E.3) to the result of the children function (Definition E.7):

$$child_n(t) = \pi_n\big(children(t)\big) \tag{E.8}$$

E.4 Basic Functions

For all various kinds of operations with ADORA syntax trees, it is necessary to retrieve particular properties of a model. The basic functions described in the following allow retrieval of various properties, such as the parts of a component, the type of a scenario, or the name of a state.

Each of the functions takes at least one argument which is usually the syntax tree of an ADORA model or a subpart of an ADORA model. The returned values are either syntax trees of ADORA models, subtrees of it, or another kind of value which represents a retrieved property. The functions are alphabetically sorted.

Function actionPart. The function $actionPart : \Delta \rightarrow \Delta$ has one argument s which denotes the syntax tree of a transition ($TransitionDefinition$). It returns a syntax subtree which describes the action part of the transition. The method returns the empty word ε if the given syntax tree does not describe a transition or if there is no action part. The function is formally defined in Listing E.1.

[4] Remember that in the following Δ represents the set of ADORA syntax trees (cf. Section 6.2.1).

Listing E.1: The function $actionPart$.

```
1   function actionPart(transition)
2     input
3       transition ∈ Δ;
4     return
5       retVal ∈ Δ;
6     declare
7       st ∈ Δ;
8     begin
9       retVal ← ε
10      if (type(transition) =
11        TransitionDefinition) begin
12        st = childOfType(transition,
13          SimpleTransition, 1);
14        if (st ≠ ε) begin
15          retVal ← childOfType(
16            st, ActionPart, 1);
17        end
18      end
19  end
```

Listing E.2: The function $attributes$.

```
1   function attributes(fs)
2     input
3       fs ∈ Δ;
4     return
5       retVal ∈ ∪_k Δ^k, where k ∈ ℕ0;
6   begin
7     retVal ← filterFsElements(fs,
8       AttributeDefinitions,
9       VariableDefinition);
10  end
```

Function attributes. The function $attributes : \Delta \rightarrow \bigcup_k \Delta^k$, where $k \in \mathbb{N}_0$ takes an argument s which describes the syntax tree of a functional specification. The function gathers the syntax subtrees of s which define attributes and returns them in a tuple. The elements in the tuple are ordered according to their occurrence in s, read from left to right in infix order. The function returns the empty tuple $\emptyset$ if s is not the syntax tree of a functional specification or there are no attributes contained in the functional specification. The function is formally defined in Listing E.2.

Function calcdistance. The function $calcdistance : \Delta \times \Delta \rightarrow \mathbb{Z}$ takes two arguments t and s, where t is a ADORA syntax tree and s is another syntax tree. If s is part of t, the function calculates the distance between the root node of s and the root node of t by counting the number of edges that have to be traversed to reach the root node of t and the root node of s. If s is not part of t, the function returns -1. It is a helper function that is used by the function $distance$ (cf. page 319). The function is formally defined in Listing E.3.

Function childOfType. The function $childOfType : \Delta \times (T \cup N) \times \mathbb{N} \rightarrow \Delta \cup \{\varepsilon\}$ has three arguments s, p and n. The argument s is an arbitrary syntax tree, p is a type label, which is either a terminal or a non-terminal symbol, and n is a natural number. The function searches for the n-th occurrence of a subtree with the type p. The subtree is searched in the (direct) subtrees of s. If no subtree is found with the given type or the subtree type occurs fewer times as specified, the empty word ε is returned. An example with the model in Fig. E.1 illustrates the function. Suppose the argument s is the subtree of the component A, p is *ComponentName*, and the argument n is *1*. In this case the method returns the left-most subtree branch defining the name of the component, i.e., $(ComponentName, (SpecialIdentifier, (Identifier, (\mathbf{A}))))$. The

Listing E.3: The function *calcdistance*.

```
1   function calcdistance(tree, subtree, level)
2     input
3       tree ∈ Δ ∧ tree ∉ T;
4       subtree ∈ Δ; level ∈ ℕ0;
5       level ∈ ℕ0
6     return
7       retVal ∈ ℤ;
8     declare
9       i ∈ ℕ0; e ∈ Δ;
10      ch ∈ ∪_k Δ^k, where k ∈ ℕ0;
11    begin
12      retVal ← -1;
13      if (¬ equals(tree, subtree)) begin
14        i ← 0;
15        ch ← children(tree);
16        while (i < arity(ch) ∧ retVal = -1)
17          begin
18          e ← π_i(ch);
19          if (e ∈ Δ) begin
20            retVal ←
21              calcdistance(e, subtree, level+1);
22          end
23          i ← i + 1;
24        end
25      end else begin
26        retVal ← level + 1;
27      end
28    end
```

function is formally defined in Listing E.4.

Function childrenSet. The function $childrenSet : \Delta \rightarrow \mathcal{P}(\Delta)$ takes an arbitrary syntax tree t as argument and returns the set of all direct subtrees of t.[5] Thus, in contrast to the function $descendants$[6], defined below, the function's result does not contain the indirect subtrees of the root node. In contrast to the *children* function of Definition E.7, this function returns an *unordered* set, whereas the children function returns an *ordered* tuple of t's children. For example, calling the function with the syntax subtree of the association defined in Fig. E.2 results in a set containing its direct subtrees, thus {**`association`**, (UniqueModelElementIdentifier, ((SpecialIdentifier,(**`'1.a.1'`**)))), **`to`**, (ElementReference, ((SpecialIdentifier,(**`'1'`**)))), (Role, (**`role`**, (RoleName, ((SpecialIdentifier, ((Identifier,(**`a`**)))))), (Cardinality, (**`(`**, **`1`**, **`,`**, **`1`**, **`)`**)))) }. The function is formally defined in Listing E.5.

Function childrenSetOfType. The function $childrenSetOfType : \Delta \times (T \cup N) \rightarrow \mathcal{P}(\Delta)$ takes the arguments t and p. The argument t is an arbitrary syntax tree and p is a type label. The function returns the set of all direct subtrees of t having the type p. If no subtree of the specified type is found, the empty set is returned. For instance, when calling the function with the syntax subtree of the component A, given in Fig. E.2, as t and *ComponentName* as argument p, a set is returned which contains one subtree representing the name of the component.[7] Thus, the result is { ($ComponentName$, ($SpecialIdentifier$, ($Identifier$, (**`A`**)))) }. The function is formally defined in Listing E.6.

Function conditionPart. The function $conditionPart : \Delta \rightarrow \Delta$ has one argument s which contains the syntax tree of a transition ($TransitionDefinition$). It returns the syntax subtree which describes the

[5] Remember that, $\mathcal{P}(\Delta)$ is the power set of Δ.

[6] The function descendants returns the set of all direct and indirect subtrees of t.

[7] One might suppose that two elements in the set are returned, because there is one component name in the header and another one in the footer of the definition. However, the returned subtrees are equal, and therefore only one element results.

Listing E.4: The function $childOfType$.

```
1   function childOfType(tree, type, occurrence)
2     input
3       tree ∈ Δ ∧ tree ∉ T;
4       type ∈ T ∪ N;
5       occurrence ∈ ℕ0;
6     return
7       retVal ∈ Δ;
8     declare
9       count ∈ ℕ0;
10      i ∈ ℕ0;
11    begin
12      i ← 0; retVal ← ε;
13      count ← 0;
14      while (i < arity(children(tree)) ∧
15              retVal = ε) begin
16        if (type(child_i(tree) = type) begin
17          count ← count + 1;
18          if (count = occurrence) begin
19            retVal ← child_i(tree);
20          end
21        end
22        i ← i + 1;
23      end
24    end
```

Listing E.5: The function $childrenSet$.

```
1   function childrenSet(tree)
2     input
3       tree ∈ Δ ∧ tree ∉ T;
4     return
5       retVal ∈ 𝒫(Δ);
6     declare
7       i ∈ ℕ0;
8     begin
9       i ← 0;
10      retVal ← ∅;
11      while (i < arity(children(tree)) ∧
12              retVal = ε) begin
13        retVal ← retVal ∪ { child_i(tree) };
14        i ← i + 1;
15      end
16    end
```

Listing E.6: The function $childrenSetOfType$.

```
1   function childrenSetOfType(tree, type)
2     input
3       tree ∈ Δ ∧ tree ∉ T;
4       type ∈ T ∪ N;
5     return
6       retVal ∈ 𝒫(Δ);
7     declare
8       i ∈ ℕ0;
9     begin
10      i ← 0;
11      retVal ← ∅;
12      while (i < arity(children(tree)) ∧
13              retVal = ε) begin
14        if (type(child_i(tree) = type) begin
15          retVal ← retVal ∪ { child_i(tree) };
16        end
17        i ← i + 1;
18      end
19    end
```

Listing E.7: The function $conditionPart$.

```
1   function conditionPart(transition)
2     input
3       transition ∈ Δ;
4     return
5       retVal ∈ Δ;
6     declare
7       st ∈ Δ;
8     begin
9       retVal ← ε;
10      if (type(transition)=TransitionDefinition)
11        begin
12          st = childOfType(
13            transition, SimpleTransition, 1);
14          if (st ≠ ε) begin
15            retVal ← childOfType(
16              st, ConditionPart, 1);
17          end
18        end
19  end
```

condition part of the transition. The method returns the empty word ε if the given syntax tree does not describe a transition or if the transition is an epsilon transition without a message reception part and a guard. The function is formally defined in Listing E.7.

Function connections. The function $connections : \Delta \rightarrow \mathcal{P}(\Delta)$ takes the argument t which is a syntax tree of type *ComponentDe nition*, *StateDe nition*, *AspectDe nition*, or *ScenarioDe nition*. The function returns the set of syntax trees of t's out-going connections. If t is not one of the above specified types, or t does not contain any out-going connections, the empty set is returned. For example, calling the function for the syntax subtree of the component A in Fig. E.1 as t, the function returns a set with one element comprising the association defined in component A, i.e., { (AssociationDefinition, (**association**, (UniqueModelElementIdentifier, ((SpecialIdentifier,(**'1.a.1'**)))), **to**, (ElementReference, ((SpecialIdentifier,(**'1'**)))), (Role, (**role**, (RoleName, ((SpecialIdentifier, ((Identifier,(**a**)))))), (Cardinality, ((**(**, **1**, **,**, **1**, **)**)))))) }. The function is formally defined in Listing E.8.

Function containsElement. The function $containsElement : \Delta \times \Delta \rightarrow \{true, false\}$ takes two arguments t and s, which must both be syntactically correct syntax (sub)trees. The function returns true, if s is a subtree of t, otherwise it returns false. The algorithmic description of the function can be found in Listing E.9.

Function createElementReferenceTree. The auxiliary function $createElementReferenceTree : T \rightarrow \Delta$ takes a token t which must recognized as <APOSTROPHIZED_IDENTIFIER>. The token is used to create the tree for an element reference, i.e., a subtree which is used to refer to another element that has a unique model element identifier. The algorithmic description of the function can be found in Listing E.10.

Function createUniqueElementIdentTree. The auxiliary function *createUniqueElementIdentTree*: $T \rightarrow \Delta$ takes an argument t which must be a token that belongs to the set of <APOSTROPHIZED_-IDENTIFIER>. The function builds a syntax tree for a unique model element identifier. The function is formally defined in Listing E.11.

Listing E.8: The function *connections*.

```
1   function connections(tree)
2     input
3       tree ∈ Δ;
4     return
5       retVal ∈ 𝒫(Δ);
6     declare
7       typeVar ∈ N;
8       e ∈ Δ;
9       temp ∈ 𝒫(Δ);
10    begin
11      temp ← ∅;
12      retVal ← ∅;
13      typeVar ← ε;
14      if (type(tree) =
15        ComponentDefinition) begin
16        typeVar ← ComponentConnections;
17      end else if (type(tree) =
18        AspectDefinition) begin
19        typeVar ← AspectConnections;
20      end else if (type(tree) =
21        StateDefinition) begin
22        typeVar ← StateConnections;
23      end else if (type(tree) =
24        ScenarioDefinition) begin
25        typeVar ← ScenarioConnections;
26      end else if (type(tree) =
27        EnvironmentObjectDefinition) begin
28        typeVar ←
29          EnvironmentObjectConnections;
30      end
31      if (typeVar ≠ ε) begin
32        e ← childOfType(tree, typeVar, 1);
33        if (e ≠ ε) begin
34          temp = childrenSet();
35        end
36      end
37      foreach e ∈ temp begin
38        if (e ∉ T) begin
39          retVal ← retVal ∪ { e }
40        end
41      end
42    end
```

Listing E.9: The function *containsElement*.

```
1   function containsElements(model, element)
2     input
3       model ∈ Δ \ T;
4       element ∈ Δ;
5     return
6       retVal ∈ { true, false };
7     declare
8       children ∈ ∪_k Δ^k, where k ∈ ℕ0;
9       i ∈ ℕ0;
10    begin
11      retVal ← false; i← 0;
12      children ← children(model);
13      while (i < arity(children) ∧
14        retVal = false)
15      begin
16        if (child_i(model) ∉ T) begin
17          if (equals(child_i(model),
18            element)
19          begin
20            retVal ← true;
21          end else begin
22            retVal ← containsElement(
23              child_i(model), element);
24          end
25        end
26        i ← i + 1;
27      end
28    end
```

Listing E.10: The function *createElementReferenceTree*.

```
1  function createElementReferenceTree(token)
2    input
3      token ∈ <APOSTROPHIZED_IDENTIFIER>;
4    return
5      retVal ∈ Δ ∧ type(retVal) = ElementReference;
6    begin
7      retVal ← ( ElementReference, ( (SpecialIdentifier, (token) ) ) );
8    end
```

Listing E.11: The function *createUniqueElementIdentTree*.

```
1  function createUniqueElementIdentTree(token)
2    input
3      token ∈ <APOSTROPHIZED_IDENTIFIER>;
4    return
5      retVal ∈ Δ ∧ type(retVal) = UniqueModelElementIdentifier;
6    begin
7      retVal ← ( UniqueModelElementIdentifier, ( (SpecialIdentifier, (token) ) ) );
8    end
```

Listing E.12: The function $dataTypeDeclarations$.

```
1   function dataTypeDeclarations
2     (fs)
3     input
4       e ∈ Δ;
5     return
6       retVal ∈ ⋃_k Δ^k,
7         where k ∈ ℕ0;
8     begin
9       retVal ← filterFsElements(
10        fs,
11        DataTypeDeclarations,
12        DataTypeDeclaration);
13    end
```

Function dataTypeDeclarations. The function $dataTypeDeclarations : \Delta \to \bigcup_k \Delta^k$, where $k \in \mathbb{N}_0$ takes one argument s which describes the syntax tree of a functional specification. It extracts the syntax subtrees of all data types declared in the functional specification and returns them in a tuple. The elements in the tuple have the same order as they occur in s. The function returns the empty tuple $\emptyset$ if s is not the syntax tree of a functional specification or s does not contain data type declarations. The function is formally defined in Listing E.12.

Function decompositionParent. The function $decompositionParent : \Delta \times \Delta \to \Delta \cup \{\varepsilon\}$ takes two syntax trees t and s as arguments. It determines the smallest subtree p such that p comprises s in its parts section which may be either of the type *StateParts*, *ComponentParts*, or *Model*.[8] Table E.2 specifies the valid part-of relationships between p and s.

Table E.2: Table defining the valid part-of relationships of ADORA elements.

Type of s	Possible types of parent nodes p
ComponentDefinition	Model, ComponentDefinition
EnvironmentObjectDefinition	Model
StateDefinition	ComponentDefinition, StateDefinition
ScenarioDefinition	ComponentDefinition

If s has a type other than those given in Table E.2, the decompositionParent function returns the value ε. The function is exemplified by Fig. E.1: when calling the function $decompositionParent$ with the syntax tree of this model as t and the subtree of the component *A* (i.e., the subtree with the type *ComponentDe nition*) as s, the function returns the subtree of the Model. A formal definition of the function can be found in Listing E.13.

Function descendants. The function $descendants : \Delta \to \mathcal{P}(\Delta)$ takes a syntax tree t as argument and returns the set of all direct and indirect subtrees of t's. For example, the set of descendants for the *ComponentName* subtree of the component *A* in Fig. E.2 is {(ComponentName, ((SpecialIdentifier, ((Identifier, (**A**)))))), ((SpecialIdentifier, ((Identifier, (**A**))))), (Identifier, (**A**)), **A** }. A formal definition of the function can be found in Listing E.14.

Function distance. The function $distance : \Delta \times \Delta \to \mathbb{Z}$ takes two syntax trees t and s as arguments and checks whether t contains s as a direct or indirect child. If s is contained in t, the number of edges that have to be traversed to get from the root node of t to the root node of s is returned. Otherwise -1 is returned. For example, calling the function with the syntax tree given in Fig. E.2 as t and the syntax subtree of the definition of component *A*, which has the root node *ComponentDe nition* returns the value 2. A formal definition of the function can be found in Fig. E.15.

[8] The smallest subtree is determined by using the *distance* function defined below.

Listing E.13: The function $decompositionParent$.

```
1   function decompositionParent(model,
2     subtree)
3     input
4       model ∈ Δ;
5       subtree ∈ Δ;
6     return
7       retVal ∈ Δ;
8     declare
9       map ⊆ (N × ℕ);
10    begin
11      map ← ∅;
12      retVal ← ε;
13      if (type(subtree) =
14        ComponentDefinition)
15      begin
16        map ← {
17          (Model, 1),
18          (ComponentDefintion, 2),
19          (AspectDefinition, 2) };
20      end
21      if (type(subtree) =
22        EnvironmentObjectDefinition)
23      begin
24        map ← {(Model, 1)};
25      end
26      if (type(subtree) =
27        StateDefinition)
28      begin
29        map ← {
30          (ComponentDefinition, 2),
31          (StateDefintion, 2),
32          (AspectDefinition, 2) };
33      end
34      if (type(subtree) =
35        AspectDefinition)
36      begin
37        map ← {
38          (Model, 1),
39          (ComponentDefintion, 2),
40          (AspectDefinition, 2) };
41      end
42      if (type(subtree) =
43        ExitPointDefinition)
44      begin
45        map ← {(AspectDefintion, 2)};
46      end
47      if (type(subtree) =
48        ScenarioDefinition)
49      begin
50        map ← {
51          (ComponentDefintion, 2),
52          (AspectDefinition, 2) };
53      end
54      if (map ≠ ∅)
55      begin
56        retVal ← treeAncestor(
57          model, subtree, map);
58      end
59    end
```

Listing E.14: The function *descendants*.

```
1   function descendants(tree)
2     input
3       tree ∈ Δ ∧ tree ∉ T;
4     return
5       retVal ∈ 𝒫(Δ)
6     declare
7       i ∈ ℕ0; ch ∈ ∪_k Δ^k, where k ∈ ℕ0;
8       e ∈ Δ;
9     begin
10      i ← 0;
11      ch ← children(tree);
12      while (i < arity(ch)) begin
13        e = π_i(ch);
14        retVal ← retVal ∪ { e };
15        if (e ∈ Δ) begin
16          retVal ← retVal ∪ descendants(e);
17        end
18        i ← i + 1;
19      end
20    end
```

Listing E.15: The function *distance*.

```
1  function distance(tree, subtree)
2    input
3      tree ∈ Δ ∧ tree ∉ T;
4      subtree ∈ Δ ∧ subtree ∉ T;
5    return
6      retVal ∈ ℕ0;
7    begin
8      retVal ← calcdistance(tree, subtree, -1);
9    end
```

Listing E.16: The function *elementReference*.

```
1  function elementReference(tree)
2    input
3      tree ∈ Δ;
4    return
5      retVal ∈ uniqueIdentifier ∈
6        <REFERENCE_STRING_LITERAL>
7        ∪ <IDENTIFIER>;
8    begin
9      if (type(tree) = ElementReference) begin
10       retVal ← specialIdentifier(child0(tree));
11     end
12   end
```

Function elementReference. The function $elementReference : \Delta \rightarrow T$ has one argument s which is an ADORA syntax tree. If s contains the syntax tree of an element reference, the token denoting element reference is returned, otherwise the return value is undefined. The algorithmic description of the function can be found in Listing E.16.

Function equals. The function $equals : \Delta \times \Delta \rightarrow \{true, false\}$ returns $true$ if two given syntax trees t_1 and t_2 are equal. Two syntax trees are equal if they are isomorphic, i.e., they are structurally equivalent and the labels of the corresponding nodes are equal. The use of $equal(t_1, t_2)$ is equivalent to the expression $t_1 = t_2$. A formal definition of the function can be found in Listing E.17.

Function lterFsElement. The function $filterFsElement : \Delta \times N \times N \rightarrow \bigcup_k \Delta^k$, where $k \in \mathbb{N}_0$, takes three arguments t, m, and p. The argument t is an ADORA syntax tree, m and p are non-terminal elements which denote the type of a subtree. The function returns all children of t which have p as type and m as the parent's type. The found elements are returned as a tuple. The order of the tuple is the order of the found elements in the tree. This function is an auxiliary function which is used by the functions *attributes* (cf. p. 313), *dataTypeDeclarations* (cf. p. 319), and *invariants* (cf. p. 329). A formal definition of the function can be found in Listing E.18.

Function lterOperationOrProperties. The function $filterOperationsOrProperties : \Delta \times N \rightarrow \bigcup_k \Delta^k$, where $k \in \mathbb{N}_0$, takes two arguments t and m. The argument t is an ADORA syntax tree, the argument m is a non-terminal element which denotes the type of a subtree. The function returns all subtrees which have m as type. The found subtrees are returned in a tuple which has the same order as the occurrence of the found elements in the tree. This function is an auxiliary function which is used by the functions *operations* (cf. p. 329) and *standardizedProperties* (cf. p. 338). A formal definition of the function can be found in Listing E.19.

Listing E.17: The function *equals*.

```
1   function equals(tree1, tree2)
2     input
3       tree1 ∈ Δ \ T;
4       tree2 ∈ Δ \ T;
5     return
6       retVal ∈ {true,false}
7     declare
8       i ∈ ℕ0;
9       ch1 ∈ (∪_k Δ^k), where k ∈ ℕ0;
10      ch2 ∈ (∪_k Δ^k), where k ∈ ℕ0;
11      e1 ∈ Δ; e2 ∈ Δ;
12    begin
13      i ← 0;
14      retVal ← true;
15      if (type(tree1) = type(tree2)) begin
16        ch1 ← children(tree1);
17        ch2 ← children(tree2);
18        if (arity(ch1) ≠ arity(ch2)) begin
19          retVal ← false;
20        end
21        while (retVal ∧ i < arity(ch1)) begin
22          e1 ← π_i(ch1);
23          e2 ← π_i(ch2);
24          if (e1 ∈ T ∧ e2 ∈ T) begin
25            if (e1 ≠ e2) begin
26              retVal ← false;
27            end
28          end else if (e1 ∈ Δ ∧ e2 ∈ Δ) begin
29            retVal ← equals(e1, e2);
30          end else begin
31            retVal ← false;
32          end
33          i ← i + 1;
34        end
35      end else begin
36        retVal ← false;
37      end
38    end
```

Listing E.18: The function *filterFsElement*.

```
1   function filterFsElement
2     (fs, superType, subType)
3     input
4       fs ∈ Δ;
5       superType ∈ N;
6       subType ∈ N;
7     return
8       retVal ∈ ∪_k Δ^k, where k ∈ ℕ0;
9     declare
10      invs ∈ ∪_k Δ^k, where k ∈ ℕ0;
11      i ∈ ℕ0;
12      j ∈ Integer0;
13      ch ∈ ∪_k Δ^k, where k ∈ ℕ0;
14    begin
15      retVal ← emtpySet;
16      if (type(fs) = FunctionalSpecification)
17        begin
18        invs = children(fs);
19        j ← 0;
20        while (j < arity(invs)) begin
21          if (π_j(invs) ∉ T ∧
22            type(π_j(invs)) = superType begin
23            i ← 0;
24            ch ← children(π_j(invs));
25            while (i < arity(ch)) begin
26              if (type(π_i(ch)) = subType) begin
27                retVal ← insert_arity(retVal)(
28                  retVal, π_i(ch));
29              end
30              i ← i + 1;
31            end
32          end
33          j ← j + 1;
34        end
35      end
36    end
```

Listing E.19: The function $\mathit{filterOperationsOrProperties}$.

```
1   function filterOperationsOrProperties
2     (fs,stType)
3     input
4       fs ∈ Δ;
5       stType ∈ N;
6     return
7       retVal ∈ ∪_k Δ^k, where k ∈ ℕ_0;
8     declare
9       i ∈ ℕ_0;
10      ch ∈ ∪_k Δ^k, where k ∈ ℕ_0;
11    begin
12      retVal ← ∅;
13      if (type(fs) = FunctionalSpecification)
14        begin
15          ch = children(fs);
16          i ← 0;
17          while (i < arity(ch)) begin
18            if (type(π_i(ch)) = stType) begin
19              retVal ←
20                insert_arity(retVal)(retVal, π_i(ch));
21            end
22            i ← i + 1;
23          end
24        end
25    end
```

Function lterProvidesRequires. The auxiliary function $\mathit{filterProvidesRequires} : \Delta \times N \times N$ has three arguments t, m, and p. It extracts either the elements defined in the *provides* or the *requires* declaration of a functional specification. The elements that should be extracted are specified by the arguments m and p. Argument m must either be the non-terminal *Provides* or *Requires*. The argument p denotes the type of the elements contained in the corresponding requires/provides declaration, i.e., *Identi er* for provides and *Quali edIdenti er* for requires declarations. The function is used by the functions $\mathit{provides}$ (cf. p. 331) and $\mathit{requires}$ (cf. p. 331). A formal definition of the function can be found in Listing E.20.

Function lterSet. The function $\mathit{filterSet} : \mathcal{P}(\Delta) \times (N \cup T \rightarrow \mathcal{P})(\Delta)$ takes two arguments m and p. The argument m denotes a set of syntax trees, and p denotes a type label, i.e., a terminal or a non-terminal symbol of the ADORA syntax. The function returns a subset of m which contains all trees of the type p. If no syntax trees in m have the type p, the empty set is returned. For example, when calling the function $\mathit{filterSet}(\mathit{descendants}(t), \mathit{Identifier})$, where t is the syntax tree of the model in Fig. E.2, a set with the two elements { (Identifier, (**A**)), (Identifier, (**a**)) } is returned. The first element is the identifier from the component name and the second element is the identifier of the association's role name. A formal definition of the function can be found in Listing E.21.

Function nd. The function $\mathit{find} : \Delta \times A \rightarrow \Delta \cup \{\epsilon\}$, where A is the set of all identifiers defined by the grammar rule *UniqueModelElementIdenti er*[9], takes two arguments t and a. The function searches in the syntax tree t for a subtree defining a (semantical) element, such as a component, an association, a transition, etc. which is identified by the unique element identifier a (cf. Section 6.1.5). Unique element identifiers are represented by subtrees of the type *UniqueModelElementIdenti er*. The label of the only leaf node of such a *UniqueModelElementIdenti er* subtree must be equal to a. If a is not contained in a *UniqueModelElementIdenti er* subtree of t, the empty word ε is returned.

The find function is defined as a recursive algorithm in Listing E.22. Calling the function $\mathit{find}(t,' 1')$, where t is the syntax tree illustrated in Fig. E.2, exemplifies how the *nd* function works. It returns the

[9] According to the grammar definition in Appendix B, the set of all unique model element identifiers consist of A = <APOSTROPHIZED_IDENTIFIER> ∪ <IDENTIFIER>.

Listing E.20: The function *filterProvidesRequires*.

```
1  function filterProvidesRequires
2    (fs, type, subType)
3    input
4      fs ∈ Δ;
5      type ∈ N;
6      subType ∈ N;
7    return
8      retVal ∈ ∪_k Δ^k, where k ∈ ℕ_0;
9    declare
10     provides ∈ Δ;
11     i ∈ ℕ_0;
12     ch ∈ ∪_k Δ^k, where k ∈ ℕ_0;
13   begin
14     i ← 0;
15     retVal ← ∅;
16     if (type(fs) = FunctionalSpecification)
17     begin
18       provides ← childOfType(fs, type, 1);
19       if (provides ≠ ε) begin
20         ch ← children(provides);
21         while (i < arity(ch)) begin
22           if (type(π_i(ch)) = subType) begin
23             retVal ← insert_arity(ch)(
24               retVal, π_i(ch));
25           end
26           i ← i + 1;
27         end
28       end
29     end
30   end
```

Listing E.21: The function *filterSet*.

```
1  function filterSet(trees, type)
2    input
3      trees ∈ P(Δ);
4      type ∈ N ∪ T;
5    return
6      retVal ∈ P(Δ);
7    declare
8      element ∈ Δ;
9    begin
10     retVal ← ∅;
11     foreach (element ∈ trees) begin
12       if (type(element) = type) begin
13         retVal ← retVal ∪ { element };
14       end
15     end
16   end
```

Listing E.22: The function *find*.

```
1   function find(tree, uniqueIdentifier)
2     input
3       tree ∈ Δ ∧ tree ∉ T;
4       uniqueIdentifier ∈
5         <APOSTROPHIZED_IDENTIFIER>
6         ∪ <IDENTIFIER> ;
7     return
8       retVal ∈ Δ;
9     declare
10      i ∈ ℕ0; ch ∈ ∪_k Δ^k, where k ∈ ℕ0;
11      e ∈ Δ;
12    begin
13      retVal ← ε; i ← 0;
14      ch ← children(tree);
15      while (i < arity(ch) ∧ retVal = ε) begin
16        e = π_i(ch);
17        if (type(e) =
18          UniqueModelElementIdentifier ∧
19          uniqueElementidentifier(e) =
20          uniqueIdentifier
21        begin
22          retVal ← tree;
23        end else if (e ∈ Δ) begin
24          retVal ← find(e, uniqueIdentifier);
25        end
26        i ← i + 1;
27      end
28    end
```

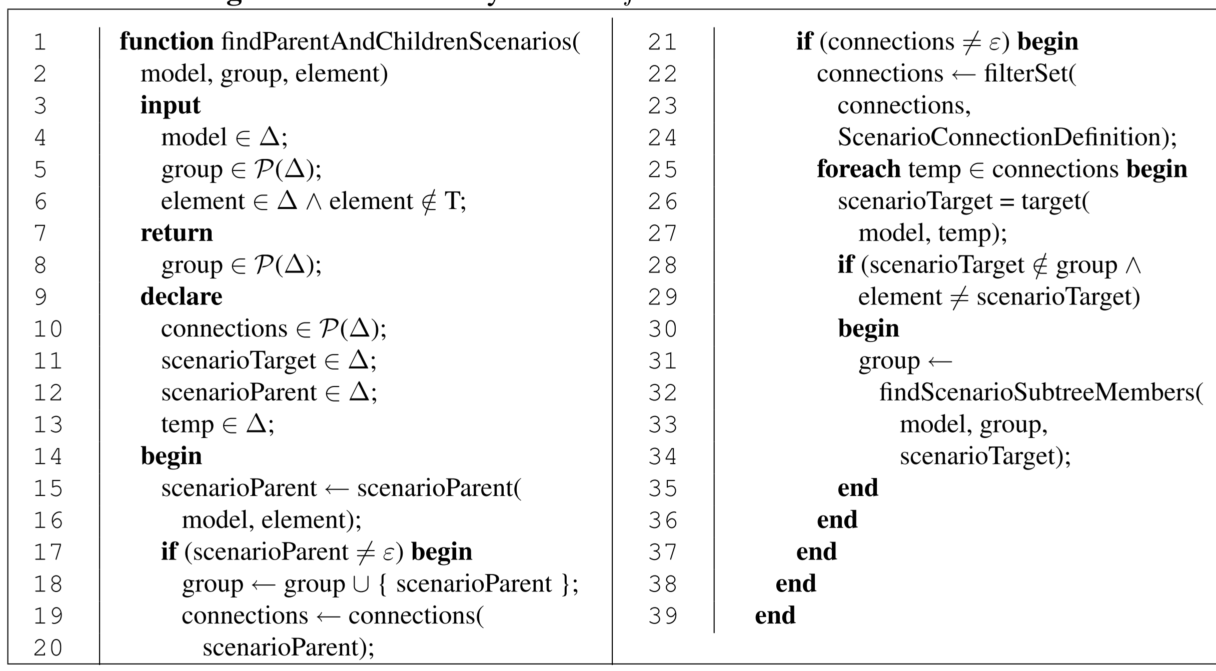
subtree for the component A, i.e., the tuple (ComponentDefinition, (**`component`**, (ComponentName, ((SpecialIdentifier, ((Identifier, (**`A`**)))))),..., **`end`**, **`component`**), (ComponentName, ((SpecialIdentifier, ((Identifier, (**`A`**))))))).

Function ndParentAndChildrenScenarios. The auxiliary function *ndParentAndChildrenScenarios* : $\Delta \times \mathcal{P}(\Delta) \times \Delta \rightarrow \mathcal{P}(\Delta)$ takes three arguments t, m, and s. The argument t is a syntax tree of an ADORA model, and m denotes the set of already found syntax trees describing scenario nodes. The argument m is actually used for the recursive descent of the function, and therefore, for an initial call m is usually the empty set $\emptyset$. The argument s is the syntax tree of a scenario node contained in t. The function returns a set containing the syntax trees of the parent node of s and all of the parent's direct and indirect child nodes. Note that s is also contained in the returned set and that the returned set is empty if either s does not describe a scenario node or if s is a root node. A formal definition of the function can be found in Listing E.23.

Function ndScenarioGroupMembers. The function $findScenarioGroupMembers : \Delta \times \mathcal{P}(\Delta) \times \Delta \rightarrow \mathcal{P}(\Delta)$ takes three arguments. The first argument is the syntax tree t of an ADORA model. The second argument m denotes a set of syntax trees which have already been found in a recursive call of the function. This argument is usually the empty set $\emptyset$ for an initial call. The third argument s is the syntax subtree contained in t describing a scenario node. The function returns the set of syntax subtrees representing the scenario nodes which belong to the same scenario group (cf. Section 5.2.4) as the scenario node described by s. Note that s is also contained in the returned set. The function returns m if s does not describe a scenario node. A formal definition of the function can be found in Listing E.24.

Function ndScenarioSubtreeMembers. The function $findScenarioSubtreeMembers : \Delta \times \mathcal{P}(\Delta) \times \Delta \rightarrow \mathcal{P}(\Delta)$ takes three arguments as input. The first argument t is the syntax tree of an ADORA model. The second argument m is a set consisting of syntax trees which describe scenario nodes that have already been found by a recursive call of the function. For an initial call of the function, this argument is usually

Listing E.23: The auxiliary function *findParentAndChildrenScenarios*.



```
1  function findParentAndChildrenScenarios(
2    model, group, element)
3    input
4      model ∈ Δ;
5      group ∈ 𝒫(Δ);
6      element ∈ Δ ∧ element ∉ T;
7    return
8      group ∈ 𝒫(Δ);
9    declare
10     connections ∈ 𝒫(Δ);
11     scenarioTarget ∈ Δ;
12     scenarioParent ∈ Δ;
13     temp ∈ Δ;
14   begin
15     scenarioParent ← scenarioParent(
16       model, element);
17     if (scenarioParent ≠ ε) begin
18       group ← group ∪ { scenarioParent };
19       connections ← connections(
20         scenarioParent);
21       if (connections ≠ ε) begin
22         connections ← filterSet(
23           connections,
24           ScenarioConnectionDefinition);
25         foreach temp ∈ connections begin
26           scenarioTarget = target(
27             model, temp);
28           if (scenarioTarget ∉ group ∧
29             element ≠ scenarioTarget)
30           begin
31             group ←
32               findScenarioSubtreeMembers(
33                 model, group,
34                 scenarioTarget);
35           end
36         end
37       end
38     end
39   end
```

Listing E.24: The auxiliary function *findScenarioGroupMembers*.

```
1  function findScenarioGroupMembers(
2    model, group, element)
3    input
4      model ∈ Δ;
5      group ∈ 𝒫(Δ);
6      element ∈ Δ ∧ element ∉ T;
7    return
8      group ∈ 𝒫(Δ);
9    begin
10     group ←
11       findScenarioSubtreeMembers(
12         model, group, element);
13     group ←
14       findParentAndChildrenScenarios(
15         model, group, element);
16   end
```

Listing E.25: The auxiliary function *findScenarioSubtreeMembers*.

```
1   function findScenarioSubtreeMembers(
2     model, group, element)
3     input
4       model ∈ Δ;
5       group ∈ 𝒫(Δ);
6       element ∈ Δ ∧ element ∉ T;
7     return
8       group ∈ 𝒫(Δ);
9     declare
10      connections ∈ 𝒫(Δ);
11      scenarioConnections ∈ 𝒫(Δ);
12      targetNode ∈ Δ;
13      e ∈ Δ;
14    begin
15      if (type(element) = ScenarioDefinition)
16      begin
17        group ← group ∪ { element };
18        connections ← connections(element);
19        scenarioConnections ← filterSet(
20          connections,
21          ScenarioConnectionDefinition);
22
23        foreach e ∈ scenarioConnections
24        begin
25          targetNode ← target(model, e);
26          if (targetNode ∉ group) begin
27            group ←
28              findScenarioSubtreeMembers(
29                model, group, targetNode);
30          end
31        end
32      end
33    end
```

the empty set $\emptyset$. The third argument s is the syntax tree of a scenario node which is part of t. The function returns a set of syntax trees which are contained in t and which describe the scenario subtree members[10] that have s as *root* node. Note that s is also contained in the result set. The function returns m if s is not the syntax tree of a scenario node. A formal definition of the function can be found in Listing E.25.

Function ndStateGroupMembers. The function $findStateGroupMembers : \Delta \times \mathcal{P}(\Delta) \times (\Delta \setminus T) \rightarrow \mathcal{P}(\Delta)$ takes three arguments. The first argument t denotes the syntax tree of an ADORA model. The second argument m denotes the set of state group members that have already been found. The argument m is used for the recursive call of the function and is usually the empty set $\emptyset$ for an initial call. The third argument s is the syntax tree of a state or a component which is part of a state group (cf. Section 5.2.3). The method returns a set of syntax trees describing states and components which are in the same state group as the element given by s. If s is not a state or a component, the function returns the empty set. A formal definition of the function can be found in Listing E.26.

Function atenTuple. The function $flatenTuple \bigcup_i M^i \times \bigcup_k M^k, \rightarrow \bigcup_l M^l$, where k, i and $l \in \mathbb{N}_0$ and M is an arbitrary set, takes two arguments. The first argument x is a tuple which contains arbitrary elements in an arbitrarily nested tuple structure. It may also contain tuples that are empty. The second argument z is a tuple to which the elements extracted from x are appended. The argument z is used for the recursive call of the function. For an initial call of the function, this argument is usually the empty tuple $\emptyset$. The function returns the attened tuple structure, i.e. all non-tuple elements which are found in the nested tuple structure are appended at the end of z. The function returns z if x does not contain a (nested) tuple structure. A formal definition of the function can be found in Listing E.27.

[10] Scenario subtree means the scenario tree in the model, *not* the syntax tree of the scenarios.

Listing E.26: The function *findStateGroupMembers*.

```
1   function findStateGroupMembers(
2     model, group, element)
3     input
4       model ∈ Δ; group ∈ 𝒫(Δ); element ∈ Δ;
5     return
6       group ∈ 𝒫(Δ);
7     declare
8       transitions ∈ 𝒫(Δ); transitions2 ∈ 𝒫(Δ);
9       connections ∈ 𝒫(Δ);
10      connections2 ∈ 𝒫(Δ);
11      targetNode ∈ Δ; sourceNode ∈ Δ;
12      e1 ∈ Δ; e2 ∈ Δ;
13    begin
14      group ← group ∪ { element };
15      connections ← connections(element);
16      transitions ← filterSet(
17        connections, TransitionDefinition);
18
19      foreach e1 in transitions begin
20        targetNode ← target(model, e1);
21        if (targetNode ∉ group) begin
22          group ← findStateGroupMembers(
23            model, group, targetNode);
24        end
25      end
26
27      connections = targetConnections(
28        model, element);
29      transitions = filterSet(
30        connections, TransitionDefinition);
31
32      foreach e1 in transitions begin
33        sourceNode ← source(model, e1);
34        if (sourceNode ≠ ε ∧
35          sourceNode ∉ group)
36        begin
37          group ← group ∪ { sourceNode };
38          connections2 ←
39            connections(sourceNode);
40          transitions2 ← filterSet(
41            connections2,
42            TransitionDefinition);
43          foreach e2 ∈ transitions2 begin
44            targetNode ← target(model, e2);
45            if (targetNode ∉ group) begin
46              group ←
47                findStateGroupMembers(
48                  model,
49                  group,
50                  targetNode);
51            end
52          end
53        end
54      end
55    end
```

Listing E.27: The function *flatenTuple*.

```
1   function atenTuple(inC, outC)
2     input
3       inC ∈ ∪_k M^k, where k ∈ ℕ_0;
4       outC ∈ ∪_k M^k, where k ∈ ℕ_0;
5     return
6       outC ∈ ∪_k M^k, where k ∈ ℕ_0;
7     declare
8       curr ∈ ∪_k M^k, where k ∈ ℕ_0;
9       i ∈ ℕ_0;
10    begin
11      i ← 0;
12      while (i < arity(inC)) begin
13        if ((π_i(inC) ∉ Δ ∧ π_i(inC) ∈ ∪_k M^k)
14        begin
15          curr ← π_i(inC);
16          outC ← atenTuple(curr, outC);
17        end else if (π_i(inC) ∈ Δ) begin
18          insert_arity(outC)(outC, π_i(inC));
19        end
20        i ← i + 1;
21      end
22    end
```

Listing E.28: The function *functionalSpec*.

```
1   function functionalSpec(e)
2     input
3       e ∈ Δ;
4     return
5       retVal ∈ Δ ∧
6         (type(retVal) = FunctionalSpecification
7         ∨ retVal = ε);
8     begin
9       retVal ← ε;
10      if (type(e) = ComponentDefinition ∨
11        type(e) = AspectDefinition)
12      begin
13        retVal ← childOfType(
14          e, FunctionalSpecification, 1);
15      end
16    end
```

Listing E.29: The function *invariants*.

```
1   function invariants(fs)
2     input
3       fs ∈ Δ;
4     return
5       retVal ∈ ⋃_k Δ^k, where k ∈ ℕ0;
6     begin
7       retVal ← filterFsElements(
8         fs, Invariants, Expression);
9     end
```

Function functionalSpec. The function *functionalSpec* : $\Delta \rightarrow \Delta$ takes one argument s that must be the syntax tree of an aspect module or a component. It returns the syntax subtree describing the functional specification of s. The function returns the empty word ε if no functional specification is defined in s, or s is not the syntax tree of a component or an aspect module. A formal definition of the function can be found in Listing E.28.

Function invariants. The function *invariants* : $\Delta \rightarrow \bigcup_k \Delta^k$, where $k \in \mathbb{N}_0$ takes an argument s which must be a syntax tree of a functional specification. It extracts the syntax trees of the expressions which are part of the invariant in the functional specification. The function returns them in a tuple. The expressions in the tuple occur in the same (infix) order as they are contained in s. The function returns the empty tuple $\emptyset$ if there is no syntax subtree in s describing invariants, or if s does not contain the syntax tree of a functional specification. A formal definition of the function can be found in Listing E.29.

Function operations. The function *operations* : $\Delta \rightarrow \bigcup_k \Delta^k$, where $k \in \mathbb{N}_0$ takes an argument s which describes the syntax tree of a functional specification. It returns the syntax subtrees of all operations that are defined in the functional specification. The result is a tuple and the elements contained in the tuple have the same (infix) order as in s. The function returns the empty tuple $\emptyset$ if s is not the syntax tree of a functional specification, or if there are no operations defined in s. A formal definition of the function can be found in Listing E.30.

Function orderedChildrenOfType. The function *orderedChildrenOfType* $\Delta \times N \times \bigcup_k \Delta^k \rightarrow \bigcup_i \Delta^i$, where $k, i \in \mathbb{N}_0$ takes three arguments. The first argument s is any ADORA syntax tree. The second argument x is an element from the set of non-terminals N which denotes the name of a type. The third

Listing E.30: The function *operations*.

```
1   function operations(fs)
2     input
3       fs ∈ Δ;
4     return
5       retVal ∈ ∪_k Δ^k, where k ∈ ℕ0;
6   begin
7     retVal ←
8       filterOperationsOrProperties(fs,
9         OperationDefinition);
10  end
```

Listing E.31: The function *orderedChildrenOfType*.

```
1   function orderedChildrenOfType
2     (element, type, foundElements)
3     input
4       element ∈ Δ;
5       type ∈ N;
6       foundElements ∈ ∪_k Δ^k, where k ∈ ℕ0;
7     return
8       retVal ∈ ∪_k Δ^k, where k ∈ ℕ0;
9     declare
10      child ∈ Δ;
11      i ∈ ℕ0;
12    begin
13      retVal ← foundElements;
14      i ← 0;
15      while (i < arity(children(element)))
16      begin
17        child ← child_i(element);
18        if (type(child) = type) begin
19          retVal ← insert_arity(retVal)(
20            retVal, child);
21        end else if (child ∉ T) begin
22          retVal ← orderedChildrenOfType(
23            child, type, retVal);
24        end
25        i ← i + 1;
26      end
27    end
```

argument y is the tuple of syntax subtrees which have been already found. This argument is used for the recursive call of the function and is usually the empty tuple $\emptyset$ for an initial call. The function seeks in the given syntax tree s for syntax subtrees of the type x in infix order and adds them at the end of y. The resulting tuple is returned by the function. The argument y is returned if s does not contain any subtrees of type x. A formal definition of the function can be found in Listing E.31.

Function partial. The function $partial : \Delta \rightarrow \{true, false\}$ in Definition E.32 takes a syntax tree $t \in \Delta$ and returns true, if at least one (direct) subelement in the tree of t is the label `partial`, otherwise it returns false. A formal definition of the function can be found in Listing E.32.

Function parts. The function $parts : \Delta \rightarrow \mathcal{P}(\Delta)$ takes an argument s which is a syntax tree of type *ComponentDe nition*, *StateDe nition*, *AspectDe nition* or *Model*. The function returns a set of subtrees

Listing E.32: The function *partial*.

$$partial(t) = \exists k \in \mathbb{N}_0 : (k < arity(children(s)) \land \pi_k = \texttt{partial})$$

Listing E.33: The function *parts*.

```
1   function parts(tree)
2     input
3       tree ∈ Δ;
4     return
5       retVal ∈ 𝒫(Δ);
6     declare
7       type ∈ N; e ∈ Δ; temp ∈ 𝒫(Δ);
8     begin
9       retVal ← ∅; temp ← ∅; type ← ε;
10      if (type(tree) = ComponentDefinition)
11      begin
12        type ← ComponentParts;
13      end else if (type(tree) = AspectDefinition)
14      begin
15        type ← AspectParts;
16      end else if (type(tree) = StateDefinition)
17      begin
18        type ← StateParts;
19      end else if (type(tree) = Model)
20      begin
21        temp ← childrenSetOfType(
22          tree, ComponentDefinition);
23        temp ← retVal ∪ childrenSetOfType(
24          tree, AspectDefinition);
25        temp ← retVal ∪ childrenSetOfType(
26          tree, EnvironmentObjectDefinition);
27      end
28      if (type ≠ ε) begin
29        retVal ← childrenSet(childOfType(
30          tree, type, 1));
31      end
32      foreach e ∈ temp begin
33        if (e ∉ T) begin
34          retVal ← retVal ∪ { e }
35        end
36      end
37    end
```

Listing E.34: The function *provides*.

```
1   function provides(fs)
2     input
3       fs ∈ Δ;
4     return
5       retVal ∈ ∪_k Δ^k, where k ∈ ℕ0;
6     begin
7       retVal ← filterProvidesRequries(
8         fs, Provides, Identifier);
9     end
```

of ADORA elements which are in a part-of relationship with the element described by t. Hence, the return value is the set of syntax trees described by the corresponding *ComponentParts*, *StateParts*, *AspectParts* or the direct children in the case of the *Model* grammar rule given in Section B.3, respectively. If no parts are contained in s, or the argument s is not one of the types above, the empty set is returned. A formal definition of the function can be found in Listings E.33.

Function provides. The function $provides : \Delta \rightarrow \bigcup_k \Delta^k$, where $k \in \mathbb{N}_0$ takes an argument s which is the syntax tree of a functional specification. It extracts all syntax subtrees of the elements defined in the *provides* statement and returns them as a tuple. The elements of the tuple occur in the same order as in s. The function returns the empty tuple $\emptyset$ if s is not the syntax tree of a functional specification or it does not contain a *provide* statement. A formal definition of the function can be found in Listing E.34.

Function requires. The function $requires : \Delta \rightarrow \bigcup_k \Delta^k$, where $k \in \mathbb{N}_0$, takes an argument s which describes the syntax tree of a functional specification. It extracts all syntax subtrees which describe the elements defined in the *requires* statement of the functional specification and returns them as a tuple. The

Listing E.35: The function $requries$.

```
1  function requires(fs)                     6  begin
2    input                                   7    retVal ← filterProvidesRequries(
3      fs ∈ Δ;                               8      fs, Requires, QualifiedIdentifier);
4    return                                  9  end
5      retVal ∈ ∪_k Δ^k, where k ∈ ℕ0;
```

elements of the tuple occur in the same order as in s. The function returns the empty tuple $\emptyset$ if s is not the syntax tree of a functional specification or it does not contain a *requires* statement. A formal definition of the function can be found in Listing E.35.

Function roleChannelNames. The function $roleChannelNames : \Delta \times \bigcup_i \Delta^i \rightarrow \bigcup_k \Delta^k$, where $i, k \in \mathbb{N}_0$ takes two arguments. The first argument s is the syntax tree of any element. The second argument m is a tuple of syntax trees. It is used for the recursion of the function and is usually the empty tuple $\emptyset$ for an initial call. The function seeks for all channel names used in *receive*, *send* and *transform* statements which refer to roles of associations. It returns a tuple of syntax trees of $SpecialIdentifier$s or $QualifiedIdentifier$s representing the names of the referred roles. The order of the elements in the tuple is their infix occurrence in s. The function returns m if there are no role names in s. A formal definition of the function can be found in Listing E.36.

Function roleName. The function $roleName : \Delta \rightarrow T$ takes one argument s which must be the syntax tree of a association role. The function extracts the name of the role and returns it. The empty word ε is returned if s does not contain the syntax tree of a role, or if the role has no name defined. A formal definition of the function can be found in Listing E.37.

Function rootScenarios. The function $rootScenarios : \Delta \times \mathcal{P}(\Delta) \rightarrow \mathcal{P}(\Delta)$ takes two arguments t and m, where t is the syntax tree of the ADORA model and m a set of syntax subtrees of t. The elements of m must be scenario nodes (subtrees of the type *ScenarioDe nition*). The function returns a set of scenario syntax trees which do not have in-coming scenario connections. Hence, the returned set of scenario nodes are the *root nodes* found in m. The elements in m not having the type *ScenarioDe nition* are ignored. If no valid elements are found in m, the empty set $\emptyset$ is returned. A formal definition of the function can be found in Listing E.38.

Function scenarioGroups. The function $scenarioGroups : \Delta \times \mathcal{P}(\Delta) \rightarrow \mathcal{P}(\mathcal{P}(\Delta))$ takes the parse tree of an ADORA model t and a set m of syntax subtrees contained in t. The syntax trees contained in m must describe scenario nodes, i.e., they must have the type *ScenarioDe nition*. Elements which do not have this type are ignored. The function takes all elements in m and computes the set of all scenario groups, where a scenario group is the set of all connected scenario nodes (cf. Section 5.2.4). There may be groups in the result set containing only one element, which is the case if a scenario node does not have any in-coming and out-going connections. If no elements with the type *ScenarioDe nition* are contained in m, the empty set is returned. A formal definition of the function can be found in Listing E.39.

Listing E.36: The function *roleChannelNames*.

```
1   function roleChannelNames
2     (model, foundNames)
3     input
4       model ∈ Δ;
5       foundNames ∈ ∪_k Δ^k, where k ∈ ℕ0;
6     return
7       retVal ∈ ∪_k Δ^k, where k ∈ ℕ0;
8     declare
9       child ∈ Δ;
10      temp ∈ Δ;
11      i ∈ ℕ0;
12    begin
13      retVal ← foundNames;
14      i ← 0;
15      if (type(model) = MessageSend ∨
16        type(model) = MessageReceive) begin
17        child ← childOfType(
18          model, AssociationRoleNameList, 1);
19        if (child ≠ ε ∧ child ∉ T)
20        begin
21          while (i < arity(children(child)))
22          begin
23            temp ← child_i(child);
24            if (type(temp) = SpecialIdentifier)
25            begin
26              retVal ←
27                insert_arity(retVal)(retVal, temp);
28            end
29            i ← i + 1;
30          end
31        end
32      end else
33      if (type(model) = TransformInput ∨
34          type(model) = TransformOutput)
35        begin
36        child ← childOfType(
37          model, RoleName, 1);
38        if (child ≠ ε) begin
39          temp ← childOfType(
40            child, SpecialIdentifier, 1);
41          retVal ←
42            insert_arity(retVal)(retVal, temp);
43        end else begin
44          temp ← childOfType(
45            model, QualifiedIdentifier, 1);
46          if (temp ≠ ε) begin
47            retVal ←
48              insert_arity(retVal)(retVal, temp);
49          end
50        end
51      end else begin
52        while (i < arity(children(model)))
53        begin
54          if (child_i(model) ∉ T) begin
55            retVal ← roleChannelNames(
56              child_i(model), retVal);
57          end
58          i ← i + 1;
59        end
60      end
61    end
```

Listing E.37: The function *roleName*.

```
1   function roleName(role)
2     input
3       role ∈ Δ;
4     return
5       roleName ∈ Δ;
6     declare
7       temp ∈ Δ;
8     begin
9       if (type(role) = Role) begin
10        temp ← childOfType(
11          role, RoleName, 1);
12        if (temp ≠ ε) begin
13          temp ← childOfType(
14            temp, SpecialIdentifier, 1);
15        end
16        roleName = temp;
17      end
18    end
```

Listing E.38: The function *rootScenarios*.

```
1   function rootScenarios(model,
2     scenarioNodes)
3     input
4       model ∈ Δ ∧ model ∉ T;
5       scenarioNodes ∈ 𝒫(Δ);
6     return
7       retVal ∈ 𝒫(Δ);
8     declare
9       nonRootNodes ∈ 𝒫(Δ);
10      connections ∈ Δ; scenarioConn ∈ 𝒫(Δ);
11      temp ∈ Δ; scenario ∈ Δ;
12    begin
13      nonRootNodes ← ∅; retVal ← ∅;
14      foreach (scenario ∈ scenarioNodes)
15      begin
16        if (type(scenario) = ScenarioDefinition)
17        begin
18          scenarioConn ← ∅;
19          connections ← childOfType(
20            scenario, ScenarioConnections, 1);
21          if (connections ≠ ε) begin
22            scenarioConn ←
23              childrenSetOfType(
24                connections,
25                  ScenarioConnectionDefinition);
26          end
27
28          foreach (temp in scenarioConn)
29          begin
30            temp ← childOfType(
31              temp, ElementReference, 1);
32            temp ← find(
33              model, elementReference(temp));
34            if (temp ≠ ε)
35            begin
36              nonRootNodes ←
37                nonRootNodes ∪ { temp };
38            end
39          end
40        end else begin
41          nonRootNodes ←
42            nonRootNode ∪ { scenario };
43        end
44      end
45      retVal ←
46        scenarioNodes \ nonRootNodes;
47    end
```

Listing E.39: The function *scenarioGroups*.

```
1   function scenarioGroups
2     (tree, scenarioElements)
3     input
4       tree ∈ Δ \ T;
5       scenarioElements ∈ 𝒫(Δ);
6     return
7       groups ∈ 𝒫(𝒫(Δ));
8     declare
9       temp ∈ 𝒫(Δ);
10      found ∈ { true, false };
11      element ∈ Δ;
12      v ∈ 𝒫(Δ);
13      s ∈ Δ;
14    begin
15      groups ← ∅;
16      foreach element ∈ scenarioElements
17      begin
18        found ← false;
19        if (element ∉ T ∧
20          type(element) = ScenarioDefinition)
21        begin
22          foreach v ∈ groups begin
23            foreach s ∈ v begin
24              if (s = element) begin
25                found ← true;
26              end
27            end
28          end
29          if (¬ found) begin
30            temp ←
31              findScenarioGroupMembers(
32                tree, ∅, element);
33            groups ← groups ∪ { temp };
34          end
35        end
36      end
37    end
```

Listing E.40: The function $scenarioParent$.

```
1   function scenarioParent(model, scenario)
2     input
3       model ∈ Δ;
4       scenario ∈ Δ;
5     return
6       retVal ∈ Δ ∪ { ε };
7     declare
8       connections ∈ P(Δ);
9       connection ∈ Δ;
10    begin
11      retVal ← ε;
12      if (type(scenario) = ScenarioDefinition)
13      begin
14        connections ← filterSet(
15          targetConnections(model, scenario),
16          ScenarioConnectionDefinition);
17        if (| connections | = 1) begin
18          connection ∈ connections;
19          retVal ← source(model, connection);
20        end
21      end
22    end
```

Function scenarioParent. The function $scenarioParent : \Delta \times \Delta \rightarrow \Delta$ takes two arguments. The first argument t is the syntax tree of an ADORA model. The second argument s is a syntax subtree of t which describes a scenario node. The function returns the syntax tree of the parent scenario, i.e., the parent node in the scenario tree which is connected by a scenario connection to scenario node described by s. The function returns the empty word ε if s does not describe a scenario node or if s specifies the root node of a scenario tree. A formal definition of the function can be found in Listing E.40.

Function scenarioSiblings. The function $scenarioSiblings : \Delta \times \Delta \rightarrow \bigcup_k \Delta^k$, where $k \in \mathbb{N}_0$, has two arguments. The first argument t is the syntax tree of an ADORA model. The second argument s is a syntax subtree of t that describes a scenario node. The function returns a tuple of syntax trees containing the siblings of the given scenario s. Note that the returned tuple also contains s. The function returns the empty tuple $\emptyset$ if s is not a scenario. If s is not part of t, the function also returns the empty tuple. A formal definition of the function can be found in Listing E.41.

Function scenarioType. The function $scenarioType : \Delta \rightarrow T$ takes one argument s, which must be the syntax tree of a scenario node. The function extracts the type of the scenario node and returns the corresponding token **`sequence`**, **`parallel`**, **`alternative`**, or **`root`**. The function returns the empty word ε if s is not the syntax tree of a scenario node. A formal definition of this function can be found in Listing E.42.

Function seekTargetConnections. The auxiliary function $seekTargetConnections : \Delta \times \Delta \times \mathcal{P}(\Delta) \rightarrow \mathcal{P}(\Delta)$ takes three arguments t, s, and m. The first argument t is the syntax tree of an ADORA model, s is a syntax tree that is part of t. Finally, m is a set of connections that have already been found. It is used for recursive descent and is usually the empty set $\emptyset$ for an initial call of the function. The function determines all connections in the given model t which have the element s as target element. It returns a set of the syntax trees of the found elements. The function returns the empty set $\emptyset$ if s is not a subtree of t, or if s is not the target of any connection. A formal definition of the function can be found in Listing E.43.

Listing E.41: The function *scenarioSiblings*.

```
1   function scenarioSiblings(model, scenario)
2   input
3     model ∈ Δ;
4     scenario ∈ Δ;
5   return
6     retVal ∈ ⋃_k Δ^k, where k ∈ ℕ_0;
7   declare
8     scenarioParent ∈ Δ;
9     connection ∈ Δ;
10    target ∈ Δ;
11  begin
12    retVal ← ∅;
13    scenarioParent ← scenarioParent(
14      model, scenario);
15    if (scenarioParent ≠ ε) begin
16      connections ←
17        filterSet(
18          connections(scenarioParent),
19          ScenarioConnection);
20      foreach connection ∈ connections
21      begin
22        if (connection ∉ T) begin
23          target ← target(model, connection);
24          insert_arity(retVal)(retVal, target);
25        end
26      end
27    end
28  end
```

Listing E.42: The function *scenarioType*.

```
1   function scenarioType(scenario)
2     input
3       scenario ∈ Δ;
4     return
5       retVal ∈ T;
6     declare
7       typetree ∈ Δ;
8     begin
9       retVal ← ε;
10      if (type(scenario) = ScenarioDefinition)
11      begin
12        typetree ← childOfType(
13          scenario, ScenarioType, 1);
14        if (typetree ≠ ε) begin
15          retVal ← child_0(typetree);
16        end
17      end
18    end
```

Listing E.43: The function $seekTargetConnections$.

```
1   function seekTargetConnection
2   (model, element, foundConn)
3   input
4     model ∈ Δ;
5     element ∈ Δ;
6     foundConn ∈ P(Δ);
7   return
8     foundConn ∈ P(Δ) ∧
9       ∀ e ∈ foundConn : type(e) ∈ {
10        TransitionDefinition,
11        AssociationDefinition,
12        ScenarioConnectionDefintion,
13        JoinRelationshipDefinition };
14  declare
15    i ∈ ℕ0;
16    occ ∈ ℕ;
17    ch ∈ Δ;
18    elementRef ∈Δ;
19    id ∈ T;
20    refId ∈ T;
21  begin
22    i ← 0;
23    id ← uniqueElementIdentifier(
24      childOfType(element,
25      UniqueModelElementIdentifier, 1));
26    refId ← ε;
27    while (i < arity(children(model))) begin
28      ch ← child_i(model);
29        if (type(ch) ∈ { TransitionDefinition,
30          AssociationDefinition,
31          ScenarioConnectionsDefinition,
32          JoinRelationshipDefinition } begin
33          if (type(ch) =
34            JoinRelationshipDefinition)
35          begin
36            occ ← 2;
37          end else begin
38            occ ← 1;
39          end
40          elementRef ← childOfType(
41            ch, ElementReference, occ);
42          refId = elementReference(
43            elementRef);
44          if (refId = id) begin
45            foundConn ← foundConn ∪ { ch };
46          end
47        end else if (ch ∉ T) begin
48          foundConn ←
49            seekTargetConnections(
50              element, ch, foundConn);
51        end
52        i ← i + 1;
53      end
54    end
```

Function seqNo. The function $seqNo : \Delta \rightarrow \mathbb{N}_0$ takes a syntax tree as argument s. It must describe a scenario node (type *ScenarioDefinition*). If the scenario has the scenario type *sequence*, the function returns a sequence number which is greater than zero. The function returns 0, if the given scenario node has not the type *sequence* or if s is not the syntax tree of a scenario. A formal definition of the function can be found in Listing E.44.

Function source. The function $source : \Delta \times \Delta \rightarrow \Delta \cup \{\varepsilon\}$ takes the two arguments t and s. The tree t is a syntax tree of an ADORA model that contains a connection described by s. Thus, s has the type *TransitionDe nition*, *AssociationDe nition*, *ScenarioDe nition*, or a *JoinRelationshipDe nition* and is a subtree of t. The function returns the source of s. Thus it returns the syntax tree of the element which the connection s is embedded in. If s does not describe a connection or is not contained in t, the empty word ε is returned. For example, a function call with the syntax tree of the model in Fig. E.2 and the syntax subtree of the association identified by *1.a.1* returns the syntax tree of the abstract object *A*, as it is the source of this association. A formal definition of the function can be found in Listing E.45.

Listing E.44: The function $seqNo$.

```
1   function seqNo(scenario)
2     input
3       scenario ∈ Δ;
4     return
5       retVal ∈ ℕ0;
6     declare
7       typetree ∈ Δ;
8       f : T → ℕ =
9         { (1,1), (2,2), (3,3), . . . , };
10    begin
11      retVal ← 0;
12      if (scenarioType(scenario) =
13        sequence)
14      begin
15        typetree ← childOfType(
16          scenario, ScenarioType, 1);
17        retVal ← f(child1(typetree));
18      end
19    end
```

Function sourceRole. The function $sourceRole : \Delta \to \Delta$ takes one argument t which must be a syntax tree of an association. The function extracts the subtree of the source role. If no source role is defined or t is not the syntax tree of an association, the empty word ε is returned. A formal definition of the function can be found in Listing E.46.

Function specialIdenti er. The function $specialIdentifier : \Delta \to T$ takes one argument s. If s is a syntax tree representing a special identifier, the function returns the token of the special identifier. If s is not the tree of a special identifier, the empty word ε is returned. A formal definition of the function can be found in Listing E.47.

Function standardizedProperties. The function $standardizedProperties : \Delta \to \bigcup_k \Delta^k$, where $k \in \mathbb{N}_0$ takes an argument s which describes the syntax tree of a functional specification. The function gathers the syntax subtrees of s which define standardized properties. They are returned as tuple, and the contained elements are ordered according to their occurrence in s. The function returns the empty tuple $\emptyset$ if s is not the syntax tree of a functional specification or there are no standardized properties in s. A formal definition of the function can be found in Listing E.48.

Function startStates. The function $startStates : \mathcal{P}(\Delta) \to \mathcal{P}(\Delta)$ takes a set m of syntax trees and returns the subset of m of all syntax trees of the type *ComponentDe nition* or *StateDe nition* with a start state indicator. The function returns the empty set if no component or state with a start state indicator is found in m. A formal definition of the function can be found in Listing E.49.

Function stateGroups. The function $stateGroups : \Delta \times \mathcal{P}(\Delta) \to \mathcal{P}(\mathcal{P}(\Delta))$ takes the syntax tree t of an ADORA model[11] and a set m of syntax trees containing the nodes of a behavior description. The elements in m should have the type *ComponentDe nition*, *ExitPointDe nition*, or *StateDe nition*. The function takes all elements of m and computes the set of all state groups which can be formed of the given set of nodes. There may be state groups in the result which consist only of one element, which is the case if a node has no in-coming and no out-going transitions. However, an exception are nodes of the type *ComponentDe nition*. Elements of these types which have no incoming and no outgoing transitions do

[11] More precisely, it can be any subtree of an ADORA model which contains all the behavior elements whose stategroups should be returned.

Listing E.45: The function *source*.

```
1  function source(model, conn)
2    input
3      model ∈ Δ ∧ model ∉ T;
4      conn ∈ Δ ∧ conn ∉ T;
5    return
6      retVal ∈ (Δ ∪ { ε });
7    declare
8      r ∈ Δ; map ⊆ (N × ℕ);
9    begin
10     retVal ← ε;
11     if (type(conn) = TransitionDefinition)
12     begin
13       map ←
14         {(ComponentDefintion, 2),
15         (StateDefinition, 2) };
16     end
17     if (type(conn) = AssociationDefinition)
18     begin
19       retVal ←
20         {(EnvironmentObjectDefinition, 2),
21         (ComponentDefintion, 2),
22         (AspectDefinition, 2),
23         (ScenarioDefinition, 2)};
24     end
25     if (type(conn) =
26       ScenarioConnectionDefinition)
27     begin
28       map ← {(ScenarioDefinition, 2) };
29     end
30     if (map ≠ ∅) begin
31       retVal ← treeAncestor(
32         model, conn, map);
33     end
34     if (type(conn) =
35       JoinRelationshipDefinition)
36     begin
37       r ← childOfType(
38         conn, "ElementReference", 1);
39       if (r ≠ ε) begin
40         retVal ← find(
41           model, elementReference(r));
42       end
43     end
44   end
```

not form a state group, i.e., they are not members of a one-element stategroup in the result. They are rather so-called embedded components (cf. Section 7.9). Furthermore, if m does not contain any element of the types mentioned above, the empty set $\emptyset$ is returned. A formal definition of the function can be found in Listing E.50.

Function target. The function $target : \Delta \times \Delta \to \Delta \cup \{\varepsilon\}$ takes two arguments t and s. The argument t is the syntax tree of an ADORA model and s the syntax tree of a connection. The connection must be part of the ADORA model represented by t. The function returns the syntax tree of the connection's target if (i) s has either the type *TransitionDe nition*, *AssociationDe nition*, *ScenarioConnectionDe nition*, or *JoinRelationshipDe nition*, and (ii) if s is contained in t, and (iii) the referenced target element exists in t.

Listing E.46: The function *sourceRole*.

```
1  function sourceRole(association)
2    input
3      association ∈ Δ;
4    return
5      retVal ∈ Δ ∪ { ε };
6    begin
7      retVal ← ε;
8      if (type(association) =
9        AssociationDefinition) begin
10       retVal ← childOfType(
11         association, Role, 1);
12     end
13   end
```

Listing E.47: The function *specialIdentifier*.

```
1   function specialIdentifier(tree)
2     input
3       tree ∈ Δ;
4     return
5       retVal ∈ uniqueIdentifier ∈
6         <REFERENCE_STRING_LITERAL>
7         ∪ <IDENTIFIER>;
8     declare
9       temp ∈ Δ;
10    begin
11      if (type(tree) = SpecialIdentifier) begin
12        temp ← child_0(tree);
13        if (type(temp) = Identifier) begin
14          retVal ← child_0(temp);
15        end else begin
16          retVal ← temp;
17        end
18      end
19    end
```

Listing E.48: The function *standardizedProperties*.

```
1   function standardizedProperties(fs)
2     input
3       fs ∈ Δ;
4     return
5       retVal ∈ ∪_k Δ^k, where k ∈ ℕ_0;
6     begin
7       retVal ←
8         filterOperationsOrProperties(fs,
9           Property);
10    end
```

Listing E.49: The function *startStates*.

```
1   function startStates(states)
2     input
3       states ∈ 𝒫(Δ);
4     return
5       retVal ∈ 𝒫(Δ);
6     declare
7       element ∈ Δ;
8     begin
9       retVal ← ∅;
10      foreach element ∈ states begin
11        if (element ∉ T ∧
12          childOfType(
13            element, start, 1) ≠ ε)
14        begin
15          retVal ← retVal ∪ { element };
16        end
17      end
18    end
```

Listing E.50: The function $stateGroups$.

```
1   function stateGroups(model,
2     statechartElements)
3     input
4       model ∈ Δ ∧ tree ∉ T;
5       statechartElements ∈ 𝒫(Δ);
6     return
7       groups ∈ 𝒫(𝒫(Δ));
8     declare
9       temp ∈ 𝒫(Δ);
10      found ∈ { true, false };
11      e ∈ Δ;
12      m ∈ 𝒫(Δ);
13      s ∈ Δ;
14    begin
15      groups ← ∅;
16      foreach e ∈ statechartElements begin
17        if (e ∉ T ∧ (type(e) = StateDefinition ∨
18          type(e) = ComponentDefinition ∨
19          type(e) = ExitPointDefinition)) begin
20          found ← false;
21          foreach m ∈ groups begin
22            foreach s ∈ m begin
23              if (s = e) begin
24                found ← true;
25              end
26            end
27          end
28          if (¬ found) begin
29            temp ← findStateGroupMembers(
30              model, ∅, e);
31            groups ← groups ∪ { temp };
32          end
33        end
34      end
35    end
```

If any one of these three conditions is not satisfied, the empty element ε is returned. For example, calling the function with the syntax tree of the model in Fig. E.2 and syntax subtree of the association identified by *1.a.1* , it returns the syntax tree of the abstract object A, as it is the target of this association. A formal definition of the function can be found in Listing E.51.

Function targetConnections. The function $targetConnections : \Delta \times \Delta \rightarrow \mathcal{P}(\Delta)$ takes two arguments. The first argument t is the syntax tree of an ADORA model. The second argument s should be a subtree of t and be identifiable with a unique model element identifier. The function returns the set of syntax trees describing the connections, i.e., transitions, associations, scenario connections, and join relationships which have s as target element. The function returns the empty set $\emptyset$ if s is not the target of any connection or it is not part of t. A formal definition of the function can be found in Listing E.52.

Function targetRole. The function $targetRole : \Delta \times \Delta \rightarrow \Delta$ takes two arguments t and s. The argument t must be the syntax tree of an ADORA model and s must be the syntax tree of an association. The function extracts the subtree of the target role of the association represented by s. If no target role is defined, or s is not a subtree of t, or s is not the syntax tree of an association, the empty word ε is returned. A formal definition of the function can be found in Listing E.53.

Function treeAncestor. The auxiliary function $treeAncestor : \Delta \times \Delta(N \times \mathbb{N})$ takes three arguments t, s, and m. The argument t is the syntax tree of an ADORA model, s any subtree of t, and m is a set which defines the properties of a valid relationship between the given s and a given t. Concretely, m contains pairs of non-terminals denoting the type of t, as well as a natural number which specifies the tree distance[12] between the root node of t and s. The function seeks for the (sub)tree in t which contains s and

[12]This is done in terms of the function *distance* (cf. p. 319).

Listing E.51: The function *target*.

```
1  function target(model, conn)
2    input
3      model ∈ Δ ∧ model ∉ T;
4      conn ∈ Δ ∧ conn ∉ T;
5    return
6      retVal ∈ (Δ ∪ { ε });
7    declare
8      occurAt ∈ ℤ;
9      e ∈ Δ;
10   begin
11     retVal ← ε;
12     occurAt ← -1;
13
14     if (type(conn) in
15       { TransitionDefinition,
16       AssociationDefinition,
17       ScenarioConnectionDefinition }
18     begin
19       occurAt ← 1;
20     end
21
22     if (type(conn) =
23       JoinRelationshipDefinition) begin
24       occurAt ← 2;
25     end
26
27     if (occurAt ≠ -1) begin
28       e ← childOfType(conn,
29         ElementReference, occurAt);
30       if (e ≠ ε) begin
31         retVal ← find(
32           model, elementReference(e));
33       end
34     end
35   end
```

Listing E.52: The function *targetConnections*.

```
1    function targetConnections(model,
2    element)
3    input
4      model ∈ Δ;
5      element ∈ Δ;
6    return
7      retVal ∈ 𝒫(Δ);
8    begin
9      retVal ← ∅;
10     retVal ← seekTargetConnections(
11       model, element, retVal);
12   end
```

Listing E.53: The function *targetRole*.

```
1   function targetRole(model, association)
2     input
3       model ∈ Δ;
4       association ∈ Δ;
5     return
6       retVal ∈ Δ ∪ { ε };
7     declare
8       temp ∈ Δ;
9       assocTargetRoles ∈ P(Δ);
10      a ∈ Δ;
11      uid ∈ Δ;
12    begin
13      retVal ← ε;
14      if (type(association) =
15        AssociationDefinition)
16      begin
17        uid ← childOfType(association,
18          UniqueModelElementIdentifier, 1);
19        temp ← target(model, association);
20        assocTargetRoles ← filterSet(
21          connections(temp),
22          AssociationRoleDefinition);
23        foreach a ∈ assocTargetRoles begin
24          temp ← childOfType(
25            a, ElementReference, 1);
26          if (elementReference(temp) =
27            uniqueElementIdentifier(uid))
28          begin
29            retVal ← childOfType(a, Role, 1);
30          end
31        end
32      end
33    end
```

whose root node is one of the types given by the relationship m and which has the specified distance to s. A formal definition of the function can be found in Listing E.54.

Function uniqueElementIdenti er. The function $uniqueElementIdentifier : \Delta \rightarrow T$ takes one argument t, where t should be the syntax tree of a unique model element identifier. It extracts the unique model element identifier terminal symbol and returns it. The function returns the empty word ε, if t is not the syntax tree of a unique model element identifier. A formal definition of the function can be found in Listing E.55.

E.5 Aspect Speci c Functions

Function containedAspects. The function $containedAspects : \Delta \times \mathcal{P}(\Delta) \rightarrow \mathcal{P}(\Delta)$ has two arguments t and m. The first argument s is the syntax tree of an ADORA model or a part of a model. The argument m is a set of syntax trees, which is used for the recursive descent of the function. It is usually the empty set $\emptyset$ when being used by an external caller. The function takes the syntax tree s and seeks for the subtrees which are aspect modules, i.e., which have the type *AspectDe nition*. If such a subtree is found, it is added to the content of m. The method returns in the end all syntax trees which are aspect modules that are contained in s. The function returns the empty set $\emptyset$ if s does not contain any aspect module. A formal definition of the function can be found in Listing E.56.

Function enclosingAspects. The function $enclosingAspects : \Delta \times \Delta \times \mathcal{P}(\Delta) \rightarrow \mathcal{P}(\Delta)$ takes three arguments t, s, and m. The argument t is the syntax tree of an ADORA model, s is the subtree of any model element contained in t. Finally, the argument m is a set of calculated elements which is used for the recursive computation of the function. It should be the empty set $\emptyset$ for the initial function call. The method returns all direct and indirect decomposition parents of s which are of the type $AspectDefinition$.

Listing E.54: The function *treeAncestor*.

```
1  function treeAncestor(model, subtree, map)
2    input
3      model ∈ Δ ∧ model ∉ T;
4      subtree ∈ Δ ∧ subtree ∉ T;
5      map ⊆ (N × ℕ);
6    return
7      retVal ∈ (Δ ∪ { ε });
8    declare
9      level ∈ ℤ; i ∈ ℕ0;
10     ch ∈ ∪_k Δ^k, where k ∈ ℕ0;
11     e ∈ Δ;
12     compare ∈ {true,false};
13     m ∈ (N × ℕ);
14   begin
15     level ← 1; retVal ← ε;
16     compare ← false;
17     if (map = ∅) compare ← true;
18     for each m ∈ map begin
19       if (π0(m) = type(model)) begin
20         level ← π1(m);
21         compare ← true;
22       end
23     end
24     if (compare) begin
25       if (distance(model, subtree) = level)
26       begin
27         retVal ← model;
28       end
29     end
30     if (retVal = ε) begin
31       ch ← children(model);
32       i ← 0;
33       while (i < arity(ch) ∧ retVal = ε) begin
34         e ← child_i(model);
35         if (e ∉ T) begin
36           retVal ←
37             treeAncestor(e, subtree, map);
38         end
39         i ← i + 1;
40       end
41     end
42   end
```

Listing E.55: The function *uniqueElementIdentifier*.

```
1  function uniqueElementIdentifier(tree)
2    input
3      tree ∈ Δ;
4    return
5      retVal ∈
6        <REFERENCE_STRING_LITERAL>
7        ∪ <IDENTIFIER>;
8    begin
9      retVal ← ε;
10     if (type(tree) =
11       UniqueModelElementIdentifier) begin
12       retVal ← specialIdentifier(child0(tree));
13     end
14   end
```

Listing E.56: The function *containedAspects*.

```
1  function containedAspects(model, contained)
2    input
3      model ∈ Δ \ T;
4      contained ∈ P(Δ);
5    return
6      contained ∈ P(Δ);
7    declare
8      children ∈ ∪_k Δ^k, where k ∈ ℕ0;
9      element ∈ Δ;
10     i ∈ ℕ0;
11   begin
12     i← 0;
13     children ← children(model);
14     while (i < arity(children)) begin
15       element ← child_i(model);
16       if (type(element) = AspectDefinition)
17       begin
18         contained ← contained ∪
19           { element };
20       end
21       if (element notIn T)
22       begin
23         contained ← containedAspects(
24           element, contained);
25       end
26       i ← i + 1;
27     end
28   end
```

Hence, the function returns the set of all aspect modules which enclose the element s. If s is not uniquely identifiable in t, i.e., if the element in s has no unique element identifier, the enclosing aspects of the first element matching s are derived. If no aspect module encloses the element defined by s, the value of parameter m is returned (i.e., the empty set). A formal definition of the function can be found in Listing E.57.

Function endTargetModules. The function $endTargetModules\ \Delta \times \Delta \times \bigcup_k \Delta^k \rightarrow \bigcup_i \Delta^i$, where $k, i \in \mathbb{N}_0$ determines the end target modules of an aspect. Its algorithmic specification is formally given in Section E.5. The function takes three arguments. The first argument t is the syntax tree of an ADORA model. The second argument s is the syntax subtree of t describing the aspect module for which the end target modules are searched. The third argument z is a tuple used to store the resulting elements found during the recursive call of the function. It is usually the empty tuple $\emptyset$ for an initial call. The function seeks for all syntax trees defining the *end target modules* in t with respect to the given aspect s. The found elements are appended to z and returned. As the final model only contains conventional language elements after the weaving, an end target module is only contained in the resulting tuple if it is a component.[13] Even though the given aspect s may have various join relationship paths to the same end target module, the target module is only contained once in the resulting tuple.[14] The function returns the value of z, if s is not an aspect or if the model does not contain join relationships. A formal definition of the function can be found in Listing E.58.

Function exitPoints. The function $exitPoints : \mathcal{P}(\Delta) \rightarrow \mathcal{P}(\Delta)$ takes the argument m which is a set of syntax trees. It returns a subset of m containing all syntax trees which describe an exit point, i.e., which have the type *ExitPointDe nition*. If m does not contain exit points, the empty set is returned. A formal definition of the function can be found in Listing E.59.

[13] However, there may be end target modules that are aspects. Nevertheless, they are irrelevant for the final model and therefore they are omitted by the $endTargetModules$ function.

[14] The first time the end target module is found, it is appended to the result tuple.

Listing E.57: The function $enclosingAspects$.

```
1   function enclosingAspects
2     (model, element, aspects)
3     input
4       model ∈ Δ \ T;
5       element ∈ Δ;
6       aspects ∈ P(Δ);
7     return
8       retVal ∈ P(Δ) ∧
9         ∀ x ∈ retVal :
10          type(x) = AspectDefinition;
11    declare
12      child ∈ Δ;
13      i ∈ ℕ0;
14    begin
15      if (type(model) = AspectDefinition) begin
16        if (containsElement(model, element))
17          begin
18            aspects ← aspect ∪ { model };
19          end
20      end
21      i ← 0;
22      while (i < arity(children(model))) begin
23        child = child_i(model);
24        if (child ∉ T) begin
25          aspects ← enclosingAspects(
26            child, element, aspects);
27        end
28        i ← i + 1;
29      end
30      retVal = aspects;
31  end
```

Function ndEoJrs. The function $findEoJrs : \Delta \times \Delta \rightarrow \bigcup_k \Delta^k$, where $k \in \mathbb{N}_0$, has two arguments. The first argument t is the syntax tree of an ADORA model. The second argument s is the syntax tree of an association (type $AssociationDefinition$). It must be a subtree of t. The function seeks for an environment object which is either the source or the target node of the given association. If an environment object is found, its *outgoing* join relationships are extracted and returned as a tuple. The function returns the empty tuple $\emptyset$ if s is not part of t, or s is not an association, or the found environment object has no outgoing join relationships. A formal definition of the function can be found in Listing E.60.

Function ndJoinRelationshipCycle. The auxiliary function $findJoinRelationshipCycle : \Delta \times \Delta \times \Delta \rightarrow \{true, false\}$ takes three arguments t, s, and a. Argument t is a syntax tree of an ADORA model, s a syntax tree of a join relationship's target, and a is the start aspect, where the join relationship is hosted. The function returns whether the source aspect and the target are reachable over a cyclic join relationship structure. This function is actually a helper for the function $hasJoinRelationshipCycles$. The actual criteria which describe when a join relationship cycle results can be found in the description in the $hasJoinRelationshipCycles$ function at page 348. A formal definition of the function is given in Listing E.61.

Function gatherAspects. The function $gatherAspects\ \Delta \times \bigcup_i \Delta^i \rightarrow \bigcup_k \Delta^k$, where $k, i \in \mathbb{N}_0$ takes two arguments. The function collects the syntax subtrees of all aspect modules found in the model and returns them as a tuple. The first argument t must be the syntax tree of an ADORA model whose aspect modules should be extracted. The second argument x is a tuple of syntax trees. Found aspects are appended to x. The argument x is used for the recursive descent of the function and is usually the empty tuple $\emptyset$ for an initial call. The order of the elements in the tuple denotes the order in which the aspects are found during the gathering when visiting t in infix order. The argument x is returned if no aspect modules are found. A formal definition of the function can be found in Listing E.62.

Listing E.58: The function $endTargetModules$.

```
1  function endTargetModules(
2    model, aspect, foundModules)
3    input
4      model ∈ Δ;
5      aspect ∈ Δ;
6      foundModules ∈ ∪_k Δ^k, where k ∈ ℕ_0;
7    return
8      foundModules ∈ ∪_k Δ^k, where k ∈ ℕ_0;
9    declare
10     ch ∈ ∪_k M^k, where k ∈ ℕ_0;
11     temp ∈ Δ; e ∈ Δ; i ∈ ℕ_0; j ∈ ℕ_0;
12     found ∈ { true, false };
13   begin
14     i ← 0; found ← false;
15     if (type(aspect) = AspectDefinition) begin
16       temp ← childOfType(
17         aspect, AspectConnections, 1);
18       if (temp ≠ ε) begin
19         ch ← children(temp);
20         while (i < arity(ch)) begin
21           e ← π_i(ch);
22           if (type(e) =
23             JoinRelationshipDefinition)
24           begin
25             temp ← targetModule(
26               model, target(model, e));
27             if (temp ≠ ε ∧ type(temp) =
28               AspectDefinition)
29             begin
30               foundModules ←
31                 endTargetModules(
32                   model,
33                   temp,
34                   foundModules);
35             end else if (temp ≠ ε ∧ type(
36               temp) = ComponentDefinition)
37             begin
38               j ← 0; found ← false;
39               while
40                 (j < artiy(foundModules))
41               begin
42                 if (π_j(foundModules) =
43                   temp)
44                 begin
45                   found = true;
46                 end
47                 j ← j + 1;
48               end
49               if (¬ found) begin
50                 foundModules ←
51                   insert_arity(foundModules)(
52                     foundModules, temp);
53               end
54             end
55           end
56           i ← i + 1;
57         end
58       end
59     end
60 end
```

Listing E.59: The function $exitPoints$.

```
1  function exitPoints(states)
2    input
3      stateGroupElements ∈ 𝒫(Δ);
4    return
5      retVal ∈ 𝒫(Δ);
6    declare
7      element ∈ Δ;
8    begin
9      retVal ← ∅;
10     foreach element ∈ states begin
11       if (element ∉ T ∧
12         type(element) = ExitPointDefinition)
13       begin
14         retVal ← retVal ∪ { element };
15       end
16     end
17 end
```

Listing E.60: The function *findEoJrs*.

```
1   function findEoJrs(model, association)
2     input
3       model ∈ Δ;
4       associations ∈ Δ;
5     return
6       retVal ∈ ∪_k Δ^k, where k ∈ ℕ_0;
7     declare
8       eo ∈ Δ;
9       j ∈ Δ;
10      jrs ∈ 𝒫(Δ);
11    begin
12      retVal ← ∅;
13      if (type(association) =
14        AssociationDefinition)
15      begin
16        eo ← ε;
17        if (type(source(model, association)) =
18          EnvironmentObjectDefinition)
19        begin
20          eo ← source(model, association);
21        end else if (type(target(
22          model, association)) =
23          EnvironmentObjectDefinition)
24        begin
25          eo ← target(model, association);
26        end
27        if (eo ≠ ε) begin
28          jrs ← filterSet(
29            connections(eo),
30            JoinRelationshipDefinition);
31          foreach j ∈ jrs begin
32            insert_arity(retVal)(retVal, j);
33          end
34        end
35      end
36    end
37
```

Function hasJoinRelationshipCycles. The function $hasJoinRelationshipCycles : \Delta \times \Delta \rightarrow \{true, false\}$ takes two arguments t and s, where t is a syntax tree of an ADORA model and s is a syntax tree of a join relationship contained in t. This function is needed since aspects may crosscut aspects. Hence cycles of crosscutting join relationships may occur, but such cycles are not meaningful. The function returns *true* if the given join relationship connects its source and target in a way that a cyclic crosscutting relationship is created. A cycle occurs, if an aspect A or one of A's descendants or ancestor aspects[15] is connected directly or transitively with an aspect B or one of B's descendant or ancestor aspects, and in turn the aspect B (or one of its descendant or ancestor aspects) is connected directly or transitively with the aspect A (or one of its descendant or ancestor aspects). A formal definition of the function can be found in Listing E.63.

Function jrHostingAspect. The function $jrHostingAspect : \Delta \times \Delta \rightarrow \Delta$ takes two arguments t and s. The argument t is the syntax tree of an ADORA model and s must be a subtree of t, and s must describe a join relationship. The function returns the syntax tree of the aspect module to which s belongs. If s is not contained in t, the empty element is returned. A formal definition of the function can be found in Listing E.64.

Function ordering. The function $ordering : \Delta \rightarrow \{$`before`, `instead`, `after`$\}$ takes one argument s. It must contain the syntax tree of a join relationship as otherwise the result of the function is undefined. It returns one of the ordering keywords `before`, `instead`, or `after` which indicate how the source element or chunk is woven with respect to the target. If the join relationship does not have

[15] An ancestor aspect is an aspect module in the decomposition hierarchy which encloses the given aspect, whereas a descendant aspect is an aspect module contained in the given aspect.

Listing E.61: The auxiliary function *findJoinRelationshipCycle*.

```
1   function findJoinRelationshipCycle(
2     model, target, startAspect)
3     input
4       model ∈ Δ \ T; target ∈ Δ;
5       startAspect ∈ Δ;
6     return
7       retVal ∈ { true, false };
8     declare
9       tempTarget ∈ Δ; tempSource ∈ Δ;
10      aspectSet ∈ 𝒫(Δ); tempElement ∈ Δ;
11      connections ∈ Δ; jrs ∈ 𝒫(Δ);
12      tempElement2 ∈ Δ;
13    begin
14      tempTarget ← ε; tempSource ← ε;
15      aspectSet ← ∅; retVal ← ε;
16
17      if (type(target) ∈ {
18        AssociationDefinition,
19        TransitionDefinition }
20      begin
21        tempTarget ← target(model, target);
22        tempSource ← source(model, target);
23        if (type(tempTarget) =
24          AspectDefinition)
25        begin
26          aspectSet ← aspectSet ∪ {
27            tempTarget };
28        end
29        if (type(tempSource) =
30          AspectDefinition)
31        begin
32          aspectSet ← aspectSet ∪ {
33            tempSource };
34        end
35        aspectSet ← enclosingAspects(
36          model, tempTarget, aspectSet);
37        aspectSet ← enclosingAspects(
38          model, tempSource, aspectSet);
39        aspectSet ← containedAspects(
40          tempTarget, aspectSet);
41        aspectSet ← containedAspects(
42          tempSource, aspectSet);
43      end else begin
44        if (type(target) =
45          AspectDefinition)
46        begin
47          aspectSet ← aspectSet ∪ { target };
48        end
49        aspectSet ← enclosingAspects(
50          model, target, aspectSet);
51        aspectSet ← containedAspects(
52          target, aspectSet);
53      end
54
55      foreach (tempElement ∈ aspectSet)
56      begin
57        if (tempElement = startAspect)
58          retVal ← true;
59      end
60
61      foreach (tempElement ∈ aspectSet)
62      begin
63        if (retVal = false) begin
64          connections ← childOfType(
65            tempElement,
66            AspectConnections, 1);
67          if (connections ≠ ε) begin
68            jrs ← childrenSetOfType(
69              connections,
70              JoinRelationshipDefinition);
71            foreach (tempElement2 ∈ jrs)
72            begin
73              if (retVal = false) begin
74                tempTarget ← target(
75                  model, tempElement2);
76                retVal ←
77                  findJoinRelationshipCycle(
78                    model,
79                    tempTarget,
80                    startAspect);
81              end
82            end
83          end
84        end
85      end
86    end
```

Listing E.62: The function $gatherAspects$.

```
1   function gatherAspects(model, aspects)
2     input
3       model ∈ Δ;
4       aspects ∈ ∪_k Δ^k, where k ∈ ℕ0;
5     return
6       retVal ∈ ∪_k Δ^k, where k ∈ ℕ0;
7     declare
8       i ∈ ℕ0;
9       child ∈ Δ;
10    begin
11      i ← 0;
12      retVal = aspects;
13      while (i < arity(model)) begin
14        child ← child_i(model);
15        if (child ∉ T) begin
16          if (type(child) = AspectDefinition)
17          begin
18            retVal ← insert_arity(retVal)(
19              retVal, child);
20          end else begin
21            retVal ← gatherAspects(
22              child, retVal);
23          end
24        end
25        i = i + 1;
26      end
27    end
```

Listing E.63: The function $hasJoinRelationshipCycles$.

```
1   function hasJoinRelationshipCycles
2     (model, jr)
3     input
4       model ∈ Δ \ T;
5       jr ∈ Δ ∧
6         type(jr) = JoinRelationshipDefinition
7     return
8       retVal ∈ { true, false };
9     declare
10      startAspect ∈ Δ;
11      target ∈ Δ;
12    begin
13      startAspect ← source(model, jr);
14      target ← target(model, jr);
15      retVal ← findJoinRelationshipCycle(
16        model, target, startAspect);
17    end
18
```

Listing E.64: The function $jrHostingAspect$.

```
1   function jrHostingAspect(model, jr)
2     input
3       model ∈ Δ;
4       jr ∈ Δ ∧
5         type(element) =
6           JoinRelationshipDefinition;
7     return
8       host ∈ Δ;
9     begin
10      host ← treeAncestor(
11        model,
12        jr,
13        { (AspectDefinition, 2) });
14    end
```

Listing E.65: The function *ordering*.

```
1   function ordering(jr)
2     input
3       jr ∈ Δ ∧
4         type(jr) = JoinRelationship;
5     return
6       ordering ∈ { before,
7         instead, after} ⊂ T;
8     begin
9       ordering ←
10        childOfType(jr, before, 1);
11      if (ordering = ε) begin
12        ordering ←
13          childOfType(jr, instead, 1);
14        if (ordering = ε) begin
15          ordering ←
16            childOfType(jr, after, 1);
17          if (ordering = ε) begin
18            ordering ← before;
19          end
20        end
21      end
22    end
```

Listing E.66: The function *priority*.

```
1   function priority(jr)
2     input
3       jr ∈ Δ ∧ type(jr) = JoinRelationship;
4     return
5       p ∈ {1, 2, . . . , 10} ⊂ ℕ;
6     declare
7       o ∈ {1, 2, . . . , 10} ⊂ T;
8       f : {1, 2, . . . , 10} → { 1, 2,. . . , 10 } =
9         { (1,1), (2,2), . . . , (10,10) };
10    begin
11      o ← child(childOfType(jr, Priority, 1),1);
12      if (o ≠ ε) begin
13        p ← f(o);
14      end else begin
15        p ← 1;
16      end
17    end
```

any of the three ordering keywords, the default keyword **before** is returned. A formal definition of the function can be found in Listing E.65.

Function priority. The function $priority : \Delta \rightarrow \{1, 2, \ldots, 10\} \subset \mathbb{N}$ takes the syntax tree of a join relationship s as argument and returns its priority. The method returns the default priority 1 if there is no priority specified in the join relationsip. A formal definition of the function can be found in Listing E.66.

Function serverComponentAssociations. The function $serverComponentAssociations : \Delta \times \Delta \rightarrow \bigcup_{k} \Delta^k$, where $k \in \mathbb{N}_0$, takes two arguments. The first argument t is the syntax tree describing an ADORA model. The second argument s is the syntax tree contained in t which must describe an aspect module. The function seeks for all associations which connect s with any (server) component. The syntax trees of the found associations are returned as a tuple. The function returns the empty tuple $\emptyset$ if s is not part of t or there are no associations connecting the aspect described by s with a server component. A formal definition of the function can be found in Listing E.67.

Function targetModule. The function $targetModule : \Delta \times \Delta \rightarrow \Delta$ takes two arguments. The first argument t is the syntax tree of an ADORA model. The second argument s should be the syntax subtree of a join relationship's target element. The element described by s must be part of the syntax tree t.

Listing E.67: The function $serverComponentAssociations$.

```
1   function serverComponentAssociations
2     (model, aspect)
3     input
4       model ∈ Δ;
5       aspect ∈ Δ;
6     return
7       retVal ∈ ⋃_k Δ^k, where k ∈ ℕ0;
8     declare
9       connections ∈ 𝒫(Δ);
10      a ∈ Δ;
11    begin
12      retVal ← ∅;
13      if (type(aspect) = AspectDefinition) begin
14        connections ← filterSet(
15          connections(aspect),
16          AssociationDefinition);
17        connections ← connections ∪
18          filterSet(
19            targetConnections(model, aspect),
20            AssociationDefinition);
21        foreach a ∈ connections begin
22          retVal ← insert_arity(retVal)(
23            retVal, a);
24        end
25      end
26    end
```

The function returns the next component ($ComponentDefinition$) or aspect ($AspectDefinition$) in the decomposition hierarchy which directly or indirectly contains s. A formal definition of the function can be found in Listing E.68.

E.6 Transformation Functions

Function adaptCloneReferences. The auxiliary function $adaptCloneReferences : \Delta \times \bigcup_i M^i$, where $i \in \mathbb{N}_0$ takes two arguments t and m. The argument t is the syntax tree of an ADORA model or any element contained in a model, and m is a tuple consisting of pairs of mappings between original and copied elements. The function seeks for all element references in t that refer to an original contained in the mapping of m. Any found reference is replaced by the reference to the corresponding cloned element. This helper function is used by the $createCloneMap$ function to substitute the original references by references to the corresponding cloned elements. A formal definition of the function can be found in Listing E.69.

Function cloneElement. The auxiliary function $cloneElement : \Delta \times \Delta \times \bigcup_k M^k \times \{true, false\} \rightarrow \bigcup_i M^i$, where $i, k \in \mathbb{N}_0$, takes four arguments. The first argument t is the syntax tree of an ADORA model. The second argument s is the syntax tree of any uniquely identifiable ADORA element contained in t. The third element m is an exsiting clone map which contains already cloned elements. The fourth argument b is a boolean value which indicates whether this method is called recursively or by another function. for an initial call of the function, b must be *false*. The function takes the syntax tree s and creates a clone of it. The clone is identical to s except for any contained unique element identifier. Moreover, any subtree of the type *ScenarioDe nition* is removed from the clone. Each unique element identifier is adapted so that it is unique within t. Consequently each subelement contained in t is also cloned. Each cloned element and each cloned subelement is appended, together with its original, as a new pair to the clone map m. A formal definition of the function can be found in Listing E.70.

Listing E.68: The function $targetModule$.

```
1   function targetModule(model, element)
2     input
3       model ∈ Δ;
4       element ∈ Δ;
5     return
6       retVal ∈ Δ ∪ { ε };
7     declare
8       temp ∈ Δ; i ∈ ℕ0;
9     begin
10      i ← 0;
11      retVal ← ε;
12      while (i < arity(children(model)) ∧
13        retVal = ε)
14      begin
15        if (child_i(model) ∉ T) begin
16          temp ← child_i(model);
17          if (temp = element) begin
18            retVal ← element;
19          end else begin
20            retVal ←
21              targetModule(temp, element);
22            if (retVal ≠ ε ∧
23              retVal = element)
24            begin
25              if (type(model) ∈ {
26                AspectDefinition,
27                ComponentDefinition }) begin
28                retVal ← model;
29              end
30            end
31          end
32        end
33        i ← i + 1;
34      end
35    end
```

Listing E.69: The function $adaptCloneReferences$.

```
1   function adaptCloneReferences
2     (element, cloneVector)
3     input
4       element ∈ Δ;
5       cloneVector ∈ ⋃_k Δ^k,
6         where k ∈ ℕ0;
7     return
8       retVal ∈ Δ;
9     declare
10      i ∈ ℕ0;
11      ch ∈ Δ;
12      ref ∈ T;
13      mappedRef ∈ Δ;
14    begin
15      retVal ← element; i ← 0;
16      while (i < arity(retVal)) begin
17        ch ← child_i(retVal);
18        if (ch ∈ Δ \ T) begin
19          if (type(ch) = ElementReference)
20          begin
21            ref = elementReference(ch);
22            mappedRef ← mappedReference(
23              ref, cloneVector);
24            if (mappedRef ≠ ε) begin
25              delete_i(retVal);
26              insert_i(mappedRef, i);
27            end
28          end else begin
29            ch ← adaptCloneReferences(
30              ch, cloneVector);
31            delete_i(retVal);
32            insert_i(ch, i);
33          end
34        end
35        i ← i + 1;
36      end
37    end
```

Listing E.70: The function $cloneElement$.

```
1  function cloneElement(model, element,
2    cloneVector, isFirstCall)
3    input
4      model ∈ Δ;
5      element ∈ Δ;
6      cloneVector ∈ ∪_k M^k, where k ∈ ℕ_0;
7      isFirstCall ∈ { true, false };
8    return
9      retVal ∈ ∪_k Δ^k, where k ∈ ℕ;
10   declare
11     ch1 ∈ Δ;
12     ch2 ∈ Δ;
13     children ∈ ∪_k Δ^k, where k ∈ ℕ;
14     original ∈ Δ;
15   begin
16     i ← 0; original ← element;
17     if (isFirstCall) begin
18       element ← removeScenarios(element);
19     end
20     while (i < arity(children(element)) begin
21       ch1 ← child_i(element);
22       if (ch1 ∈ Δ \ T) begin
23         if (type(ch1) =
24           UniqueModelElementidentifier)
25         begin
26           ch2 = createClonedId(
27             model, element);
28         end else begin
29           cloneVector = cloneElement(
30             model, ch1, cloneVector, false);
31           ch2 = π_{arity(cloneVector)−1}(
32             cloneVector);
33           children = children(element);
34           children ← delete_i(children);
35           children ← insert_i(ch2);
36           delete_1(element);
37           element ← insert_1(element,
38             children);
39         end
40       end
41       i ← i + 1;
42     end
43
44     if (childOfType(element,
45       UniqueModelElementIdentifier, 1) ≠ ε)
46     begin
47       children = ∅;
48       insert_0(children, original);
49       insert_1(children, element);
50       insert_{arity(cloneVector)}(cloneVector,
51         children);
52     end
53     if (¬ isFirstCall ∧
54         (childOfType(element,
55         UniqueModelElementIdentifier, 1)
56         = ε)
57         ) begin
58       insert_{arity(cloneVector)}(cloneVector,
59         element);
60     end
61     retVal ← cloneVector;
62   end
```

Function createAdditionalExitStateClone. The function $createAdditionalExitStateClone : \Delta \times \Delta \rightarrow \bigcup_k M^k$, where $k \in \mathbb{N}_0$, takes two arguments and creates a new state and a new exit transition. These elements are needed in the case of the *before* weaving of a behavior chunk (cf. Section G.1.1. The first argument t is the syntax tree of the model. The second argument s is the syntax tree of the exit point for which the additional state has to be created. The function returns a tuple with one pair $(s, c) \in M$. The syntax tree s is the syntax tree of the (original) exit point, whereas c is the created additional state. This tuple can be part of a clone map. It can be added to the clone map by the function $createCloneMap$ (see below). Thus, the exit point s is mapped to the additional state. Furthermore, the syntax tree of the state c also contains a transition which is re exive. The weaving process reassigns the target of this transition to the crosscut transition. The function returns the empty tuple $\emptyset$ if s is not the syntax tree of an exit point. A formal definition of the function can be found in Listing E.71.

Listing E.71: The function $createAdditionalExitStateClone$.

```
1   function createAdditionalExitStateClone(
2     model,exitPoint)
3     input
4       model ∈ Δ;
5       exitPoint ∈ Δ;
6     return
7       map ∈ Δ × Δ;
8     declare
9       name ∈ Δ; cloneid ∈ Δ;
10      transitionDefCh ∈ ∪_k Δ^k, where k ∈ ℕ0;
11      transition ∈ Δ; transitionId ∈ T;
12      targetRef ∈ Δ; transitionUId ∈ Δ;
13      stateDefCh ∈ Δ; state ∈ Δ;
14    begin
15      map ← ∅;
16      if (type(exitPoint) = ExitPointDefinition)
17      begin
18        cloneid ← createCloneId(model,
19          exitPoint);
20        name ← childOfType(model,
21          StateName, 1);
22        if (name = ε ∨ specialIdentifier(
23          childOfType(name,
24          SpecialIdentifier, 1)) = ε)
25        begin
26          name ← generateUniqueName(
27            model, exitPoint);
28        end
29        transitionId ←
30          uniqueElementIdentifier(cloneid);
31        transitionUId ←
32          createUniqueElementIdentTree(
33            generateUniqueElementIdentifier(
34              model, transitionId));
35        targetRef ←
36          createElementReferenceTree(
37            transitionId);
38        transitionDefCh ←
39          ( transitionUId, to, targetRef );
40        transition ← ( TransitionDefinition,
41          transitionDefCh );
42        stateDefCh ← ( state, name,
43          cloneid, connections,
44          transition, end,
45          connections, end,
46          state, name );
47        state ← ( StateDefinition,
48          stateDefCh );
49        map ← ( exitPoint, state );
50      end
51    end
```

Function createCloneId. The auxiliary function $createCloneId : \Delta \times \Delta \rightarrow \Delta$ takes two arguments. The first argument t is the syntax tree of an ADORA model and s a syntax subtree contained in t. The ADORA element specified by argument s must be identified by a unique model element identifier. The function takes this unique model element idenfier and generates a new identifier which is unique with respect to the elements in t. It is returned as a syntax tree of the type *UniqueModelElementIdenti er*. The function is used by $createAdditionalExitStateClone$. There is no formal description of it, as the present work does not anticipate an identifier scheme for ADORA models.

Function createCloneMap. The function $createCloneMap : \Delta \times \mathcal{P}(\Delta) \times \bigcup_k M^k \rightarrow \bigcup_j M^j$, where $k \in \mathbb{N}_0$, $j \in \mathbb{N}_0$, and M is a pair of ADORA syntax trees, computes the mapping between a set of given elements and the corresponding clone copies of these elements. It takes three arguments: the first argument t is the syntax tree of an ADORA model. The second argument m is a set of syntax trees contained in t and which have a unique model element identifier. The elements in m must be subtrees of t. The third argument c is a tuple of a previously computed clone map which is incorporated by the function into the computed clone map. The argument c can be $\emptyset$ if no clone map has been previously computed.

The function takes the elements in m and creates a tuple which contains pairs of ADORA syntax trees. A pair is a mapping between an original element and a clone copy of it. The clone copy gets a

Listing E.72: The function $createCloneMap$.

```
1   function createCloneMap
2     (model, elements, cloneMap)
3     input
4       model ∈ Δ; elements ∈ 𝒫(Δ);
5       cloneMap ∈ ∪_k M^k, where k ∈ ℕ0;
6     return
7       retVal ∈ ∪_k M^k, where k ∈ ℕ0;
8     declare
9       i ∈ ℕ0; element ∈ Δ; clone ∈ Δ;
10      newCh ∪_k M^k, where k ∈ ℕ0;
11      ch1 ∈ ∪_k M^k, where k ∈ ℕ0;
12    begin
13      retVal ← cloneMap; i ← 0;
14      foreach element in elements begin
15        if (element ∈ Δ minus T) begin
16          retVal = cloneElement(model,
17            element, retVal, true);
18        end
19      end
20      while (i < arity(retVal)) begin
21        ch1 ← π_i(retVal); element ← π_0(ch1);
22        clone ← π_1(ch1);
23        clone ←
24          adaptCloneReferences(clone, retVal);
25        newCh ← ∅; insert_0(newCh, element);
26        insert_1(newCh, clone); delete_i(retVal);
27        insert_i(retVal, newCh);
28      end
29    end
```

new model element identifier which is unique in the syntax tree t. The return value of the function is a tuple containing pairs of the form $(o_i, c_i) \in M$, where o_i is the syntax tree of an element contained in m and c_i the corresponding clone. Other uniquely identified elements[16] described by subtrees of o_i are also cloned and receive their own entries in the resulting clone mapping list. However, scenarios which are contained as subtree of an element in m are not cloned, but removed from the clone copies. This happens due to the fact that scenarios build an orthogonal decomposition hierarchy which is cloned separately. Furthermore, references between cloned elements are adapted. Thus, if o_i refers o_j in the clone map, then correspondingly c_i refers to c_j. A formal definition of the function can be found in Listing E.72.

Function ndClone. The function $\mathit{findClone} : \Delta \times \bigcup_k M^k \rightarrow \Delta$, where $k \in \mathbb{N}_0$, maps a given original to a corresponding clone copy of a syntax tree. The first argument o denotes the syntax tree of an original element whose clone copy is searched. The second argument m is the clone map consisting of the mapping pairs between the original syntax trees and the cloned syntax trees. It is produced by the functions $createCloneMap$ and $createAdditionalExitStateClone$ which are described above. The function seeks for the pair (o_i, c_i) in m which has an element o_i that has the same unique element identifier as o and returns c_i. The empty word ε is returned if there is no element found. A formal definition of the function can be found in Listing E.73.

Function ndOrderingGroups. The auxiliary function $\mathit{findOrderingGroups} : \bigcup_k \Delta^k \rightarrow M^3$, where $k \in \mathbb{N}_0$, takes an argument m which is a tuple of syntax trees that describe join relationships. It groups the join relationships according to their ordering *before*, *instead*, *after* and returns a triple (b, i, a), where b is the group of *before*, i the group of *instead*, and a the group of *after* join relationships. Syntax trees in m which are not join relationships are ignored. If no join relationships for a group are found, the group is

[16] For example, a given component x may have parts, such as states. The parts have an own unique identifier and are contained as sub tree in x.

Listing E.73: The function *findClone*.

```
1   function findClone(origInp, cloneMap)
2     input
3       origInp ∈ Δ;
4       cloneMap ∈ ∪_k M^k, where k ∈ ℕ0;
5     return
6       clone ∈ Δ;
7     declare
8       identifier ∈ Δ;
9       orig ∈ Δ;
10      origKey ∈ Δ;
11      i ∈ ℕ0;
12    begin
13      identifier ← childOfType(origInp,
14        UniqueModelElementIdentifier, 1);
15      i ← 0; clone ← ε;
16      while (i < arity(cloneMap) ∧ clone = ε)
17      begin
18        orig ← π0(πi(cloneMap));
19        origKey ← childOfType(orig,
20          UniqueModelElementIdentifier, 1);
21        if (identifier = origKey) begin
22          clone ← π1(πi(cloneMap));
23        end
24        i ← i + 1;
25      end
26    end
```

represented by the an empty tuple $\emptyset$. The function is used *topologicJrSort*. Its formal definition can be found in Listing E.74.

Function rstJr. The function *firstJr* : $\bigcup_k \Delta^k \rightarrow \Delta$, where $k \in \mathbb{N}_0$, takes one argument m which must be a topologically sorted list of join relationships created by the function *topologicJrSort* (cf. Section 9.1 and Section E.6). The argument m is a list of elements $(m_0, m_1, \ldots, m_n)$, where m_i contains tuples impacting the same target. An element m_k is a three-tuple $(m_k^b, m_k^i, m_k^a) \in M$, where m_k^b is a tuple of all *before*, m_k^i of all *instead*, and m_k^a a tuple of all *after* join relationships pointing at the same target. Note that tuples and subtuples may be empty, i.e., contain no join relationships. The function returns the first element in this tuple structure. The search is started at the left-most element. The function returns the empty word ε, if m does not contain any join relationships. A formal definition of the function can be found in Listing E.75.

Function gatherJrs. The auxiliary Function *gatherJrs* : $\Delta \times \Delta \times \bigcup_k \Delta^k \rightarrow \bigcup_p \Delta^p$, where $k, p \in \mathbb{N}_0$ takes three arguments. The first argument t is a syntax tree of an ADORA model. The second argument m is also a syntax tree of an ADORA model which is used for the recursive descent of the function. For an initial call, usually $m = t$. The third argument is a tuple of join relationships which is also used for the recursive descent. It is usually the empty tuple $\emptyset$ for the initial call of the function. The function traverses the model and returns all found join relationships in a tuple. The order in the tuple represents the order in which the join relationships are found in the model. A formal definition of the function is given in Listing E.76.

Function generateUniqueElementIdenti er. The auxiliary function *generateUniqueElementIdenti er* : $\Delta \times T \rightarrow T$ takes two arguments. The first argument t is the syntax tree of an ADORA model. The second argument m must be a unique model element identifier. The function generates a new unique element id which is unique with respect to t and m and returns the corresponding token. The function is used by *createAdditionalExitStateClone*. There is no formalized description of this function, because

Listing E.74: The function *findOrderingGroups*.

```
1   function findOrderingGroups(jrs)
2     input
3       jrs ∈ ∪_k Δ^k, where k ∈ ℕ_0 ∧
4         ∀ l ∈ ℕ_0 :
5           0 ≤ l < arity(jrs) ⇒
6             type(π_l(jrs)) =
7             JoinRelationshipDefinition;
8     return
9       retVal ∈ M^3;
10    declare
11      before ∈ ∪_k M^k, where k ∈ ℕ_0;
12      instead ∈ ∪_k M^k, where k ∈ ℕ_0;
13      after ∈ ∪_k M^k, where k ∈ ℕ_0;
14      curr ∈Δ;
15      ordering ∈ { before, instead,
16        after };
17      i ∈ ℕ_0;
18    begin
19      before ← ∅; instead ← ∅;
20      after ← ∅; retVal ← ∅; i ← 0;
21      curr ← ε; ordering ← ε;
22      while (i < arity(jrs)) begin
23        curr ← π_i(jrs);
24        ordering ← ordering(curr);
25        if (ordering = before)
26        begin
27          insert_arity(before)(before, curr);
28        end else if (ordering = instead)
29        begin
30          insert_arity(insert)(insert, curr);
31        end else if (ordering = after)
32        begin
33          insert_arity(after)(after, curr);
34        end
35        i ← i + 1;
36      end
37      before ← prioritySort(before);
38      instead ← prioritySort(instead);
39      after ← prioritySort(after);
40      insert_arity(retVal)(retVal, before);
41      insert_arity(retVal)(retVal, instead);
42      insert_arity(retVal)(retVal, after);
43    end
```

Listing E.75: The function *firstJr*.

```
1   function firstJr(list)
2     input
3       list ∈ ∪_k Δ^k, where k ∈ ℕ_0;
4     return
5       retVal ∈ Δ;
6     begin
7       retVal ← π_0( atenTuple(list, ∅));
8     end
```

Listing E.76: The function $gatherJrs$.

```
1   function gatherJrs(model, subTree, retVal)
2     input
3       model ∈ Δ;
4       subTree ∈ Δ;
5     return
6       retVal ∈ ∪_k Δ^k, where k ∈ ℕ₀;
7     declare
8       i ∈ ℕ₀;
9     begin
10      i ← 0;
11      while (i < arity(children(model))) begin
12        if (type(child_i(subTree)) =
13          JoinRelationshipDefinition)
14        begin
15          if (type(source(model,
16            child_i(subTree))) ≠
17            EnvironmentObjectDefinition)
18          begin
19            insert_arity(retVal)(retVal, child_i);
20          end
21        end else if (child_i(subTree) ∉ T) begin
22          retVal ← gatherJrs(
23            model, child_i(subTree), retVal);
24        end
25        i ← i + 1;
26      end
27    end
```

the present work does not anticipate an identifier scheme for ADORA models.

Function generateUniqueName. The auxiliary function $generateUniqueName : \Delta \times \Delta \rightarrow \Delta$ takes two arguments. The first argument t is a syntax tree of an ADORA model, and the second argument s is a syntax subtree of t which describes an element contained in the model. The function extracts the name syntax tree of s and creates a new unique name with respect to the context of s. The corresponding syntax tree of the generated name is returned. This fucntion is used by $createAdditionalExitStateClone$. There is no concrete formal description as the present work does not anticipate a naming scheme for ADORA models.

Function generateUniqueRole. The function $generateUniqueRole : \Delta \times \Delta \times \Delta \rightarrow \Delta$ takes three arguments. The first argument t is the syntax tree of an ADORA model. The second argument a is a syntax subtree of t and describes an association. The third argument s is a syntax subtree of t which specifies a role of the association described by a. The function returns the syntax tree of a role which is based on s and which has a unique name with respect to t and a. The cardinality of the returned role is the same as given by s. The function returns the empty word ε if any of the following conditions is fulfilled: (i) a is not an association, (ii) s is not role, (iv) s is not contained in a, or (v) a is not contained in t. Note that there is no formal algorithmic description of this function. This is due to the fact that the function depends on the naming schema for roles that is not anticipated by the present work.

Function identicalElement. The function $identicalElement : \Delta \times \Delta \rightarrow \Delta$ takes two arguments. The first argument t is the syntax tree of an ADORA model. The second argument s is the syntax tree describing an element tagged by a unique model element identifier. The function reads the model element identifier of s and retrieves the syntax subtree in t which is tagged by the same unique model element identifier as that of s. The function returns the empty word ε if no element with the same identifier as s is contained in t.

The function is necessary because the syntax tree of an element may change during the time taken for a model transformation. Therefore, an element need not necessarily be exactly equal at two different

Listing E.77: The function $identicalElement$.

```
1   function identicalElement(model, element)
2     input
3       model ∈ Δ;
4       element ∈ Δ;
5     return
6       retVal ∈ Δ
7     declare
8       idtree ∈ Δ;
9       id ∈ T ∪ ∅;
10    begin
11      retVal ← ε;
12      idtree ← childOfType(element,
13        UniqueModelElementIdentifier, 1);
14      id ← uniqueElementIdentifier(idtree);
15      retVal ← find(model, id);
16    end
```

points in time, and a comparison for equality between trees of the same model element but with different model versions may fail. However, despite changes, the element can still be identified uniquely. Therefore, this function can be employed to retrieve a given element from a different model version resulting during the transformation of a model syntax tree. When using this function for the description of weaving transformations, there is the convention that all properties of the retrieved syntax tree are the same as for the originating tree, unless stated otherwise. A formal definition of the function can be found in Listing E.77.

Function isPredecessorGroup. The auxiliary function $isPredecessorGroup : \Delta \times \bigcup_k M^k \times \bigcup_p \Delta^p$, where $k, p \in \mathbb{N}_0$, takes three arguments. The first argument t is the syntax tree of an ADORA model. The second and third argument m and q contain a multiple nested-tuple structure consisting of join relationship syntax trees. The function checks for each join relationship in m whether a predecessor join relationship in p exists.[17] If at least one join relationship of m has a predecessor in p, the method returns true, otherwise the method returns false. The function $isPredecessorGroup$ is used by the function $sortTargetGroups$ to determine the weaving order of transitively crosscutting aspects. Its formal definition can be found in Listing E.78.

Function mappedReference. The auxiliary function $mappedReference : T \times \bigcup_k M^k \rightarrow \Delta$ takes two arguments. The first argument r should be a terminal symbol that is a unique element identifier. The second argument must be clone map produced by $createCloneMap$. The function takes r and searches for an orginal which has a unique model element identifier that is equal to r. It returns the clone copy of the found original. If there is no element with an identifier r in the clone map, the function returns the empty word ε. This function is used by $adpatCloneReferences$. Its formal definition can be found in Listing E.79.

Function prioritySort. The auxiliary function $prioritySort : \bigcup_k \Delta^k \rightarrow \bigcup_k \Delta^k$, where $k \in \mathbb{N}_0$ takes one argument m which must be a tuple of join relationship syntax trees. It returns a tuple which contains all elements in m sorted according to their priority. Syntax trees in m which do not describe join relationships are interpreted with the lowest priority. The function is used by $findOrderingGroups$, and is described more formally in Listing E.80.

[17] The predecessor relationship between two join relationships is explained in Section 9.1.1.

Listing E.78: The function $isPredecessorGroup$.

```
1   function isPredecessorGroup
2     (model, targetGroup, compareGroup)
3     input
4       model ∈ Δ;
5       targetGroup ∈ ⋃_k M^k, where k ∈ ℕ0;
6       compareGroup ∈ ⋃_k M^k, where k ∈ ℕ0;
7     return
8       isBefore ∈ { true, false };
9     declare
10      encAspects ∈ P(Δ); i ∈ ℕ0; j ∈ ℕ0;
11      tg1 ∈ ⋃_k M^k, where k ∈ ℕ;
12      tg2 ∈ ⋃_k M^k, where k ∈ ℕ;
13      e1 ∈ Δ;
14    begin
15      isBefore ← false;
16      tg1 ←  atenChildren(targetGroup, ε);
17      tg2 ←  atenChildren(compareGroup, ε);
18      i ← 0;
19      while (i < arity(tg1) ∧
20        isBefore = false) begin
21        e1 ← jrHostingAspect(model, π_i(tg1));
22        j ← 0;
23        while (j < arity(tg2) ∧
24          isBefore ≠ false) begin
25          encAspects ← enclosingAspects(
26            model, π_j(tg2), ∅);
27          if (e1 ∈ encAspects) begin
28            isBefore ← true;
29          end
30          j ← j + 1;
31        end
32        i ← i + 1;
33      end
34    end
```

Listing E.79: The function $mappedReference$.

```
1   function mappedReference(id, cloneVector)
2     input
3       id ∈ T;
4       cloneVector ∈ ⋃_k M^k,
5         where k ∈ ℕ0;
6     return
7       retVal ∈ Δ;
8     declare
9       ch1 ∈ ⋃_k M^k,
10        where k ∈ ℕ0;
11      orig ∈ Δ; clone ∈ Δ;
12      ref ∈ Δ; i ∈ ℕ0; sid ∈ T;
13    begin
14      retVal ← ε;
15      i ← 0;
16      while (i < arity(cloneVector)) begin
17        ch1 ← π_i(cloneVector);
18        orig ← π_0(ch1);
19        clone ← π_1(ch1);
20        ref ← childOfType(orig,
21          UniqueModelElementidentifier, 1)
22        if (ref ≠ ε) begin
23          sid = uniqueElementIdentifier(ref);
24          if (sid = id) begin
25            ref ←childOfType(clone,
26              UniqueModelElementIdentifier,
27              1);
28            sid ← uniqueElementIdentifier(
29              ref);
30            retVal ← ( ElementReference,
31              ( ( SpecialIdentifier, ( sid ) ) );
32          end
33        end
34        i ← i + 1;
35      end
36    end
```

Listing E.80: The function $prioritySort$.

```
1   function prioritySort(jrs)
2     input
3       jrs ∈ ⋃_k M^k, where k ∈ ℕ0 ∧
4         ∀ x ∈ jrs :
5           type(x) = JoinRelationshipDefinition;
6     return
7       jrs ∈ ⋃_k M^k, where k ∈ ℕ0;
8     declare
9       i ∈ ℕ0;
10      j ∈ ℕ0;
11      value ∈ ℕ;
12      t1 ∈ Δ;
13    begin
14      i ← 1;
15      while (i < arity(jrs)) begin
16        t1 ← π_i(jrs);
17        value ← priority(t1);
18        j ← j - 1;
19        while (j >= 0 ∧
20          priority(π_j(jrs) < value) begin
21          insert_{j+1}(jrs, π_j(jrs));
22          delete_{j+2}(jrs);
23          j ← j - 1;
24        end
25        insert_{j+1}(jrs, t1);
26        delete_{j+2}(jrs);
27        i ← i + 1;
28      end
29    end
```

Function removeScenarios. The auxiliary function $removeScenarios : \Delta \to \Delta$ has one argument s which is an ADORA syntax tree. The function traverses s and removes all subtrees of the type *ScenarioDe nition*. The function is used by $cloneElement$, because the weaving of embedded components and scenarios in aspects is done in two different steps of the weaving (cf. Section 9.2.2, 9.2.4 and 9.2.5). A formal definition of $removeScenarios$ can be found in Listing E.81.

Function sortTargetGroups. The auxiliary function $sortTargetGroups : \Delta \times \bigcup_k \Delta^k \to \bigcup_k \Delta^k$, where $k \in \mathbb{N}_0$ takes two arguments. The first argument t is the syntax tree of an ADORA model. The second argument m is a tuple which contains tuples of join relationship groups. Each group contains join relationships which have the same target. The join relationships in m must be part of t. The function returns the target groups in m sorted by the predecessor relationship (cf. Section 9.1.1 and p. 360) and is used by $topologicJrSort$. It is described formally in Listing E.82.

Function topologicJrSort. The function $topologicJrSort : \Delta \to \bigcup_k M^k$, where $k \in \mathbb{N}$ implements the topological sorting of the join relationships. It takes an ADORA model syntax tree t and returns a tuple containing the syntax trees of all topologically ordered join relationships, except the join relationships between crosscutting environment objects. The empty tuple $\emptyset$ is returned if there are no join relationships in t_0. The resulting tuple has the structure $(m_0, m_1, \ldots, m_p)$, where $p \in \mathbb{N}_0$. The element $m_k \in M$ denotes a tuple of the join relationships impacting the same target. It is structured in turn as a tuple of the form (m_k^b, m_k^i, m_k^a), where m_k^b denotes the join relationships with a *before*, m_k^i with an *instead* and m_k^a with an *after* ordering. The tuple m_k^o, where o denotes the ordering index $b, i,$ or a, is in turn a tuple containing the syntax trees of the join relationships $(j_{k,0}^o, j_{k,1}^o, \ldots, m_{k,q}^o)$ ordered according to their priority. The function is described formally in Listing E.83.

Listing E.81: The function *removeScenarios*.

```
1  function removeScenarios(element)
2    input
3      element ∈ Δ;
4    return
5      element ∈ Δ;
6    declare
7      i ∈ ℕ0; j ∈ ℕ0;
8      child ∈ Δ;
9      children ∈ ⋃_k Δ^k, where k ∈ ℕ0;
10     compParts ∈ ⋃_k Δ^k, where k ∈ ℕ0;
11     removeCompParts ∈ { true, false };
12   begin
13     i ← 0;
14     children ← children(element);
15     while (i < arity(children)) begin
16       child ← π_i(children);
17       if (type(child) = ScenarioDefinition)
18       begin
19         children ← delete_i(children);
20       end else if (child ∉ T) begin
21         child ← removeScenarios(child);
22         children ← delete_i(children);
23         children ← insert_i(children, child);
24       end
25       if (type(child) = ComponentParts)
26       begin
27         removeCompParts ← true;
28         compParts ← children(child);
29         j ← 0;
30         while (j < arity(compParts)) begin
31           if (π_j(compParts) ∉ T) begin
32             removeCompParts ← false;
33           end
34           j ← j + 1;
35         end
36         if (removeCompParts) begin
37           children ← delete_i(children);
38         end
39       end
40       i ← i + 1;
41     end
42     element ← delete_1(element);
43     element ← insert_1(element, children);
44 end
```

Listing E.82: The function *sortTargetGroups*.

```
1  function sortTargetGroups(model,
2    targetGroups)
3    input
4      model ∈ Δ;
5      targetGroups ∈ ⋃_k M^k, where k ∈ ℕ0;
6    return
7      targetGroups ∈ ⋃_k M^k, where k ∈ ℕ0;
8    declare
9      currTg ∈ ⋃_k M^k, where k ∈ ℕ0;
10     i ∈ ℕ0;
11     j ∈ ℕ0;
12   begin
13     i ← 1;
14     while (i < arity(targetGroups)) begin
15       currTg ← π_i(targetGroups);
16       j ← i - 1;
17       while (j >= 0 ∧ isPredecessorGroup(
18         model, currTg, π_j(targetGroups)))
19       begin
20         insert_{j+1}(targetGroups,
21           π_j(targetGroups));
22         delete_{j+2}(targetGroups);
23         j ← j - 1;
24       end
25       insert_{j+1}(targetGroups, currTg);
26       delete_{j+2}(targetGroups);
27       i ← i + 1;
28     end
29 end
```

Listing E.83: The function $topologicJrSort$.

```
1  function topologicJrSort(model)
2    input
3      model ∈ Δ \ T;
4    return
5      retVal ∈ ∪_k M^k, where k ∈ ℕ_0;
6    declare
7      jrs ∈ ∪_k Δ^k, where k ∈ ℕ_0;
8      i ∈ ℕ_0;
9      targetGroup ∈ ∪_k M^k, where k ∈ ℕ_0;
10     currJr ∈ Δ;
11   begin
12     jrs ← gatherJrs(model, model, ∅);
13     retVal ← ∅;
14     while (arity(jrs) > 0) begin
15       targetGroup ← ∅;
16       insert_0(targetGroup, π_0(jrs));
17       delete_0(jrs);
18
19       i ← 0;
20       while (i < arity(jrs)) begin
21         currJr ← π_i(jrs);
22         if (target(model, currJr) =
23           target(model,
24             π_0(targetGroup)))
25         begin
26           insert_arity(targetGroup)(
27             targetGroup, currJr);
28           delete_i(jrs);
29         end
30         i ← i + 1;
31       end
32
33       targetGroup ← findOrderingGroups(
34         targetGroup);
35       insert_arity(retVal)(retVal,
36         targetGroup);
37     end
38     retVal ← sortTargetGroups(
39       model, retVal);
40   end
```

Appendix F

Formal Language Constraints of Aspect-Oriented Constructs

This section provides the formal definition of the language constraints needed for the aspect-oriented elements presented in Chapter 7. Note that the functions used for describing the predicates can be found in the catalog in Appendix E.

F.1 Aspect Module

The footer and the header of an textual aspect module definition must contain the same name. This is ensured by the constraint in Definition C_2, where *t* is the syntax tree of the aspect module. The constraint is leniently checked at a user-defined time.

$$\begin{aligned} &childOfType(t, AspectName, 1) = childOfType(t, AspectName, 2) \wedge \\ &childOfType(t, AspectName, 1) \neq {}'' \end{aligned} \tag{C_2}$$

F.2 Behavior Description

F.2.1 State Groups Must Be Well-Formed

A state group in an aspect module must have either one start state or one exit point, but not both. This constraint is formally expressed in Definition C_3. In this formula, t and s are the syntax trees of the ADORA model and the aspect module, respectively. This constraint is leniently enforced and must be satisfied before weaving the model.

$$\forall x \in stateGroups(t, parts(s)) : \\ \big((|exitPoints(x)| = 1 \wedge |startStates(x)| = 0) \vee \\ (|exitPoints(x)| = 0 \wedge |startStates(x)| = 1)\big) \tag{C_3}$$

F.2.2 No Out-Going Join Relationships from Crosscutting Statecharts

No state or component which is part of a crosscutting statechart, i.e., a state group with one start state, may have an outgoing join relationship. This is expressed by the Constraint C_4. In this definition, $\mathcal{J}$ denotes the set of all join relationships of the aspect module described by the syntax tree s. The term t is the syntax tree of the ADORA model. This constraint is strictly enforced before a join relationship is inserted in the model. In the case of a violation, the insertion is not allowed.

$$\forall x \in stateGroups(t, parts(s)) : \\ \big((|startStates(x)| > 0) \Rightarrow \forall j \in \mathcal{J} : (\forall y \in x : source(j) \neq y)\big) \\ \text{where } \mathcal{J} = filterSet(connections(s), JoinRelationshipDefinition) \tag{C_4}$$

F.2.3 No Crossing of the Aspect Border by Transitions

A transition originating in a state group which is part of an aspect may not cross the border of the aspect. This fact is expressed by the constraint in Definition C_5, where s is the syntax tree of the aspect module and t the syntax tree of the corresponding ADORA model. The constraint is strictly enforced.

$$stateGroups(t, parts(s)) \neq \emptyset \Rightarrow \forall x \in stateGroups(t, parts(s)) : \forall y \in x : y \in descendants(s) \tag{C_5}$$

F.2.4 Transitions May Connect to Exit Points

In the previous versions of ADORA, no exit point element existed. As it exists in the aspect-oriented version, transitions must be connectable to them. The corresponding language constraint must be overridden by the formal constraint given in Definition C_6. In the predicate s is the syntax tree of the transition and t the syntax tree of the ADORA model. This constraint is strictly enforced.

$$type(target(t, s)) \in \{ComponentDefinition, StateDefinition, ExitPointDefinition\} \tag{C_6}$$

F.3 User View

F.3.1 No Crossing of Aspect Border by a Scenario Connection

The connections between the nodes of a scenario group may not cross the border of an aspect. This language constraint is expressed in Definition C_7, where t is the syntax tree of the ADORA model and s the syntax tree of the aspect module containing the scenario elements to be check. This constraint is strictly enforced.

$$\begin{aligned} &\forall x \in \mathit{scenarioGroups}(t, \mathit{parts}(s)) : \\ &\quad \forall y \in x : \mathit{filterSet}(\mathit{connections}(y), \mathit{ScenarioConnection}) \neq \emptyset \Rightarrow \\ &\quad\quad \forall z \in \mathit{filterSet}(\mathit{connections}(y), \mathit{ScenarioConnection}) : \\ &\quad\quad\quad \mathit{target}(t, z) \in \mathit{descendants}(s) \end{aligned} \tag{C_7}$$

F.3.2 Well-Formed Scenario Chunks

A well-formed scenario chunk is a scenario group which has to satisfy one the following two conditions. First, if there is a join relationship outgoing from the scenario group, no associations may be outgoing from any node in the scenario group. Second, a scenario group does not need to have an out-going join relationship, but if there is one, it has to originate from the root node of the scenario group. The following two auxiliary predicates are used to express the corresponding constraints.

- The predicate $\mathit{sgHasAssociation} : \mathcal{P}(\Delta) \rightarrow \{\mathit{true}, \mathit{false}\}$ takes a set of syntax trees m as argument. Each syntax tree in m describes a scenario node which is member of the same scenario group. The method returns true if m contains at least one node which is connected with an association to another element. This predicate is formally specified in Definition F.1. If the input is invalid, false is returned.
- The predicate $\mathit{sgHasJoinRelationship} : \Delta \times \mathcal{P}(\Delta) \times \Delta \rightarrow \{\mathit{true}, \mathit{false}\}$ takes three arguments. The argument t is the syntax tree of the ADORA model, m is a set of syntax trees forming a scenario group, and j is a syntax tree of a join relationship. The predicate returns true if at least one node contained in m is the source of j. This predicate is formally given in Definition F.2.

$$\begin{aligned} &\mathit{sgHasAssociation}(m) = \exists y \in m : \\ &\quad (\mathit{filterSet}(\mathit{connections}(y), \mathit{AssociationDefinition}) \cup \\ &\quad \mathit{filterSet}(\mathit{targetConnections}(t, y), \mathit{AssociationDefinition}) \neq \emptyset) \end{aligned} \tag{F.1}$$

$$\mathit{sgHasJoinRelationship}(t, m, j) = \exists y \in m : \mathit{source}(t, j) = y \tag{F.2}$$

The language constraint for well-formed scenario chunks is given by Definition C_8, where s represents the syntax tree of the aspect module and t the syntax tree of the corresponding ADORA model. It is strictly enforced.

$$\begin{aligned}
&\forall x \in \mathit{scenarioGroups}(t, \mathit{parts}(s)) : \neg \mathit{sgHasAssociation}(x) \Rightarrow \\
&\quad \exists \mathcal{J} = \mathit{filterSet}(\mathit{connections}(s), \mathit{JoinRelationshipDefinition}) : \\
&\qquad \forall j \in \mathcal{J} : \\
&\qquad\quad \big(\mathit{sgHasJoinRelationship}(t, x, j) \Rightarrow \\
&\qquad\qquad \forall z \in x \setminus \mathit{rootScenarios}(x) : \\
&\qquad\qquad\quad \mathit{source}(t, j) \neq z \\
&\qquad\quad \big)
\end{aligned} \tag{C_8}$$

F.3.3 Well-Formed Crosscutting Scenariocharts

A well-formed crosscutting scenariochart must satisfy three conditions. First, it must be connected by an association to at least one environment object. Second, no join relationships may originate in any of the nodes of the scenario group, and third, non-root nodes may not have an association connected to another node.

This constraint is strictly enforced. It is specified in Definition C_9, where s is the syntax tree of the aspect module and t the syntax tree of the ADORA model. The constraint uses the auxiliary predicates $\mathit{sgHasJoinRelationship}$ and $\mathit{sgHasAssociations}$ which are defined above.

$$\begin{aligned}
&\forall x \in \mathit{scenarioGroups}(t, \mathit{parts}(s)) : \mathit{sgHasAssociation}(x) \Rightarrow \\
&\quad (\forall y \in x \setminus \mathit{rootScenarios}(x) : \\
&\qquad \mathit{filterSet}(\mathit{connections}(y), \mathit{AssociationDefinition}) \cup \\
&\qquad \mathit{filterSet}(\mathit{targetConnections}(t, y), \mathit{AssociationDefinition}) = \emptyset) \wedge \\
&\quad \forall j \in \mathcal{J} : \neg \mathit{sgHasJoinRelationship}(t, x, j) \\
&\\
&\quad \text{where } \mathcal{J} = \mathit{filterSet}(\mathit{connections}(s), \mathit{JoinRelationshipDefinition})
\end{aligned} \tag{C_9}$$

F.3.4 Disallowed Embedding of A Scenario Node in a Component Belonging to a Statechart

A scenario node may not be part of a component which belongs to a statechart. This constraint is leniently enforced before an aspect-oriented model is woven. It is formally specified in C_{10}, where t is the syntax tree of the ADORA model and s the syntax tree of the aspect to be check.

$$
\begin{aligned}
&\forall x \in \mathit{filterSet}(\mathit{descendants}(s), \mathit{ScenarioDefinition}) : \\
&\quad \exists y = \mathit{decompositionParent}(t, x) : \\
&\quad\quad \mathit{type}(y) = \mathit{ComponentDefinition} \Rightarrow \\
&\quad\quad\quad \exists z = \mathit{connections}(y) \cup \mathit{targetConnections}(t, y) : \\
&\quad\quad\quad\quad \mathit{filterSet}(z, \mathit{TransitionDefinition}) = \emptyset
\end{aligned} \tag{C_{10}}
$$

F.4 Join Relationships

F.4.1 Constituents of Non-Partial Join Relationships

A non-partial join relationship can either connect the state of a behavior chunk with a transition, a scenario chunk with a scenario node, or an environment object with an environment object. The formal definition of this language constraint can be found in Definition C_{11}, where t is the syntax tree of the ADORA model and s is the sub-tree describing the join relationship. The set S contains the syntax trees of elements that are part of the aspect to which the join relationship s belongs and can be the syntax trees of all scenario root nodes, all components that are part of a statechart, and all states are parts of the aspect where s originates. This constraint is strictly enforced.

$$
\begin{aligned}
&\neg \mathit{partial}(s) \Rightarrow \\
&\quad \mathit{source}(t, s) \in S \ \wedge \big(\\
&\quad\quad (\mathit{type}(\mathit{source}(t, s)) = \mathit{ScenarioDefinition} \wedge \mathit{type}(\mathit{target}(t, s)) = \mathit{ScenarioDefinition}) \vee \\
&\quad\quad (\mathit{type}(\mathit{source}(t, s)) = \mathit{StateDefinition} \wedge \mathit{type}(\mathit{target}(t, s)) = \mathit{TransitionDefinition}) \vee \\
&\quad\quad (\mathit{type}(\mathit{source}(t, s)) = \mathit{EnvironmentObjectDefinition} \wedge \\
&\quad\quad \mathit{type}(\mathit{target}(t, s)) = \mathit{EnvironmentObjectDefinition})\big)
\end{aligned} \tag{C_{11}}
$$

$$
\begin{aligned}
\text{where } S =\, &\mathit{rootScenarios}(\mathit{parts}(\mathit{jrHostingAspect}(t, s))) \cup \\
&\mathit{filterSet}(\mathit{parts}(\mathit{jrHostingAspect}(t, s)), \mathit{StateDefinition}) \cup \\
&\{x | x \in \mathit{filterSet}(\mathit{parts}(\mathit{jrHostingAspect}(t, s)), \mathit{ComponentDefinition}) \wedge \\
&\quad \exists y \in \mathit{stategroups}(\mathit{parts}(\mathit{jrHostingAspect}(t, s))) : x \in y\}
\end{aligned}
$$

F.4.2 Constituents of Partial Join Relationships

Partial join relationships may connect either an aspect, a state or a scenario with either an aspect, a transition, a state, an association, a component, or a scenario. Moreover, an environment object may also be connected by a partial join relationship with another environment object. This constraint is strictly enforced and formally defined in Definition C_{12}, where t is the syntax tree of the ADORA model and s is the join relationship contained in t.

$$\begin{aligned}
&\mathit{partial}(s) \Rightarrow \big(\mathit{source}(t,s) \in S \cup \{\mathit{jrHostingAspect}(t,s)\} \wedge \\
&\quad (\mathit{type}(\mathit{target}(t,s)) = \mathit{AspectDefinition} \vee \mathit{type}(\mathit{target}(t,s)) = \mathit{TransitionDefinition} \vee \\
&\quad \mathit{type}(\mathit{target}(t,s)) = \mathit{StateDefinition} \vee \mathit{type}(\mathit{target}(t,s)) = \mathit{AssociationDefinition} \vee \\
&\quad \mathit{type}(\mathit{target}(t,s)) = \mathit{ComponentDefinition} \vee \mathit{type}(\mathit{target}(t,s)) = \mathit{ScenarioDefinition})\big) \vee \\
&\quad (\mathit{type}(\mathit{source}(t,s)) = \mathit{EnvironmentObjectDefinition} \wedge \\
&\quad \mathit{type}(\mathit{target}(t,s)) = \mathit{EnvironmentObjectDefinition})
\end{aligned} \tag{C_{12}}$$

$$\begin{aligned}
\text{where } S =\, &\mathit{rootScenarios}(\mathit{parts}(\mathit{jrHostingAspect}(t,s))) \cup \\
&\mathit{filterSet}(\mathit{parts}(\mathit{jrHostingAspect}(t,s)), \mathit{StateDefinition}) \cup \\
&\{x \mid x \in \mathit{filterSet}(\mathit{parts}(\mathit{jrHostingAspect}(t,s)), \mathit{ComponentDefinition}) \wedge \\
&\quad \exists y \in \mathit{stategroups}(\mathit{parts}(\mathit{jrHostingAspect}(t,s))) : x \in y\}
\end{aligned}$$

F.4.3 Join Relationships Connecting to Scenariochart Root Nodes

A concrete join relationship connecting a scenario chunk node and the root node of a target scenario must have the ordering *instead*. This constraint is leniently enforced before the aspect-oriented model is woven. Its formal Definition can be found in Constraint C_{13}, where t is the syntax tree of the ADORA model and s the syntax tree of the join relationship.

$$\begin{aligned}
&\mathit{type}(\mathit{source}(t,s)) = \mathit{ScenarioDefinition} \wedge \\
&\mathit{type}(\mathit{target}(t,s)) = \mathit{ScenarioDefinition} \wedge \\
&\mathit{partial}(s) = \mathit{false} \Rightarrow \\
&\quad \exists x = \mathit{descendants}(\mathit{targetModule}(t, \mathit{target}(t,s))) : \\
&\quad\quad \exists y = \mathit{rootScenarios}(t, \mathit{filterSet}(x, \mathit{ScenarioDefinition})) : \\
&\quad\quad\quad \mathit{target}(t,s) \in y \Rightarrow \\
&\quad\quad\quad\quad \mathit{ordering}(s) = \texttt{\textbf{instead}}
\end{aligned} \tag{C_{13}}$$

F.4.4 No Cycles in Join Relationships

A join relationship may not lead to a cycle within the aspect structure. This constraint is strictly enforced, and it is formally expressed by the Constraint C_{14}. The variable t denotes the syntax tree of the ADORA model, and s the syntax tree of the join relationship.

$$\mathit{hasJoinRelationshipCycles}(t,s) = \mathit{false} \tag{C_{14}}$$

F.4.5 Border Crossing of the Join Relationship

A join relationship may not cross the border of the aspect's parent. This constraint is strictly enforced and formally given in Definition C_{15}. The variable t denotes the syntax tree of the ADORA model and s the syntax tree of join relationship.

$$target(t, s) \in descendants(decompositionParent(t, jrHostingAspect(t, s))) \tag{C_{15}}$$

F.4.6 Priority within the Range of 1–10

The priority of a join relationship must be between 1 and 10. This constraint is formally defined by the Definition C_{16}, where s is the syntax tree of the join relationship. This constraint is leniently enforced.

$$priority(s) \in \{1, 2 \ldots 10\} \subset \mathbb{N} \tag{C_{16}}$$

F.5 Crosscutting Environment Objects

F.5.1 Only One Join Relationship between Two Environment Objects

Between the same two environment objects, more than one join relationship is not meaningful. This restriction is strictly enforced and specified more formally by Constraint C_{17}, where s denotes the syntax tree of the environment object and t the syntax tree of the model.

$$\begin{aligned} &\forall x \in filterSet(connections(s), JoinRelationshipDefinition) : \\ &\quad \forall y \in filterSet(connections(s), JoinRelationshipDefinition) : \\ &\qquad (x \neq y) \Rightarrow target(t, x) \neq target(t, y) \end{aligned} \tag{C_{17}}$$

F.5.2 Crosscutting Environment Objects Must Be Connected To a Scenariochart

A crosscutting environment object must be connected with an association to a crosscutting scenariochart. The corresponding constraint is leniently enforced before the weaving is executed. It is described formally in C_{18}, where s denotes the syntax tree of the environment object and t the syntax tree of the ADORA model.

$$\begin{aligned}&\mathit{filterSet}(\mathit{connections}(s), \mathit{JoinRelationshipDefinition}) \neq \emptyset \Rightarrow \\ &\quad \exists y \in \mathit{filterSet}(\mathit{connections}(s), \mathit{AssociationDefinition}) \cup \\ &\qquad \mathit{filterSet}(\mathit{targetConnection}(t, s), \mathit{AssociationDefinition}) : \\ &\qquad\quad \mathit{enclosingAspects}(t, \mathit{target}(t, y), \emptyset) \cup \\ &\qquad\quad \mathit{enclosingAspects}(t, \mathit{source}(t, y), \emptyset) \neq \emptyset\end{aligned} \tag{C_{18}}$$

F.5.3 No Association of a Crosscutting Environment to More than One Aspect

A crosscutting environment object may be connected to several different crosscutting scenariocharts. However, these scenariocharts must be part of the same aspect. This constraint is leniently checked before the weaving of a model and formally described in Constraint C_{19}, where t denotes the syntax tree of the ADORA model and s the syntax tree of the environment object.

$$\begin{aligned}&\mathit{filterSet}(\mathit{connections}(s), \mathit{JoinRelationshipDefinition}) \neq \emptyset \Rightarrow \\ &\quad \big(\forall x \in \mathcal{A} : \mathit{type}(\mathit{source}(t, x)) = \mathit{EnvironmentObjectDefinition} \Rightarrow \\ &\qquad \big(\forall y \in \mathcal{A} : x \neq y \wedge \mathit{source}(t, y) = s \wedge \mathit{type}(\mathit{target}(t, y)) \neq \mathit{EnvironmentObjectDefinition} \Rightarrow \\ &\qquad\qquad \mathit{enclosingAspects}(t, \mathit{target}(t, y)) = \mathit{enclosingAspects}(t, \mathit{target}(t, x))\big) \\ &\quad \big) \wedge \big(\forall x \in \mathcal{A} : \mathit{type}(\mathit{target}(t, x)) = \mathit{EnvironmentObjectDefinition} \Rightarrow \\ &\qquad\quad \big(\forall y \in \mathcal{A} : x \neq y \wedge \mathit{target}(t, y) = s \wedge \mathit{type}(\mathit{source}(t, y)) \neq \mathit{EnvironmentObjectDefinition} \Rightarrow \\ &\qquad\qquad\quad \mathit{enclosingAspects}(t, \mathit{source}(t, y)) = \mathit{enclosingAspects}(t, \mathit{source}(t, x))\big) \\ &\quad \big)\end{aligned} \tag{C_{19}}$$

where $\mathcal{A} = \mathit{filterSet}(\mathit{targetConnections}(t, s), \mathit{AssociationDefinition}) \cup \mathit{filterSet}(\mathit{connections}(s), \mathit{AssociationDefinition})$

F.6 Aspect Decomposition

F.6.1 Aspect-Re ned Aspect Modules May Contain only Aspect Modules

An aspect containing embedded aspects may not contain other elements than aspects. This constraint is strictly enforced and formally given in Definition C_{20}, where s is the syntax tree of an aspect.

$$\begin{aligned}&(\exists x \in \mathit{parts}(s) : \mathit{type}(x) = \mathit{AspectDefinition}) \Rightarrow \\ &\quad (\forall y \in \mathit{parts}(s) : \mathit{type}(y) = \mathit{AspectDefinition})\end{aligned} \tag{C_{20}}$$

F.6.2 Associations Originating within an Aspect May not Cross the Border of the Aspect

If one constituent node of an association is part of an aspect A, then the other constituent node must also be part of A. This fact is formally expressed in the Constraint C_{21}, where s denotes the syntax tree of the association and t the syntax tree of the ADORA model. This constraint is strictly enforced.

$$\mathit{enclosingAspects}(t, \mathit{target}(t,s), \emptyset) = \mathit{enclosingAspects}(t, \mathit{source}(t,s), \emptyset) \tag{C_{21}}$$

F.6.3 Components and Aspects Must Be Connectable

Aspects and components must be connectable by associations, which extends the original meaning of associations in ADORA. This constraint must be strictly enforced. It is formally specified in Constraint C_{22}, where s is the syntax tree of the association and t the syntax tree of the ADORA model.

$$\begin{aligned}
(\mathit{type}(\mathit{target}(t,s)) = \; & \mathit{ComponentDefinition} \;\wedge \\
\mathit{type}(\mathit{source}(t,s)) = \; & \mathit{ComponentDefinition}) \;\vee \\
(\mathit{type}(\mathit{target}(t,s)) = \; & \mathit{AspectDefinition} \;\wedge \\
\mathit{type}(\mathit{source}(t,s)) = \; & \mathit{ComponentDefinition}) \;\vee \\
(\mathit{type}(\mathit{target}(t,s)) = \; & \mathit{ScenarioDefinition} \;\wedge \\
\mathit{type}(\mathit{source}(t,s)) = \; & \mathit{EnvironmentObjectDefinition}) \;\vee \\
(\mathit{type}(\mathit{target}(t,s)) = \; & \mathit{EnvironmentObjectDefinition} \;\wedge \\
\mathit{type}(\mathit{source}(t,s)) = \; & \mathit{EnvironmentObjectDefinition}) \;\vee \\
\\
(\mathit{type}(\mathit{target}(t,s)) = \; & \mathit{ScenarioDefinition} \;\wedge \\
\mathit{type}(\mathit{source}(t,s)) = \; & \mathit{ComponentDefinition} \;\wedge \\
\mathit{childOfType}(\mathit{source}(t,s), \mathbf{external}) \neq \; & \varepsilon) \;\vee \\
(\mathit{type}(\mathit{source}(t,s)) = \; & \mathit{AspectDefinition} \;\wedge \\
\mathit{type}(\mathit{target}(t,s)) = \; & \mathit{ComponentDefinition}) \;\vee \\
(\mathit{type}(\mathit{source}(t,s)) = \; & \mathit{ScenarioDefinition} \;\wedge \\
\mathit{type}(\mathit{target}(t,s)) = \; & \mathit{EnvironmentObjectDefinition}) \;\vee \\
(\mathit{type}(\mathit{source}(t,s)) = \; & \mathit{ScenarioDefinition} \;\wedge \\
\mathit{type}(\mathit{target}(t,s)) = \; & \mathit{ComponentDefinition} \;\wedge \\
\mathit{childOfType}(\mathit{target}(t,s), \mathbf{external}) \neq \; & \varepsilon)
\end{aligned} \tag{C_{22}}$$

Appendix G

Formal Weaving Semantics

The present appendix gives the formal description of the weaving semantics which was delineated informally in Chapter 9. Section 9.4.3 explained the schema by which the present work describes the formal weaving semantics. It uses an axiomatic semantics which employs various functions that operate on ADORA syntax trees. These functions allow retrieval of particular properties of a model represented by the syntax tree. In the most cases, the names of functions are self-documenting. However, all of them are explained in detail and formally described in the catalog of functions in Appendix E.

Analogously to the informal description in Chapter 9, the formal description is split into two parts. Section G.1 describes the weaving semantics of non-partial elements and Section G.2 the semantics for partial elements.

G.1 Formal Weaving Semantics of Non-Partial Elements

This section presents the weaving semantics of non-partial aspect-oriented elements.

G.1.1 Weaving Semantics of Behavior Chunks

This section describes formally the weaving semantics of behavior chunks whose informal description is presented in Section 9.2.1.

Given Elements

The operation ϕ_g weaves a behavior chunk. For its formal description, several elements are given as prerequisites. The model t_{g-1} is the input and t_g the output model. Note that the model ϕ_{g-1} is either an intermediate model resulting from the weaving of a previous join relationship, or it is the initial aspect-oriented model. The term $jrlist$ defined in Formula (G.1) denotes the topologically sorted list of join relationships. The current join relationship j is defined by the formula given in (G.2). Furthermore, the entry point of the behavior chunk is given by the term $entryState$ specified in Formula (G.3). A short hand $targetTransition$ for the target of j, i.e., the transition which is crosscut by the behavior chunk, is defined in Formula (G.4).

$$jrlist = topologicalJrSort(t_{g-1}) \tag{G.1}$$
$$j = firstJr(jrlist) \tag{G.2}$$
$$entryState = source(t_{g-1}, j) \tag{G.3}$$
$$targetTransition = target(t_{g-1}, j) \tag{G.4}$$

Precondition of ϕ_g

In order that the postcondition of ϕ_g can be satisfied when executing the operation, the precondition in (G.5) must be satisfied. It states that the source node of the join relationship must either be a state or a component, and the target must be a transition. Note that the join relationship can also be partial when executing ϕ_g (cf. Section G.2.1).

$$\begin{aligned}&\big(type(entryState) = StateDefinition \lor \\ &type(entryState) = ComponentDefinition\big) \land \\ &type(targetTransition) = TransitionDefinition\end{aligned} \tag{G.5}$$

Postcondition of ϕ_g

The postcondition that must be satisfied in order that a behavior chunk can be woven consists of several predicates which must all evaluate to true so far that the postcondition of ϕ_g to be satisfied. The terms defined in the formulae (G.6)–(G.10) are used to simplify the postcondition predicates in the following: $stateGroup$ is a set of syntax trees describing the nodes of the behavior chunk which will be woven by ϕ_g into the target module.[1] The expression $exitPoints$ specifies the set of exit points of the behavior chunk. Even though it is a *set* of exit points, there can only be one exit point, as defined by the language constraint in Section 7.4.2. The target module $tModule$ denotes the aspect or component which contains the target transition $targetTransition$ after the weaving of the behavior chunk. The term $exitState$ denotes the target state of the crosscut transition.

$$stateGroup = findStateGroupMembers(t_{g-1}, \emptyset, entryState) \tag{G.6}$$
$$exitPoints = exitPoints(stateGroup) \tag{G.7}$$
$$tModule = identicalElement(t_g, targetModule(t_{g-1}, j)) \tag{G.8}$$
$$exitState = target(t_{g-1}, targetTransition) \tag{G.9}$$

The term $cloneMap$ defined in Formula (G.10) denotes the tuple which contains the mapping between the original states, components, and transitions of the behavior chunk and the clone copies. Two different cases have to be distinguished. In the case the join relationship j has a *before* ordering, an additional state and an additional transition have to be introduced besides the cloned elements of the behavior chunk. This is done by the function $createAdditionalExitStateClone$. The resulting map contains a mapping between the exit point of the behavior chunk and an extra state. In the case of an *instead* and *after* join

[1] Remember that the transitions of the state group are contained in the description of the statechart nodes.

relationship, only the states, components, and transitions of the behavior chunk are cloned: there is no mapping between the exit point and an extra state and an extra transition.

$$cloneMap = \begin{cases} createCloneMap(t_{g-1}, stateGroup \backslash exitPoints, & \textbf{if } ordering(j) = \\ \quad createAdditionalExitStateClone(t_{g-1}, exitPoints)) & \quad \texttt{\textbf{before}} \\ & \\ createCloneMap(t_{g-1}, stateGroup \backslash exitPoints, \emptyset) & \textbf{if } ordering(j) \neq \\ & \quad \texttt{\textbf{before}} \end{cases} \tag{G.10}$$

The actual postcondition of ϕ_g consists of several predicates. The first predicate in (G.11) specifies that copies of the states and the corresponding transitions of the behavior chunk are found in the target module after the weaving.

$$\begin{aligned} &\big(\forall x \in stateGroup \backslash exitPoints : \\ &\quad identicalElement(t_g, findClone(x, cloneMap)) \in parts(tModule)\big) \wedge \\ &\big(ordering(j) = \texttt{\textbf{before}} \Rightarrow \forall x \in exitPoints : \\ &\qquad\qquad identicalElement(t_g, findClone(x, cloneMap)) \in parts(tModule)\big) \end{aligned} \tag{G.11}$$

The predicate in (G.12) specifies how the cloned behavior chunk is connected to the crosscut behavior in the case that j is a *before* join relationship (cf. Fig 9.5 (a)). The extra state contained in the clone map is part of the target module. It has an outgoing exit transition that has no condition part but the same action part as the crosscut transition.[2] The exit transition is connected to the target state ($exitState$) of the crosscut transition. Furthermore, the crosscut transition is reassigned to the entry point of the cloned behavior chunk and its action part is deleted.

$$\begin{aligned} &ordering(j) = \texttt{\textbf{before}} \Rightarrow \\ &\quad \big(\forall x \in exitPoints : \\ &\qquad \big(\forall y \in connections(findClone(x, cloneMap)) : \\ &\qquad\quad \exists z = identicalElement(t_g, y) \ : \\ &\qquad\qquad target(t_g, z) = identicalElement(t_g, exitState) \wedge \\ &\qquad\qquad actionPart(z) = actionPart(targetTransition) \wedge \\ &\qquad\qquad conditionPart(z) = \varepsilon \\ &\qquad \big) \\ &\quad \big) \wedge \big(\\ &\qquad \exists reassignedTransition = identicalElement(t_g, targetTransition) : \\ &\qquad\quad conditionPart(reassignedTransition) = \\ &\qquad\qquad conditionPart(targetTransition) \wedge \\ &\qquad\quad actionPart(reassignedTransition) = \varepsilon \wedge \\ &\qquad\quad target(t_g, reassignedTransition) = findClone(entryState, cloneMap) \\ &\quad \big) \end{aligned} \tag{G.12}$$

[2] The extra state and the additional transition are created by the $createCloneMap$ function.

The situation illustrated by Fig. 9.5 (b) is described in Predicate (G.13). It formally describes how the cloned behavior chunk is connected to the crosscut behavior in the case that j has an *instead* ordering. The clone of the exit state's incoming transition is reconnected to the target state of the crosscut transition. The crosscut transition is reassigned to the entry state of the cloned behavior chunk and its action part is deleted.

$$
\begin{aligned}
&ordering(j) = \mathbf{instead} \Rightarrow \\
&\quad \big(\forall x \in exitPoints : \\
&\qquad \big(\forall y \in targetConnections(t_{g-1}, x) : \\
&\qquad\quad \exists z = identicalElement(t_g, findClone(y, cloneMap)) : \\
&\qquad\qquad target(t_g, z) = identicalElement(t_g, exitState) \\
&\qquad \big) \\
&\quad \big) \wedge \big(\\
&\qquad \big(\exists reassignedTransition = identicalElement(t_g, targetTransition) : \\
&\qquad\quad conditionPart(reassignedTransition) = conditionPart(targetTransition) \wedge \\
&\qquad\quad actionPart(reassignedTransition) = \varepsilon \wedge \\
&\qquad\quad target(t_g, reassignedTransition) = findClone(entryState, cloneMap) \\
&\qquad \big) \\
&\quad \big)
\end{aligned}
\tag{G.13}
$$

Figure 9.5 (c) is described in Predicate (G.14) and specifies how the cloned behavior chunk is connected to the crosscut behavior for an *after* join relationship. Basically, the transformation does the same as for the *instead* case, except that the action part is not deleted from the crosscut transition.

$$
\begin{aligned}
&ordering(j) = \mathbf{after} \Rightarrow \\
&\quad \big(\forall x \in exitPoints : \\
&\qquad \big(\forall y \in targetConnections(t_{g-1}, x) : \\
&\qquad\quad \exists z = identicalElement(t_g, findClone(y, cloneMap)) : \\
&\qquad\qquad target(t_g, z) = identicalElement(t_g, exitState) \\
&\qquad \big) \\
&\quad \big) \wedge \big(\\
&\qquad \big(\exists reassignedTransition = identicalElement(t_g, targetTransition) : \\
&\qquad\quad conditionPart(reassignedTransition) = conditionPart(targetTransition) \wedge \\
&\qquad\quad actionPart(reassignedTransition) = actionPart(targetTransition) \wedge \\
&\qquad\quad target(t_g, reassignedTransition) = findClone(entryState, cloneMap) \\
&\qquad \big) \\
&\quad \big)
\end{aligned}
\tag{G.14}
$$

If a target transition is impacted by more than one join relationship, the weaving needs a processing of the subsequently woven join relationships, as illustrated in Fig. 9.6, and Fig. 9.7. The processing is

described by additional predicates. In (G.15) several additional terms are defined. They help to simplify the subsequent predicates.

$$
\begin{aligned}
beforeGroup &= \pi_0(\pi_0(jrlist)) \\
insteadGroup &= \pi_1(\pi_0(jrlist)) \\
afterGroup &= \pi_2(\pi_0(jrlist))
\end{aligned}
\tag{G.15}
$$

The handling of multiple join relationships impacting the same target transition, as illustrated by Fig. 9.6 (a)–(c), is described by Predicate (G.16). It describes how subsequently woven join relationships are handled if j is a *before* join relationship. All join relationships targeting the same transition in t_{g-1} which are woven after j are reassigned to the newly created $exitTransition$[3].

$$
\begin{aligned}
&ordering(j) = \mathbf{before} \Rightarrow \big(\\
&\quad \exists! y \in exitPoints : \\
&\quad\quad \exists! u = identicalElement(t_g, findClone(y, cloneMap)) : \\
&\quad\quad\quad \exists! exitTransition \in connections(u) : \\
&\quad\quad\quad\quad \big(\forall i \in \mathbb{N} : 1 \leq i < arity(beforeGroup) \Rightarrow \\
&\quad\quad\quad\quad\quad target(t_g, identicalElement(t_g, \pi_i(beforeGroup))) = exitTransition \\
&\quad\quad\quad\quad \big) \wedge \\
&\quad\quad\quad\quad \big(\forall i \in \mathbb{N}_0 : 0 \leq i < arity(insteadGroup) \Rightarrow \\
&\quad\quad\quad\quad\quad target(t_g, identicalElement(t_g, \pi_i(insteadGroup))) = exitTransition \\
&\quad\quad\quad\quad \big) \wedge \\
&\quad\quad\quad\quad \big(\forall i \in \mathbb{N}_0 : 0 \leq i < arity(afterGroup) \Rightarrow \\
&\quad\quad\quad\quad\quad target(t_g, identicalElement(t_g, \pi_i(afterGroup))) = exitTransition \\
&\quad\quad\quad\quad \big) \\
&\big)
\end{aligned}
\tag{G.16}
$$

The handling of multiple join relationships impacting the same target transition, illustrated by Fig. 9.7 (a) and (b), is described by Predicate (G.17). It specifies how subsequently woven join relationships are handled if j is an *instead* relationship.

[3] There is only one exit transition in the set.

$$
\begin{aligned}
&ordering(j) = \texttt{\textbf{instead}} \Rightarrow \big(\\
&\quad \exists! x \in exitPoints : \\
&\quad\quad \exists! y \in targetConnections(t_{g-1}, x) : \\
&\quad\quad\quad \exists! exitTransition = identicalElement(t_g, findClone(y, cloneMap)) : \\
&\quad\quad\quad\quad \big(\forall i \in \mathbb{N} : 1 \leq i < arity(insteadGroup) \Rightarrow \big(\\
&\quad\quad\quad\quad\quad \exists jr = identicalElement(t_g, \pi_i(insteadGroup)) : \\
&\quad\quad\quad\quad\quad\quad target(t_g, jr) = exitTransition \wedge \\
&\quad\quad\quad\quad\quad\quad ordering(jr) = \texttt{\textbf{after}} \\
&\quad\quad\quad\quad\quad \big) \wedge \big(\\
&\quad\quad\quad\quad\quad\quad \forall i \in \mathbb{N}_0 : 0 \leq i < arity(afterGroup) \Rightarrow \\
&\quad\quad\quad\quad\quad\quad\quad target(t_g, identicalElement(t_g, \pi_i(afterGroup))) = exitTransition \\
&\quad\quad\quad\quad\quad \big) \\
&\quad\quad\quad\quad \big) \\
&\big)
\end{aligned}
\tag{G.17}
$$

Predicate (G.18) describes the handling of the subsequently woven join relationships if j is an *after* join relationship, as illustrated by Fig. 9.7 (c). The target of such a join relationship is reassigned to the exit transition of the woven crosscutting behavior.

$$
\begin{aligned}
&ordering(j) = \texttt{\textbf{after}} \Rightarrow \big(\\
&\quad \exists! x \in exitPoints : \\
&\quad\quad \exists! y \in targetConnections(t_{g-1}, x) : \\
&\quad\quad\quad \exists! exitTransition = identicalElement(t_g, findClone(y, cloneMap)) : \\
&\quad\quad\quad\quad \big(\forall i \in \mathbb{N} : 1 \leq i < arity(afterGroup) \Rightarrow \\
&\quad\quad\quad\quad\quad target(t_g, identicalElement(t_g, \pi_i(afterGroup))) = exitTransition \\
&\quad\quad\quad\quad \big) \\
&\big)
\end{aligned}
\tag{G.18}
$$

Finally, the woven join relationship j must be removed from the model before the next weaving step is entered, which is describe in Predicate G.19.

$$identicalElement(t_g, j) = \varepsilon \tag{G.19}$$

G.1.2 Formal Weaving Semantics of Scenario Chunks

The weaving semantics of scenario chunks has been informally presented in Section 9.2.2. This section describes the weaving semantics for scenariocharts formally.

Given Elements

The operation ϕ_g weaves a scenario chunk into a target module. There are various elements given for the execution of this operation. The input model of the function is denoted in the following as t_{g-1}, the output model as t_g.[4] The term $jrlist$ in (G.20) denotes the list of topologically sorted join relationships. The term j in Formula (G.21) defines the join relationship j which is woven by ϕ_g. Furthermore, the formulas (G.22) and (G.23) define the root node of the scenario chunk and the target scenario node, respectively.

$$jrlist = topologicalJrSort(t_{g-1}) \tag{G.20}$$
$$j = firstJr(jrlist) \tag{G.21}$$
$$chunkRoot = source(t_{g-1}, j) \tag{G.22}$$
$$targetNode = target(t_{g-1}, j) \tag{G.23}$$

Precondition of ϕ_g

The precondition given by Predicate (G.24) must be fulfilled in order to guarantee the correct execution of ϕ_g for scenario chunks, i.e., the postcondition ϕ_g. It states that the source and the target of the join relationship must be a scenario node. Note that the join relationship can be partial (cf. Section G.2.1).

$$\begin{aligned} &type(chunkRoot) = ScenarioDefinition \wedge \\ &type(targetNode) = ScenarioDefinition \end{aligned} \tag{G.24}$$

Postcondition of ϕ_g

The postcondition of ϕ_g needs to satisfy several predicates. For a simpler definition of them, the auxiliary terms in (G.25)–(G.30) are defined. Term $tModule$ in (G.25) defines the target module, i.e., the aspect or the component which contains the target of the join relationship, in the resulting model t_g. In (G.26), the term $scenarioChunk$ represents all nodes which are part of the scenario chunk. The term $cloneMap_g$ in (G.27) specifies the clone map for all scenario chunk nodes. It has an index, as it is accessed later by the weaving operation which weaves embedded components (cf. Section G.1.5). In (G.28), a short hand for the clone copy of the chunk's root node is given. Formula (G.29) specifies the woven clone $wovenChunkRoot$ of the root chunk node in the resulting model t_g. Finally, in (G.30), the term $targetSiblings$ defines a tuple containing all siblings of the target scenario node also containing the target node.

$$tModule = targetModule(t_g, identicalElement(t_g, targetNode)) \tag{G.25}$$
$$scenarioChunk = findScenarioGroupMembers(t_{g-1}, \emptyset, chunkRoot) \tag{G.26}$$
$$cloneMap_g = createCloneMap(t_{g-1}, scenarioChunk, \emptyset) \tag{G.27}$$
$$clonedChunkRoot = findClone(chunkRoot, cloneMap_g) \tag{G.28}$$
$$wovenChunkRoot = identicalElement(t_g, clonedChunkRoot) \tag{G.29}$$
$$targetSiblings = scenarioSiblings(t_{g-1}, targetNode) \tag{G.30}$$

[4] The model t_{g-1} results either from the weaving of another join relationship or it is the initial model.

The first predicate that must be satisfied by the postcondition of ϕ_g for the weaving of scenario chunks is given in (G.31). It states that the clone copies of the scenario chunk are inserted into the target module. The root of the cloned chunk is connected with the parent of the target node. Furthermore, the cloned chunk root node gets the scenario type of the target node. This predicate must be satisfied for each of the cases described by Figure 9.8 (a)–(e). Note that scenario chunk nodes can be contained in embedded components (cf. Section 7.5.2). However, such nodes are first woven into the target module as direct children, as the embedded component will be woven at a later point in time (cf. Section G.1.5).

$$\begin{aligned}
&\big(\forall x \in \mathit{scenarioChunk} \setminus \{\mathit{chunkRoot}\} : \\
&\qquad \mathit{findClone}(x, \mathit{cloneMap}_g) \in \mathit{parts}(\mathit{tModule})\big) \wedge \\
&\big(\mathit{scenarioParent}(t_g, \mathit{wovenChunkRoot}) = \\
&\qquad\qquad \mathit{identicalElement}(t_g, \mathit{scenarioParent}(t_{g-1}, \mathit{targetNode}))\big) \wedge \\
&\big(\mathit{wovenChunkRoot} \in \mathit{parts}(\mathit{tModule})\big) \wedge \\
&\big(\mathit{scenarioType}(\mathit{wovenChunkRoot}) = \mathit{scenarioType}(\mathit{targetNode})\big)
\end{aligned} \tag{G.31}$$

In the case, where the behavior chunk is woven into a scenario of the type *sequence*, further predicates are needed to describe the weaving semantics. For a simple definition of them, several terms are introduced. First, the term $\mathit{greaterSiblings}$ in (G.32) must be defined. It describes the set of target node siblings that have a sequence number which is equal or greater than the target node. Note that the target node is also contained in the set $\mathit{greaterSiblings}$.

$$\mathit{greaterSiblings} = \begin{cases}
\{x \mid \exists i \in \mathbb{N}_0 : & \textbf{if } \mathit{scenarioType}(\\
\quad 0 \le i < \mathit{arity}(\mathit{targetSiblings}) \wedge & \quad \mathit{targetNode}) = \\
\quad \mathit{seqNo}(\mathit{targetNode}) \le & \quad \texttt{sequence} \\
\qquad \mathit{seqNo}(\pi_i(\mathit{targetSiblings})) \wedge & \\
\quad x = \pi_i(\mathit{targetSiblings})\} & \\
\emptyset & \textbf{else}
\end{cases} \tag{G.32}$$

The weaving semantics for the situation in Fig. 9.8 (a) is described by the Predicate (G.33). It describes how a chunk is woven *before* a target node that has the scenario type *sequence*. The cloned root node of the chunk gets the sequence number of the target scenario node. The sequence numbers of the target scenario as well as the sequence number of its siblings with a greater sequence number are increased by one.

$$\begin{aligned}
&\mathit{scenarioType}(\mathit{targetNode}) = \texttt{sequence} \wedge \mathit{ordering}(j) = \texttt{before} \Rightarrow \big(\\
&\qquad \mathit{seqNo}(\mathit{wovenChunkRoot}) = \mathit{seqNo}(\mathit{targetNode}) \wedge \\
&\qquad \forall y \in \mathit{greaterSiblings} : \\
&\qquad\quad \mathit{seqNo}(y) + 1 = \mathit{seqNo}(\mathit{identicalElement}(t_g, y)) \\
&\big)
\end{aligned} \tag{G.33}$$

For the definition of the *instead* weaving semantics, the auxiliary term $tScenarioGroup$ given in (G.34) is defined. It specifies the *target scenario group* $tScenarioGroup$, i.e., the scenario group, to which the target node belongs.

$$tScenarioGroup = findScenarioGroupMembers(t_{g-1}, \emptyset, targetNode) \tag{G.34}$$

In the case that the target node is the root of a scenario tree, as illustrated Fig. 9.8 (b), the terms in the Formulae (G.35) and (G.36) are required. They specify the incoming ($sRootAssoc$) and outgoing ($tRootAssoc$) associations of the (target) root node, respectively.

$$\begin{aligned} tRootAssoc = \{u | \exists! x \in\ & rootScenarios(t_{g-1}, tScenarioGroup): \\ & \exists z = targetConnections(t_{g-1}, x): \\ & \forall y \in filterSet(z, AssociationDefinition): \\ & y = u\} \end{aligned} \tag{G.35}$$

$$\begin{aligned} sRootAssoc = \{u | \exists! x \in\ & rootScenarios(t_{g-1}, targetScenarioGroup): \\ & \exists z = connections(x): \\ & \forall y \in filterSet(z, AssociationDefinition): \\ & y = u\} \end{aligned} \tag{G.36}$$

Predicate (G.37) actually describes how a scenario chunk impacting a root scenario, as shown in Fig. 9.8 (b), is woven. It defines that the target scenario and the corresponding scenario group are removed from the model. The source and target associations are connected to the woven clone copy of the scenario chunk.

$$\begin{aligned} & ordering(j) = \mathbf{instead} \wedge targetNode \in rootScenarios(t_{g-1}, tScenarioGroup) \Rightarrow \Big(\\ & \qquad \big(\forall x \in tScenarioGroup: \\ & \qquad\qquad identicalElement(t_g, x) = \varepsilon \\ & \qquad \big) \wedge \big(\forall x \in tRootAssoc: \\ & \qquad\qquad target(t_g, identicalElement(t_g, x)) = wovenChunkRoot \\ & \qquad \big) \wedge \big(\forall x \in sRootAssoc: \\ & \qquad\qquad source(t_g, identicalElement(t_g, x)) = wovenChunkRoot \\ & \qquad \big) \\ & \Big) \end{aligned} \tag{G.37}$$

In Predicate (G.38), the weaving semantics for the situation illustrated in Fig. 9.8 (c) is specified formally. It describes the semantics of the weaving of an *instead* join relationship which targets a non-root scenario node of type *sequence*. The cloned scenario chunk root in the woven model has the same sequence number as the target node. Apart from the case where the target node is of type *sequence*, the predicate also handles the *instead* weaving for target nodes which have another type than *sequence*: the target node of any type, e.g., *alternative*, *sequence*, or *parallel*, as well as all child nodes and grand-child nodes are removed from the model.

$$
\begin{aligned}
&ordering(j) = \texttt{instead} \wedge \\
&targetNode \notin rootScenarios(t_{g-1}, tScenarioGroup) \Rightarrow \big(\\
&\quad scenarioType(targetNode) = \texttt{sequence} \Rightarrow \big(\\
&\qquad seqNo(wovenChunkRoot) = seqNo(targetNode) \wedge \\
&\qquad \forall y \in greaterSiblings : \\
&\qquad\quad y \neq targetNode \Rightarrow \\
&\qquad\qquad seqNo(y) = seqNo(identicalElement(t_g, y)) \\
&\quad \big) \wedge \forall y \in findScenarioSubtreeMembers(t_{g-1}, \emptyset, targetNode) : \\
&\qquad\quad identicalElement(t_g, y) = \varepsilon \\
&\big)
\end{aligned} \tag{G.38}
$$

Last but not least, the weaving semantics for a scenario chunk that is woven with an *after* ordering is given in (G.39). The predicate specifies that the cloned root node of the chunk gets the sequence number of the target node, incremented by one. Moreover, the siblings with a greater sequence number than the target node have their sequence number incremented by one.

$$
\begin{aligned}
&scenarioType(targetNode) = \texttt{sequence} \wedge ordering(j) = \texttt{after} \Rightarrow \big(\\
&\qquad seqNo(wovenChunkRoot) = seqNo(targetNode) + 1 \wedge \\
&\qquad \forall y \in greaterSiblings : \\
&\qquad\quad seqNo(y) > seqNo(targetNode) \Rightarrow \big(\\
&\qquad\qquad seqNo(identicalElement(t_g, y)) = seqNo(y) + 1 \\
&\qquad\quad \big) \\
&\big)
\end{aligned} \tag{G.39}
$$

There must be some more predicates which specify the case where multiple join relationships impact the same target node, as illustrated by Figure 9.9 and 9.10. This is necessary in order that the subsequently woven join relationship result in a logically correct model (cf. Section 9.2.2). However, for this post processing, only the cases listed in Fig. 9.10 are relevant, as they give rise to a need for a manipulation of the remaining join relationships.

Two auxiliary terms are introduced to simplify the following postcondition predicates of ϕ_g. The additional terms $insteadGroup$ and $afterGroup$ in formulae (G.40) and (G.41) define the join relationships impacting the same target as j. They specify each as an n-tuple containing the *instead* or an *after* join relationships, respectively.

$$insteadGroup = \pi_1(\pi_0(jrlist) \tag{G.40}$$

$$afterGroup = \pi_2(\pi_0(jrlist) \tag{G.41}$$

The situation where more than one *instead* join relationship impacts the same root scenario node must be handled as illustrated in Fig. 9.10 (a). The formal definition of this semantics is given in Predicate

(G.42). In the case described, the *instead* join relationships with a lower priority than j are removed from the model.

$$\begin{aligned}
&scenarioType(targetNode) = \texttt{sequence} \wedge ordering(j) = \texttt{instead} \wedge \\
&targetNode \in rootScenarios(t_{g-1}, tScenarioGroup) \Rightarrow \big(\\
&\qquad \forall i \in \mathbb{N} : 1 \leq i < arity(insteadGroup) \Rightarrow \\
&\qquad\quad identicalElement(t_g, \pi_i(insteadGroup)) = \varepsilon \\
&\big)
\end{aligned} \tag{G.42}$$

Predicate (G.43) handles the case where two or more *instead* join relationships impact the same target node, as illustrated in Fig. 9.10 (b). The target of any other *instead* join relationship processed after j is reassigned to the woven chunk root node. Moreover, its ordering is set to *after*.

$$\begin{aligned}
&scenarioType(targetNode) = \texttt{sequence} \wedge ordering(j) = \texttt{instead} \wedge \\
&targetNode \notin rootScenarios(t_{g-1}, targetScenarioGroup) \Rightarrow \big(\\
&\qquad \forall i \in \mathbb{N} : 1 \leq i < arity(insteadGroup) \Rightarrow \big(\\
&\qquad\quad \exists! u = identicalElement(t_g, \pi_i(insteadGroup)) : \\
&\qquad\qquad target(t_g, u) = wovenChunkRoot \wedge \\
&\qquad\qquad ordering(u) = \texttt{after} \\
&\qquad \big) \\
&\big)
\end{aligned} \tag{G.43}$$

Figure 9.10 (c) illustrates the semantics for the post processing of *after* join relationships which impact the same scenario node as j with an *instead* ordering. The formal description of the corresponding post processing step is given in Predicate (G.44). The target of all following *after* join relationships is reassigned to the woven clone of the chunk root node.

$$\begin{aligned}
&scenarioType(targetNode) = \texttt{sequence} \wedge ordering(j) = \texttt{instead} \wedge \\
&\qquad \forall i \in \mathbb{N}_0 : 0 \leq i < arity(afterGroup) \Rightarrow \big(\\
&\qquad\quad \exists! u = identicalElement(t_g, \pi_i(afterGroup)) : \\
&\qquad\qquad target(t_g, u) = wovenChunkRoot \\
&\qquad \big) \\
&\big)
\end{aligned} \tag{G.44}$$

Figure 9.10 (d) exemplifies how to handle the situation where two or more *after* join relationships impact the same scenario node. The semantics is similar to the *instead/after* case. It is described in Predicate (G.45).

$$
\begin{aligned}
&\mathit{scenarioType}(\mathit{targetNode}) = \texttt{sequence} \wedge \mathit{ordering}(j) = \texttt{after} \wedge \\
&\qquad \forall i \in \mathbb{N} : 1 \leq i < \mathit{arity}(\mathit{afterGroup}) \Rightarrow \big(\\
&\qquad\qquad \exists! u = \mathit{identicalElement}(t_g, \pi_i(\mathit{afterGroup})) : \\
&\qquad\qquad\qquad \mathit{target}(t_g, u) = \mathit{wovenChunkRoot} \\
&\qquad \big) \\
&)
\end{aligned}
\tag{G.45}
$$

Finally, the woven join relationship j must be removed from the model before the next weaving operation is executed. This fact is specified in Predicate G.46.

$$\mathit{identicalElement}(t_g, j) = \varepsilon \tag{G.46}$$

G.1.3 Formal Weaving Semantics of Crosscutting Statecharts

This section presents the formal the weaving semantics for crosscutting statecharts, which was informally presented in Section 9.2.3. The operation ϕ_m weaves all crosscutting statecharts contained in the model t_{m-1}. It is the first operation after all join relationships are woven. The weaving consists of multiple substeps. First, all aspects in the model are determined, then, the *end target modules* are located for each aspects. This is done by the algorithm specified by the function $\mathit{endTargetModules}$ described in Section 9.1. After the execution of ϕ_m, the crosscutting statecharts are woven into the corresponding end target modules.

Given Elements

The index m represents the number of weaving steps executed previously. It is specified in Formula (G.47). Correspondingly, the model t_{m-1} denotes the model before weaving the crosscutting statecharts.

$$m = \mathit{arity}(\mathit{flatenTuple}(\mathit{topologicalJrSort}(t_0))) + 1 \tag{G.47}$$

Precondition of ϕ_m

The precondition stated in Predicate (G.48) must be fulfilled before the operation ϕ_m is executed. It states, that all join relationships of the aspect-oriented models must be woven.

$$\mathit{arity}(\mathit{flatenTuple}(\mathit{topologicalJrSort}(t_{m-1}))) = 0 \tag{G.48}$$

Postcondition of ϕ_m

For the execution of the weaving operation, all aspects in the model need to be processed. They are given by the term $\mathit{aspects}$ in formula (G.49).

$$\mathit{aspects} = \mathit{gatherAspects}(t_0) \tag{G.49}$$

After executing ϕ_m, the main predicate of the postcondition given in (G.50) must be satisfied. It ensures all crosscutting statecharts contained in the aspects of the given model are woven into the corresponding end target modules. It therefore employs the subpredicate *ccStatechartsWoven*.

$$\begin{aligned}
&\forall i \in \mathbb{N}_0 : 0 \leq i < arity(aspects) \Rightarrow \\
&\quad \exists tModules = endTargetModules(t_0, \pi_i(aspects), \emptyset) : \\
&\quad\quad \forall k \in \mathbb{N}_0 : 0 \leq k < arity(tModules) \Rightarrow \\
&\quad\quad\quad \exists j = (\sum_{0 \leq d < i} arity(endTargetModules(t_0, \pi_d(aspects), \emptyset))) + k + m : \\
&\quad\quad\quad\quad ccStatechartsWoven(\pi_i(aspects), \pi_k(tModules), j)
\end{aligned} \tag{G.50}$$

The subpredicate $ccStatechartsWoven(aspect, targetModule, h)$ ensures that the crosscutting statecharts of a given $aspect$ are woven into the given end target module $targetModule$. The parameter h is the index of the current substep. The model t_{h-1} represents the model before weaving the crosscutting statechart into the target module and t_h the model after the weaving.

Furthermore, several auxiliary terms are defined in the formulas (G.51)–(G.53). They are used to simplify the subpredicates in the following. Formula (G.51) describes the set of nodes and transitions which are part of a crosscutting statechart in the given $aspect$.

$$\begin{aligned}
&crosscuttingStatechartNodes = \\
&\quad \{x | \exists! w = identicalElement(t_{h-1}, aspect) : \\
&\quad\quad \exists y = stateGroups(t_{h-1}, parts(w)) : \\
&\quad\quad\quad \exists z \in y : \\
&\quad\quad\quad\quad \big(|startStates(z)| = 1 \land \\
&\quad\quad\quad\quad \forall v \in z : v = x\big) \\
&\quad \}
\end{aligned} \tag{G.51}$$

Formula (G.52) defines the clone map for all elements that are part of a crosscutting statechart in the $aspect$. In (G.53), a short hand $tModule$ of the target module in the *resulting* model is defined. Thus, $tModule$ is the target module that contains the woven crosscutting statecharts.

$$cloneMap = createCloneMap(t_{h-1}, crosscuttingStatechartNodes, \emptyset) \tag{G.52}$$

$$tModule = identicalElement(t_h, targetModule) \tag{G.53}$$

The actual predicate defining the result of $ccStatechartsWoven$ is given in (G.54) and states that the clone copies of the statecharts of the $aspect$ are contained in $tModule$ after the execution of ϕ_m.

$$\begin{aligned}
&\forall x \in crosscuttingStatechartNodes : \\
&\quad findClone(x, cloneMap) \in parts(tModule)
\end{aligned} \tag{G.54}$$

G.1.4 Formal Weaving Semantics of Crosscutting Scenariocharts

This section presents the formal weaving semantics for crosscutting scenariocharts, which has been informally presented in Section 9.2.4.

Given Elements

The weaving operation ϕ_p weaves the crosscutting scenariocharts. Several elements are given for the formal specification of this operation. The term $aspects$ in (G.55) is shorthand for all aspect modules of the initial model. The term defined by Formula (G.56) describes the number of sub steps d needed for the weaving of the crosscutting statecharts (cf. Section G.1.3). The formula in (G.57) describes the index p of the operation ϕ_p. It is based on the index m, defined in (G.47), and d. Correspondingly, the model before the weaving the crosscutting scenariocharts is denoted as t_{p-1}.

$$aspects = gatherAspects(t_0) \tag{G.55}$$

$$d = \sum_{k=0}^{arity(aspects)} arity(endTargetModules(t_0, \pi_k(aspects), \emptyset)) \tag{G.56}$$

$$p = m + d \tag{G.57}$$

Precondition of ϕ_p

Before ϕ_p can be executed, the weaving of the crosscutting statechart (ϕ_m) must be finished.

Postcondition of ϕ_p

After the execution of ϕ_p, the main predicate of the postcondition given in (G.58) must be satisfied. It ensures that for all aspects in the given model the crosscutting scenariocharts are woven with the corresponding end target modules. It employs the subpredicate *ccScenariochartsWoven*.

$$\begin{aligned}
&\forall i \in \mathbb{N}_0 : 0 \leq i < arity(aspects) \Rightarrow \\
&\quad \exists tModules = endTargetModules(t_0, \pi_i(aspects), \emptyset) : \\
&\quad\quad \forall k \in \mathbb{N}_0 : 0 \leq k < arity(tModules) \Rightarrow \\
&\quad\quad\quad \exists j = (\sum_{0 \leq d < i} arity(endTargetModules(t_0, \pi_d(aspects), \emptyset))) + k + p : \\
&\quad\quad\quad\quad ccScenariochartsWoven(\pi_i(aspects), \pi_k(tModules), j)
\end{aligned} \tag{G.58}$$

The subpredicate $ccScenariochartsWoven(aspect, targetModule, h)$ takes one $aspect$ and one of its end target modules $(targetModule)$ and the index h. This subpredicate ensures that the crosscutting scenariocharts of the $aspect$ are woven into the $targetModule$. The argument h is an index which denotes the step number executed in the transformation process. There are several auxiliary formulae and three main predicates describing this subpredicate. The auxiliary formulas are given in (G.59)–(G.65). In (G.59), the shorthand for the target module after the weaving of the crosscutting scenariochart is defined.

$$tModule = identicalElement(t_h, targetModule) \tag{G.59}$$

The term $ccSNodes_h$ in (G.60) represents the scenario nodes which are part of a crosscutting scenariochart in the current aspect.[5] The term has an index h, because it is accessed again in a later substep, when the embedded components are woven (cf. the weaving of embedded components in Section G.1.5).

$$\begin{aligned}
&ccSNodes_h = \\
&\quad \{x | \exists oAspect = identicalElement(t_{h-1}, aspect) : \\
&\qquad \exists w = filterSet(descendants(oAspect), ScenarioDefinition) : \\
&\qquad\quad \exists y \in scenarioGroups(t_{h-1}, w) : \\
&\qquad\qquad \big(\exists! z \in rootScenarios(t_{h-1}, y) : \\
&\qquad\qquad\quad (filterSet(connections(z), AssociationDefinition) \cup \\
&\qquad\qquad\quad (filterSet(targetConnections(t_{h-1}, z), AssociationDefinition) \neq \emptyset) \\
&\qquad\qquad \big) \wedge \forall z \in y : z = x \\
&\quad \}
\end{aligned} \tag{G.60}$$

Formula (G.61) contains the roots nodes of the set crosscutting scenariochart nodes. The clone map of all scenario nodes is defined in (G.62). The term $cloneMap_h$ has an index h and is therefore accessible for later stages in the weaving process (cf. the weaving of embedded components in Section G.1.5).

$$rootNodes = rootScenarios(t_{h-1}, ccSNodes_h) \tag{G.61}$$

$$cloneMap_h = createCloneMap(t_{h-1}, ccSNodes_h, \emptyset) \tag{G.62}$$

Formula (G.63) defines the associations which are connected to the root nodes of the scenariocharts and which have environment objects or external components as a source. Note that this term is defined for simplifying the next term definition.

$$\begin{aligned}
&tgRootAssocs = \\
&\quad \{y | \forall x \in rootNodes : \\
&\qquad \forall z \in filterSet(targetConnections(t_{h-1}, x), AssociationDefinition) : \\
&\qquad\quad y = z \\
&\quad \}
\end{aligned} \tag{G.63}$$

In (G.64), the set of all incoming and outgoing associations from the scenariochart root nodes are defined. The defined term $rootAssociations_h$ has an index, as it is accessed during later stages of the weaving process (cf. the weaving of environment objects in Section G.1.6).

[5] A crosscutting scenariochart is a scenario group which has at least one association connecting its root node with an environment object (cf. Section 7.5.2).

$$
\begin{aligned}
&rootAssociations_h = \\
&\quad \{y | \forall x \in rootNodes : \\
&\quad\quad \forall z \in \mathit{filterSet}(connections(x), AssociationDefinition) : \\
&\quad\quad\quad x = z \\
&\quad \} \cup tgRootAssocs
\end{aligned} \tag{G.64}
$$

Formula (G.65) specifies the clone map $associationCloneMap_h$ for all nodes, as well as all incoming and outgoing associations of the root nodes. It has an index as it is accessed in subsequent weaving steps (cf. the weaving of environment objects in Section G.1.6).

$$
associationCloneMap_h = createCloneMap(t_{h-1}, rootAssociations, cloneMap_h) \tag{G.65}
$$

In (G.66), the first predicate of $ccScenariochartsWoven$ is given. It describes that the clone copies of the crosscutting scenariochart nodes are contained in the target module after the weaving. Note that the scenario nodes of a crosscutting scenariochart may be part of an embedded component. However, the operation ϕ_p does not handle the placement of the cloned scenario nodes in the corresponding embedded components, as they are cloned later in the weaving process. Instead, they are first placed directly as parts of the end target module.

$$
\begin{aligned}
&\big(\forall x \in ccSNodes_h \setminus rootNodes : \\
&\quad \mathit{findClone}(x, cloneMap_h) \in parts(tModule)\big) \wedge \\
&\big(\forall x \in rootNodes : \\
&\quad identicalElement(t_h, \mathit{findClone}(x, cloneMap_h)) \in parts(tModule)\big)
\end{aligned} \tag{G.66}
$$

The predicate given in (G.67) specifies how the incoming and outgoing associations of the scenariochart root nodes are handled. If the source of an original association is a scenario node, the cloned association is an outgoing association of the cloned scenario node. If the source of an original association is an environment object or an external component, the clone of the association is outgoing from that node.

$$
\begin{aligned}
&\forall x \in rootAssociations_h : \big(\\
&\quad \exists associationSource = source(t_{h-1}, x) :\\
&\quad\quad \exists clonedConn = identicalElement(t_h, findClone(x, associationCloneMap_h)) :\\
&\quad\quad\quad type(associationSource) = ScenarioDefinition \Rightarrow \big(\\
&\quad\quad\quad\quad \exists clonedAssocSource = findClone(associationSource, cloneMap_h) :\\
&\quad\quad\quad\quad\quad \exists clonedRoot = identicalElement(t_h, clonedAssocSource) :\\
&\quad\quad\quad\quad\quad\quad clonedConn \in connections(clonedRoot)\\
&\quad\quad\quad \big) \wedge \big((type(associationSource) = EnvironmentObjectDefinition \vee\\
&\quad\quad\quad\quad (type(associationSource) = ComponentDefinition \wedge\\
&\quad\quad\quad\quad childOfType(associationSource, \mathtt{\textbf{external}}, 1) \neq \varepsilon)) \Rightarrow \big(\\
&\quad\quad\quad\quad\quad clonedConn \in connections(identicalElement(t_h, associationSource))\\
&\quad\quad\quad\quad \big)\\
&\quad\quad\quad \big)\\
&\big)
\end{aligned}
\tag{G.67}
$$

Finally, the Predicate (G.68) states, that the original associations between the environment object (or an external component) and the crosscutting scenariochart of the *aspect* are removed from the model. This is done in the last substep when the crosscutting scenariocharts of the aspect are woven into the last target module.

$$
\begin{aligned}
&\exists l = arity(aspects) :\\
&\quad \big(\sum_{d=0}^{d=l} arity(endTargetModules(t_0, \pi_d(aspects), \emptyset))\big) + p = h \Rightarrow\\
&\quad\quad \forall x \in rootAssociations_h : \big(\\
&\quad\quad\quad identicalElement(t_h, x) = \varepsilon\\
&\quad\quad \big)
\end{aligned}
\tag{G.68}
$$

G.1.5 Formal Weaving Semantics of Embedded Components

The weaving semantics of embedded components has been explained informally in Section 9.2.5 and is specified formally in the present section.

Given Elements

The operation ϕ_q weaves the embedded components contained in the aspects of model t_0 into the corresponding end target modules. For the definition of the pre- and postcondition of ϕ_q, several formulas and predicates are required. The term *aspects* in (G.69) is a shorthand that represents all aspect modules in t_0. Formula in (G.70) defines a term d that is short hand for the number of steps needed for the weaving of the crosscutting scenariocharts, described in Section G.1.4. It calculates the number of end target modules

summed up for each aspect. The formula in G.71 specifies the index q of the operation ϕ_q. It is based on the indices p (cf. Section G.1.4, G.57) and d (G.70). Correspondingly, the model before weaving any embedded components is denoted as t_{q-1}.

$$aspects = gatherAspects(t_0) \tag{G.69}$$

$$d = \sum_{k=0}^{arity(aspects)} arity(endTargetModules(t_0, \pi_k(aspects), \emptyset)) \tag{G.70}$$

$$q = p + d \tag{G.71}$$

Precondition of ϕ_q

Before the operation ϕ_q can be executed, the weaving of the crosscutting scenariocharts (ϕ_p) must be finished.

Postcondition of ϕ_q

The postcondition consists of the main predicate given in (G.72). It ensures that for all aspects in the given model, the embedded components are woven into the target modules. Therefore, it employs the subpredicate $embeddedComponentsWoven$.

$$\begin{aligned}
&\forall i \in \mathbb{N}_0 : 0 \leq i < arity(aspects) \Rightarrow \\
&\quad \exists tModules = endTargetModules(t_0, \pi_i(aspects), \emptyset) : \\
&\quad\quad \forall k \in \mathbb{N}_0 : 0 \leq k < arity(tModules) \Rightarrow \\
&\quad\quad\quad \exists j = (\sum_{0 \leq d < i} arity(endTargetModules(t_0, \pi_d(aspects), \emptyset))) + k + q : \\
&\quad\quad\quad\quad embeddedComponentsWoven(\pi_i(aspects), \pi_k(tModules), j)
\end{aligned} \tag{G.72}$$

The subpredicate $embeddedComponentsWoven(aspect, targetModule, h)$ ensures that all embedded components contained in the $aspect$ are woven into the given end target module $targetModule$. It is based on several formulas and predicates. The argument h denotes the substep for the weaving and the term t_h denotes in the following the model which results from this sub step. In (G.73), the term $tModule$ is specified which is shorthand for the target module contained in the resulting model.

$$tModule = identicalElement(t_h, targetModule) \tag{G.73}$$

The term $eComponents$ in (G.74) describes the embedded components contained in the $aspect$.

$$
\begin{aligned}
& eComponents = \\
& \quad \{x | \exists y \in \mathit{filterSet}(parts(aspect), ComponentDefinition) : \\
& \quad\quad \exists z = identicalElement(t_{h-1}, y) : \\
& \quad\quad\quad \big(\exists conn = connections(z) : \\
& \quad\quad\quad\quad \exists tConn = targetConnections(z, t_{h-1}) : \\
& \quad\quad\quad\quad\quad (\mathit{filterSet}(conn, TransitionDefinition) \cup \\
& \quad\quad\quad\quad\quad \mathit{filterSet}(tConn, TransitionDefinition) = \emptyset)\big) \wedge \\
& \quad\quad x = z \\
& \quad \}
\end{aligned}
\tag{G.74}
$$

The term $cloneMap$ given in (G.75) defines a shorthand for the clone map of all embedded components. Remember that the cloned embedded components do not contain any scenarios, as they are removed when being cloned by the $createCloneMap$ function. This is necessary because scenarios which are part of scenario chunks and crosscutting scenariocharts in the aspect have been cloned and woven separately in the previous steps.

$$
cloneMap = createCloneMap(eComponents, \emptyset) \tag{G.75}
$$

The first predicate of the postcondition given in (G.76) of the operation ϕ_q defines that for each embedded component of the $aspect$, a corresponding clone is copied into the target module of the resulting model.

$$
\begin{aligned}
& \big(\forall x \in eComponents : \\
& \quad \exists z = identicalElement(t_h, \mathit{findClone}(x, cloneMap)) : \\
& \quad\quad z \in parts(tModule)\big) \\
& \big)
\end{aligned}
\tag{G.76}
$$

There are two further predicates which ensure a proper assignment of the crosscutting scenariochart nodes to the woven embedded components. There are some auxiliary formulas defined first. The formula given in (G.77) defines a shorthand c for the index of the clone map used for the weaving of the crosscutting scenariocharts for the $aspect$ and the $targetModule$. The term $ccSNodes$ in (G.78) denotes all scenario nodes of crosscutting statecharts that have been woven into the $targetModule$ from the $aspect$ by function ϕ_p. The term $ccSNodes$ is originally defined in Formula (G.60). The clone map for the woven crosscutting scenariochart nodes is given in (G.79).

$$
c = p + (h - q) \tag{G.77}
$$

$$
ccSNodes = ccSNodes_c \tag{G.78}
$$

$$
scenarioCloneMap = cloneMap_c \tag{G.79}
$$

Crosscutting scenariocharts are woven by the operation ϕ_p (cf. Section G.1.4) as direct subparts into the target module, no matter whether they are part of an embedded component or not. Predicate (G.80)

specifies that the woven crosscutting scenariochart nodes whose originals are a part of an embedded component are moved into the corresponding clone of the embedded components.

$$
\begin{aligned}
& ccSNodes \neq \emptyset \Rightarrow \big(\\
& \quad \forall z \in ccSNodes : \\
& \quad\quad \exists! y = identicalElement(t_{c-1}, z) : \\
& \quad\quad\quad \exists! parent = identicalElement(t_{h-1}, decompositionParent(t_{c-1}, y)) : \\
& \quad\quad\quad\quad type(parent) = ComponentDefinition \Rightarrow \big(\\
& \quad\quad\quad\quad\quad \exists! clonedScenario = findClone(z, scenarioCloneMap) : \\
& \quad\quad\quad\quad\quad\quad \exists! clonedParent = findClone(parent, cloneMap) : \\
& \quad\quad\quad\quad\quad\quad\quad identicalElement(t_h, clonedScenario) \in \\
& \quad\quad\quad\quad\quad\quad\quad\quad parts(identicalElement(t_h, clonedParent)) \\
& \quad\quad\quad\quad \big) \\
& \big)
\end{aligned}
\tag{G.80}
$$

Apart from the nodes of the crosscutting scenariocharts, the nodes of the scenario chunks may also have to be moved into a woven embedded component. For this purpose, the auxiliary term *jrs* in (G.81) that represents the join relationships of the model t_0 has to be defined.

$$
jrs = flatenTuple(topologicalJrSort(t_0)) \tag{G.81}
$$

Predicate (G.82) specifies that the scenario nodes of the woven *scenario chunks* whose originals are embedded in components must be moved into the corresponding embedded component in the target. For this purpose, the scenario chunks and the clone map created in the corresponding step are used.

$$
\begin{aligned}
&\forall k \in \mathbb{N}_0 : 0 \leq k < arity(jrs) \Rightarrow \big(\\
&\quad \exists jr = \pi_k(jrs) : \\
&\quad\quad identicalElement(t_{h-1}, jrHostingAspect(t_0, jr)) = aspect \Rightarrow \big(\\
&\quad\quad\quad \exists srcNode = source(t_0, jr) : \\
&\quad\quad\quad\quad type(srcNode) = ScenarioDefinition \Rightarrow \big(\\
&\quad\quad\quad\quad\quad \forall y \in findScenarioGroupMembers(t_0, \emptyset, srcNode) : \\
&\quad\quad\quad\quad\quad\quad \exists oScNode = identicalElement(t_k, y) : \\
&\quad\quad\quad\quad\quad\quad\quad \exists cScNode = findClone(oScNode, cloneMap_{k+1}) : \\
&\quad\quad\quad\quad\quad\quad\quad\quad \exists wScNode = identicalElement(t_{k+1}, cScNode) : \\
&\quad\quad\quad\quad\quad\quad\quad\quad\quad \exists oParent = decompositionParent(t_k, oScNode) : \\
&\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad type(oParent) = ComponentDefinition \Rightarrow \big(\\
&\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \exists oEmbComp = identicalElement(t_{h-1}, oParent) : \\
&\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \exists cEmbComp = findClone(oEmbComp, cloneMap) : \\
&\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad identicalElement(t_h, wScNode) \in \\
&\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad parts(identicalElement(t_h, cEmbComp)) \\
&\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \big) \\
&\quad\quad\quad\quad \big) \\
&\quad\quad \big) \\
&\big)
\end{aligned}
\tag{G.82}
$$

G.1.6 Formal Weaving Semantics of Environment Objects

The informal weaving semantics definition of the crosscutting environment objects is given in Section 9.2.6. This section will give the corresponding formal description.

Given Elements

The operation ϕ_u weaves the crosscutting environment objects contained in the model t_0 into the crosscut end target modules. For the definition of ϕ_u's pre- and postcondition, several terms and predicates are needed. The term *aspects* in (G.83) is a shorthand term that describes all aspect modules in t_0. The term d in Formula (G.84) represents the number of steps needed for the weaving of embedded components, described in Section G.1.5. It is computed from the number of end target modules summed for each aspect in the model t_0. The formula in G.85 describes the index u of the operation ϕ_u. It is based on the indices q, defined in (G.71), and d. Correspondingly, the model before the weaving of the crosscutting environment objects is denoted as t_{u-1}.

$$aspects = gatherAspects(t_0) \tag{G.83}$$

$$d = \sum_{k=0}^{arity(aspects)} arity(endTargetModules(t_0, \pi_k(aspects), \emptyset)) \tag{G.84}$$

$$u = q + d \tag{G.85}$$

Precondition of ϕ_u

The precondition for executing ϕ_u is that the weaving of the embedded components (ϕ_q) has finished.

Postcondition of ϕ_u

After the execution of ϕ_u, the crosscutting environment objects are woven and the postcondition must be fulfilled. It consists of the main predicate given in (G.86). This predicate ensures that for all aspects in the given model, the associated crosscutting environment objects are woven into the crosscut environment objects.[6] It therefore employs the subpredicate $ccEnvironmentObjectsWoven$.

$$\begin{aligned}
&\forall i \in \mathbb{N}_0 : 0 \leq i < arity(aspects) \Rightarrow \\
&\quad \exists tModules = endTargetModules(t_0, \pi_i(aspects), \emptyset) : \\
&\quad\quad \forall k \in \mathbb{N}_0 : 0 \leq k < arity(tModules) \Rightarrow \\
&\quad\quad\quad \exists j = (\sum_{0 \leq d < i} arity(endTargetModules(t_0, \pi_d(aspects), \emptyset))) + k + u : \\
&\quad\quad\quad\quad ccEnvironmentObjectsWoven(\pi_i(aspects), \pi_k(tModules), j)
\end{aligned} \tag{G.86}$$

The subpredicate $ccEnvironmentObjectsWoven(aspect, targetModule, h)$ ensures that the crosscutting environment objects associated with the $aspect$ are woven. The end target module $targetModule$ contains the crosscutting scenariochart associated with the environment objects in the woven model (cf. operation ϕ_p, Section G.1.4). The index h denotes the number of the weaving substep. Correspondingly, t_{h-1} is the model before the crosscutting environment objects associated with $aspect$ are woven.

For simplifying $ccEnvironmentObjectsWoven$ in the following, some shorthand terms are introduced in (G.87)–(G.91). The index c in (G.87) denotes the index of the weaving step when the crosscutting scenariocharts of the $aspect$ have been woven into the $targetModule$. The term $rAssocs$ in (G.88) denotes the original associations between the root nodes of the crosscutting scenariocharts of the $aspect$ and the environment objects. This term is originally defined in Formula (G.64). Finally, the term $aCloneMap$ in (G.89) denotes the clone map of the associations in $rAssocs$, defined originally by formula (G.65).

$$c = p + h - u \tag{G.87}$$

$$rAssocs = rootAssociations_c \tag{G.88}$$

$$aCloneMap = associationCloneMap_c \tag{G.89}$$

[6] An environment object is associated with an aspect if one of its scenariocharts is connected by an association to the environment object.

The term in (G.90) defines the subset $sEoAssocs \subseteq rAssocs$ of associations which are connected to the root node of a scenariochart that have an environment object as *source* node.

$$\begin{aligned}
&sEoAssocs = \\
&\quad \{x | \exists y \in rAssocs : \\
&\quad\quad type(source(t_{c-1}, y)) = EnvironmentObjectDefinition \wedge \\
&\quad\quad x = y \\
&\quad \}
\end{aligned} \tag{G.90}$$

Correspondingly, in (G.91), the subset $tEoAssocs \subseteq rAssocs$ contains the associations which have an environment object as *target* node.

$$\begin{aligned}
&tEoAssocs = \\
&\quad \{x | \exists y \in rAssocs : \\
&\quad\quad type(target(t_{c-1}, y)) = EnvironmentObjectDefintion \wedge \\
&\quad\quad x = y \\
&\quad \}
\end{aligned} \tag{G.91}$$

The predicates given in (G.92)–(G.95) must be satisfied in order to satisfy the postcondition of sub-predicate *ccEnvironmentObjectsWoven*. Predicate (G.92) states how an association *A* between an environment object *E* and the root node of a woven crosscutting statechart *R* must be processed, in the case that *E* is the source of *A*. *A* is cloned as many times as there are join relationships outgoing from *E* and crosscutting other environment objects. The clones are connected to the corresponding targets of the join relationships. In terms of Fig. 9.17 and assuming that *E1* is the source of γ and δ, this predicate describes the creation of the clones ω, σ, η, and φ and their connecting to the corresponding target environment object.

$$\begin{aligned}
&\forall sAssoc \in sEoAssocs : \\
&\quad \exists eojrs = findEoJrs(t_c, sAssoc) : \\
&\quad\quad \forall k \in \mathbb{N}_0 : 0 \leq k < arity(eojrs) \Rightarrow \\
&\quad\quad\quad \exists! jr = identicalElement(t_{h-1}, \pi_k(eojrs)) : \\
&\quad\quad\quad\quad \exists! clonedAssoc = \\
&\quad\quad\quad\quad\quad identicalElement(t_{h-1}, findClone(sAssoc, aCloneMap)) : \\
&\quad\quad\quad\quad\quad \exists! newCloneMap = createCloneMap(t_{h-1}, \{clonedAssoc\}, \emptyset) : \\
&\quad\quad\quad\quad\quad\quad \exists! reClonedAssoc = findClone(clonedAssoc, newCloneMap) : \\
&\quad\quad\quad\quad\quad\quad\quad \exists! newClonedAssoc = identicalElement(t_h, reClonedAssoc) : \\
&\quad\quad\quad\quad\quad\quad\quad\quad identicalElement(t_h, target(t_{h-1}, jr)) = \\
&\quad\quad\quad\quad\quad\quad\quad\quad\quad source(t_h, newClonedAssoc) \wedge \\
&\quad\quad\quad\quad\quad\quad\quad\quad target(t_h, newClonedAssoc) = \\
&\quad\quad\quad\quad\quad\quad\quad\quad\quad identicalElement(t_h, target(t_{h-1}, clonedAssoc))
\end{aligned} \tag{G.92}$$

The predicate in (G.93) specifies the equivalent weaving semantics for the situation where the crosscutting environment object is the target of the association.

$$
\begin{aligned}
&\forall tAssoc \in tEoAssocs : \\
&\quad \exists eojrs = findEoJrs(t_c, tAssoc) : \\
&\quad\quad \forall k \in \mathbb{N}_0 : 0 \leq k < arity(eojrs) \Rightarrow \\
&\quad\quad\quad \exists! jr = identicalElement(t_{h-1}, \pi_k(eojrs)) : \\
&\quad\quad\quad\quad \exists! clonedAssoc = \\
&\quad\quad\quad\quad\quad identicalElement(t_{h-1}, findClone(tAssoc, aCloneMap)) : \\
&\quad\quad\quad\quad\quad \exists! newCloneMap = createCloneMap(t_{h-1}, \{clonedAssoc\}, \emptyset) : \\
&\quad\quad\quad\quad\quad\quad \exists! reClonedAssoc = findClone(clonedAssoc, newCloneMap) : \\
&\quad\quad\quad\quad\quad\quad\quad \exists! newClonedAssoc = identicalElement(t_h, reClonedAssoc) : \\
&\quad\quad\quad\quad\quad\quad\quad\quad identicalElement(t_h, target(t_{h-1}, jr)) = \\
&\quad\quad\quad\quad\quad\quad\quad\quad\quad target(t_h, newClonedAssoc) \wedge \\
&\quad\quad\quad\quad\quad\quad\quad\quad source(t_h, newClonedAssoc) = \\
&\quad\quad\quad\quad\quad\quad\quad\quad\quad identicalElement(t_h, source(t_{h-1}, clonedAssoc))
\end{aligned}
\tag{G.93}
$$

Predicate (G.94) and (G.95) specify the removal of the associations introduced when weaving the crosscutting scenariocharts. They also define that the crosscutting environment objects and the corresponding join relationships must be removed. The removal of these elements must not be executed before the last end target module of the *aspect* is processed. With respect to Fig. 9.17, Predicate (G.94) denotes the removal of γ and δ, *E1*, and the join relationships, under the assumption that they have *E1* as source. Predicate (G.95) describes the same semantics for the case where the associations γ and δ have the environment object *E1* as target.

$$
\begin{aligned}
&sEoAssocs \neq \emptyset \wedge \\
&\quad \exists tm = endTargetModules(t_0, aspect, \emptyset) : \\
&\quad\quad \pi_{arity(tm)-1}(tm) = targetModule \Rightarrow \\
&\quad\quad\quad \forall sAssoc \in sEoAssocs : \\
&\quad\quad\quad\quad \big(arity(findEoJrs(t_{c-1}, sAssoc)) > 0 \Rightarrow \\
&\quad\quad\quad\quad\quad identicalElement(t_h, source(t_{c-1}, sAssoc)) = \varepsilon\big) \wedge \\
&\quad\quad\quad\quad \big(\exists! clonedAssoc = \\
&\quad\quad\quad\quad\quad identicalElement(t_{h-1}, findClone(sAssoc), aCloneMap) : \\
&\quad\quad\quad\quad\quad\quad clonedAssoc \neq \varepsilon \wedge \\
&\quad\quad\quad\quad\quad\quad identicalElement(t_h, clonedAssoc) = \varepsilon \wedge \\
&\quad\quad\quad\quad\quad\quad targetRole(t_h, clonedAssoc) = \varepsilon\big) \wedge \\
&\quad\quad\quad\quad \big(\exists eojrs = findEoJrs(t_{c-1}, sAssoc) : \\
&\quad\quad\quad\quad\quad \forall k \in \mathbb{N}_0 : 0 \leq k < arity(eojrs) \Rightarrow \\
&\quad\quad\quad\quad\quad\quad identicalElement(t_h, \pi_k(eojrs)) = \varepsilon\big)
\end{aligned}
\tag{G.94}
$$

$$
\begin{aligned}
&tEoAssocs \neq \emptyset \wedge \\
&\quad \exists tm = endTargetModules(t_0, aspect, \emptyset) : \\
&\quad\quad \pi_{arity(tm)-1}(tm) = targetModule \Rightarrow \\
&\quad\quad\quad \forall tAssoc \in tEoAssocs : \\
&\quad\quad\quad\quad \big(arity(findEoJrs(t_c, tAssoc)) > 0 \Rightarrow \\
&\quad\quad\quad\quad\quad identicalElement(t_h, target(t_{c-1}, tAssoc)) = \varepsilon\big) \wedge \\
&\quad\quad\quad\quad \big(\exists! clonedAssoc = \\
&\quad\quad\quad\quad\quad identicalElement(t_{h-1}, findClone(tAssoc), aCloneMap) : \\
&\quad\quad\quad\quad\quad\quad clonedAssoc \neq \varepsilon \wedge \\
&\quad\quad\quad\quad\quad\quad identicalElement(t_h, clonedAssoc) = \varepsilon \wedge \\
&\quad\quad\quad\quad\quad\quad targetRole(t_h, clonedAssoc) = \varepsilon\big) \wedge \\
&\quad\quad\quad\quad \big(\exists eojrs = findEoJrs(t_c, tAssoc) : \\
&\quad\quad\quad\quad\quad \forall k \in \mathbb{N}_0 : 0 \leq k < arity(eojrs) \Rightarrow \\
&\quad\quad\quad\quad\quad\quad identicalElement(t_h, \pi_k(eojrs)) = \varepsilon\big)
\end{aligned}
\tag{G.95}
$$

G.1.7 Formal Weaving Semantics of the Functional Speci cation

This section presents the formal weaving semantics for the functional specification of aspects which was presented informally in Section 9.2.7.

Given Elements

The operation ϕ_v weaves the functional specifications contained in the aspects of a model into the corresponding end target modules. For the definition of the pre- and postcondition of ϕ_v, several terms and predicates are required. The term *aspects* in (G.96) is a shorthand term that represents all aspect modules in the aspect-oriented model t_0. In (G.97) the number of steps d of the previous operation ϕ_u is calculated. The term in G.98 describes the index v of the operation ϕ_v. It is based on the index u, originally defined in (G.85), and on d. The term t_{v-1} stands for the model before weaving the functional specification.

$$aspects = gatherAspects(t_0) \tag{G.96}$$

$$d = \sum_{k=0}^{arity(aspects)} arity(endTargetModules(t_0, \pi_k(aspects), \emptyset)) \tag{G.97}$$

$$v = u + d \tag{G.98}$$

Precondition of ϕ_v

The operation ϕ_u must be finished before ϕ_v can start.

Postcondition of ϕ_v

There are various predicates that must be satisfied by the operation ϕ_v. In (G.99), the main predicate for the postcondition of ϕ_v is given. It ensures that the functional specification of each aspect is woven into each of its *end target modules*. The actual postcondition is specified in detail by the subpredicate $functionalSpecificationWoven$.

$$\begin{aligned}
&\forall i \in \mathbb{N}_0 : 0 \leq i < arity(aspects) \Rightarrow \\
&\quad \exists tModules = endTargetModules(t_0, \pi_i(aspects), \emptyset) : \\
&\quad\quad \forall k \in \mathbb{N}_0 : 0 \leq k < arity(tModules) \Rightarrow \\
&\quad\quad\quad \exists j = (\sum_{0 \leq d < i} arity(endTargetModules(t_0, \pi_d(aspects), \emptyset))) + k + v : \\
&\quad\quad\quad\quad functionalSpecificationWoven(\pi_i(aspects), \pi_k(tModules), j)
\end{aligned} \tag{G.99}$$

The subpredicate $functionalSpecificationWoven(oAspect, targetModule, h)$ describes the postcondition for the weaving of the functional specification of $oAspect$ into the target module $targetModule$. The argument h denotes the index of the current substep. The model t_{h-1} denotes the model before weaving aspects into $targetModule$, t_h is the model after the weaving.

For specifying the subpredicate, a set of shorthand terms are required, which are defined in the following. The following formulae (G.100)–(G.108) define terms that describe tuples containing the elements of an aspect's functional specification.

$$aspect = identicalElement(t_{h-1}, oAspect) \tag{G.100}$$
$$aFunctionalSpec = functionalSpec(aspect) \tag{G.101}$$
$$aProvides = provides(aFunctionalSpec) \tag{G.102}$$
$$aRequires = requires(aFunctionalSpec) \tag{G.103}$$
$$aProperties = standardizedProperties(aFunctionalSpec) \tag{G.104}$$
$$aInvariants = invariants(aFunctionalSpec) \tag{G.105}$$
$$aDataType = dataTypeDeclarations(aFunctionalSpec) \tag{G.106}$$
$$aAttributes = attributes(aFunctionalSpec) \tag{G.107}$$
$$aOperations = operations(aFunctionalSpec) \tag{G.108}$$

The formulae (G.109)–(G.117) define the shorthand for elements of the target module's functional specification.

$$tModule = identicalElement(t_{h-1}, targetModule) \tag{G.109}$$
$$tFunctionalSpec = functionalSpec(tModule) \tag{G.110}$$
$$tProvides = provides(tFunctionalSpec) \tag{G.111}$$
$$tRequires = requires(tFunctionalSpec) \tag{G.112}$$
$$tProperties = invariants(tFunctionalSpec) \tag{G.113}$$
$$tInvariants = standardizedProperties(tFunctionalSpec) \tag{G.114}$$
$$tDataType = dataTypeDeclarations(tFunctionalSpec) \tag{G.115}$$
$$tAttributes = attributes(tFunctionalSpec) \tag{G.116}$$
$$tOperations = operations(tFunctionalSpec) \tag{G.117}$$

The terms (G.118)–(G.125) specify the elements of the woven functional specification of the $tModule$ in the resulting model t_h.

$$wFunctionalSpec = functionalSpec(identicalElement(t_h, tModule)) \tag{G.118}$$
$$wProvides = provides(wFunctionalSpec) \tag{G.119}$$
$$wRequires = requires(wFunctionalSpec) \tag{G.120}$$
$$wInvariants = invariants(wFunctionalSpec) \tag{G.121}$$
$$wProperties = standardizedProperties(wFunctionalSpec) \tag{G.122}$$
$$wDataType = dataTypeDeclarations(wFunctionalSpec) \tag{G.123}$$
$$wAttributes = attributes(wFunctionalSpec) \tag{G.124}$$
$$wOperations = operations(wFunctionalSpec) \tag{G.125}$$

For each part of the functional specification, such as the attributes, the invariants, etc., a predicate has to be defined, which describes the result of the weaving operation. Predicate (G.126) defines the postcondition for the *provides* elements. It states that all elements declared in the list of provided elements of the aspect's functional specification are appended to the list of provided elements of the target module. The predicates (G.127)–(G.132) describe the weaving semantics correspondingly for the other elements of the functional specification.

$$\begin{aligned}
&\forall k \in \mathbb{N}_0 : 0 \le k < arity(aProvides) + arity(tProvides) \Rightarrow \big(\\
&\quad (k < arity(tProvides) \Rightarrow \\
&\qquad\qquad \pi_k(tProvides) = \pi_k(wProvides)) \wedge \\
&\quad (k \ge arity(tProvides) \Rightarrow \\
&\qquad\qquad \pi_{k-arity(tProvides)}(aProvides) = \pi_k(wProvides)) \\
&\big)
\end{aligned} \tag{G.126}$$

$$\begin{aligned}
&\forall k \in \mathbb{N}_0 : 0 \leq k < arity(aRequires) + arity(tRequries) \Rightarrow \big(\\
&\quad (k < arity(tRequires) \Rightarrow \\
&\qquad \pi_k(tRequires) = \pi_k(wRequries)) \wedge \\
&\quad (k \geq arity(tRequires) \Rightarrow \\
&\qquad \pi_{k-arity(tRequires)}(aRequires) = \pi_k(wRequires)) \\
&\big)
\end{aligned} \tag{G.127}$$

$$\begin{aligned}
&\forall k \in \mathbb{N}_0 : 0 \leq k < arity(aInvariants) + arity(tInvariants) \Rightarrow \big(\\
&\quad (k < arity(tInvariants) \Rightarrow \\
&\qquad \pi_k(tInvariants) = \pi_k(wInvariants)) \wedge \\
&\quad (k \geq arity(tInvariants) \Rightarrow \\
&\qquad \pi_{k-arity(tInvariants)}(aInvariants) = \pi_k(wInvariants)) \\
&\big)
\end{aligned} \tag{G.128}$$

$$\begin{aligned}
&\forall k \in \mathbb{N}_0 : 0 \leq k < arity(aProperties) + arity(tProperties) \Rightarrow \big(\\
&\quad (k < arity(tProperties) \Rightarrow \\
&\qquad \pi_k(tProperties) = \pi_k(wProperties)) \wedge \\
&\quad (k \geq arity(tProperties) \Rightarrow \\
&\qquad \pi_{k-arity(tProperties)}(aProperties) = \pi_k(wProperties)) \\
&\big)
\end{aligned} \tag{G.129}$$

$$\begin{aligned}
&\forall k \in \mathbb{N}_0 : 0 \leq k < arity(aDataTypes) + arity(tDataTypes) \Rightarrow \big(\\
&\quad (k < arity(tDataTypes) \Rightarrow \\
&\qquad \pi_k(tDataTypes) = \pi_k(wDataTypes)) \wedge \\
&\quad (k \geq arity(tDataTypes) \Rightarrow \\
&\qquad \pi_{k-arity(tDataTypes)}(aDataTypes) = \pi_k(wDataTypes)) \\
&\big)
\end{aligned} \tag{G.130}$$

$$\begin{aligned}
&\forall k \in \mathbb{N}_0 : 0 \leq k < arity(aAttributes) + arity(tAttributes) \Rightarrow \big(\\
&\quad (k < arity(tAttributes) \Rightarrow \\
&\qquad \pi_k(tAttributes) = \pi_k(wAttributes)) \wedge \\
&\quad (k \geq arity(tAttributes) \Rightarrow \\
&\qquad \pi_{k-arity(tAttributes)}(aAttributes) = \pi_k(wAttributes)) \\
&\big)
\end{aligned} \tag{G.131}$$

$$
\begin{aligned}
&\forall k \in \mathbb{N}_0 : 0 \leq k < arity(aOperations) + arity(tOperations) \Rightarrow \Big(\\
&\quad (k < arity(tOperations) \Rightarrow \\
&\qquad \pi_k(tOperations) = \pi_k(wOperations)) \land \\
&\quad (k \geq arity(tOperations) \Rightarrow \\
&\qquad \pi_{k-arity(tOperations)}(aOperations) = \pi_k(wOperations)) \\
&\Big)
\end{aligned}
\tag{G.132}
$$

G.1.8 Formal Weaving Semantics of Server Components

The formal definition for the weaving of server components is given informally in Section 9.2.8. This section presents the corresponding formal specification.

Given Elements

The operation ϕ_w weaves the associations between aspects and server components into the target modules of the aspects and adapts the role names referred to by the modules involved. For the definition of the pre- and postcondition of ϕ_w, several formulae and predicates are given. The term $aspects$ in (G.133) is a shorthand term that describes all aspect modules in the aspect-oriented model t_0. The shorthand term d in (G.134) describes the number of steps needed for the weaving of the functional specification, described in Section G.1.7. It sums up the number of end target modules for each aspect of the given model. The term in G.135 describes the index w of the operation ϕ_w. It is based on the indices v, defined in (G.98), and d. Correspondingly, the model before weaving the server components is denoted as t_{w-1}, the model after the weaving as t_w.

$$aspects = gatherAspects(t_0) \tag{G.133}$$

$$d = \sum_{k=0}^{arity(aspects)} arity(endTargetModules(t_0, \pi_k(aspects), \emptyset)) \tag{G.134}$$

$$w = v + d \tag{G.135}$$

Precondition of ϕ_w

The operation ϕ_v must be finished before ϕ_w can be executed.

Postcondition of ϕ_w

After the execution of ϕ_w, the associations between each aspect and the associated server components are woven with the corresponding end target modules and the behavior description is adapted. The postcondition of ϕ_w is described by the main predicate given in (G.136). It ensures that for each combination of an aspect, a target module, and an association connecting the aspect with a server component, the association is woven correctly in the resulting model. For this purpose, the subpredicate $scWoven$ is employed. Note that $calcIndex$ in Predicate (G.136) is a helper function that is used to compute the corresponding index

of the substep. Furthermore, the term $numScSubSteps$ in (G.137) describes the total number of weaving substeps. It is used by the subpredicate $scWoven$ to calculate the index of the model which results after the weaving of all server components.

$$\begin{aligned}
&\forall l \in \mathbb{N}_0 : 0 \leq l < arity(aspects) \Rightarrow \\
&\quad \exists tModules = endTargetModules(t_0, \pi_l(aspects), \emptyset) : \\
&\quad\quad \forall i \in \mathbb{N}_0 : 0 \leq i < arity(tModules) \Rightarrow \\
&\quad\quad\quad \exists assocs = serverComponentAssociations(t_0, \pi_i(aspects)) : \\
&\quad\quad\quad\quad \forall k \in \mathbb{N}_0 : 0 \leq k < arity(assocs) \Rightarrow \\
&\quad\quad\quad\quad\quad scWoven(\pi_l(aspects), \pi_i(tModules), \pi_k(Assocs), \\
&\quad\quad\quad\quad\quad\quad\quad calcIndex(aspects, l, i, k), numScSubSteps)
\end{aligned} \tag{G.136}$$

where $calcIndex(aspects, l, i, k) =$

$$\sum_{0 \leq d < l} \big(arity(endTargetModules(t_0, \pi_d(aspects), \emptyset)) \big(arity(serverComponentAssociations(t_0, \pi_d(aspects))))\big) + (i-1)\big(arity(serverComponentAssociations(t_0, \pi_l(aspects)))\big) + k$$

$$numScSubSteps = \sum_{0 \leq d < arity(aspects)} \big(arity(endTargetModules(t_0, \pi_d(aspects), \emptyset))\big) \big(arity(serverComponentAssociations(t_0, \pi_d(aspects)))\big) \tag{G.137}$$

The subpredicate $scWoven(oAspect, oTargetModule, oAssoc, h, j)$ ensures that the original association $oAssoc$ between a given $oAspect$ and a server component is woven for the given target module $oTargetModule$. The argument h is the index of the current weaving substep. Correspondingly, the input model of a substep is t_{h-1} and the resulting model t_h. The argument j denotes the number of substeps that need to be executed to weave all server components of a model. Thus, t_{h+j} represents the model after the weaving of the server components.

The subpredicate $scWoven$ consists of several terms and predicates. The term $aspect$ in (G.138) denotes the aspect $oAspect$ before the weaving of its associations. In (G.139), the shorthand term for the end target module contained in the resulting model is given. The term $aCloneMap$ in (G.141) defines the clone map for the given association. It provides the mapping between the original association and the clone of the association between the server component and the target module $tModule$. In (G.142), the term $associationClone$ defines the clone of the $association$.

$$aspect = identicalElement(t_{h-1}, oAspect) \tag{G.138}$$
$$tModule = identicalElement(t_h, oTargetModule) \tag{G.139}$$
$$association = identicalElement(t_{h-1}, oAssoc) \tag{G.140}$$
$$aCloneMap = createCloneMap(t_{h-1}, \{association\}, \emptyset) \tag{G.141}$$
$$associationClone = findClone(t_{h-1}, aCloneMap) \tag{G.142}$$

The term $severComponent$ in formula (G.143) denotes the server component which is connected by the given $association$ to the given $aspect$.

$$serverComponent = \begin{cases} source(t_{h-1}, association) & \textbf{if } type(target(t_{h-1}, association)) = \\ & \quad AspectDefinition \\ target(t_{h-1}, association) & \textbf{else} \end{cases} \tag{G.143}$$

The predicates in (G.144)–(G.147) describe how the cloned association is connected to the server component and the given target module in the resulting model t_h. Predicates (G.144) and (G.145) describe the case where the server component is the source of the association. Predicates (G.146) and (G.147) describe the case where the aspect is the source.

$$\begin{aligned} &source(t_{h-1}, association) = serverComponent \Rightarrow \\ &\quad source(t_h, associationClone) = identicalElement(t_h, serverComponent) \end{aligned} \tag{G.144}$$

$$\begin{aligned} &target(t_{h-1}, association) = aspect \Rightarrow \\ &\quad target(t_h, associationClone) = tModule \end{aligned} \tag{G.145}$$

$$\begin{aligned} &target(t_{h-1}, association) = serverComponent \Rightarrow \\ &\quad target(t_h, associationClone) = identicalElement(t_h, serverComponent) \end{aligned} \tag{G.146}$$

$$\begin{aligned} &source(t_{h-1}, association) = aspect \Rightarrow \\ &\quad source(t_h, associationClone) = tModule \end{aligned} \tag{G.147}$$

Apart from specifying the source and the target of the cloned association, the role of the *clonedAssociation* in model t_h must be substituted with roles names that are unique in the model. Formula (G.148) and (G.149) create a source and a target role which are based on the roles attached to $association$ but which have a unique name with respect to the model t_{h-1}.

$$\begin{aligned} clonedSourceRole = generateUniqueRole(&t_{h-1}, association, \\ &sourceRole(association)) \end{aligned} \tag{G.148}$$

$$\begin{aligned} clonedTargetRole = generateUniqueRole(&t_{h-1}, association, \\ &targetRole(t_{h-1}, association)) \end{aligned} \tag{G.149}$$

The predicates (G.151) and (G.151) specify that the cloned source and target role are attached as roles of the cloned association in model t_h.

$$sourceRole(identicalElement(t_h, associationClone)) = clonedSourceRole \tag{G.150}$$

$$targetRole(identicalElement(t_h, associationClone)) = clonedTargetRole \tag{G.151}$$

The original $association$ role names must be replaced by the generated roles names, in every *receive* and *send* statement of transitions, the functional specifications, and the transform expressions of scenarios. The formulae (G.152) and (G.153) define the term $clonedSourceRoleName$ and $clonedTargetRoleName$ that represent the bare role names[7] as special identifier or qualified identifier. Formulae (G.154) and (G.155) contain the bare source and target role name of the original $association$.

$$clonedSrcRoleName = roleName(clonedSourceRole) \tag{G.152}$$

$$clonedTgtRoleName = roleName(clonedTargetRole) \tag{G.153}$$

$$srcRoleName = roleName(sourceRole(association)) \tag{G.154}$$

$$tgtRoleName = roleName(targetRole(t_{h-1}, association)) \tag{G.155}$$

In the following formulas, several terms are defined in order to simplify the predicates which specify the replacing of the roles in the behavior description of the server component and the target module. In (G.156), the term $targetModuleParts$ specifies all parts in the target module in t_{h-1}.

$$targetModuleParts = parts(identicalElement(t_{h-1}, oTargetModule)) \tag{G.156}$$

The formula (G.157) defines the term $targetModuleTransitions$ which represents all transitions outgoing from the elements that are contained as direct children in the target module.[8]

$$\begin{aligned} & targetModuleTransitions = \\ & \quad \{x | \exists y \in targetModuleParts : \\ & \qquad \exists z \in filterSet(connections(y), TransitionDefinition) : \\ & \qquad\quad x = z \\ & \quad \} \end{aligned} \tag{G.157}$$

Correspondingly to the parts and transitions of the target module, the parts of the server component and the corresponding transitions are defined by the terms in formulae (G.158) and (G.159).

$$serverComponentParts = parts(identicalElement(t_{h-1}, serverComponent)) \tag{G.158}$$

$$\begin{aligned} & serverComponentTransitions = \\ & \quad \{x | \exists y \in serverComponentParts : \\ & \qquad \exists z \in filterSet(connections(y), TransitionDefinition) : \\ & \qquad\quad x = z \\ & \quad \} \end{aligned} \tag{G.159}$$

[7] That means without the cardinalities of the role.

[8] Note that only the transitions outgoing from the direct children in the target module are in a position to access the roles of the woven association. In contrast, transitions contained in the children are not allowed to refer to the $association$ roles.

The term in (G.160) unifies the transitions contained in the server component and the target module into one set. It is used by the following predicates.

$$oTransitions = targetModuleTransitions \cup serverComponentTransitions \tag{G.160}$$

The following two predicates specify that all referred role names of the original $association$, contained in the transitions of the server component and the current target module, are replaced by the role names of the cloned transition. This semantics is specified for receive statements in (G.161) and for send statements in (G.162).

$$\begin{aligned}
&\forall oTrans \in oTransitions : \big(\\
&\quad \exists wTrans = identicalElement(t_h, oTrans) : \\
&\quad \exists eTrans = identicalElement(t_{h+j}, oTrans) : \\
&\quad \exists oRecs = orderedChildrenOfType(oTrans, MessageReceive, \emptyset) : \\
&\quad \exists wRecs = orderedChildrenOfType(wTrans, MessageReceive, \emptyset) : \\
&\quad \exists eRecs = orderedChildrenOfType(eTrans, MessageReceive, \emptyset) : \\
&\quad\; \big(\forall i \in \mathbb{N}_0 : 0 \leq i < arity(eRecs) \Rightarrow \exists eChn = channelNames(\pi_i(eRecs)) : \\
&\qquad\quad (\forall k \in \mathbb{N}_0 : 0 \leq k < arity(eChn) \Rightarrow \\
&\qquad\qquad \pi_k(eChn) \neq srcRoleName \wedge \pi_k(eChn) \neq tgtRoleName)\big) \wedge \\
&\quad\; \forall i \in \mathbb{N}_0 : 0 \leq i < arity(oRecs) \Rightarrow \big(\\
&\qquad \exists oChn = channelNames(\pi_i(oRecs)) : \\
&\qquad\quad \forall l \in \mathbb{N}_0 : 0 \leq l < arity(oChn) \Rightarrow \big(\\
&\qquad\qquad \exists wChn = channelNames(\pi_i(wRecs)) : \\
&\qquad\qquad\quad \pi_l(oChn) = srcRoleName \Rightarrow \big(\\
&\qquad\qquad\qquad \big(\exists k \in \mathbb{N}_0 : 0 \leq k < arity(wChn) \wedge \pi_k(wChn) = clonedSrcRoleName\big) \\
&\qquad\qquad\quad \big) \wedge \pi_l(oChn) = tgtRoleName \Rightarrow \big(\\
&\qquad\qquad\qquad \big(\exists k \in \mathbb{N}_0 : 0 \leq k < arity(wChn) \wedge \pi_k(wChn) = clonedTgtRoleName\big) \\
&\qquad\qquad\quad \big)\big)\big)\big)
\end{aligned} \tag{G.161}$$

$$
\begin{aligned}
&\forall oTrans \in oTransitions : \big(\\
&\quad \exists wTrans = identicalElement(t_h, oTrans) : \\
&\quad \exists eTrans = identicalElement(t_{h+j}, oTrans) : \\
&\quad \exists oSends = orderedChildrenOfType(oTrans, MessageSend, \emptyset) : \\
&\quad \exists wSends = orderedChildrenOfType(wTrans, MessageSend, \emptyset) : \\
&\quad \exists eSends = orderedChildrenOfType(eTrans, MessageSend, \emptyset) : \\
&\quad\; \big(\forall i \in \mathbb{N}_0 : 0 \leq i < arity(eSends) \Rightarrow \exists eChn = channelNames(\pi_i(eSends)) : \\
&\qquad\; (\forall k \in \mathbb{N}_0 : 0 \leq k < arity(eChn) \Rightarrow \\
&\qquad\quad \pi_k(eChn) \neq srcRoleName \wedge \pi_k(eChn) \neq tgtRoleName)\big) \wedge \\
&\quad\; \forall i \in \mathbb{N}_0 : 0 \leq i < arity(oSends) \Rightarrow \big(\\
&\qquad \exists oChn = channelNames(\pi_i(oSends)) : \\
&\qquad\; \forall l \in \mathbb{N}_0 : 0 \leq l < arity(oChn) \Rightarrow \big(\\
&\qquad\quad \exists wChn = channelNames(\pi_i(wSends)) : \\
&\qquad\quad\; \pi_l(oChn) = srcRoleName \Rightarrow \big(\\
&\qquad\qquad \big(\exists k \in \mathbb{N}_0 : 0 \leq k < arity(wChn) \wedge \pi_k(wChn) = clonedSrcRoleName\big) \\
&\qquad\quad\; \big) \wedge \pi_l(oChn) = tgtRoleName \Rightarrow \big(\\
&\qquad\qquad \big(\exists k \in \mathbb{N}_0 : 0 \leq k < arity(wChn) \wedge \pi_k(wChn) = clonedTgtRoleName\big) \\
&\qquad\quad\; \big)\big)\big)\big)
\end{aligned}
\tag{G.162}
$$

The term $oTargetFuncSpec$ in Formula (G.163) is short hand for the functional specification of the target module *before* weaving the $association$. The term in (G.164) is an abbreviation for the functional specification *after* the weaving of the $association$, and the term in (G.165) describes the functional specification of the target module after weaving all server components. The formulae (G.166), (G.167), and (G.168) describe a tuple of all message send statements contained in the functional specification of the target module *before* and *after* the weaving of the $association$, and after weaving all server components, respectively.

$$oTargetFuncSpec = functionalSpec(identicalElement(t_{h-1}, oTargetModule)) \tag{G.163}$$

$$wTargetFuncSpec = functionalSpec(tModule) \tag{G.164}$$

$$eTargetFuncSpec = functionalSpec(identicalElement(t_{h+j}, oTargetModule)) \tag{G.165}$$

$$oTargetFsMsgSends = orderedChildrenOfType(oTargetFuncSpec, MessageSend) \tag{G.166}$$

$$wTargetFsMsgSends = oderedChildrenOfType(wTargetFuncSpec, MessageSend) \tag{G.167}$$

$$eTargetFsMsgSends = oderedChildrenOfType(eTargetFuncSpec, MessageSend) \tag{G.168}$$

Predicate (G.169) specifies that after the weaving, all the role names of the original $association$ occurring in message send statements of the functional specification are substituted after the weaving by the corresponding generated role names.

$$
\begin{aligned}
&\forall i \in \mathbb{N}_0 : 0 \leq i < \mathit{arity}(\mathit{oTargetFsMsgSends}) \Rightarrow \big(\\
&\quad \exists \mathit{oMsgSend} = \pi_i(\mathit{oTargetFsMsgSends}) : \\
&\quad \exists \mathit{wMsgSend} = \pi_i(\mathit{wTargetFsMsgSends}) : \\
&\quad \exists \mathit{eMsgSend} = \pi_i(\mathit{eTargetFsMsgSends}) : \\
&\quad\ \big(\exists \mathit{eChn} = \mathit{channelNames}(\mathit{eMsgSend}) : \\
&\quad\quad \forall k \in \mathbb{N}_0 : 0 \leq k < \mathit{arity}(\mathit{eChn}) \Rightarrow \\
&\quad\quad\quad \pi_k(\mathit{eChn}) \neq \mathit{srcRoleName} \wedge \pi_k(\mathit{eChn}) \neq \mathit{tgtRoleName} \\
&\quad\ \big) \wedge \exists \mathit{oChn} = \mathit{channelNames}(\mathit{oMsgSend}) : \\
&\quad\quad\quad \forall k \in \mathbb{N}_0 : 0 \leq k < \mathit{arity}(\mathit{oChn}) \Rightarrow \big(\\
&\quad\quad\quad\quad \exists \mathit{wChn} = \mathit{channelNames}(\mathit{wMsgSend}) : \\
&\quad\quad\quad\quad\quad \pi_k(\mathit{oChn}) = \mathit{srcRoleName} \Rightarrow \big(\\
&\quad\quad\quad\quad\quad\quad \big(\exists l \in \mathbb{N}_0 : 0 \leq l < \mathit{arity}(\mathit{wChn}) \wedge \pi_l(\mathit{wChn}) = \mathit{clonedSrcRoleName}\big) \\
&\quad\quad\quad\quad\quad \big) \wedge \pi_k(\mathit{oChn}) = \mathit{tgtRoleName} \Rightarrow \big(\\
&\quad\quad\quad\quad\quad\quad \big(\exists l \in \mathbb{N}_0 : 0 \leq l < \mathit{arity}(\mathit{wChn}) \wedge \pi_l(\mathit{wChn}) = \mathit{clonedTgtRoleName}\big) \\
&\quad\quad\quad\quad \big)\big)\big)
\end{aligned}
\tag{G.169}
$$

Correspondingly to (G.163)–(G.169), in (G.170)–(G.176), the replacement of the association roles is described for the functional specification of the server component.

$$\mathit{oSCompFuncSpec} = \mathit{functionalSpec}(\mathit{identicalElement}(t_{h-1}, \mathit{oTargetModule})) \tag{G.170}$$
$$\mathit{wSCompFuncSpec} = \mathit{functionalSpec}(\mathit{tModule}) \tag{G.171}$$
$$\mathit{eSCompFuncSpec} = \mathit{functionalSpec}(\mathit{identicalElement}(t_{h+j}, \mathit{oTargetModule})) \tag{G.172}$$
$$\mathit{oSCompFsMsgSends} = \mathit{orderedChildrenOfType}(\mathit{oSCompFuncSpec}, \mathit{MessageSend}) \tag{G.173}$$
$$\mathit{wSCompFsMsgSends} = \mathit{oderedChildrenOfType}(\mathit{wSCompFuncSpec}, \mathit{MessageSend}) \tag{G.174}$$
$$\mathit{eSCompFsMsgSends} = \mathit{oderedChildrenOfType}(\mathit{eSCompFuncSpec}, \mathit{MessageSend}) \tag{G.175}$$

$$
\begin{aligned}
&\forall i \in \mathbb{N}_0 : 0 \leq i < arity(oSCompFsMsgSends) \Rightarrow \big(\\
&\quad \exists oMsgSend = \pi_i(oSCompFsMsgSends) : \\
&\quad \exists wMsgSend = \pi_i(wSCompFsMsgSends) : \\
&\quad \exists eMsgSend = \pi_i(eSCompFsMsgSends) : \\
&\quad\ \big(\exists eChn = channelNames(eMsgSend) : \\
&\quad\quad \forall k \in \mathbb{N}_0 : 0 \leq k < arity(eChn) \Rightarrow \\
&\quad\quad\ \ \pi_k(eChn) \neq srcRoleName \wedge \pi_k(eChn) \neq tgtRoleName \\
&\quad\ \big) \wedge \exists oChn = channelNames(oMsgSend) : \\
&\quad\quad\ \ \forall k \in \mathbb{N}_0 : 0 \leq k < arity(oChn) \Rightarrow \big(\\
&\quad\quad\quad \exists wChn = channelNames(wMsgSend) : \\
&\quad\quad\quad\ \ \pi_k(oChn) = srcRoleName \Rightarrow \big(\\
&\quad\quad\quad\quad \big(\exists l \in \mathbb{N}_0 : 0 \leq l < arity(wChn) \wedge \pi_l(wChn) = clonedSrcRoleName\big) \\
&\quad\quad\quad\ \ \big) \wedge \pi_k(oChn) = tgtRoleName \Rightarrow \big(\\
&\quad\quad\quad\quad \big(\exists l \in \mathbb{N}_0 : 0 \leq l < arity(wChn) \wedge \pi_l(wChn) = clonedTgtRoleName\big) \\
&\quad\quad\quad \big)\big)\big)
\end{aligned}
\tag{G.176}
$$

The references to the roles of the$association$ must also be adapted in the transform expressions of the target module and the server component. Formula (G.177) specifies a term $targetModuleScenarios$ which represents all scenarios contained as direct children in the target module.[9] Similarly, in the term of Formula (G.178), the scenario nodes contained in the server component are given. Finally, the term $oScenarios$ defines in (G.179) the unified set of all scenarios contained in the target module and the server component.

$$
\begin{aligned}
&targetModuleScenarios = \\
&\quad \{x | \exists y \in filterSet(targetModuleParts, ScenarioDefinition) : \\
&\quad\quad x = y \\
&\quad \}
\end{aligned}
\tag{G.177}
$$

$$
\begin{aligned}
&serverCompScenarios = \\
&\quad \{x | \exists y \in filterSet(targetModuleParts, ScenarioDefinition) : \\
&\quad\quad x = y \\
&\quad \}
\end{aligned}
\tag{G.178}
$$

$$
oScenarios = targetModuleScenarios \cup serverCompScenarios \tag{G.179}
$$

In order that the weaving operation ϕ_w works correctly, the Predicate (G.180) must be satisfied. It specifies that all occurrences of references to the source and target role names of the *original association*

[9] Only the transformation expressions of the scenarios which are direct children can access the roles of the associations.

contained in transform input expressions must be replaced by the corresponding roles names of the *cloned association*. Predicate (G.181) specifies the corresponding weaving semantics for all *transform output* expressions.

$$\begin{aligned}
&\forall oScen \in oScenarios : \big(\\
&\quad \exists wScen = identicalElement(t_h, oScen) : \\
&\quad \exists eScen = identicalElement(t_{h+j}, oScen) : \\
&\quad\; \exists oTform = orderedChildrenOfType(oScen, TransformInput, \emptyset) : \\
&\quad\; \exists wTform = orderedChildrenOfType(wScen, TransformInput, \emptyset) : \\
&\quad\; \exists eTform = orderedChildrenOfType(eScen, TransformInput, \emptyset) : \\
&\quad\quad \big(\forall k \in \mathbb{N}_0 : 0 \le k < arity(eTform) \Rightarrow \exists eChn = channelNames(\pi_k(eTform)) : \\
&\quad\quad\; \forall i \in \mathbb{N}_0 : 0 \le i < arity(eChn) \Rightarrow \\
&\quad\quad\quad \pi_i(eChn) \ne srcRoleName \wedge \pi_i(eChn) \ne tgtRoleName \\
&\quad\quad \big) \wedge \big(\forall i \in \mathbb{N}_0 : 0 \le i < arity(oTform) \Rightarrow \big(\\
&\quad\quad\; \exists oChn = channelNames(\pi_i(oTform)) : \\
&\quad\quad\quad \forall l \in \mathbb{N}_0 : 0 \le l < arity(oChn) \Rightarrow \big(\\
&\quad\quad\quad\; \exists wChn = channelNames(\pi_i(wTform)) : \\
&\quad\quad\quad\quad \pi_l(oChn) = srcRoleName \Rightarrow \big(\\
&\quad\quad\quad\quad\; \big(\exists k \in \mathbb{N}_0 : 0 \le k < arity(wChn) \wedge \pi_k(wChn) = clonedSrcRoleName\big) \\
&\quad\quad\quad\quad \big) \wedge \pi_l(oChn) = tgtRoleName \Rightarrow \big(\\
&\quad\quad\quad\quad\; \big(\exists k \in \mathbb{N}_0 : 0 \le k < arity(wChn) \wedge \pi_k(wChn) = clonedTgtRoleName\big) \\
&\quad\quad\quad \big)\big)\big)\big)\big)
\end{aligned} \tag{G.180}$$

$$
\begin{aligned}
&\forall oScen \in oScenarios : \big(\\
&\quad \exists wScen = identicalElement(t_h, oScen) : \\
&\quad \exists eScen = identicalElement(t_{h+j}, oScen) : \\
&\quad\; \exists oTform = orderedChildrenOfType(oScen, TransformOutput, \emptyset) : \\
&\quad\; \exists wTform = orderedChildrenOfType(wScen, TransformOutput, \emptyset) : \\
&\quad\; \exists eTform = orderedChildrenOfType(eScen, TransformOutput, \emptyset) : \\
&\quad\quad \big(\forall k \in \mathbb{N}_0 : 0 \leq k < arity(eTform) \Rightarrow \exists eChn = channelNames(\pi_k(eTform)) : \\
&\quad\quad\; \forall i \in \mathbb{N}_0 : 0 \leq i < arity(eChn) \Rightarrow \\
&\quad\quad\quad \pi_i(eChn) \neq srcRoleName \wedge \pi_i(eChn) \neq tgtRoleName \\
&\quad\quad \big) \wedge \big(\forall i \in \mathbb{N}_0 : 0 \leq i < arity(oTform) \Rightarrow \big(\\
&\quad\quad\; \exists oChn = channelNames(\pi_i(oTform)) : \\
&\quad\quad\quad \forall l \in \mathbb{N}_0 : 0 \leq l < arity(oChn) \Rightarrow \big(\\
&\quad\quad\quad\; \exists wChn = channelNames(\pi_i(wTform)) : \\
&\quad\quad\quad\quad \pi_l(oChn) = srcRoleName \Rightarrow \big(\\
&\quad\quad\quad\quad\; \big(\exists k \in \mathbb{N}_0 : 0 \leq k < arity(wChn) \wedge \pi_k(wChn) = clonedSrcRoleName\big) \\
&\quad\quad\quad\quad \big) \wedge \pi_l(oChn) = tgtRoleName \Rightarrow \big(\\
&\quad\quad\quad\quad\; \big(\exists k \in \mathbb{N}_0 : 0 \leq k < arity(wChn) \wedge \pi_k(wChn) = clonedTgtRoleName\big) \\
&\quad\quad\quad \big)\big)\big)\big)
\end{aligned} \tag{G.181}
$$

Finally, the associations between the server component and the aspect must be deleted, after the weaving of the server components. The corresponding semantics is described in Predicate G.182.

$$
\begin{aligned}
&identicalElement(t_{h+j}, association) = \varepsilon \wedge \\
&targetRole(t_{h+j}, association)) = \varepsilon
\end{aligned} \tag{G.182}
$$

G.1.9 Formal Semantics of the Postprocessing

The informal description of the post processing is given in Section 9.2.10. The present section gives the corresponding formal weaving semantics.

Given Elements

The operation ϕ_n describes the weaving semantics for the handling of the aspect modules. The index n is calculated as by $n = w + numScSubSteps$, where $numScSubSteps$ is given by formula (G.137). It denotes the number of transformation steps that have been executed by the operation ϕ_w to weave the server components.

Precondition of ϕ_n

Before executing ϕ_n, the operation, ϕ_w must first finish its execution.

Postcondition of ϕ_n

After the last transformation step, the resulting model t_n must not contain any aspect modules. This semantics is specified by the predicate in (G.183).

$$filterSet(descendants(t_n), AspectDefinition) = \emptyset \tag{G.183}$$

G.2 Formal Weaving Semantics for Partial Aspect-Oriented Elements

G.2.1 Formal Weaving Semantics of Partial Join Relationships Connecting Scenario/Behavior Chunks with a Scenario/Transition

The formal definition of the partial weaving semantics extends the weaving operation ϕ_g. Thus, the pre- and postcondition given in Section G.1.1 and Section G.1.2 have to be extended.

Given Elements

For the description of the partial join relationship, the given elements of Section G.1.1 and Section G.1.2 are also applied. The term *jrlist* in (G.184), denotes the list of topologically sorted join relationships. The current (partial) join relationship is given by (G.185). The source and the target element of the join relationship are defined in (G.186) and (G.187). The term g denotes the index of the operation. Correspondingly, t_{g-1} is the model before weaving the partial join relationship and t_g the model after the weaving.

$$jrlist = topologicalJrSort(t_{g-1}) \tag{G.184}$$
$$j = firstJr(jrlist) \tag{G.185}$$
$$sourceElement = source(t_{g-1}, j) \tag{G.186}$$
$$targetElement = target(t_{g-1}, j) \tag{G.187}$$

Precondition

The precondition for the weaving of behavior and scenario chunks, which is given in Predicate (G.5) and (G.24), is extended by Predicate (G.188). It states that the join relationship between a behavior/scenario chunk and the target transition/scenario must be partial in order that the following postcondition can be satisfied.

$$\begin{aligned}
&\big((\mathit{type}(\mathit{sourceElement}) = \mathit{ScenarioDefinition} \wedge \\
&\quad \mathit{type}(\mathit{targetElement}) = \mathit{ScenarioDefinition}) \vee \\
&(\mathit{type}(\mathit{sourceElement}) \in \{\mathit{StateDefinition}, \mathit{ComponentDefinition}\} \wedge \\
&\quad \mathit{type}(\mathit{targetElement}) = \mathit{TransitionDefinition})\big) \wedge \\
&\mathit{partial}(j) = \mathit{true}
\end{aligned} \tag{G.188}$$

Postcondition

After the execution the weaving operation ϕ_g, the target module is set to partial if the woven join relationship is partial. Predicate (G.189) states this fact.

$$\mathit{partial}(\mathit{identicalElement}(t_g, \mathit{targetModule}(t_{g-1}, \mathit{targetElement}))) = \mathit{true} \tag{G.189}$$

G.2.2 Formal Weaving Semantics of Partial Join Relationships Connecting other Elements

The operation ϕ_g is also responsible for weaving abstract join relationships which do not connect a behavior/scenario chunk and a transition/scenario, respectively (cf. Section 9.3.2). The operation has to satisfy the given pre- and postcondition.

Given Elements

Several predicates and terms are required for specifying the semantics of ϕ_g. The term *jrlist* in (G.190), denotes the list of topologically sorted join relationships. The current (partial) join relationship j is given by (G.191), and its source and the target element are defined in (G.192) and (G.193), respectively. The term g denotes the index of the operation. Correspondingly, t_{g-1} is the model before and t_g after the weaving of j.

$$\mathit{jrlist} = \mathit{topologicalJrSort}(t_{g-1}) \tag{G.190}$$

$$j = \mathit{firstJr}(\mathit{jrlist}) \tag{G.191}$$

$$\mathit{sourceElement} = \mathit{source}(t_{g-1}, j) \tag{G.192}$$

$$\mathit{targetElement} = \mathit{target}(t_{g-1}, j) \tag{G.193}$$

Precondition

The Precondition in (G.194) specifies that the extension of the operation t_g must process the partial join relationships which are not handled by the semantics described in Section G.2.1, G.1.1, or G.1.2.

$$\begin{aligned}
&\big((type(sourceElement) = ScenarioDefinition \wedge \\
&\quad type(targetElement) \neq ScenarioDefinition) \vee \\
&\quad (type(sourceElement) \in \{StateDefinition, ComponentDefinition\} \wedge \\
&\quad type(targetElement) \neq TransitionDefinition) \vee \\
&\quad (type(sourceElement) = AspectDefinition)\big) \wedge \\
&partial(j) = true
\end{aligned} \tag{G.194}$$

Postcondition

The postcondition for ϕ_g consists of several terms and Predicates. The origin of j is an aspect module. A shorthand term for it before the weaving of j is defined in (G.195). The term $tElement$ in (G.196) represents the target element after the weaving of j. The term in (G.198) denotes the informal descriptions of the target element before and in (G.197) after the weaving, respectively. In (G.199) the informal description of the aspect module is specified as the term $commentAspect$. The term $textModelAspect$ in (G.200) represents the textual description of the aspect module.[10] It is needed, as the textual description of the aspect module will be injected during the weaving process into the target module.

$$aspect = jrHostingAspect(t_{g-1}, j) \tag{G.195}$$

$$tElement = identicalElement(t_g, targetElement) \tag{G.196}$$

$$wCommentTElement = \pi_1(childOfType(tElement, InformalDescription, 1)) \tag{G.197}$$

$$oCommentTElement = \pi_1(childOfType(targetElement, InformalDescription, 1)) \tag{G.198}$$

$$commentAspect = \pi_1(childOfType(aspect, InformalDescription, 1)) \tag{G.199}$$

$$textModelAspect = \rho^{-1}(identicalElement(t_0, aspect)) \tag{G.200}$$

Predicate (G.201) defines the weaving semantics in the case that the abstract join relationship j has a *before* ordering. The comment and the textual representation of the aspect are woven before the comment of the target module. The bullet (•) denotes the concatenation of informal descriptions.

$$\begin{aligned}
&ordering(j) = \mathbf{before} \Rightarrow \\
&\quad wCommentTElement = commentAspect \bullet textModelAspect \bullet oCommentTElement
\end{aligned} \tag{G.201}$$

Similarly, Predicate (G.202) specifies the weaving semantics for the partial join relationship j with an *instead* ordering. The comment on the target module is replaced by the comment and the textual representation of the aspect module.

$$\begin{aligned}
&ordering(j) = \mathbf{instead} \Rightarrow \\
&\quad wCommentTElement = commentAspect \bullet textModelAspect
\end{aligned} \tag{G.202}$$

[10] Here the original aspect of the model t_0 is used, as some join relationships of the aspect may have already been deleted by previously executed weaving operations.

Predicate (G.203) specifies the weaving semantics for the partial join relationship j with an *after* ordering. The informal description of the aspect and its textual representation are inserted after the comment on the target module.

$$\begin{aligned} &ordering(j) = \textbf{after} \Rightarrow \\ &\quad wCommentTElement = oCommentTElement \bullet commentAspect \bullet textModelAspect \end{aligned} \tag{G.203}$$

Predicate (G.204) specifies that if and only if the target element is a module, i.e., an aspect or a component, is it set to partial after the weaving of j. If the term $targetElement$ does not denote a module $tModule$, e.g., in the case where it is an association, it is equal to ε and the partial property is not set.

$$partial(tElement) = true \tag{G.204}$$

The operation ϕ_g needs a post processing of all *instead* join relationships which impact the same target, as otherwise only the comment on the last woven join relationship is contained in the informal description of the target. If there is more than one *instead* join relationship impacting the same target, only the first one is woven with an *instead* semantics. The weaving order of the subsequently processed join relationships is changed to *after*. The term defined in (G.205) represents the *instead* group of all join relationships impacting the same target element as j. Predicate (G.206) describes the actual semantics.

$$insteadGroup = \pi_1(\pi_0(jrlist)) \tag{G.205}$$

$$\begin{aligned} &ordering(j) = \textbf{instead} \Rightarrow \Big(\\ &\quad \forall i \in \mathbb{N} : 1 \leq i < arity(insteadGroup) \Rightarrow \\ &\qquad \exists! jr = identicalElement(t_g, \pi_i(insteadGroup)) : \\ &\qquad\quad ordering(jr) = \textbf{after} \\ &\Big) \end{aligned} \tag{G.206}$$

Finally, the woven abstract join relationship j must be removed from the resulting model. This is defined by Predicate (G.207).

$$identicalElement(t_g, j) = \varepsilon \tag{G.207}$$

G.2.3 Formal Weaving Semantics of Partial Join Relationships between Crosscutting Environment Objects

The weaving of partial join relationships between (crosscutting) environment objects was informally described in Section 9.3.3. The formal weaving semantics extends the weaving semantics of the non-partial case described in Section G.1.6.

Given Elements

Only the term given in (G.208) is needed in the following for the formal description. It denotes a tuple of all join relationships contained in the aspect-oriented model.

$$jrs = gatherJrs(t_0, t_0, \emptyset) \tag{G.208}$$

Moreover, the index v which is defined in (G.98) is also used in the following. It denotes the index of the weaving step directly following the processing of the crosscutting environment objects.

Precondition of ϕ_u

The extension of ϕ_u has the same precondition as ϕ_u, i.e., the weaving of the embedded components (ϕ_q) must have finished.

Postcondition of ϕ_u

The predicate in (G.209) describes the postcondition for the extension of ϕ_u. It states that an environment object that is targeted by partial join relationship is partial after the weaving of all crosscutting environment objects.

$$\begin{aligned} &\forall k \in \mathbb{N}_0 : 0 \leq k < arity(jrs) \ \Rightarrow \\ &\quad \exists tElement = identicalElement(t_{v-1}, target(t_{v-1}, \pi_k(jrs))) : \\ &\quad\quad type(tElement) = EnvironmentObject \wedge partial(\pi_k(jrs)) = true \Rightarrow \\ &\quad\quad\quad partial(tElement) = true \end{aligned} \tag{G.209}$$

G.2.4 Formal Weaving Semantics of Partial Aspect Modules

For the description of the weaving semantics of partial aspect modules, the operation ϕ_g defined in Section G.1.1 and G.1.2 is extended.

Given Elements

There are several given elements required for describing the semantics of the extended operation ϕ_g. The term g denotes the index of the operation. Correspondingly, t_{g-1} is the model before weaving the join relationship and t_g denotes the model after the weaving. The term $jrlist$ in (G.210), defines the list of topologically sorted join relationships. The term j in (G.211) represents the currently processed join relationship. A short hand for the (partial) aspect is given by the term $aspect$ defined in (G.212), and the source and the target element of j are defined in (G.213) and (G.214), respectively.

$$jrlist = topologicalJrSort(t_{g-1}) \tag{G.210}$$

$$j = firstJr(jrlist) \tag{G.211}$$

$$aspect = jrHostingAspect(t_{g-1}, j) \tag{G.212}$$

$$sourceElement = source(t_{g-1}, j) \tag{G.213}$$

$$targetElement = target(t_{g-1}, j) \tag{G.214}$$

Precondition

The precondition for the extended operation ϕ_g is given in Predicate (G.215). It must be fulfilled in order that the corresponding postcondition can be satisfied. It expresses that the extension for the operation ϕ_g is only executed if the following conditions are satisfied. First, the join relationship connects a behavior/scenario with a scenario node or a state/component with a transition, and second, the aspect where j originates is partial. Note that the case where the aspect is partial but the precondition is not satisfied is handled by the operation described in Section G.2.2.

$$\begin{aligned}&\big((type(sourceElement) = ScenarioDefinition \wedge \\ &\quad type(targetElement) = ScenarioDefinition) \vee \\ &(type(sourceElement) \in \{StateDefinition, ComponentDefinition\} \wedge \\ &\quad type(targetElement) = TransitionDefinition)\big) \wedge partial(aspect)\end{aligned} \tag{G.215}$$

Postcondition

The postcondition is defined by several terms and predicates. The term in (G.216) is short hand for the target module before the weaving. The Formula (G.217) defines the target module after the weaving. In (G.218), the informal description of the target module *before* the weaving is given. In (G.219) the corresponding informal description *after* the weaving is specified. The term (G.220) defines a short hand for the informal description of the aspect module.

$$targetModule = targetModule(t_{g-1}, targetElement) \tag{G.216}$$
$$tModule = identicalElement(t_g, targetModule) \tag{G.217}$$
$$oCommentTModule = \pi_1(childOfType(targetModule, InformalDescription, 1)) \tag{G.218}$$
$$wCommentTModule = \pi_1(childOfType(tModule, InformalDescription, 1)) \tag{G.219}$$
$$commentAspect = \pi_1(childOfType(aspect, InformalDescription, 1)) \tag{G.220}$$

Predicate (G.221) defines the weaving semantics for the informal description of a partial aspect. It is appended to the comment on the target module. The bullet (•) denotes a concatenation of informal descriptions.

$$wCommentTModule = oCommentTModule \bullet commentAspect \tag{G.221}$$

Predicate (G.222) defines that the partial property of the target module is set after the weaving of j.

$$partial(tModule) = true \tag{G.222}$$

Appendix H

More Details on the Tool Implementation

This appendix elaborates on particular facets of the ADORA implementation which are roughly discussed in Chapter 11. Section H.1 particularizes how the visual information is stored in the proposed meta-model implementation. In Section H.2, the rules for mapping the ADORA EBNF grammar given in Appendix B to an object-oriented meta-model are discussed in detail. Finally, Section H.3 discusses some details about the implementation of the constraint checking plug-in.

H.1 Relating the Visual Representation to the Model Elements

The part of the meta-model implementation presented in Section 11.2 neither deals with the spatial layout nor with the visibility information of the ADORA language elements. However, the graphical visualization of a model element, as well as the abstraction mechanisms presented in Section 5.1.4 need this kind of information.

The object-oriented meta-model implementation distinguishes between the actual model information and the information needed for the visual representation. Both parts are strongly separated from each other. The visual information is encapsulated in its own class hierarchy which is connected to the corresponding model element. The model elements and the corresponding visual representation are referenced by the class *Speci cation* of the object-oriented meta-model implementation. The corresponding organization in the meta-model is shown in Fig. H.1.

An ADORA *Speci cation* consists of the *Model* information and a separate set of *Representations*. In turn, a *Model* consists of a set of *Node* and *Connection* objects which comprise the model information. Furthermore, a set of *Representation* objects provides the spatial layout and visibility information of the model element and is associated with the object of the corresponding model element.

Figure H.2 is another view of the meta-model implementation. It shows some of the subclasses of the *Representation* class which are associated with the corresponding model objects. For example, an instance of the class *ComponentRepresentation* contains the layout and visibility information for model elements of the class *Component*.

The abstract super class of all representations is *Representation*. It encapsulates the basic layout and visibility information which is common to all representation subclasses. Each concrete representation, e.g., the class *ComponentRepresentation*, extends this base class and specifies further visualization attributes specific to the model element.

Each representation object is a facade [Gamm95] of the corresponding model element. Thus, all

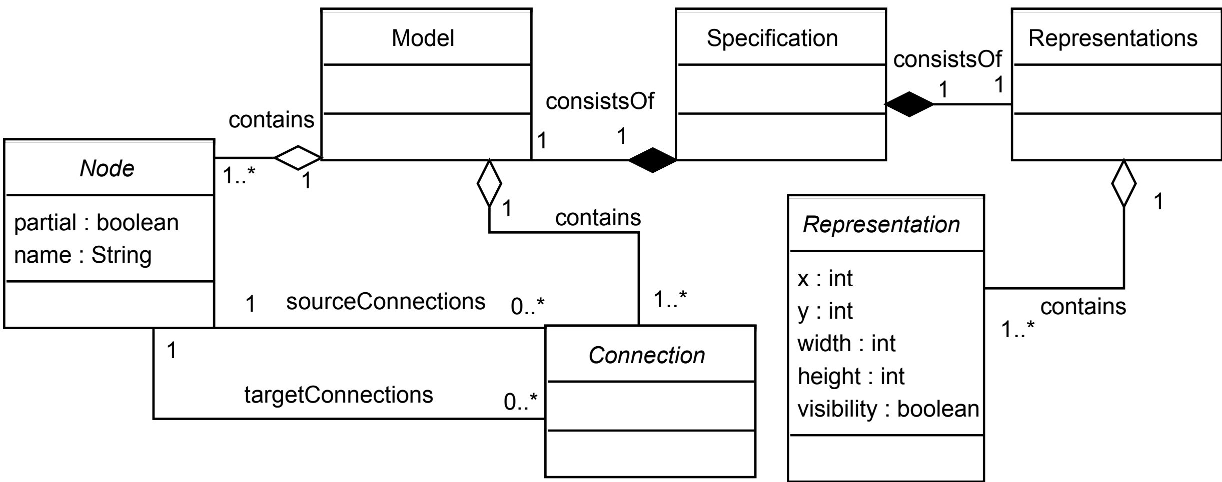


Figure H.1: An overview of the relationships between the specification, the ADORA model nodes, and its representation.

model properties are accessed through the corresponding representation object. A controller of the MVC architecture in the ADORA tool accesses the representation instances of a model element and not directly the object containing the properties of the model element. In Fig. H.2, this fact is given by the class *AdoraModelElementController* which represents the abstract base class of all controllers in the ADORA tool.[1] Using the representation classes as facade allows having more than one visual representation[2] of the same specification. This is also the basic requirement for multi-user editing of models [Moda03].

Note that the architecture for the layout information of an ADORA model which is presented above denotes the desired final state of implementation. However, the version 1.1 of the ADORA meta-model implementation that has been finished at the time of writing is in an intermediate state. The visual representation properties are currently not separated from the model information. Thus they are both contained in the model classes. Nonetheless, it is planned to separate the layout information from the model information in a future release of the tool.

H.2 Mapping between EBNF Rules and Classes in the Object-Oriented Meta-model

The actual mapping from the EBNF grammar to the classes of the meta-model implementation follows a set of rules. The most important ones are brie y discussed in the following. They are exemplified by the grammar rules given in Table 11.1 and the class model shown in Fig. 11.3:

[1]Correspondingly, the class *AdoraView* represents the abstract base class of all views of the MVC pattern in the ADORA tool.

[2]In terms of the MVC pattern, a visual representation is a view.

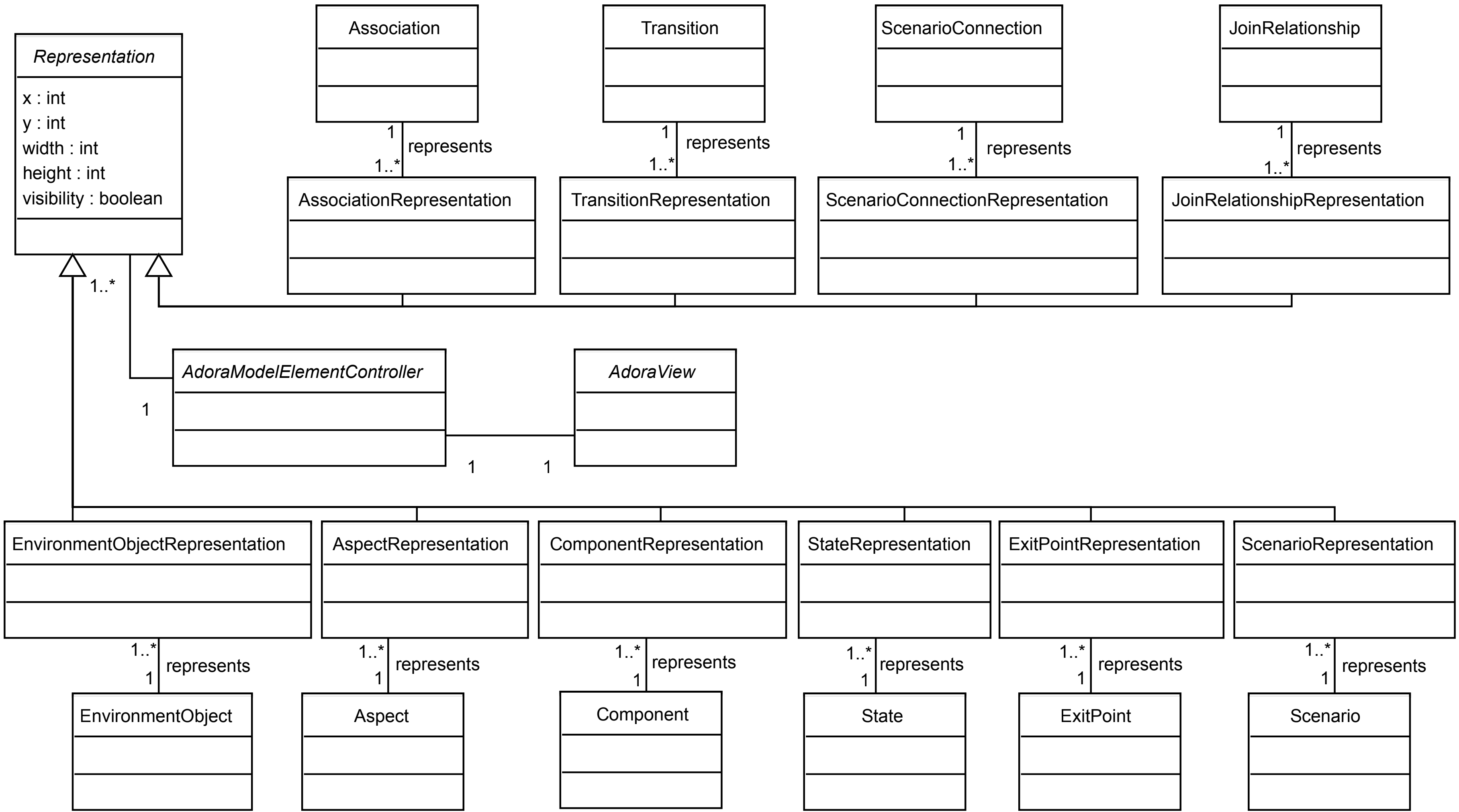


Figure H.2: An overview of the representation of the meta-model elements.

- If a grammar rule $p_i \in P$ is referred to only by one other grammar rule p_k, and p_k is mapped to a class C_k, then the information contained in p_i is represented by C_k as an attribute.[3]

  Example (H.1). The auxiliary rule *ComponentName* is only referred to by the grammar rule *ComponentDe nition*, and therefore it is mapped to the attribute *name* of the class *Component*. However, due to the generalization of the class structure, which is discussed above, the attribute *name* is declared in the class *Node*.

- If there is a rule p_i which is mapped to a class C_i, the the information of any terminal symbols that are part of p_i is mapped to C_i as an attribute. Nevertheless, pure syntactical elements, such as the semicolon used as separator between textual elements, are not mapped, as they do not contain any semantical information.

  Example (H.2). The optional terminals **`partial`**, **`external`**, and **`start`** denote specific properties of a component and are mapped to the boolean attributes *partial*, *external*, and *start* in the class *Component*. However, due to the generalization in the class structure, the attribute *partial* is defined in the class *Node*.

  Example (H.3). The terminal **`is`** in the *ComponentDe nition* rule is purely syntactical and is therefore not mapped to an attribute of the class *Component*.

- The information described by a grammar rule $p_i \in P$ is represented by a separate class C_i of the meta-model implementation if p_i is referred to by more than one other grammar rules. Thus, if $P_s \subset P$ denotes the set of grammar rules which refer to p_i, the condition $|P_s| > 1$ must hold in order that the rule p_i is mapped to C_i. Moreover, each class C_k in the object-oriented meta-model which is derived from a rule $p_k \in P_s$ must be connected to C_i either by an association, a composition or an aggregation relationship.

  Example (H.4). The rule *ComponentDe nition* in Table 11.1 is referred to by several other rules, therefore it manifests in a class. Thus, the class *Component* in Fig. 11.3 represents an abstract object or an object set in the meta-model.

  Example (H.5). The grammar rule *Cardinality* referred to by the *ComponentDe nition* rule is mapped to a class, as it is also referred to by several other rules, such as the *AssociationDe nition*, the *ScenarioDe nition*, and the *EnvironmentObjectDe nition* rule (cf. the full grammar in Appendix B). The reference becomes manifest in the *composition relationship* between the class *Association*, the class *Scenario*, the class *EnvironmentObject*, the class *Component* and the *Cardinality* class.

- The *UniqueModelElementIdenti er*, which is used in textual ADORA models to identify model elements uniquely, is not mapped to the meta-model, either as class or as an attribute. This is due to the fact that in object-oriented systems, objects are implicitly identified uniquely.

  Example (H.6). To be uniquely identifiable within a model, a textual description of a component, which is specified by the *ComponentDe nition* rule, has a unique model element identifier. However, the corresponding class in the meta-model implementation does not contain a corresponding attribute or anything similar.

[3] P denotes the set of all grammar rules, cf. p. 88.

- The relationships between different language elements defined by the grammar are expressed either by the nesting of the elements or by referencing them over an identifier relationship, i.e., by a name or a unique model element identifier (cf. Section 6.1.5). Relationships between the textual elements are mapped to relationships between the corresponding classes.

  Example (H.7). The rule *Cardinality* represents the cardinality of an association, a component, a scenario, or an environment object. Thus, the cardinality rule becomes manifest in the class *Cardinality*, whereas the reference between a component, an association, a scenario, or an environment object and the corresponding cardinality is mapped to a composition relationship between the corresponding classes.

  Example (H.8). The reference to *ComponentParts* in the rule *ComponentDe nition* denotes a part-of relationship between the component and its parts. This part-of relationship is mapped to the aggregation named *parts* between the classes *Container* and a *Node*.[4]

  Example (H.9). In textual models, the relationship between a connection and its source node is expressed by embedding the connection's textual description in the textual description of the source node, whereas the relationship between the connection and its target node is expressed by a reference (cf. Section 6.1.6).
  Each type of connection in the ADORA meta-model implementation is represented by a subclass of the *Connection* class. Thus, for each rule specifying a connection, there is a corresponding subclass. Furthermore, the relationship between the source node and the target node is expressed by a corresponding relationship in the meta-model implementation.
  The rule *ComponentDe nition* refers to the rule *ComponentConnections* which specifies the outgoing connections. This relationship is mapped to the association *sourceConnections* between the classes *Node* and *Connection* in the meta-model implementation. The reference to the target node is described correspondingly by the association *targetConnections*.[5]

- Model parts which have no graphical representation, such as transition labels or the functional specification of a component, are handled as text in the meta-model implementation. However, such an element is transformed to a syntax tree if there is the need to process it, e.g., when it has to be interpreted during the simulation or transformed during the weaving of a model. Therefore, these elements are also mapped to a class in the meta-model implementation. Nevertheless, the corresponding classes are not shown by Fig. 11.3.

  Example (H.10). The *FunctionalSpeci cation* rule is represented mapped to a text attribute in the *Component* class.

H.3 Using the Constraints Checking Plug-in and ICL

This section discusses the use of the constraints checking plug-in in the ADORA tool. As discussed in Section 11.3, leniently enforced constraints are implemented by using ICL and the constraints checking

[4] Apart from components, states and aspects can also contain nodes as parts. Therefore, this relationship is generalized and defined by the super-classes.

[5] The source or target connection relationship is generalized, as all nodes may have incoming or outgoing connections.

Listing H.1: Constraint C_3 formulated for the ADORA tool in ICL.

```
1   constraint StateGroupsInAspectMustBeWellFormed
2     category "weaving"
3     description "Ensures that the state groups of an aspect are well-formed"
4   on (aspect : Aspect) is
5
6     aspect#log.stateGroups().size() > 0 implies
7       (for-all x : StateGroup in aspect.stateGroups() |
8         x.exitPoints().size() == 1 and x.startStates().size() == 0 or
9         x.exitPoints().size() == 0 and x.startStates().size() == 1)
10
10      ifFalse (@result.setMessage("The state groups of the aspect" +
10              "are not well-formed!");@)
11  end
```

plug-in. Listing H.1 illustrates what an ICL constraint for the presented meta-model implementation looks like. It describes the leniently checked Constraint C_3 from page 366 which is enforced before weaving a model. It states that all the state groups of an aspect are well-formed, i.e., that they contain either exactly one start state or exactly one exit point but not both.

An ICL constraint has a name which is used to identify it uniquely. It may optionally have a category and a description. The category is used to define when the constraint has to be satisfied. An ICL constraint has at least one argument. The first argument defines the scope in the model for which the constraint is responsible. In the example, the constraint references objects of the type *Aspect*. The following first-order predicate can refer to this argument by the specified name. All properties, i.e., attributes and methods, defined in the meta-model class can be used to phrase the first order-predicate. For example, the method *stateGroups()* of the class *Aspect* is used to retrieve the set of all state groups in the aspect.[6]

The identifiers in an ICL predicate can be annotated with a *#log* ag. It means that the corresponding element is logged in the case of a false evaluation of the corresponding expression part. After checking the constraint, the logged elements which were involved in the constraint violation can be retrieved, which allows the error to be more precisely located in the model.

A leniently enforced constraint can be violated until a specific point in time, e.g., before a model is woven or simulated. For assigning the constraints to a specific point in time, each constraint can be assigned to a category. Other plug-ins using the constraint checker plug-in can initiate the checking of a certain constraint category. For example, the simulation or the weaving plug-in can tell the constraint checker to check constraints that belong to the category *weaving* or *simulation*, respectively. In the example of Listing H.1, the constraint is assigned to the category weaving and is therefore checked before the model is woven.

Checked constraints which fail are reported by the constraints plug-in. Figure H.3 shows a screen shot of the constraints plug-in after the checking of the constraint in Fig. C_3 has failed. In the case presented, the state group of the aspect shown is malformed because it contains a start state as well as an exit point. The corresponding tool window indicates the violation with a message which can be employed by the user to navigate to the malformed element in the model.

[6] Note that the *StateGroup* Method referred to by the ICL constraint is not shown in the meta-model design of Fig. 11.3 on page 226.

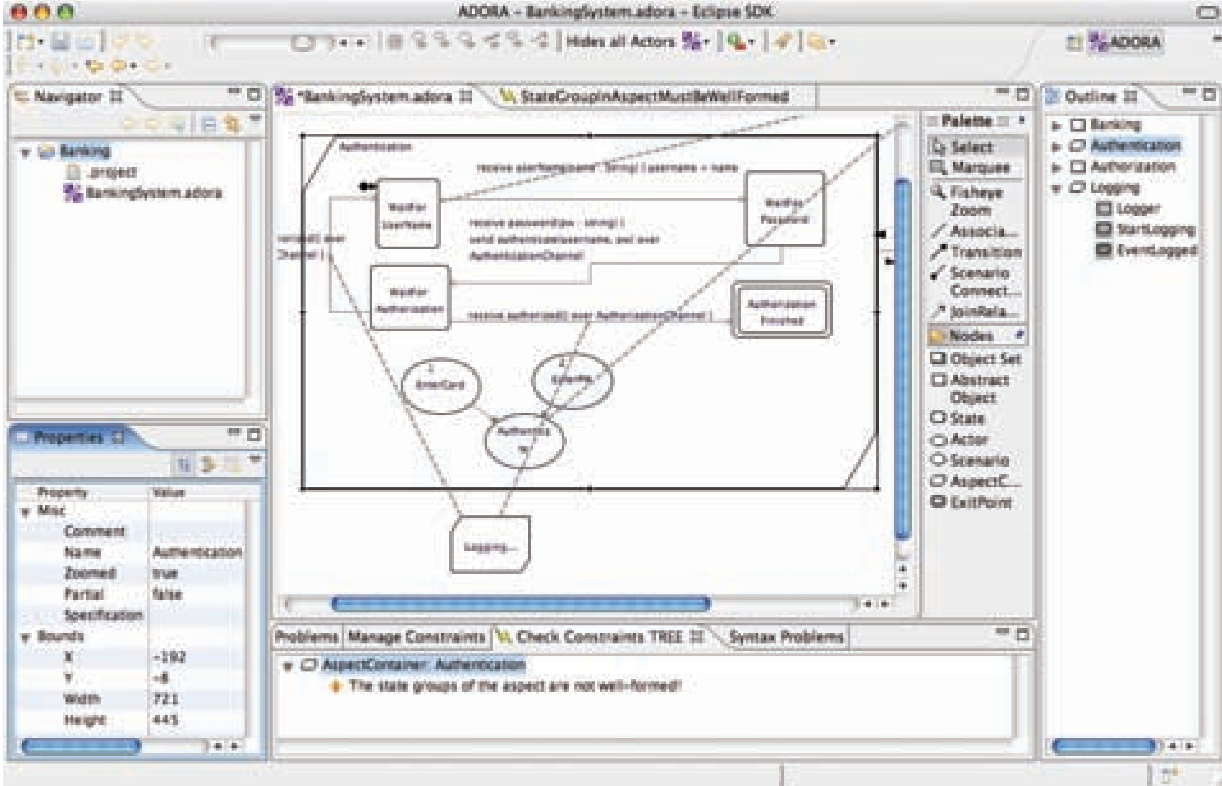


Figure H.3: A screen shot of the constraint checker view in the ADORA tool.

Bibliography

[Akc 92] Akşit, M., Bergmans, L., and Vural, S. An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach. In O. L. Madsen (Ed.), *Proceedings of the 6th European Conference on Object-Oriented Programming (ECOOP 92)*, volume 615, pages 372–395, Utrecht, Netherlands, 1992. Springer.

[Aldr04] Aldrich, J. Open Modules: Reconciling Extensibility and Information Hiding. In *Proceedings of the Workshop on Software Engineering Properties of Languages for Aspect Technologies, held in conjunction with AOSD 04*, 2004.

[Aldr05] Aldrich, J. Open Modules: Modular Reasoning about Advice. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP 05)*, Edinburgh, UK, 2005.

[Alex03] Alexander, I. Misuse Cases: Use Cases with Hostile Intent. *Software, IEEE*, 20(1):58–66, Jan/Feb 2003.

[Arau03] Araújo, J. and Moreira, A. M. D. An Aspectual Use-Case Driven Approach. In *Proceedings of the VIII Jornadas de Ingeniería de Software y Bases de Datos (JISBD 03)*, pages 463–468, Alicante, Spain, 2003.

[Arau04] Araújo, J., Whittle, J., and Kim, D.-K. Modeling and Composing Scenario-Based Requirements with Aspects. In *12th IEEE Requirements Engineering Conference*, pages 58–59, 2004.

[Arau05] Araújo, J., Baniassad, E., Clements, P., Moreira, A., Rashid, A., and Tekinerdogan, B. *Early Aspects: The Current Landscape – Draft*. Technical Report CMU/SEI-2005-TN-xxx, Carnegie Mellon University (CMU), 2005.

[Balz05] Balzer, S., Eugster, P., and Meyer, B. Can Aspects Implement Contracts? In *Proceedings of the 2nd International Workshop on Rapid Integration of Software Engineering Techniques (RISE)*, volume 3943 of *Lecture Notes in Computer Science*, pages 145–157, Heraklion, Crete, Greece, September 2005. Springer.

[Bani04a] Baniassad, E. and Clarke, S. Finding Aspects in Requirements with Theme/Doc. In *Proceedings of the Workshop on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design, held in conjunction with AOSD 04*, Lancaster, UK, 2004.

[Bani04b] Baniassad, E. and Clarke, S. Investigating the Use of Clues for Scaling Document-Level Concern Graphs. In *Proceedings of the Workshop on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design, held in conjunction with OOPSLA 04*, Vancouver, Canada, Oct. 2004.

[Bani04c] Baniassad, E. and Clarke, S. Theme: An Approach for Aspect-Oriented Analysis and Design. In *Proceedings of the International Conference on Software Engineering (ICSE 04)*, Edinburgh, UK, 2004.

[Bara04] Barais, O., Cariou, E., Duchien, L., Pessemier, N., and Seinturier, L. TranSAT: A Framework for the Specification of Software Architecture Evolution. In *Proceedings of the 1st Workshop on Coordination and Adaptation Techniques for Software Entities (WCAT), held in conjunction with ECOOP 04*, Oslo, Norway, June 2004.

[Basi75] Basili, V. R. and Turner, A. J. Iterative Enhancement: A Practical Technique for Software Development. *IEEE Trans. Software Eng.*, 1(4):390–396, 1975.

[Berg01] Bergmans, L. and Akşit, M. Composing Crosscutting Concerns Using Composition Filters. *Communications of the ACM*, 44(10):51–57, 2001.

[Bern99a] Berner, S., Glinz, M., and Joos, S. A Classification of Stereotypes for Object-Oriented Modeling Languages. In *Proceedings of the 2nd International Conference on the Uni ed Modeling Language (UML 99)*, volume 1723 of *Lecture Notes in Computer Science*, pages 249–264. Springer, 1999.

[Bern99b] Berner, S., Schett, N., Xia, Y., and Glinz, M. *An Experimental Validation of the ADORA Language*. Technical Report 99.07, Department of Informatics, University of Zurich, 1999.

[Bern02] Berner, S. *Modellvisualisierung für die Spezi kationssprache* ADORA *[Model Visualization for the Speci cation Language* ADORA *(in German)]*. PhD thesis, University of Zurich, 2002.

[Bind99] Binder, R. V. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, 1999.

[Bjor78] Bjørner, D. and Jones, C. B. Eds. *The Vienna Development Method: The Meta-Language*, volume 61 of *Lecture Notes in Computer Science*. Springer, 1978.

[Blos00] Blosser, J. Explore the Dynamic Proxy API – Use Dynamic Proxies to Bring Strong Typing to Abstract Data Types. *JavaWorld, http://www.javaworld.com*, November 2000.

[Boeh81] Boehm, B. W. *Software Engineering Economics*. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[Boeh88] Boehm, B. W. A Spiral Model of Software Development and Enhancement. *IEEE Computer*, 21(5):61–72, 1988.

[Bone04] Bonér, J. AspectWerkz - Dynamic AOP for Java. In *Proceeding of the 3rd International Conference on Aspect-Oriented Software Development (AOSD 04)*, Lancaster, UK, 2004.

[Bono04] Bonomo, I. *Aspektorientierte Software-Entwicklung – unter besonderer Berücksichtigung der begrif ichen Zusammenhänge und der Einbettung in den Entwicklungsprozess [Aspect-Oriented Software Development - With Respect to the Terminological Interrelationships and their Embedding in the Software Process (in German)]*. Diploma Thesis, University of Zurich, 2004.

[Brab05] Brabrand, C., Møller, A., and Schwartzbach, M. I. Dual Syntax for XML Languages. In *Proceedings of the Workshop on Database Programming Languages (DBPL 05)*, volume 3774 of *Lecture Notes in Computer Science*, pages 27–41, Trondheim, Norway, 2005. Springer.

[Brow96] Brownsword, L. and Clements, P. *A Case Study in Successful Product Line Development*. Technical Report CMU/SEI-96-TR-016, Carnegie Mellon University (CMU), 1996.

[Burk04] Burke, B. Aspect-Oriented Annotations. *OnJava, http://www.onjava.com*, September 2004.

[Buxt69] Buxton, J. N. and Randell, B. Eds. *Software Engineering Techniques: Report of a conference sponsored by the NATO Science Committee*. Brussels, Scientific Affairs Division, NATO. Scientific Affairs Division, North Atlantic Treaty Organization (NATO), Rome, Italy, Oct. 1969.

[Cap93] Cap, C. H. *Theoretische Grundlagen der Informatik [Theorectical Foundations of Computer Science (in German)]*. Springer, Wien, 1993.

[Cede05] Cederqvist, P. *Version Management with CVS*. Free Software Foundation, version 1.11.22 edition, 2005. CVS Version 1.11.22.

[Chec03] Chechik, M., Devereux, B., Easterbrook, S., and Gurfinkel, A. Multi-Valued Symbolic Model-Checking. *ACM Transactions on Software Engineering and Methodology*, 12(4):371–408, 2003.

[Chit05] Chitchyan, R., Rashid, A., Sawyer, P., Bakker, J., Alarcón, M. P., Garcia, A., Tekinerdogan, B., Clarke, S., and Jackson, A. *Survey of Aspect-Oriented Analysis and Design Approaches*. AOSD-Europe Project Deliverable No. AOSD-Europe-ULANC-9, University of Lancaster, May 2005. Editor(s): R. Chitchyan, A. Rashid.

[Chit06] Chitchyan, R., Sampaio, A., Rashid, A., and Rayson, P. A Tool Suite for Aspect-Oriented Requirements Engineering. In *Proceedings of the International Workshop on Early aspects, held in conjunction with ICSE 06*, pages 19–26, New York, NY, USA, 2006. ACM.

[Chit07] Chitchyan, R., Rashid, A., Rayson, P., and Waters, R. Semantics-based composition for aspect-oriented requirements engineering. In *AOSD 07: Proceedings of the 6th international conference on Aspect-oriented software development*, pages 36–48, New York, NY, USA, 2007. ACM Press.

[Chun00] Chung, L., Nixon, B., Yu, E., and Mylopoulos, J. *Non-functional Requirements in Software Engineering*. Kluwer Academic Publishers, Boston, USA, 2000.

[Clar84] Clark, L. A linguistic contribution to goto-less programming. *Commun. ACM*, 27(4):349–350, 1984.

[Clar05] Clarke, S. and Baniassad, E. *Aspect-Oriented Analysis and Design - The Theme Approach*. Addison-Wesley, Upper Saddle River, New Jersey, USA, 2005.

[Clel06] Cleland-Huang, J., Settimi, R., Zou, X., and Solc, P. The Detection and Classification of Non-Functional Requirements with Application to Early Aspects. In *Proceedings of*

the 14th IEEE International Requirements Engineering Conference (RE 06), pages 36–45, Minneapolis, USA, 2006.

[Clem01] Clements, P. and Northrop, L. *Software Product Lines: Practices and Patterns*. Addison-Wesley Professional, 2001.

[Cock01] Cockburn, A. *Writing Effective Use Cases*. Addison-Wesley, 2001.

[Cock02] Cockburn, A. *Agile Software Development*. Addison-Wesley, 2002.

[Coly04] Colyer, A., Rashid, A., and Blair, G. *On the Separation of Concerns in Program Families*. Technical report, Lancaster University, January 2004.

[Coly06] Colyer, A., Harrop, R., Vasseur, R. J. A., Beuche, D., and Beust, C. Point/Counterpoint. *IEEE Software*, 23(1):72–75, Jan.-Feb. 2006.

[Cono99] Conolly, T., Begg, C., and Strachan, A. *Database Systems – Second Edition*. Addison-Wesley Longman Limited, Harlow, Essex, England, 1999.

[Cons00] Constantinides, C. A., Bader, A., Elrad, T. H., Netinant, P., and Fayad, M. E. Designing an Aspect-Oriented Framework in an Object-Oriented Environment. *ACM Computing Survey*, page 41, 2000.

[Cons04] Constantinides, C., Skotiniotis, T., and Störzer, M. Panel Discussion: AOP Considered Harmful. In *Proceedings of the 1st European Interactive Workshop on Aspect Systems (EIWAS)*, Berlin, Germany, September 2004.

[Corn70] Corneil, D. G. and Gotlieb, C. C. An efficient algorithm for graph isomorphism. *J. ACM*, 17(1):51–64, 1970.

[Cram07] Cramer, C. *Verwendung und Prüfung von Integritätsbedingungen in der Modellierungssprache* ADORA *[Using and Checking of Integrity Constraints in the* ADORA *Modeling Language (in German)]*. Diploma thesis, University of Zurich, May 2007.

[Dard93] Dardenne, A., van Lamsweerde, A., and Fickas, S. Goal-Directed Requirements Acquisition. *Science of Computer Programming*, 20(1-2):3–50, 1993.

[Dari96] Darimont, R. and van Lamsweerde, A. Formal Refinement Patterns for Goal-Driven Requirements Elaboration. In *Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering (SIGSOFT 96)*, pages 179–190, New York, NY, USA, 1996. ACM.

[Dari97] Darimont, R., Delor, E., Massonet, P., and van Lamsweerde, A. GRAIL/KAOS: An Environment for Goal-Driven Requirements Engineering. In *Proceedings of the International Conference on Software Engineering (ICSE 97)*, pages 612–613, Kyoto, Japan, 1997.

[Davi92] Davis, A. M. *Software Requirements: Objects, Functions and States*. Prentice Hall, 1992.

[Davi93] Davis, A., Overmyer, S., Jordan, K., Caruso, J., Dandashi, F., Dinh, A., Kincaid, G., Ledeboer, G., Reynolds, P., Sitaram, P., Ta, A., and Theofanos, M. Identifying and Measuring Quality in a Software Requirements Specification. In *Proceedings of the First International Software Metrics Symposium*, 1993.

[Davi01] Davies, N., Cheverst, K., Mitchell, K., and Efrat, A. Using and Determining Location in a Context-Sensitive Tour Guide. *IEEE Computer*, 8(34):35–41, 2001.

[DeMa78] DeMarco, T. *Structured Anaysis and System Speci cation*. Yourdon Press, New York, USA, 1978.

[Dijk68] Dijkstra, E. W. Letters to the Editor: Go to Statement Considered Harmful. *Communications of the ACM*, 11(3):147–148, 1968.

[Dijk72] Dijkstra, E. W. The Humble Programmer. *Communications of the ACM*, 15(10):859–866, 1972.

[Dijk76] Dijkstra, E. W. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[Dijk82] Dijkstra, E. W. Essay: On the Role of Scientific Thought, EWD. 447, 30 August 1974, Neuen, Netherlands. In *Selected Writings on Computing: A Personal Perspective*. Springer-Verlag, 1982.

[Duan07] Duan, C. and Cleland-Huang, J. A Clustering Technique for Early Detection of Dominant and Recessive Cross-Cutting Concerns. In *Proceedings of the Workshop on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design, held in conjunction with ICSE 07*, Minneapolis, USA, 2007.

[Eade91] Eades, P., Lai, W., Misue, K., and Sugiyama, K. Preserving the Mental Map of a Diagram. In *Proceedings of Compugraphics 91*, pages 34–43, 1991.

[Ecli07a] Eclipse Foundation. *The Eclipse Homepage*. http://www.eclipse.org, October 2007.

[Ecli07b] Eclipse Foundation. *The Graphical Editing Framework (GEF) — Homepage*. http://www.eclipse.org/gef/, November 2007.

[Eigl04] Eiglsperger, M., Gutwenger, C., Kaufmann, M., Kupke, J., Jünger, M., Leipert, S., Klein, K., Mutzel, P., and Siebenhaller, M. Automatic Layout of UML Class Diagrams in Orthogonal Style. *Information Visualization*, 3(3):189–208, 2004.

[Elra01] Elrad, T., Akşit, M., Kiczales, G., Lieberherr, K., and Ossher, H. Discussing Aspects of AOP. *Communication of the ACM*, 44(10):33–38, 2001.

[Faga86] Fagan, M. E. Advances in Software Inspections. *IEEE Transactions on Software Engineering*, 12(7):744–751, 1986.

[Feli02] Felici, M. *Requirements Evolution — Understanding Formally Software Engineering Processes within Industrial Contexts*. Technical Report Bando n. 203.15.11, Consiglio Nazionale delle Ricerche, LFCS, Division of Informatics, The University of Edinburgh, 2002.

[Film00] Filman, R. E. and Friedman, D. P. Aspect-Oriented Programming Is Quantification and Obliviousness. In *Proceedings of the Workshop on Advanced Separation of Concerns, held in conjunction with OOPSLA 2000*, pages 21–35, Minneapolis, USA, 2000. Addison-Wesley.

[Fink92] Finkelstein, A., Kramer, J., Nuseibeh, B., and Goedicke, M. Viewpoints: A Framework for Integrating Multiple Perspectives in System Development. *International Journal of Software Engineering and Knowledge Engineering*, 2(1):31–58, 1992.

[Fink96] Finkelstein, A. and Sommerville, I. The Viewpoints FAQ. *Software Engineering Journal*, 11(1):2–4, 1996.

[Floy84] Floyd, C. A Systematic Look at Prototyping. In R. Budde, K. Kuhlenkamp, L. Mathiassen, and R. Züllighoven (Eds.), *Approaches to Prototyping*, pages 1–18, Berlin, 1984. Springer.

[Gal01] Gal, A., Schröder-Preikschat, W., and Spinczyk, O. AspectC++: Language Proposal and Prototype Implementation. In *Proceedings of the Workshop on Advanced Separation of Concerns in Object-Oriented Systems, held in conjunction OOPSLA 2001*, Tampa, Florida, October 2001.

[Gamm95] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns*. Addison-Wesley Professional, 1995.

[Glin95] Glinz, M. An Integrated Formal Model of Scenarios Based on Statecharts. In W. Schäfer and P. Botella (Eds.), *Proceedings of the 5th European Software Engineering Conference (ESEC 95)*, pages 254–271. Lecture Notes in Computer Science 989, Springer, 1995.

[Glin02a] Glinz, M. Statecharts For Requirements Specification – As Simple As Possible, As Rich As Needed. In *Proceedings of Workshop on Scenarios and State Machines: Models, Algorithms, and Tools, held in conjunction with ICSE 02*, 2002.

[Glin02b] Glinz, M., Berner, S., and Joos, S. Object-Oriented Modeling with ADORA. *Information Systems*, 27(6):425–444, 2002.

[Glin05] Glinz, M. *Software Engineering – Eine Einführung [Software Engineering – An Introduction (in German)].* University of Zurich, 2005. Lecture Notes in Computer Science.

[Glin07a] Glinz, M. On Non-Functional Requirements. In *Proceedings of the 15th IEEE International Requirements Engineering Conference (RE 07)*, Delhi, India, 2007.

[Glin07b] Glinz, M. and Gall, H. *Script for the Bachelor Course in Software Engineering*. Univeristy of Zurich, 2007. Lecture Notes, Winter Term 2005/2006.

[Glin07c] Glinz, M., Seybold, C., and Meier, S. Simulation-Driven Creation, Validation and Evolution of Behavioral Requirements Models. In *Proceedings of the Dagstuhl-Workshop Modellbasierte Entwicklung eingebetteter Systeme (MBEES 2007), Informatik-Bericht 2007-01*, pages 103–112., Dagstuhl, Germany, 2007. TU Braunschweig.

[Gray01] Gray, J., Bapty, T., Neema, S., and Tuck, J. Handling Crosscutting Constraints in Domain-Specific Modeling. *Communications of the ACM*, 44(10):87–93, 2001.

[Gree59] Greenwald, I. D. and Kane, M. The Share 709 System: Programming and Modification. *Journal of the ACM*, 6(2):128–133, 1959.

[Gris00a] Griss, M. L. Implementing Product-Line Features by Composing Aspects. In *Proceedings of the 1st Conference on Software Product Lines: Experience and Research Directions*, pages 271–288, Norwell, MA, USA, 2000.

[Gris00b] Griss, M. L. Implementing Product-Line Features with Component Reuse. In *Proceeding of the 6th International Conerence on Software Reuse (ICSR)*, pages 137–152, 2000.

[Groh04] Groher, I. and Baumgarth, T. Aspect-Orientation from Design to Code. In *Proceedings of the Workshop on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design, held in conjunction with AOSD 2004*, 2004.

[Grun99] Grundy, J. C. Aspect-Oriented Requirements Engineering for Component-Based Software Systems. In *Proceedings of the 4th IEEE International Symposium on Requirements Engineering (RE 99)*, pages 84–91, Washington, DC, USA, 1999.

[Grun00] Grundy, J. Multi-Perspective Specification, Design and Implementation of Software Components Using Aspects. *International Journal of Software Engineering and Knowledge Engineering*, 10(6):713–734, December 2000.

[Gutt77] Guttag, J. V. Abstract Data Type and the Development of Data Structures. *Communications of the ACM*, 20(6):396–404, 1977.

[Hann01] Hannemann, J. and Kiczales, G. Overcoming the Prevalent Decomposition of Legacy Code. In *Proceedings of the Workshop on Advanced Separation of Concerns, held in conjunction with ICSE 01*, Toronto, Canada, 2001.

[Hans75] Hansen, P. B. The Programming Language Concurrent Pascal. *IEEE Transactions on Software Engineering*, 1(2):199–207, June 1975.

[Hare87] Harel, D. A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[Harr93] Harrison, W. and Ossher, H. Subject-Oriented Programming: A Critique of Pure Objects. *SIGPLAN Notice*, 28(10):411–428, 1993.

[Harr02] Harrison, W. H., Ossher, H. L., and Tarr, P. L. *Asymmetrically vs. Symmetrically Organized Paradigms for Software Composition*. Tech. rep., IBM Research Division, 2002. http://domino.watson.ibm.com/library/cyberdig.nsf/0/2a4097e93456d0cf85256ca9006dac29?Open Document.

[Heck06] Heckel, R. Graph Transformation in a Nutshell. In *Proceedings of the School on Foundations of Visual Modelling Techniques (FoVMT 06) of the SegraVis Research Training Network*, volume 148 of *Electronic Notes in TCS*, pages 187–198. Elsevier, 2006.

[Herr02] Herrmann, S. Composable Designs with UFA. In *Proceedings of the Workshop on Aspect-Oriented Modeling with UML, held in conjunction with AOSD 02*, Enschede, Netherlands, 2002.

[Ho00] Ho, W.-M., Pennaneac'h, F., and Plouzeau, N. UMLAUT: A Framework for Weaving UML-based Aspect-Oriented Designs. In *Proceedings of the 33rd Conference on Technology of Object-Oriented Languages and Systems (TOOLS 00)*, pages 324–334, 2000.

[Hoar69] Hoare, C. A. R. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, 1969.

[Hoar74] Hoare, C. A. R. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 17(10):549 – 557, 1974.

[IEEE90] IEEE — The Institute of Electrical and Electronics Engineers. *Standard Glossary of Software Engineering Terminology*. IEEE Std. 610.12-1990, Dec. 1990.

[IEEE98] IEEE — The Institute of Electrical and Electronics Engineers. *IEEE Recommended Practice for Software Requirements Speci cations*. IEEE Std 830-1998, 1998.

[IEEE00] IEEE — The Institute of Electrical and Electronics Engineers. *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems*. IEEE Std. 1471-2000, 2000.

[ISO 96] ISO — The International Standardization Organization. *Information Technology - Syntactic Metalanguage - ExtendedBNF (Backus Naur Form)*. ISO/IEC Std. 14977:1996, 1996.

[ISO 01] ISO — The International Standardization Organization. *ISO/IEC 9126-1:2001: Software engineering – Product quality – Part 1: Quality model*. ISO/IEC Std. 9126-1, 2001.

[ISO 05] ISO — The International Standardization Organization. *ISO/IEC 9899: TC, Programming languages – C*. ISO/IEC Std. 9899, 2005.

[Jack75] Jackson, M. *Principles of Program Design*. Academic Press, New York, 1975.

[Jaco92] Jacobson, I., Christerson, M., Jonsson, P., and Övergaard, G. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 4th edition edition, 1992.

[Jaco03] Jacobson, I. Use Cases and Aspects – Working Seamlessly Together. *Journal of Object Technology*, 2(4):7–28, 2003.

[Jaco05] Jacobson, I. and Ng, P.-W. *Aspect-Oriented Software Development with Use Cases*. Addison-Wesley, Pearson Education, 2005.

[Joos99] Joos, S. ADORA-*L - eine Modellierungssprache zur Spezi kation von Software-Anforderungen [*ADORA-*L – A Modeling Language for Specifying Software Requirements (in German)]*. PhD thesis, University of Zurich, Zurich, 1999.

[Kand03] Kandé, M. M. *A Concern-Oriented Approach To Software Architecture*. Thesis no. 2796 (2003), Ecole Politechnique Fédérale de Lausanne (EPFL), 2003.

[Kang90] Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E., and Peterson, A. S. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report CMU/SEI-90-TR-021, Carnegie Mellon University, 1990.

[Katz04] Katz, S. and Rashid, A. *PROBE: From Requirements and Design to Proof Obligations for Aspect-Oriented Systems*. Technical Report COMP-002-2004, Computing Department Lancaster University, Lancaster, UK, 2004.

[Keen88] Keene, S. E. *A programmer s guide to object-oriented programming in Common LISP*. Addison-Wesley, Longman Publishing, Boston, MA, USA, 1988.

[Kell57] Kelly, P. J. A Congruence Theorem for Trees. *Paci c Journal of Mathematics*, 7:961âAŞ–968, 1957.

[Kicz91] Kiczales, G., des Rivieres, J., and Bobrow, D. G. *The Art of the Metaobject Protocol*. The MIT Press, Cambridge, Massachusetts, 1991.

[Kicz97] Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. Aspect-Oriented Programming. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 97)*, pages 327–353, 1997.

[Kicz01a] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. An Overview of AspectJ. In *Proceedings of the European Conference on Object-oriented Programming (ECOOP 01)*, pages 327–353, 2001.

[Kicz01b] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. Getting Started with AspectJ. *Communications of the ACM*, 44(10):59–65, 2001.

[Koto98] Kotonya, G. and Sommerville, I. *Requirements Engineering*. John Wiley & Sons, 1998.

[Kreb04] Krebs, J. *Entwicklung einer Ausführungsmaschine für die Simulation / Animation von formalen* ADORA*-Modellen [Development of an execution machine for the simulation / animation of formal* ADORA *models (in German)]*. Diploma thesis, University of Zurich, 2004.

[Kule04] Kulesza, U., Garcia, A., and Lucena, C. Towards a Method for the Development of Aspect-Oriented Generative Approaches. In *Proceedings of the Workshop on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design, held in conjucntion with OOPSLA 04*, Vancouver, Canada, 2004.

[Ladd03] Laddad, R. *AspectJ in Action, Practical Aspect-Oriented Programming*. Manning Publications Company, New York, 2003.

[Ladd05] Laddad, R. *AOP and Metadata: A Perfect Match*. http://www.ibm.com/developerworks/java/library/j-aopwork3/, March 2005.

[Lams01] van Lamsweerde, A. Goal-Oriented Requirements Engineering: A Guided Tour. In *Proceedings of the 5th IEEE International Symposium on Requirements Engineering, 2001.*, pages 249–262, 2001.

[Lehm97] Lehman, M. M., Ramil, J., Wernick, P. D., Perry, D. E., and M.Turski, W. Metrics and Laws of Software Evolution — The Nineties View. In *Proceedings of the 4th International Software Metrics Symposium*, pages 20–32, 1997.

[Leve93] Leveson, N. and Turner, C. S. An Investigation of the Therac-25 Accidents. *IEEE Computer*, 26(7):18–41, 1993.

[Lieb89] Lieberherr, K. J. and Holland, I. Formulations and Benefits of the Law of Demeter. *SIGPLAN Notice*, 24(3):67–78, 1989.

[Lieb97] Lieberherr, K. J. and Orleans, D. Preventive Program Maintenance in Demeter/Java (Research Demonstration). In *Proceedings of the International Conference on Software Engineering (ICSE 97)*, pages 604–605, Boston, MA, 1997. ACM Press.

[Lieb01] Lieberherr, K., Orleans, D., and Ovlinger, J. Aspect-Oriented Programming with Adaptive Methods. *Communications of the ACM*, 44(10):39–41, 2001.

[Mari04] Marin, M., van Deursen, A., and Moonen, L. Identifying Aspects Using Fan-in Analysis. In *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE2004).*, pages 132–141, 2004.

[Meie05] Meier, S. and Glinz, M. Problems when Introducing Aspect-Oriented Constructs in Models of Functional Requirements and Possible Solutions to these Problems. In *Workshop on Models and Aspects: Handling Cross-Cutting Concerns in MDSD, held in conjunction with ECOOP 05*, 2005.

[Meie06] Meier, S., Reinhard, T., Seybold, C., and Glinz, M. Aspect-Oriented Modeling with Integrated Object Models. In *Proceedings of the Modellierung 06*, pages 129–144, 2006.

[Meie07] Meier, S., Reinhard, T., Stoiber, R., and Glinz, M. Modeling and Evolving Crosscutting Concerns in ADORA. In *Proceedings of the Workshop on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design, held in conjunction with ICSE 07*, Minneapolis, USA, May 2007.

[Meie09a] Meier, S. *A Concurrency Control Mechanism and Redundancy Avoidance for Behaivor Descriptions of the* ADORA *language*. Technical Report ifi-2009.05, Department of Informatics, 2009.

[Meie09b] Meier, S. *An Empirical Validation of the Aspect-Oriented Extension for the* ADORA *language*. Technical Report ifi-2009.06, Department of Informatics, 2009.

[Mens05] Mens, T., Straeten, R. V. D., and Simmonds, J. *Software Evolution with UML and XML, H. Yang (ed.)*, chapter A Framework for Managing Consistency of Evolving UML Models, pages 1–31. Idea Group Publishing, 2005.

[Meye92] Meyer, B. Applying “Design by Contract”. *IEEE Computer*, 25(10):40–51, Oct. 1992.

[Moda03] Modarres, H. *Design and Implementation of a Repository for Distributed Multi-User Graphical Editors*. Diploma thesis, University of Zurich, 2003.

[More02] Moreira, A., Araújo, J., and Brito, I. Crosscutting Quality Attributes for Requirements Engineering. In *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering (SEKE 02)*, pages 167–174, Ischia, Italy, July 2002.

[More05a] Moreira, A., Araújo, J., and Rashid, A. A Concern-Oriented Requirements Engineering Model. In O. Pastor and J. F. e Cunha (Eds.), *Proceedings of the 17th Conference on Advanced Information Systems Engineering (CAiSE 05)*, number 3520 in Lecture Notes in Computer Science, pages 293–308. Springer, June 2005.

[More05b] Moreira, A., Araújo, J., and Rashid, A. Multi-Dimensional Separation of Concerns in Requirements Engineering. In *Proceedings of the 13th IEEE International Conference on Requirments Engineering (RE 05)*, pages 285–296, Paris, France, Sept. 2005.

[Morg94] Morgan, C. and Vickers, T. *On the Re nement Calculus*. Springer, 1994.

[Myer04] Myers, G. J., Badgett, T., Thomas, T. M., and Sandler, C. *The Art of Software Testing — Revised and Updated Second Edition*. John Wiley and Sons, 2004.

[Nuse04] Nuseibeh, B. Keynote on Crosscutting Requirements. In *Proceedings of the 3rd International Conference on Aspect-oriented Software Development (AOSD 04)*, pages 3–4, Lancaster, UK, 2004.

[OMaG03a] OMaG — Object Management Group. *Overview and Guide to OMG s Model-Driven Architecture (MDA)*. OMG document: omg/03-06-01, 2003.

[OMaG03b] OMaG — Object Management Group. *UML 2.0 Superstructure Speci cation*. OMG document: ptc/03-08-02, 2003.

[OMaG06] OMaG — Object Management Group. *UML 2.0 Object Constraint Language*. OMG document: 06-05-0, 2006.

[OMaG07] OMaG — Object Management Group. *OMG Systems Modeling Language*. OMG document: formal/07-09-01, 2007.

[Orle01] Orleans, D. and Lieberherr, K. DJ: Dynamic Adaptive Programming in Java. In *Proceedings of the Third International Conference on Meta-level Architectures and Separation of Crosscutting Concerns (Re ection 01)*, Lecture Notes in Computer Science, pages 73–80, Kyoto, Japan, September 2001. Springer.

[Ossh01] Ossher, H. and Tarr, P. Using Multidimensional Separation of Concerns to (Re)Shape Evolving Software. *Communications of the ACM*, 44(10):43–50, 2001.

[Oste05] Ostermann, K., Mezini, M., and Bockisch, C. Expressive Pointcuts for Increased Modularity. In A. P. Black (Ed.), *Proceedings of the European Conference on Object-Oriented Programming (ECOOP 05)*, volume 3586 of *LNCS*, pages 214–240. Springer, 2005.

[Paak95] Paakki, J. Attribute Grammar Paradigms – A High-Level Methodology in Language Implementation. *ACM Computing Surveys*, 27(2):197–225, 1995.

[Parn72] Parnas, D. L. On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[Parn76] Parnas, D. L., Handzel, G., and Würges, H. Design and Specification of the Minimal Subset of an Operating System Family. *IEEE Transactions on Software Engineering*, SE-2(4):301–307, Dec. 1976.

[Perr87] Perry, D. E. Software Interconnection Models. In *Proceedings of the 9th International Conference on Software Engineering (ICSE 87)*, pages 61–69, Monterey, CA, USA, March 1987. ACM Press.

[Petr62] Petri, C. A. *Kommunikation mit Automaten [Communication Using Automata (in German)]*. PhD thesis, University of Bonn, 1962.

[Petr95] Petre, M. Why Looking Isn't Always Seeing: Readership Skills and Graphical Programming. *Communications of the ACM*, 38(6):33–43, Communications of the ACM 1995.

[Pint03] Pinto, M., Fuentes, L., and Troya, J. M. DAOP-ADL: An Architecture Description Language for Dynamic Component and Aspect-Based Development. In *Proceedings of the 2nd International Conference on Generative Programming and Component Engineering (GPCE 03)*, pages 118–137, Erfurt, Germany, 2003. Springer.

[Plot81] Plotkin, G. D. *A Structural Approach to Operational Semantics*. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[Pohl05] Pohl, K., Böckle, G., and van der Linden, F. J. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.

[Pool01] Poole, J. D. Model-Driven Architecture: Vision, Standards and Emerging Technologies. In *Proceedings of the Workshop on Metamodeling and Adaptive Object Models, held in conjuction with ECOOP 01*, 2001.

[Rand96] Randell, B. The 1968/69 NATO Software Engineering Reports. In *Proceedings of the Seminar "History of Software Engineering"*, Schloss Dagstuhl, Germany, August 1996.

[Rash03] Rashid, A., Moreira, A., and Araújo, J. Modularisation and Composition of Aspectual Requirements. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD 03)*, pages 11–20, Boston, USA, 2003.

[Rash06] Rashid, A. and Moreira, A. Domain Models are NOT Aspect Free. In *Proceedings of the Conference on Model-Driven Engineering Languages and Systems (MoDELS 06)*, pages 155–169, 2006.

[Rech97] Rechenberg, P. and Pomberger, G. Eds. *Informatik Handbuch — Erste Au age [Computer Science Compendium — 1st Edition (in German)]*. Hanser Verlag, München, 1997.

[Reen79] Reenskaug, T. *Models — Views — Controllers*. Technical Note, December 1979.

[Rein06] Reinhard, T., Seybold, C., Meier, S., Glinz, M., and Merlo-Schett, N. Human-Friendly Line Routing for Hierarchical Diagrams. In *Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE 06)*, pages 273–276., Tokyo, Japan, 2006.

[Rein07] Reinhard, T., Meier, S., and Glinz, M. An Improved Fisheye Zoom Algorithm for Visualizing and Editing Hierarchical Models. In *Proceedings of the 2nd Workshop on Requirements Visualization (REV 07), held in conjunction with RE 07*, 2007.

[Rein08] Reinhard, T., Meier, S., Stober, R., Cramer, C., and Glinz, M. Tool support for the navigation in graphical models. In *Proceedings of the 30th International Conference on Software Engineering (ICSE 08)*, pages 823–826, Leipzig, Germany, 2008.

[RERG07] RERG — Requirements Engineering Research Group. ADORA *Tool Web Site*. http://www.ifi.unizh.ch /req/adora/update, September 2007.

[Robe99] Robertson, S. and Robertson, J. *Mastering the Requirements Process*. Addison-Wesley, Pearson Education, 1999.

[Rose04] Rosenhainer, L. Identifying Crosscutting Concerns in Requirements Specifications. In *Proceedings of the Workshop on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design, held in conjunction with AOSD 04*, 2004.

[Royc70] Royce, W. Managing the Development of Large Software Systems. In *Proceedings IEEE WESCON 70, Reprinted in the Proceedings 9th International Conference on Software Engineering (ICSE), 1987*, pages 328–338, August 1970.

[Roze97] Rozenberg, G. Ed. *Handbook of Graph Grammars and Computing by Graph Transformation: Volume 1: Foundations*. World Scientific Publishing, 1997.

[Rumb05] Rumbaugh, J., Jacobson, I., and Booch, G. *The Uni ed Modeling Language Reference Manual – Second Edition*. Addison-Wesley, Pearson Education, 2005.

[Rupp04] Rupp, C., Günther, A., Götz, R., Hahn, J., Cziharz, T., Haupt, A., and Schüpferling, D. *Requirements Engineering and Requirements Mangement — 3rd Edition [Requirements-Engineering und -Management (in German) — 3. Au age]*. Carl Hanser, 2004.

[Saku04] Sakurai, K., Masuhara, H., Ubayashi, N., Matsuura, S., and Komiya, S. Association Aspects. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD 04)*, pages 16–25, Lancaster, UK, 2004.

[Samp05] Sampaio, A., Loughran, N., Rashid, A., and Rayson, P. Mining Aspects in Requirements. In *Proceedings of the Workshop on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design, held in conjunction with AOSD 05*, 2005.

[Sawy96] Sawyer, P., Sommerville, I., and Viller, S. *PREview: Tackling the Real Concerns of Requirements Engineering*. Technical report, Cooperative Systems Engineering Group, 1996.

[Scha96] Schach, S. R. *Classical and Object-Oriented Software Engineering*. Irwin, 1996.

[Sche98] Schett, N. *Konzeption und Realisierung einer Notation zur Formulierung von Integritätsbedingungen für* ADORA*-Modelle [A Notation for the De ntion of Integrity Constraints in* ADORA *Models – Conception and Implementation (in German)]*. Diploma thesis, University of Zurich, 1998.

[Sche04] Schenk, F. *Konzeption und Umsetzung einer Stimuli-Ein-/Ausgabeschnittstelle für die Simulation von Anforderungsmodellen [Conception and Implementation of a Stimuli Input/Output Interface for the Simulation of Requirements Models (in German)]*. Diploma thesis, University of Zurich, 2004.

[Schm06] Schmidt, D. C. Guest Editor's Introduction: Model-Driven Engineering. *IEEE Computer*, 39(2):25–31, 2006.

[Schu98] Schürr, A., Winter, A. J., and Zündorf, A. *Handbook on Graph Grammars: Applications, Vol. 2. Grzegorz Rozenberg (ed.)*, chapter The Progres Approach: Language and Environment, pages 487–550. World Scientific, 1998.

[Scot71] Scott, D. and Strachey, C. Towards a Mathematical Semantics for Computer Languages. *Computers and Automata*, pages 19–46, 1971.

[Seyb03] Seybold, C., Glinz, M., Meier, S., and Merlo-Schett, N. An Effective Layout Adaptation Technique for a Graphical Modeling Tool. In *Proceedings of the 25th International Conference on Software Engineering (ICSE 03)*, pages 826–827, Portland, USA, 3–10 2003.

[Seyb04a] Seybold, C., Meier, S., and Glinz, M. Evolution of Requirements Models by Simulation. In *Proceedings of the 7th International Workshop on Principles of Software Evolution (IWPSE 04), held in conjunction with RE 04*, pages 43–48, Kyoto, Japan, 2004.

[Seyb04b] Seybold, C., Meier, S., and Glinz, M. *Simulation of Semi-Formal Requirements Models as a Means for their Validation and Evolution*. Technical Report 2004.02, Department of Informatics, University of Zurich, 2004.

[Seyb06a] Seybold, C. *Simulation und Evolution von teilformalen* ADORA-*Modellen [Simulation and Evolution of Semi-Formal* ADORA *Models (in German)]*. PhD thesis, University of Zurich, 2006.

[Seyb06b] Seybold, C., Meier, S., and Glinz, M. Scenario-Driven Modeling and Validation of Requirements Models. In *Proceedings of 5th International Workshop on Scenarios and State Machines: Models, Algorithms and Tools, held in conjunction with ICSE 06*, Shanghai, China, May 2006.

[Somm97] Sommerville, I. and Sawyer, P. *Requirements Engineering — A good Practice Guide*. John Wiley & Sons, Inc., 1997.

[Spiv89] Spivey, J. M. *The Z Notation: A Reference Manual*. Prentice-Hall, Inc., 1989.

[Stac73] Stachowiak, H. *Allgemeine Modelltheorie [General Model Theory (in German)]*. Springer-Verlag, Wien, 1973.

[Stan02] Stanley M. Sutton, J. and Rouvellou, I. Modeling of Software Concerns in Cosmos. In *Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD 02)*, pages 127–133, Enschede, Netherlands, 2002.

[Stei02] Stein, D., Hanenberg, S., and Unland, R. A UML-Based Aspect-Oriented Design Notation for AspectJ. In *Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD 02)*, pages 106–112, New York, NY, USA, 2002. Enschede, Netherlands.

[Stei05] Steimann, F. Domain Models are Aspect-Free. In *Proceedings of the Conference on Model-Driven Engineering Languages and Systems (MoDELS 05)*, volume 3713/2005 of *Lecture Notes in Computer Science*, pages 171–185. Springer, 2005.

[Stei06] Steimann, F. The Paradoxical Success of Aspect-Oriented Programming. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Languages, Systems, and Applications (OOPSLA 06)*, pages 481–497, Montréal, Canada, 2006.

[Stoi07] Stoiber, R., Meier, S., and Glinz, M. Visualizing Product Line Domain Variability by Aspect-oriented Modeling. In *Proceedings of the 2nd Workshop on Requirements Visualization (REV 07)*, 2007.

[Stoi08] Stoiber, R., Reinhard, T., and Glinz, M. Visualization support for software product line modeling. In *Proceedings of the 2nd International Workshop on Visualization in Software Product Line Engineering (ViSPLE 08) held in conjunction with SPLC 08*, Limerick, Ireland, 2008.

[Stor05] Störzer, M. and Graf, J. Using Pointcut Delta Analysis to Support Evolution of Aspect-Oriented Software. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM 05)*, pages 653–656, Budapest, Hungary, 2005.

[Sull05] Sullivan, K., Griswold, W. G., Song, Y., Cai, Y., Shonle, M., Tewari, N., and Rajan, H. Information hiding interfaces for aspect-oriented design. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 166–175, New York, NY, USA, 2005. ACM.

[Sun 07a] Sun Microsystems. *The Java Standard Edition Homepage*. http://java.sun.com/javase/, October 2007.

[Sun 07b] Sun Microsystems. *The JavaCC Homepage*. https://javacc.dev.java.net/, November 2007.

[Sutt03] Sutton, S. M. Concerns in a Requirements Model - A Small Case Study. In *Proceedings of the Workshop on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design, held in conjunction with AOSD 03*, Boston, USA, 2003.

[Sutt04] Sutton, S. M. and Rouvellou, I. Concern Modeling for Aspect-Oriented Software Development. In R. E. Filman, T. Elrad, S. Clarke, and M. Akşit (Eds.), *Aspect-Oriented Software Development*, pages 479–505. Addison-Wesley, 2004.

[Tane97] Tanenbaum, A. S. and Woodhull, A. S. *Operating Systems — 2nd Edition*. Prentice-Hall, 1997.

[Tarr99] Tarr, P. L., Ossher, H., Harrison, W. H., and Sutton, S. M. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proceedings of the 21st International Conference on Software Engineering (ICSE 99)*, pages 107–119, 1999.

[Tenn76] Tennent, R. D. The Denotational Semantics of Programming Languages. *Communications of the ACM*, 19(8):437–453, 1976.

[Ullm76] Ullmann, J. R. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, 1976.

[Vass04] Vasseur, A. Dynamic AOP and Runtime Weaving for Java - How does AspectWerkz Address it? In *Proceedings of the Dynamic AOP Workshop, held in conjunction with AOSD 04*, March 2004.

[W3C 07] W3C — World Wide Web Consortium. *XSL Transformations (XSLT) Version 2.0*, January 2007.

[Wage02] Wagelaar, D. and Bergmans, L. Using a Concept-Based Approach to Aspect-Oriented Software Design. In *Proceedings of the Workshop on Aspect-Oriented Modeling, held in conjunction with AOSD 02*, Enschede, Netherlands, March 2002.

[Whit00] Whittle, J. and Schumann, J. Generating Statechart Designs From Scenarios. In *Proceedings of the 24th International Conference on Software Engineering (ICSE 00)*, pages 314–323, 2000.

[Whit04] Whittle, J. and Araújo, J. Scenario Modelling with Aspects. *IEE Software*, 151(4):157–171, 2004.

[Wirt70] Wirth, N. *The Programming Language Pascal*. Technical Report 1, Fachgruppe Computer-Wissenschaften, ETH Zürich, Nov. 1970. Published also at Acta Informatica 1, 35 — 63 (1971).

[Wund05] Wunderlich, L. *AOP – Aspektorientierte Programmierung in der Praxis [AOP – Aspect-Oriented Programming in Practice (in German)]*. entwickler.press, Frankfurt, Germany, 2005.

[WWWC06] WWWC — World Wide Web Consortium, T. Bray, J. Paoli and M. C. Sperberg-McQueen, E. Maler, F. Yergeau, and J. Cowan (Eds). *XML 1.1 (Second Edition) — W3C Recommendation*. http://www.w3.org/TR/2006/REC-xml11-20060816/, August 2006.

[Xia04] Xia, Y. *A Language De nition Method for Visual Speci cation Languages*. PhD thesis, University of Zurich, 2004.

[Yu01] Yu, E. Agent-Orientation as a Modelling Paradigm. *Wirtschaftsinformatik*, 43(2):123–132, April 2001.

[Yu04] Yu, Y., do Prado Leite, J. C. S., and Mylopoulos, J. From Goals to Aspects: Discovering Aspects from Requirements Goal Models. In *12th IEEE Requirements Engineering Conference (RE 04)*, pages 38–47, Kyoto, Japan, 2004.

Curriculum Vitae

Name:	Silvio Rochus Meier
Date of Birth:	May 8th, 1973
Place of Citizenship:	Muri (AG), Switzerland

1980-1985	Grammar school, Muri (AG)
1985-1990	Secondary- and District-School, Muri (AG)
1990-1994	High school with main focus on economics and business administration, Wohlen (AG)
1994-1995	One semester of computer science, Swiss Federal Institute of Technology, Zurich
1995	IT Supporter, Zurich Insurances, Zurich
1995-2001	Studies of computer science, University of Zurich, and Student worker: IT support and software developer
2001-2002	Freelance software engineer
2002-2008	Assistant and doctoral student at the University of Zurich