TECHNISCHE UNIVERSITEIT EINDHOVEN
Department of Mathematics and Computer Science

Automating Model Transformations using UML Functions

By
René Christian Ladan

Supervisors:

dr. Michel R. V. Chaudron (TU/e)
S. 'Thiel' C. Chang (Getronics PinkRoccade)
Jelle D. Gerbrandy (Getronics PinkRoccade)

Eindhoven, November 2006

Preface

"If we knew what it was we were doing, it would not be called research, would it?"
—Albert Einstein

This thesis describes a method for using a derivative of the template mechanism as specified by the Universal Modeling Language (UML) to transform UML models automatically. UML is a graphical notation used to describe software system at any level, ranging from the end user view to the implementation view. The derivatives, called UML functions, will be used to describe transformations. These transformations can be applied to a UML model by using a tool which executes these functions. This tool is also a research topic.

The Company

This project is done at both the Eindhoven University of Technologies (TU/e) and the sector Public of Getronics PinkRoccade (GPR). GPR is a company which manufactures and maintains software systems for many customers like other companies and government institutes. Sector Public has its own research and development section. The sector consists of the following parts: Information Systems, Software Migration Factories and Products, Professional Services, Sourcing and Transition, and Focus Sales.

Abstract

Software development projects have grown significantly over the past fifty years. To keep up with this growth new development methods and languages have been developed. One of these methods is UML. It is widely used in the software industry.

This thesis describes two methods which use UML to describe transformations on UML models: the UML Component Method and the Aspect Oriented Software Design paradigm (AOSD, [9]). Most of these transformations can be automated. These automations consist of small UML snippets which specify how to transform input elements to output elements. The automations have a foundation on a formal calculus.

A tool is used to perform these transformations on the input UML models. A small case will be transformed using the automations and the tool to show their validity.

Thanks

I wish to thank the following people, listed in no particular order:

- Thiel and Marieke Chang for their enormous hospitality;
- My supervisors at GPR, Thiel Chang and Jelle Gerbrandy;
- My supervisor at TU/e, Michel Chaudron;
- Michael Kay for writing [10], which turned out to be a great introduction to the implementation language of the tool, and for answering my newbie questions on the XSLT mailing lists.

Contents

Preface **1**
The Company . . . 3
Abstract . . . 3
Thanks . . . 3

List of Figures **7**

List of Tables **10**

1 Introduction **13**
1.1 Problem Description . . . 13
1.2 Research Goal . . . 13
1.3 Research Questions . . . 14
1.4 Thesis Outline . . . 14

2 Case Descriptions **15**
2.1 Overview of the UML Component Method . . . 15
2.2 Overview of AOSD . . . 18
2.2.1 AOSD according to Jacobson . . . 18
2.2.2 Aspect Oriented Programming . . . 20
2.3 Cases for the UML Component Method . . . 20
2.3.1 Hotel Reservation System . . . 22
2.3.2 Student Mark Registration System . . . 26
2.4 Transformation Types for the UML Component Method . . . 36
2.5 Case for AOSD . . . 37
2.6 Transformation Types for AOSD . . . 38
2.7 Answered Research Questions . . . 38

3 An Analysis of Possible Solutions **39**
3.1 Selection Criteria . . . 39
3.2 Existing Methods . . . 39
3.2.1 XMI.difference . . . 40
3.2.2 XMI-based Model Transformations . . . 40
3.2.3 Proposals of DSTC . . . 40
3.2.4 Python . . . 41
3.3 Comparison of Methods . . . 42
3.4 Transformations: UML 1.X or UML 2.0 ? . . . 43
3.5 Answered Research Questions . . . 43

4 Theoretic Foundations **45**
4.1 Model Transformation . . . 45
4.2 UML Function Model . . . 48
4.3 UML Functions . . . 51
4.3.1 General UML Functions . . . 52
4.3.2 Parametrized Elements . . . 53
4.3.3 Combining UML Diagrams . . . 60
4.3.4 AOSD Extensions . . . 61
4.3.5 Nested Function Application . . . 62
4.4 Answered Research Questions . . . 62

5 A Proof of Concept **65**
5.1 Transformation Functions . . . 65
5.1.1 Functions for the UML Component Method . . . 65
5.1.2 Functions for AOSD . . . 71
5.2 Design of TPUPT . . . 81
5.3 An Actual Conversion . . . 85
5.3.1 Business Interfaces . . . 85
5.3.2 System Interfaces . . . 88
5.3.3 Component and System Architecture . . . 91
5.4 Answered Research Questions . . . 91

6 Conclusions **97**
6.1 Final Conformance to the Requirements . . . 97
6.2 Summary . . . 99
6.3 Things Learned . . . 99
6.3.1 XSLT . . . 99
6.3.2 Writing User-oriented Documents . . . 100
6.3.3 XMI 1.2 . . . 100
6.3.4 AOSD . . . 100
6.3.5 UML Component Method . . . 100
6.3.6 λ Calculus . . . 100
6.4 Future Directions . . . 100

Bibliography **101**

A UML Legend **105**
A.1 Use Case Diagram . . . 105
A.2 Sequence Diagram . . . 105
A.3 Class Diagram . . . 105
A.4 Activity Diagram . . . 108
A.5 AOSD Class Diagram . . . 109
A.6 UML Function Diagram . . . 109

B Introduction to XML **113**

C Mapping UML Elements to XMI **115**
C.1 Package . . . 116
C.2 Interface . . . 116
C.3 Class . . . 117
C.4 Operation . . . 117
C.5 Attribute . . . 118
C.6 Object . . . 118
C.7 Association . . . 118
C.8 AssociationEnd . . . 119

C.9 Dependency 120
C.10 UseCase 120
C.11 Actor 120
C.12 ClassifierRole 120
C.13 Message 121
C.14 CallAction 121
C.15 TaggedValue 121

D Introduction to XSLT and XPath **123**

List of Figures

2.1 The Layers of the UML Component Method . . . 16
2.2 The Models of the UML Component Method . . . 17
2.3 Aspect *Logging* for Classes Implementing *IMarkMgt* . . . 20
2.4 A Complete Use Case Slice . . . 21
2.5 Business Concept Model for the Hotel Reservation System . . . 22
2.6 Business Type Model for the Hotel Reservation System . . . 23
2.7 Use Case Model for the Hotel Reservation System . . . 24
2.8 Sequence Diagram Detailing the *Take Up Reservation* Use Case . . . 24
2.9 Sequence Diagram Detailing the Included *Identify Reservation* Use Case . . . 25
2.10 Business Interface for the <<core>> Class *Customer* . . . 26
2.11 System Interface for the Use Case *Take Up Reservation* . . . 27
2.12 Component Interfaces for *ReservationSystem* and *Customer* . . . 27
2.13 <<boundary>> and <<control>> Class for the Use Case *Take Up Reservation* . . . 28
2.14 Business Concept Model of the Student Mark Registration System . . . 28
2.15 Use Case Diagram of the Student Mark Registration System . . . 28
2.16 Business Process of the Student Mark Registration System . . . 29
2.17 Student Views Marks . . . 30
2.18 Teacher Views Marks . . . 31
2.19 Teacher Adds Marks . . . 32
2.20 Business Type Model of the Student Mark Registration System . . . 33
2.21 Business Interfaces of the Student Mark Registration System with *Manually* Added Methods 34
2.22 System Interfaces of the Student Mark Registration System . . . 35
2.23 Component and System Architecture of the Student Mark Registration System . . . 36
2.24 Class *doIMM* before Weaving . . . 37
2.25 Class *doIMM* after Weaving . . . 37

4.1 Transformation Goal Legend . . . 45
4.2 Transformation Goal, Part 1 . . . 46
4.3 Transformation Goal, Part 2 . . . 46
4.4 Transformation Goal, Part 3 . . . 46
4.5 Transformation Goal, Part 4 . . . 46
4.6 Transformation Goal, Part 5 . . . 47
4.7 Transformation Goal . . . 47
4.8 The Ideal World . . . 48
4.9 Transforming Package *O* with Function *A* . . . 51
4.10 A Package which Violates Rule 3 . . . 53
4.11 Transformations Use β-reduction . . . 54
4.12 Class *C* Gets Renamed to *res*1 by Function *B* . . . 55
4.13 A Function with a Correct Actual Parameter and a Wrong Actual Parameter . . . 59
4.14 Different Transformations Require Separate Invocations . . . 59
4.15 Combining Different Diagram Types . . . 61
4.16 Functions $add - a$ and $add - b$ . . . 63

4.17 Applying Function $add - a$ in a Nested Way . . . 64

5.1 Function for the Business Interface . . . 66
5.2 Extended Function for the Business Interface . . . 67
5.3 Function for the System Interfaces . . . 68
5.4 Proxied Elements of the System Interfaces Function . . . 68
5.5 Extended Function for the System Interfaces . . . 69
5.6 Proxied Elements of the Extended System Interfaces Function . . . 70
5.7 Function for the Component and System Architecture . . . 72
5.8 Proxied Elements of the Component and System Architecture Function . . . 72
5.9 Function *ucm_to_ca* for the Alternative Component and System Architecture . . . 73
5.10 Proxied Elements for Function *ucm_to_ca* . . . 73
5.11 Function *ucm_to_cs* for the Alternative Component and System Architecture . . . 74
5.12 Proxied Elements for Function *ucm_to_cs* . . . 74
5.13 Function *btm_to_cs* for the Alternative Component and System Architecture . . . 74
5.14 Function $extract - infotype$ . . . 75
5.15 Function to Extract a Class from a Non Use Case Specific Slice . . . 76
5.16 Function to Extract a Class from a Use Case Specific Slice . . . 77
5.17 Function to Substitute Pointcut Definitions . . . 79
5.18 Function to Merge an Extension Class into a Normal Class . . . 80
5.19 XSLT Template Flow Diagram of tpupt.xsl . . . 82
5.20 XSLT Template Flow of the Generated Script . . . 86
5.21 Interface Diagram of libxmi.xsl . . . 87
5.22 Generating the Business Interface for the *Customer* Class . . . 87
5.23 Generating the Business Interface for the *Room* Class . . . 88
5.24 Generated Business Interface for the *Customer* Class . . . 89
5.25 Generated Business Interface for the *Room* Class . . . 89
5.26 Generating the System Interface for the Hotel . . . 90
5.27 Proxied Elements for Generating the System Interface . . . 91
5.28 Generated System Interface for the *Take Up Reservation* Use Case . . . 92
5.29 Generating the Component Specification for the *Customer* Class . . . 93
5.30 Generating the Component Specification for the *Room* Class . . . 93
5.31 Generating the Component Architecture for the *Identify Reservation* Use Case . . . 94
5.32 Proxied Elements for the *Identify Reservation* Use Case . . . 94
5.33 Generating the Component Architecture for the *Take Up Reservation* Use Case . . . 94
5.34 Proxied Elements for the *Take Up Reservation* Use Case . . . 94
5.35 Generating the Component Specification for the *ReservationSystem* Package . . . 95
5.36 Proxied Elements for the Component Specification . . . 95
5.37 Generated Component Architecture for the *Identify Reservation* Use Case . . . 96
5.38 Generated Component Architecture for the *Take Up Reservation* Use Case . . . 96
5.39 Generated Component Specifications . . . 96

A.1 Legend for Use Cases Diagrams . . . 106
A.2 Legend for Sequence Diagrams . . . 106
A.3 Legend for Class Diagrams . . . 107
A.4 Legend for Activity Diagrams . . . 108
A.5 Legend for AOSD . . . 109
A.6 Legend for Our Self-defined Functions . . . 110

List of Tables

3.1 Summary of Possible Solutions . . . 42

4.1 Sugared Versions of Lists . . . 49
4.2 Sugared Versions of Lambda Expressions . . . 50
4.3 Parameter Names . . . 57
4.4 Parameter Variable Definitions . . . 57

5.1 Tabular Overview of tpupt.xsl . . . 83
5.2 Tabular Overview of the Generated Script . . . 84
5.3 Tabular Overview of libxmi.xsl . . . 85

Chapter 1

Introduction

"When your work speaks for itself, don't interrupt."
—Henry J. Kaiser

This chapter provides some introductory information. It will describe the research questions and the research goals. It will also give an outline for the rest of this thesis.

1.1 Problem Description

Like other industries, the software industry increasingly focuses on rationalizing production processes to increase quality, flexibility, and profits and to decrease costs.

Correctly specifying the interfaces of the requested software components makes it easier to change software. The UML Component Method is a UML application which focuses on the interface specification of software components. Although many requirements can be handled by individual components, some concerns which are called cross-cutting concerns cannot be attributed to a single component. These concerns are addressed with AOSD.

GPR wants to apply a model driven approach to both the UML Component Method and to AOSD by using a development environment which uses UML as modeling language. The problem is that these tools can handle neither the UML Component Method nor AOSD without some preprocessing. This preprocessing transforms the input models to models adhering to the internal UML meta-model of these tools. The problem with contemporary tools is that their implementation of the UML template mechanism is insufficient. Another problem is that the standard language for model transformations, "Query, Views, and Transformations" (QVT) is not yet finalized. These two problems are the main reason that transformation methods of existing tools are tool specific, and therefore not future proof.

1.2 Research Goal

The goal of this project is to provide a generic method of using UML functions with UML and Model Driven Architecture (MDA) tools. This method should be applicable to both the UML Component Method and AOSD. It should be tool-independent, stable over time, and implementations of it should be easy to maintain.

The method uses template derivatives (functions) to describe transformations. These transformations can be automatically instantiated into the model. The instantiations are done by a script which is automatically generated for the model. Up to now the actions described in the functions are done manually for each project.

So the research goal consists of two parts:

- design the functions for the UML Component Method and for AOSD;
- design and implement the script that executes the transformations.

The questions of section 1.3 should be answered as a result of working on these goals.

1.3 Research Questions

The research questions for this project are:

1. What kinds of transformations are needed for the UML Component Method?
2. What kinds of transformations are needed for AOSD?
3. How can models be transformed?
4. What does the meta-model used to model the transformations look like?
5. Why was this specific meta-model chosen?
6. How are transformations with different types of diagrams specified?
7. What do the functions for the UML Component Method look like?
8. What do the functions for AOSD look like?
9. What is the final solution and why is it chosen?
10. How is the final solution implemented?
11. How can function applications be nested?

Although not a real research question, the question which UML version will be used and why it will be used is also interesting.

1.4 Thesis Outline

Chapter 2 describes the cases used in this project. These include a hotel reservation system and a student mark registration system. These cases will be used to identify the requirements of the UML functions when used for the UML Component Method. It will later extend the student mark registration system with logging extensions, which are specified as aspects. This extension will be used to identify the requirements of the UML functions when used for AOSD.

Chapter 3 proposes some possible solutions along with some existing solutions in related fields. One of these solutions will be chosen as a basis for the final solution. This is done by identifying some requirements and investigating how well the different proposals meet these requirements. Because UML plays a major role in the next chapter, the last section of this chapter is dedicated to the question which UML version to use.

Chapter 4 describes the underlying theory of the final solution. It describes a scheme for transforming models, which will be centered around the notion of UML functions. It also describes a higher level model of these functions. This meta-model provides a formal basis for the UML functions. The last section of this chapter describes how to use these functions.

Chapter 5 describes the final solution in full detail. This solution consists of three major parts: the UML functions for the two methods, the scripts to execute these functions, and an example case showing the results of some of the functions in conjunction with the scripts.

Chapter 6 provides a conformance checklist, a short summary, an overview of the things which I have learned during this project, and some future directions.

The end of each chapter summarizes which research questions as defined in chapter 1.3 have been answered in that chapter.

Chapter 2

Case Descriptions

"The only thing worse than generalizing from one example is generalizing from no examples at all."
—FreeBSD Developer's Handbook, chapter 1.3

This chapter introduces the cases used in the project. It describes the cases themselves, how they are used, and which transformation types are required for the cases. Note that the cases do not describe actual transformations. They only specify the desired output of the transformations. First, this chapter explains some concepts needed to understand the cases.

2.1 Overview of the UML Component Method

The UML Component Method is a method to model software components in UML. For this project the definition of a UML Component as found in [4] is taken. In this definition components have an interface specification. The definition also states that components should adhere to some environment standard (e.g. a software or a business environment). Two components should be exchangeable if their interfaces match, regardless of their implementation. These components are designed to bundle several parts of a subsystem into a subsystem residing one level higher in the system. Components can and usually do interact with each other in several well-known configurations (e.g. client/server, or pipe/filter). Examples of components are the various parts of a student registration system where one component adds new marks and another component retrieves marks. The method works with the four layers shown in figure 2.1. These layers are the data layer, the business layer, the system layer, and the user interface layer:

- The data layer contains the business data.
- The business layer contains the code to maintain the various business rules. It also processes requests from the system layer. This results in storing data into the data layer or retrieving data from the data layer.
- The system layer contains the logic to "implement" the user interface, e.g. to validate input forms. This layer communicates with the business layer to store (literal) data entered at the user interface into the database.
- The user interface layer controls the user interface. It sends requests to the system layer and receives requests from this layer.

The methods contained in the business and system layer do not necessarily form a one to one mapping. Methods present in one layer do not need to have a corresponding method in the other layer. The reason for this is that the business layer is tuned for the data layer and needs to support many kinds of system layers, while the system layer is tuned for the user interface layer.

The UML Component Method uses several models. Figure 2.2 shows that these can be organized into a tree. The UML Component Method starts with 2 or 3 inputs: the business concept model, the use case model, and optionally the business process description:

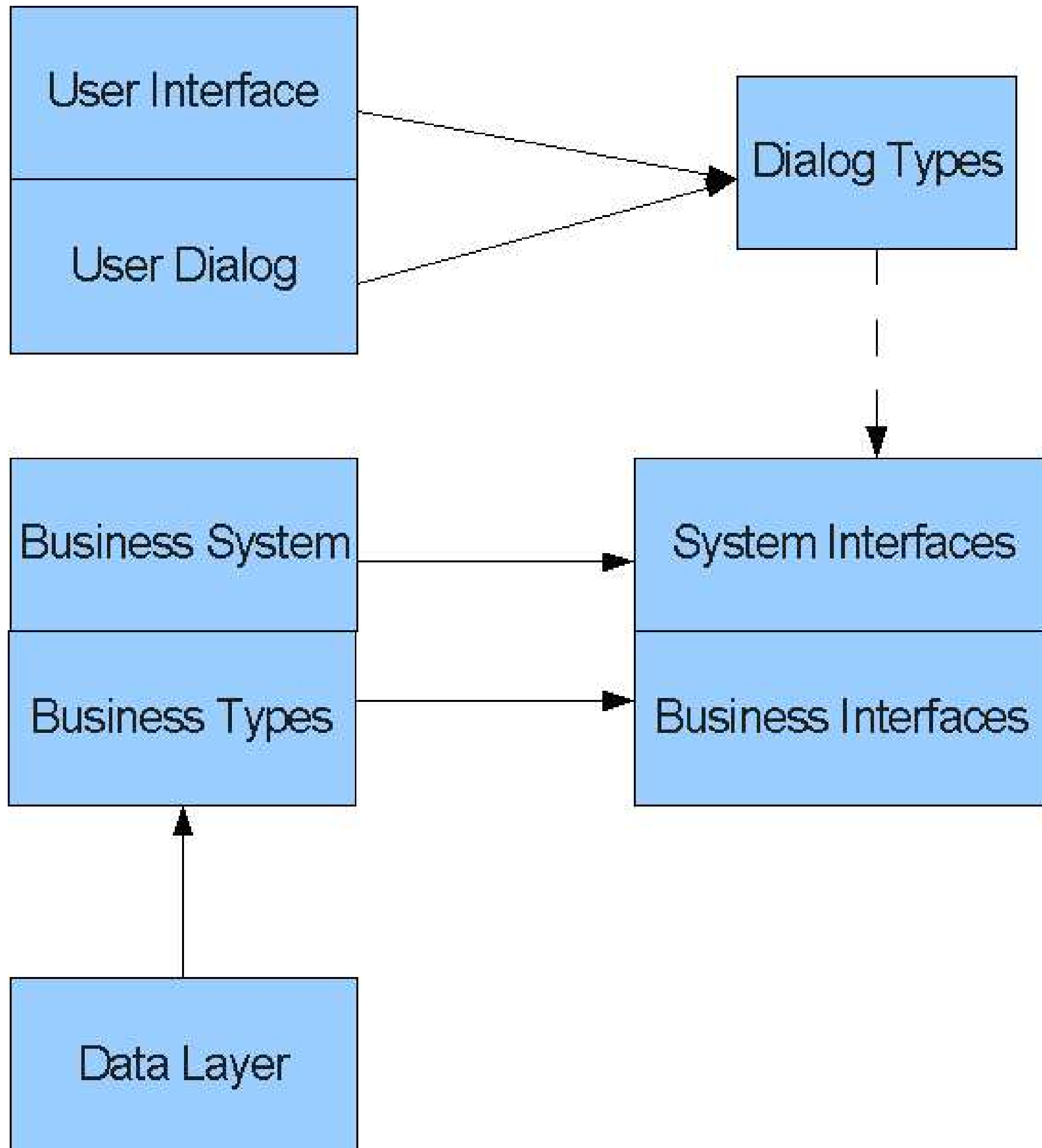

Figure 2.1: The Layers of the UML Component Method

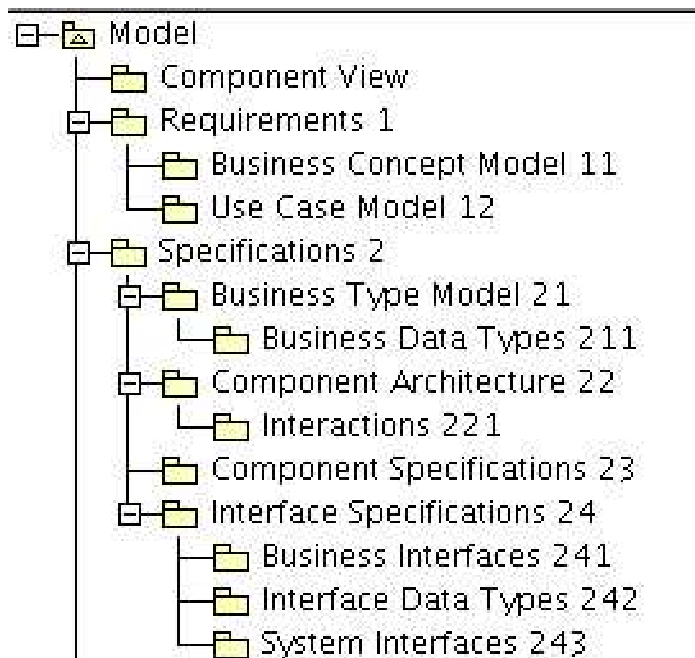

Figure 2.2: The Models of the UML Component Method

- The business concept model is a high-level diagram of the business concepts. It can model people, objects, and processes.
- The use case model describes all the use cases relevant to the system. Unlike most other models in this method, the use case model is seen from the user: it states all possible user actions per user.
- The business process diagram complements the use case model. It describes the user actions as seen from the system. The abstraction level of this diagram is lower than that of the use case model. Business process diagrams are only provided for completeness. The method itself does not need them.

These inputs are refined into other diagrams. These latter diagrams serve as inputs for (partial) automatic conversions. The refinements are:

- The conversion of the use case diagram into a set of sequence diagrams. Each use case of the use case diagram should be extended into a sequence diagram. A sequence diagram describes all the action steps of a use case.
- The business concept model can be fine-tuned into a business type model. This model states what information needs to be represented at the business layer of a software system. People who design the business type model typically perform the following operations on the business concept model:
  - Deleting unneeded classes.
  - Specifying multiplicities, directions, and names of associations.
  - Adding attributes to classes.
  - Merging or splitting classes.

  Classes in the business type model are stereotyped as either <<core>> or <<type>>. <<core>> classes represent a main business concept and do not depend on other classes. All other classes have stereotype <<type>>. The business type model also states the business rules to which the business and/or system layer must adhere.

The automatic conversions can be defined using the above refinements. The conversions are automated because it makes work for other people more consistent, faster, less errorprone, and less repetitive. For the UML Component Method the conversions are as follows:

- Converting the business type model into business interfaces which reside at the business layer. The people who design the business type model designate certain classes which they find important as <<core>> classes. A business interface is created for each <<core>> class. The name of the interface should refer to that class in the business type model. The methods of the interfaces can partially be derived from methods of interfaces at the system layer because the latter usually passes requests to interfaces at the business layer. The methods at the business interfaces can also be derived from the business rules or be freshly invented. In general no rigid automation exists to derive these methods.

- The sequence diagrams form the base for the system interfaces. The system interfaces reside at the system layer. The name of the interfaces should refer to a sequence diagram. Each sequence diagram should have a related system interface. The methods of a system interface, including their parameters and return type, can be derived from the messages of the corresponding sequence diagram.

- The last diagram, the Component and System Architecture, relates the different layers of the UML Component Method. This diagram provides an overview of the architecture. A new component specification is added for each business interface and associated to this interface. All system interfaces are part of one or more component specifications which describe the system itself. These component specifications are related to the business layer by means of the business interfaces. Each business interface has its own component specification to manage the information base which the business interface provides. This information base is derived from the <<core>> class belonging to this business interface and extended with optional attributes from associated classes. These classes must have stereotype <<type>>.

2.2 Overview of AOSD

Crosscutting concerns in large-scale applications are dealt with throughout the software lifecycle. Two crosscutting phenomena exist: scattering and tangling. Scattering means that a use case is realized by different parts of components, tangling means that a component part realizes parts of more than one use case. These two problems occur often and lead to code in which the use cases are almost unrecognizably dissolved into the code. AOSD is a method which tries to solve these two problems.

This section is based on [8, 16], and appendix A of [9]. It is divided into two subsections. The first subsection discusses AOSD according to Jacobson. The second subsection discusses programming for AOSD.

2.2.1 AOSD according to Jacobson

AOSD is based on AOP ([9], Preface) but also on existing techniques like object orientation, component-based development, design patterns, object-oriented frameworks, UML, use case development, and more. It does not compete with existing techniques but extends them.

With AOSD the overall functionality of a software system is modeled through use case diagrams. In AOSD use cases are kept separate from the initial design up to the final code implementation. Use cases can be sliced into use cases modules. These modules normally crosscut the traditional component models, because use case modules are concerned with only one use case, while component models are normally concerned with parts of different use cases. Using AOSD quality-of-service or cross-cutting concerns like logging, security, debugging, persistence, but also exception handling can be kept separate from the main business concerns. Both functional concerns and cross-cutting concerns can be modeled with use cases. AOSD enables full parallel development of large software systems: start with the main functionality and add secondary functionality and cross-cutting concerns as the system evolves.

A system can be developed by starting with a base use case and adding nonintrusive extensions to it. This is possible because extensions that extend the behavior of the base case but do not change the base case itself. These extensions can be modeled in UML as extension use cases. The base and extension use cases themselves can be detailed into state or sequence diagrams. Extension points are assigned to base use cases. Unlike a class inheriting from its parent class an extension use case cannot exist without its base use case and can only execute when its base is executing.

So the basic procedure for AOSD is:

1. Find and specify each use case.

2. Design and code each use case.

3. Compose the use case slices for each component.

4. Test each use case.

Step 3 is expected to become redundant in the foreseeable future.

Two use cases are peer use cases if they have no relationships between them from a use case modeling perspective and if they impact the same set of classes. Each peer use case can be modeled using a use case slice and aspects. This modeling preserves use case modularity. Each peer use case can be specified at a more detailed level with modeling techniques such as interaction diagrams, tables, and text.

The overlaying of different slices on each other (like overhead projector sheets) is called weaving. The overlay order, also known as the use case structure, is specified in the different slices. Weaving use cases can happen at precompile time, compile time, or runtime. Both extensions use cases and peer use cases need to be woven. Weaving extension use cases is just a special case of weaving peer use cases.

The internal building blocks of a system are introduced in the design model. These blocks are not represented by use cases but by use case realizations. Such a realization is a UML collaboration which describes the participation, interaction, and responsibilities of different components. Each realization is a different concern in the design model since each use case is a different concern in the use case model.

UML supports extensions, but only in the use case diagram. Extension support needs to be added to other modeling elements such as collaborations, classes, and operations. Note that these extensions can also imply changes to the platform or infrastructure, or improvements on architecture or refactoring.

Aspects are part of a use case slice. Use case slices overlay an existing set of classes. As opposed to aspects, use case slices can not only extend existing classes but also add new model elements like classes or interfaces to a package. Aspects are also limited in that they can only parametrize pointcuts for existing join points. A join point is a point where the main program and an aspect meet, a pointcut is a description of a set of join points.

Aspects are modeled as packages with a stereotype <<aspect>> in which the class extensions (mostly operations) are modeled per class. These extensions can also specify when the added operations have to be activated. Figure 2.3 shows an aspect *Logging*. This aspect adds an operation extension *logLogin* to the method *login* of the class *mm*. It also adds operations extensions *logAddMark* and *logQueryMarks*. The figure represents what will later be called an actual parameter. The method specified in *struct* is the structural context of the aspect, the one specified in *when* is the behavioral context. The method specification (*logLogin* in this example) and the *when* specification can also include wildcards to gain flexibility. Analogous to <<around>> aspects are <<before>>, <<after>>, and <<afterReturning>> aspects. These aspects work in a similar way, for them the operation specified in *name* is always executed.

Figure 2.4 shows a complete use case slice containing all the information for one use case. The collaboration describes how the elements contained in the use case slice interact (here the class *ProcessLogs* and the aspect *Logging*). The class is added verbatim to the element structure. The aspect is woven into the corresponding classes of the element structure.

A complete use case slice contains one collaboration, one or more aspects, and zero or more classes. Non-use-case-specific slices containing the common parts of a system also exist. AOSD, like the UML Component Method, can make use of the four layer model. This includes using control and boundary classes.

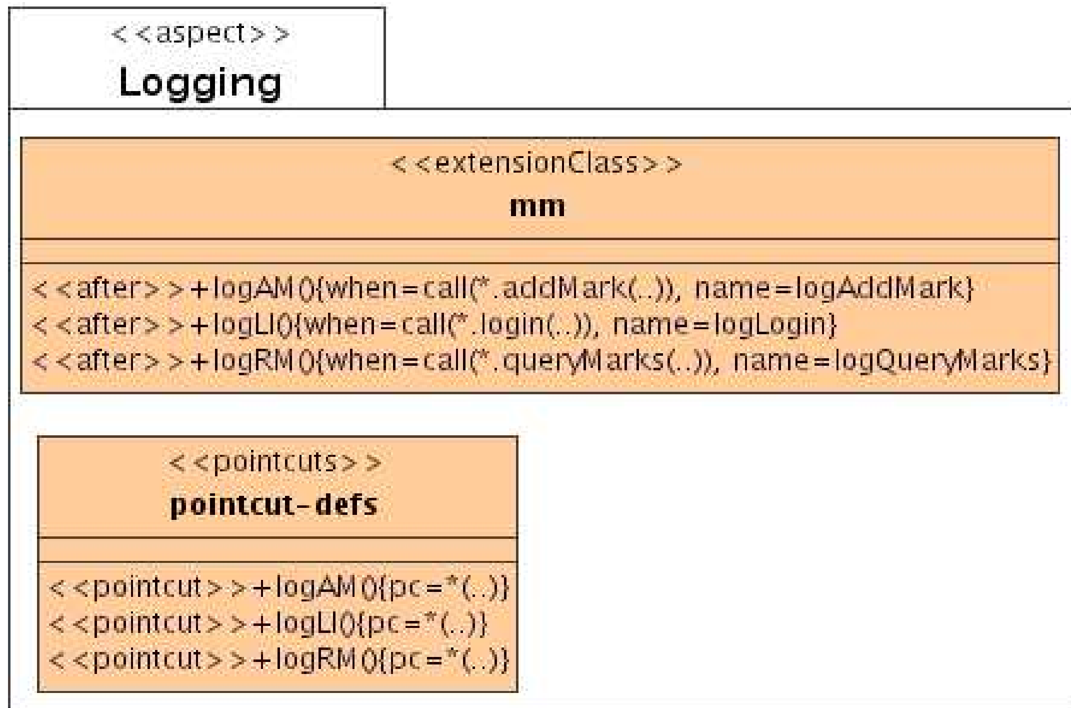

Figure 2.3: Aspect *Logging* for Classes Implementing *IMarkMgt*

2.2.2 Aspect Oriented Programming

A goal of AOP is to modularize concerns through abstractions.

Support for the programming part of AOSD, Aspect Oriented Programming (AOP), is available for most common languages. An overview of such languages is available at [21]. AOP is the first concrete ”implementation” of extension use cases. ”Implementations” of peer use cases have been around longer. They have been managed manually by their developers. The first language to support AOP was AspectJ ([1]) which is based on Java.

AOP has no construct which corresponds to a use case. It does not need it either, since AOP is a programming technique. AOSD does have constructs which correspond to use case modules. Such a construct contains a use case realization and a set of component slices that realize the use case. Each slice includes the design and implementation of that part of the use case which the component realizes.

In the context of AOP a use case realization is either part of the base program or an aspect. Peer use cases are mostly part of the base program, they are implemented as traditional code. Extension use cases are mostly implemented as aspects. An aspect is a piece of code which implements crosscutting behavior.

Extension points, called join points in AOP, are selected using some mechanism, e.g. regular expressions. The behavior of an extension is called an advice in AOP. Such an advice gets executed at each join point for which the mechanism selects this join point.

2.3 Cases for the UML Component Method

This section presents two cases for the UML Component Method: a hotel reservation system and a student mark registration system. The first case is based on an existing case (see [4]), the second case is newly created and therefore more elaborated.

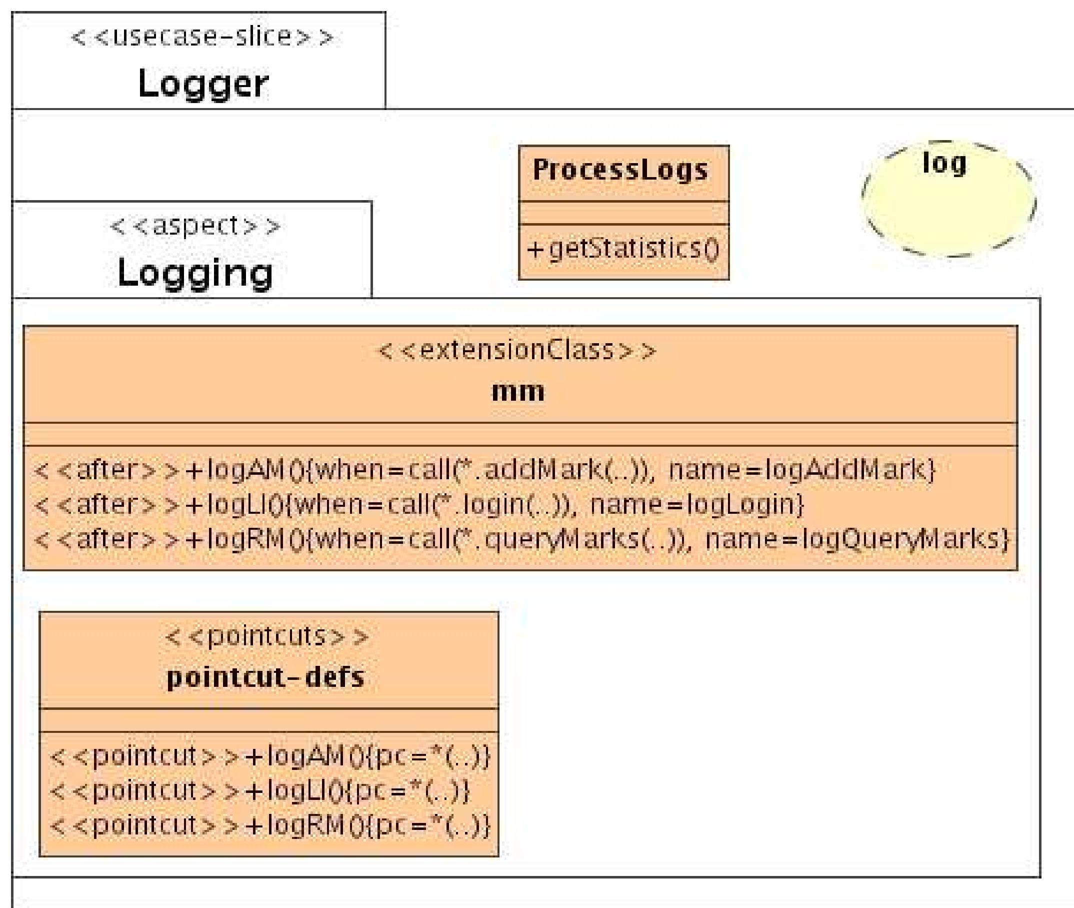

Figure 2.4: A Complete Use Case Slice

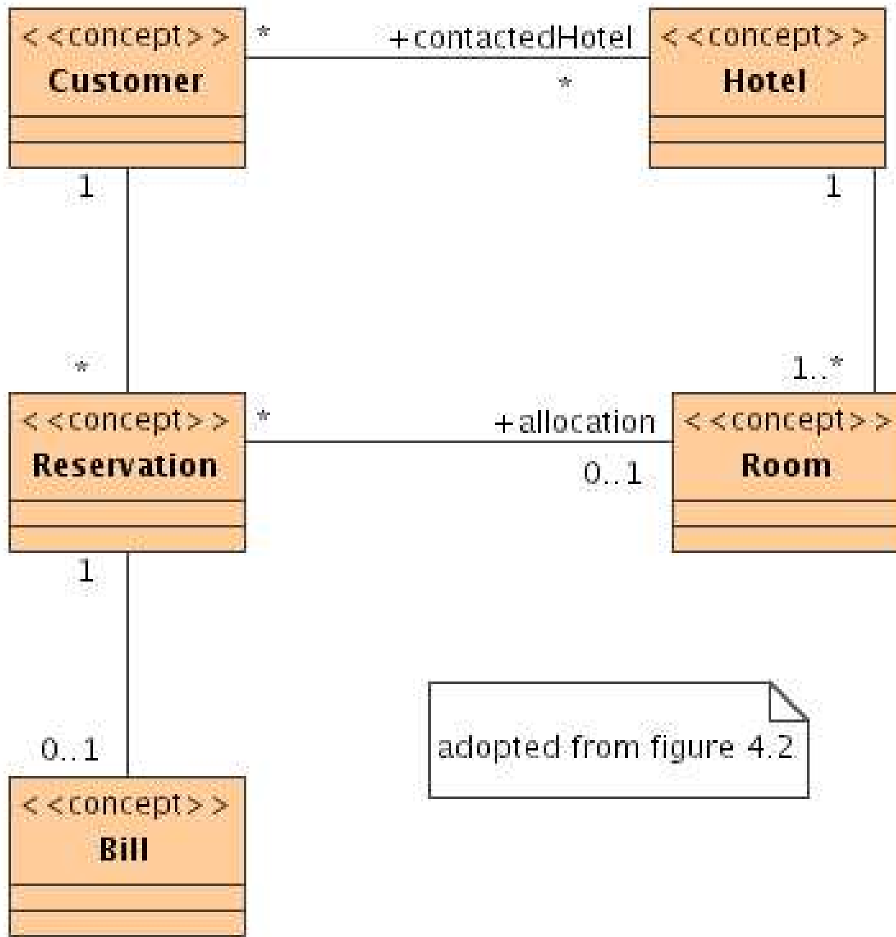

Figure 2.5: Business Concept Model for the Hotel Reservation System

2.3.1 Hotel Reservation System

This case consists of a part of a hotel reservation system as described in [4]. This case is used as an example for the UML Component method. It is also used to test the transformation functions. It only uses the model parts that are relevant to the topic being researched and may also change the model as originally defined.

Figure 2.5 shows that a customer can request an online reservation by selecting a hotel, a period, and a room type. In this figure a *Customer* contacts a *Hotel* which consists of at least one *Room*. A *Reservation* is then bound to the *Customer* and a *Room* and an optional *Bill* is started. The business concept model is manually refined into a business type model. The result of this refinement is shown in figure 2.6. This figure omits the *Bill* and *Hotel* classes, as these two classes are not necessary for the software system which implements the reservation system.

Figure 2.7 shows the two use cases for the customer. Note that these use cases are packed in a package *ReservationSystem*. Only the case of taking up a reservation is considered. In the *Make Reservation* use case the customer makes a reservation for which the system provides a reservation tag to the customer. This use case will not be considered further, as the more elaborate *Take Up Reservation* use case will be described. These two use cases are roughly analogous. The package *ReservationSystem*, which models a system, will be used later on.

Figure 2.8 and 2.9 show the sequence diagrams when a customer takes up a reservation. In this example figure 2.9 could be included in figure 2.8 but it is also used for other use cases in [4].

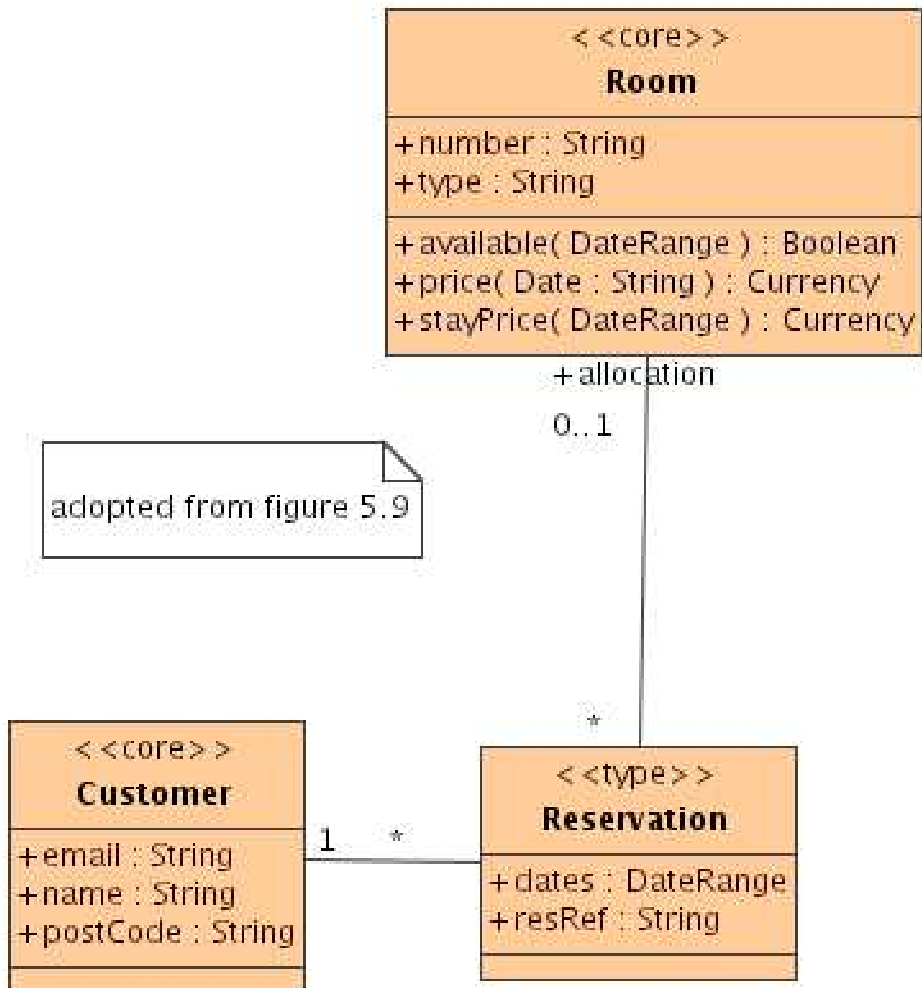

Figure 2.6: Business Type Model for the Hotel Reservation System

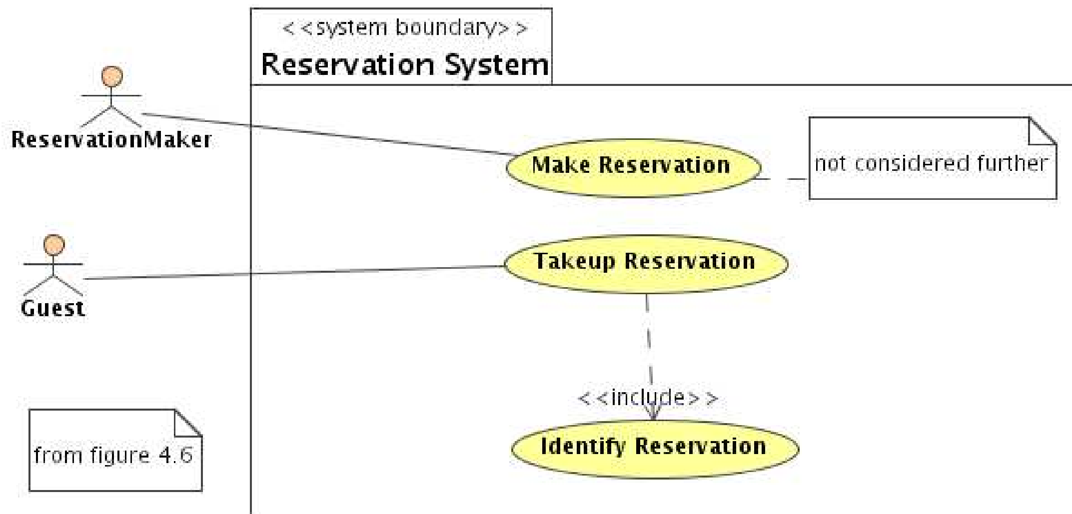

Figure 2.7: Use Case Model for the Hotel Reservation System

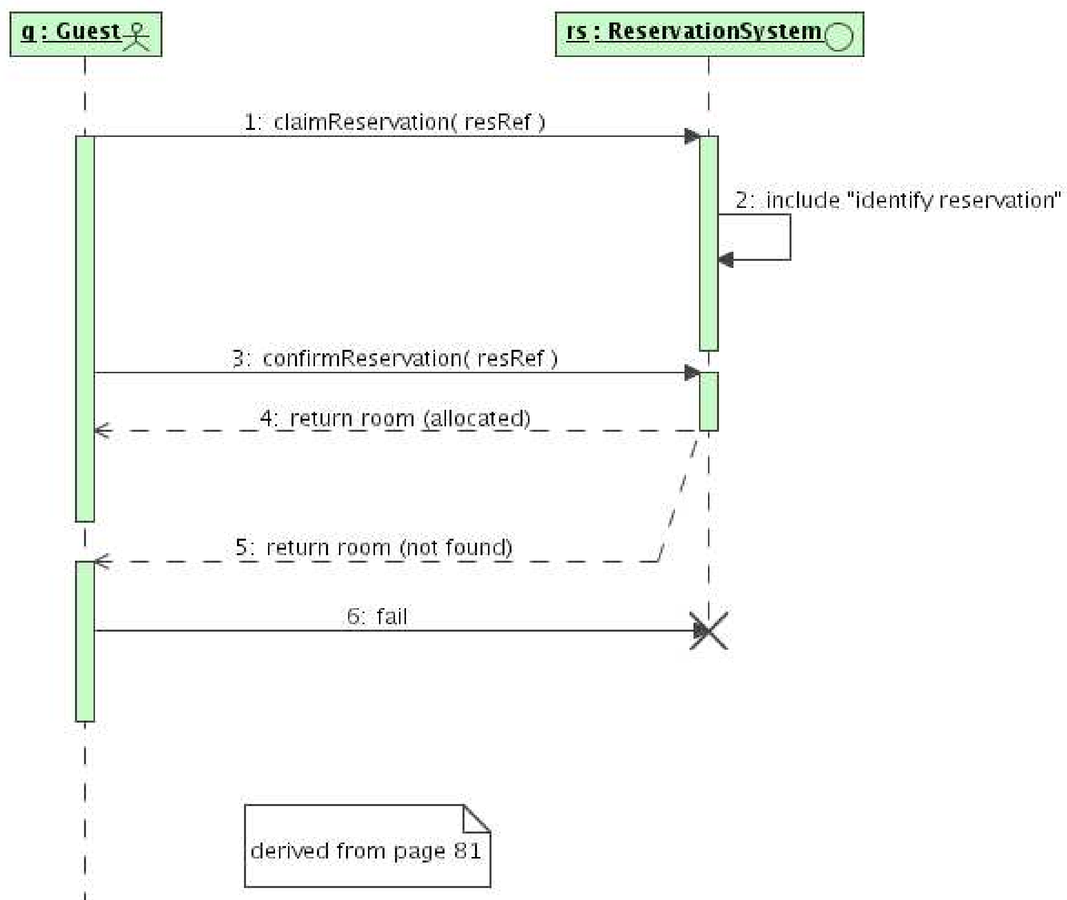

Figure 2.8: Sequence Diagram Detailing the *Take Up Reservation* Use Case

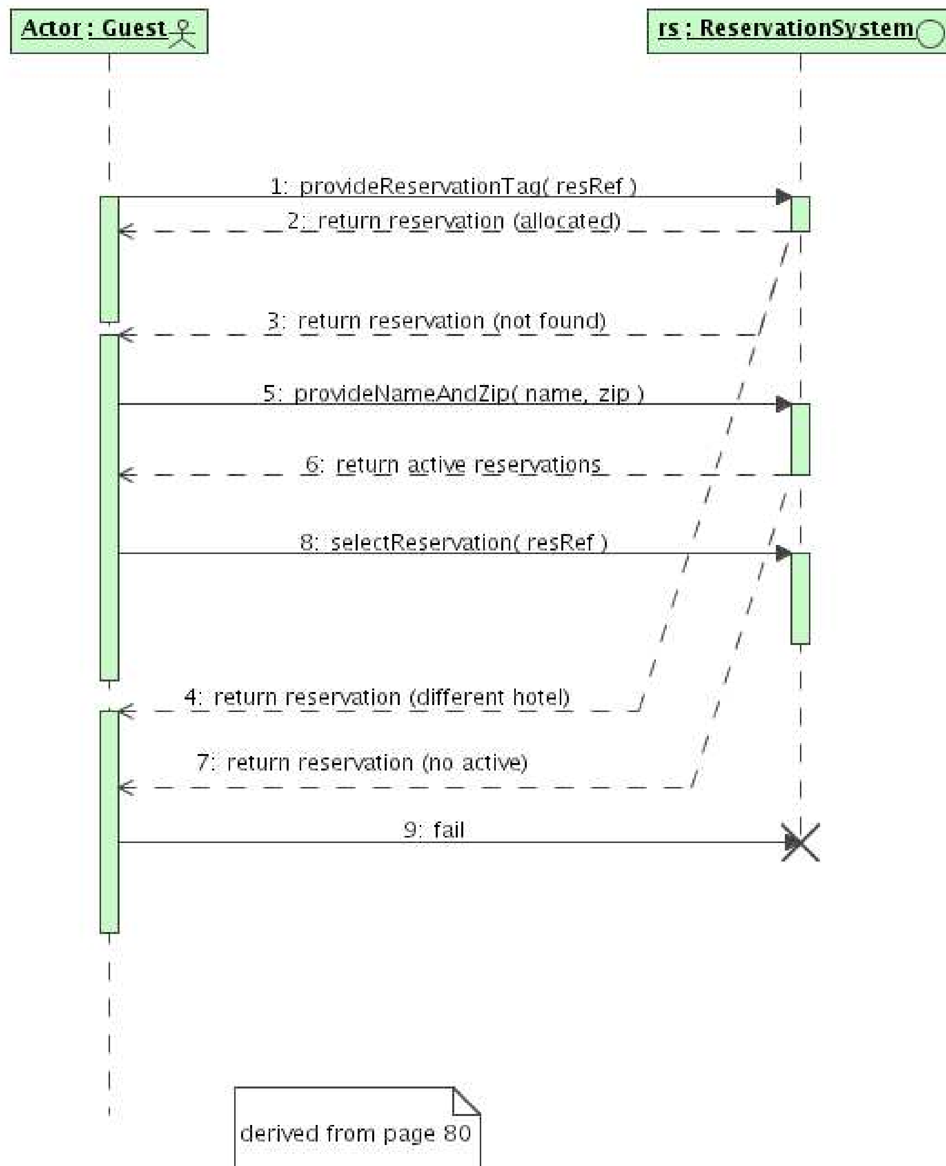

Figure 2.9: Sequence Diagram Detailing the Included *Identify Reservation* Use Case

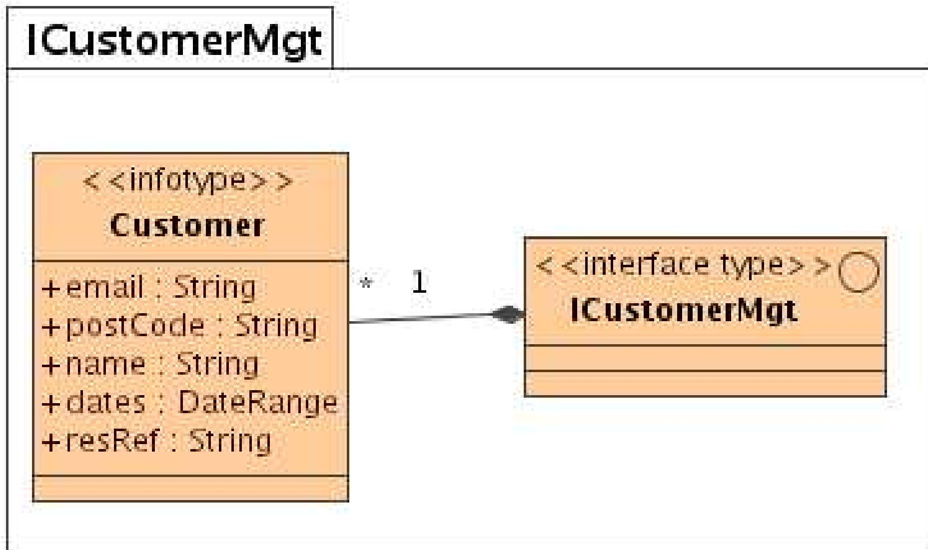

Figure 2.10: Business Interface for the <<core>> Class *Customer*

The business interface which is derived from the <<core>> class *Customer* of the business type model is shown in figure 2.10. The business interface for the class *Room* is analogous. The <<infotype>> class *Customer* contains all the attributes from the <<core>> class *Customer* and all attributes of all <<type>> classes associated with class *Customer*.

The system interface which is derived from the use case *Take Up Reservation* is shown in figure 2.11. The methods contained in this interface are derived from the 'call' messages of the sequence diagram in figure 2.8 and of the messages of all included sequence diagrams, here the sequence diagram in figure 2.9. Any <<infotype>> classes which are used in the signatures of the interface methods are associated with the interface through a composition, here <<infotype>> class *Room*.

For this case the component and system architecture is given in parts:

- An empty component specification is generated for each use case package and each <<core>> class. Figure 2.12 shows the component specifications for the use case package *ReservationSystem* and the <<core>> class *Customer*. The component specification for the <<core>> class *Room* is analogous.

- A package is created for each stand-alone use case. This package contains a <<boundary>> class and a <<control>> class to implement the use case. This is shown in figure 2.13.

2.3.2 Student Mark Registration System

This example provides a walkthrough of the UML Component Method of Daniels and Cheesman. This case is used as a pilot to determine what the functions for the UML Component Method should look like. This is done by determining which parts of which steps can be automated and what these automations should look like. The case is also used as an example of weaving model parts into a reference architecture. A reference architecture is some layered architecture like the one in figure 2.1. This is already an implicit part of the UML Component Method.

The system describes a student mark registration system. In this system students can view their own marks and teachers can add and view student marks. The input of this case consists of three diagrams. This example provides three input diagrams: the business concept model in figure 2.14, the use case diagram in figure 2.15, and the optional business process in figure 2.16.

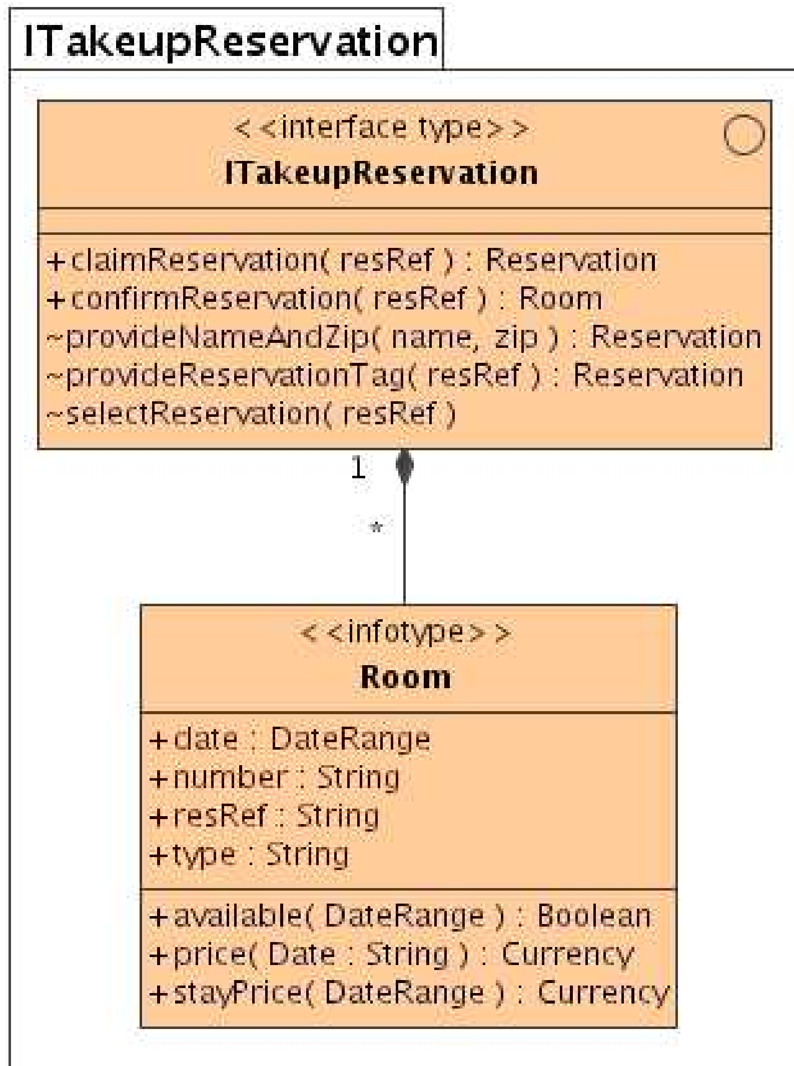

Figure 2.11: System Interface for the Use Case *Take Up Reservation*

Figure 2.12: Component Interfaces for *ReservationSystem* and *Customer*

Figure 2.13: <<boundary>> and <<control>> Class for the Use Case *Take Up Reservation*

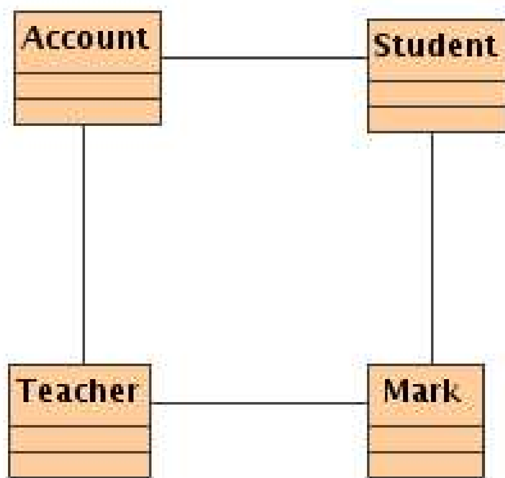

Figure 2.14: Business Concept Model of the Student Mark Registration System

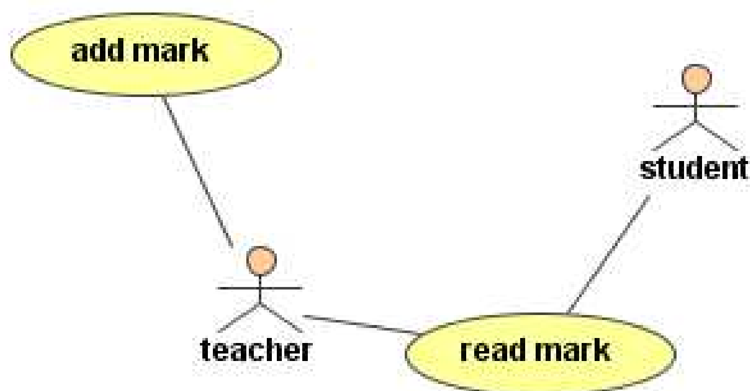

Figure 2.15: Use Case Diagram of the Student Mark Registration System

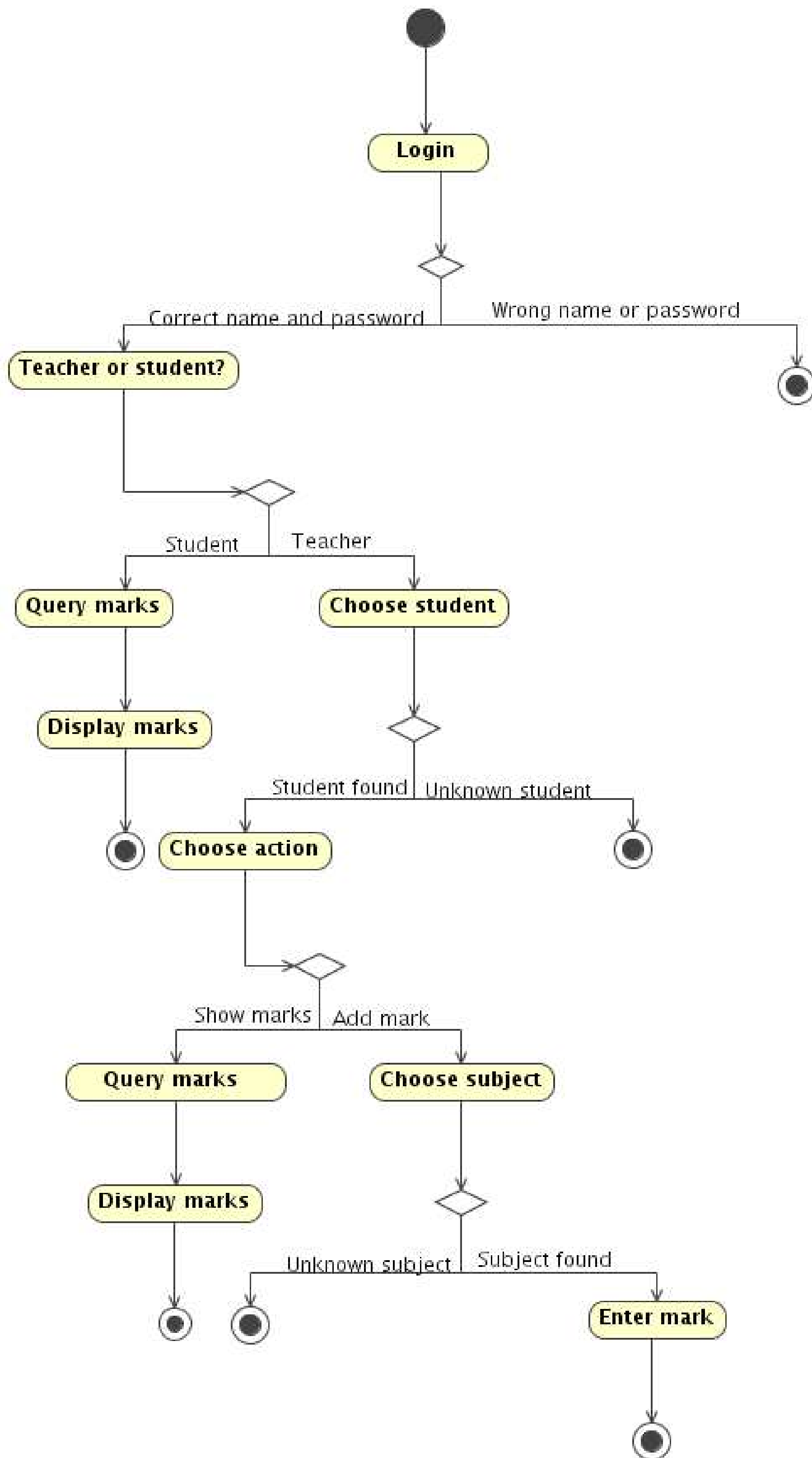

Figure 2.16: Business Process of the Student Mark Registration System

Figure 2.17: Student Views Marks

Figure 2.14 models four classes: *Account*, *Student*, *Teacher*, and *Mark*. The idea is that a *Teacher* gives a *Mark* to a *Student* and that a *Student* can view this and other *Mark*s. Both *Teacher*s and *Student*s have an *Account*, which is required to access the system. In figure 2.15 a *Teacher* can view and add marks while a *Student* can only view marks.

In figure 2.16 the business process starts with logging in to the system. If the user provides the right credentials, a menu appears depending on the role of the user: teachers are first presented with an extra menu in which they choose a student. The teacher can then choose to add or view marks of this student. If the teacher wants to add a mark, a subject has to be chosen after which the mark is entered. If the teacher or student wants to view marks, a menu appears with which the results can be queried after which the queried results are shown. If something goes wrong, the system halts, possibly displaying an error message. The system also terminates successfully after marks are added or displayed. In this example figure 2.17, 2.18, and 2.19 respectively represent the cases of a student viewing his own marks, a teacher viewing the marks of a student, and a teacher adding a mark. In figure 2.17 the interactions are as follows:

1. Student logs on to the system.
2. Login is successful, continue with step 4.
3. Student is unknown, fail.
4. Student queries and views marks, exit successfully.
5. Failure step.

In figure 2.18 the interactions are as follows:

1. Teacher logs on to the system.
2. Login is successful, continue with step 4.
3. Teacher is unknown, fail.
4. Teacher selects a student.
5. Student is known, continue with step 7.

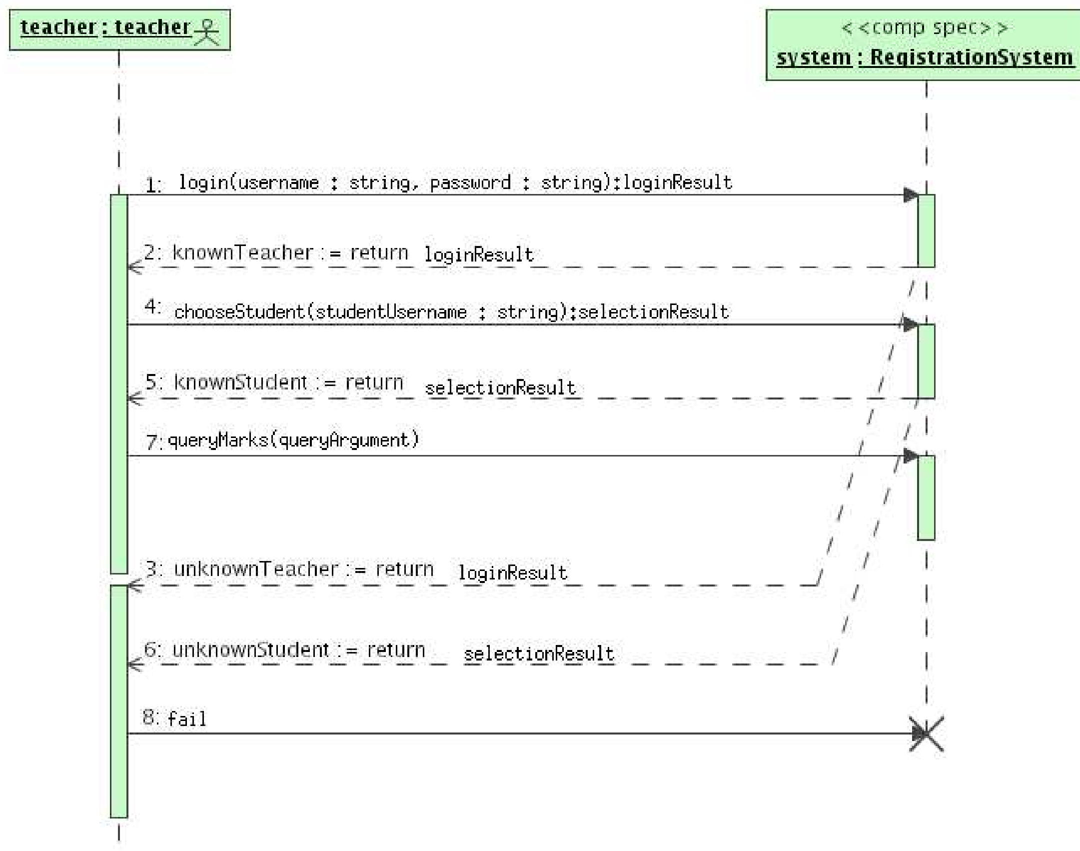

Figure 2.18: Teacher Views Marks

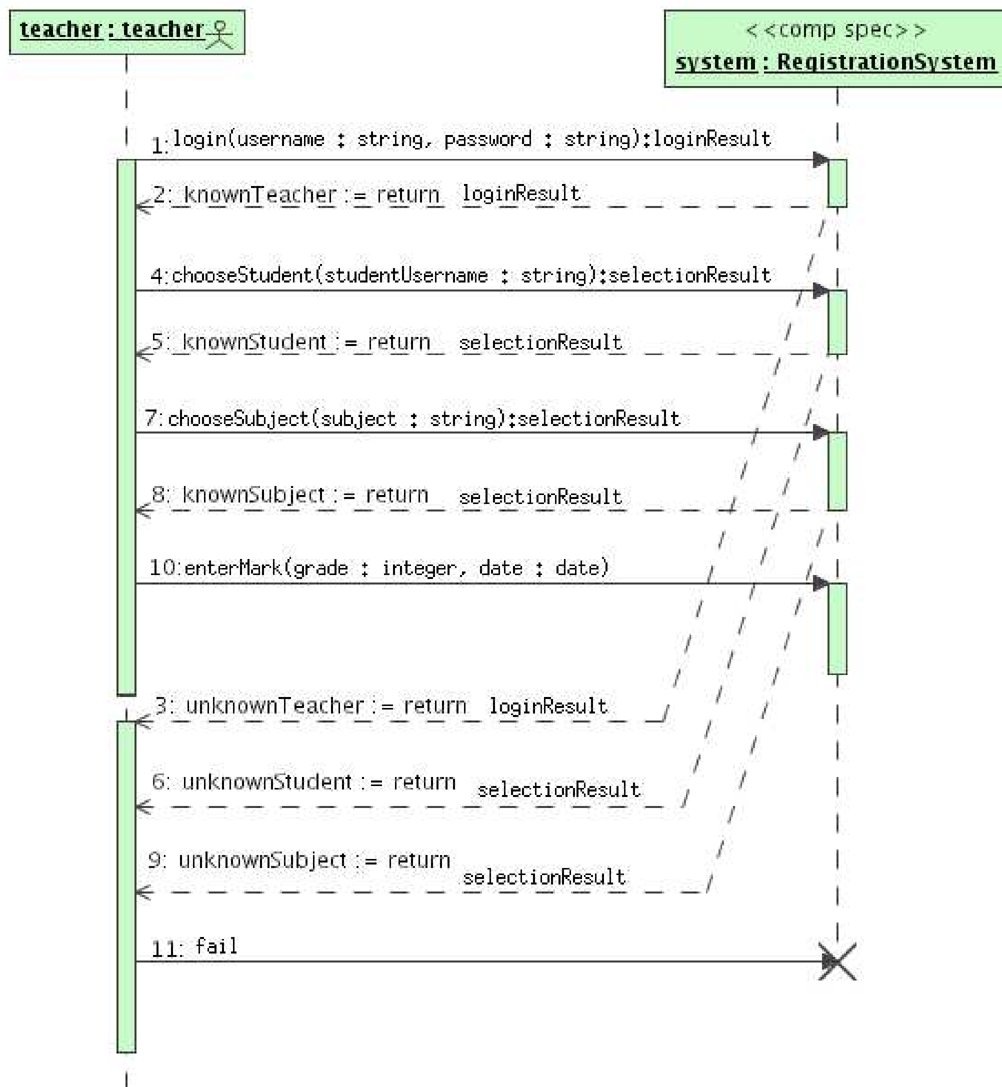

Figure 2.19: Teacher Adds Marks

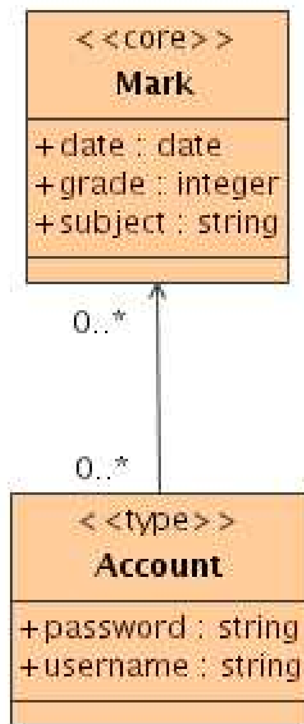

Figure 2.20: Business Type Model of the Student Mark Registration System

6. Student is unknown, fail.
7. Teacher queries and views marks, exit successfully.
8. Failure step.

In figure 2.19 the interactions are as follows:

1. Teacher logs on to the system.
2. Login is successful, continue with step 4.
3. Teacher is unknown, fail.
4. Teacher selects a student.
5. Student is known, continue with step 7.
6. Student is unknown, fail.
7. Teacher selects a subject.
8. Subject is known, continue with step 10.
9. Subject is unknown, fail.
10. Teacher adds a mark to this subject for the chosen student, exit successfully.
11. Failure step.

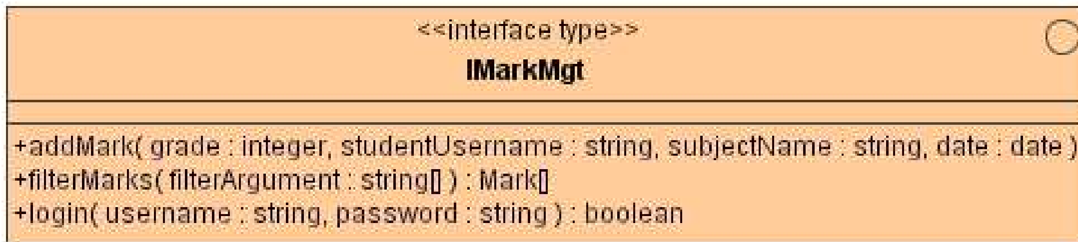

Figure 2.21: Business Interfaces of the Student Mark Registration System with *Manually* Added Methods

In this example (figure 2.20) *Mark* is a <<core>> class while *Account* is a normal <<type>> class. Students and teachers themselves do not have to be represented in this software system, their account suffices. A general account class suffices because a teacher account is identical to a student account. This is why the *Student* and *Teacher* classes of the business concept model have been removed. In this business type model the added directions are from *Account* to *Mark*. This way the *password* fields of *Account* classes are protected from the *Mark* class. The multiplicities between the *Account* classes and the *Mark* class are many to many: a certain mark can be given to any student by any teacher, and a certain teacher can give any mark to a certain student.

The business type model also defines several business rules. These rules, which must be adhered to by the business layer of the Student Mark Registration System, are:

- All dates are in the format $yyyy - mm - dd$.
- Each teacher and each student has a username and a password.
- The username of each teacher and each student is unique.
- Each student can only view his or her marks.
- No student can edit marks.
- Each teacher can view all marks of all students.
- Each teacher can add marks for any student for any subject.
- If a subject has multiple marks, then their dates must be different.

This example has only one <<core>> class in the business type model, which means that only one interface exists at the business layer. This interface, *IMarkMgt*, is shown in figure 2.21. The methods of *IMarkMgt* are derived from methods of interfaces at the system layer and will be explained at the description of the system interfaces.

This example features three system interfaces: *IStudentViewMarks*, *ITeacherViewMarks*, and *ITeacherAddMark*. These are shown in figure 2.22. They are derived from respectively figure 2.17, 2.18, and 2.19. Because the use cases are independent of each other, the system interfaces are also independent of each other.

A summary of the methods of all system interfaces follows:

- Methods appearing in multiple interfaces:
  - $chooseStudent(studentUsername : string) : boolean$.
    Try to choose the student by user name. The user name is specified in $studentUsername$. If the student exists, choose it and return $true$, otherwise return $false$.
  - $login(username : string, password : string) : boolean$.
    Try to log in the user with the username specified in *username* and the password specified in $password$. If the credentials are correct, log in and return $true$, otherwise return $false$.

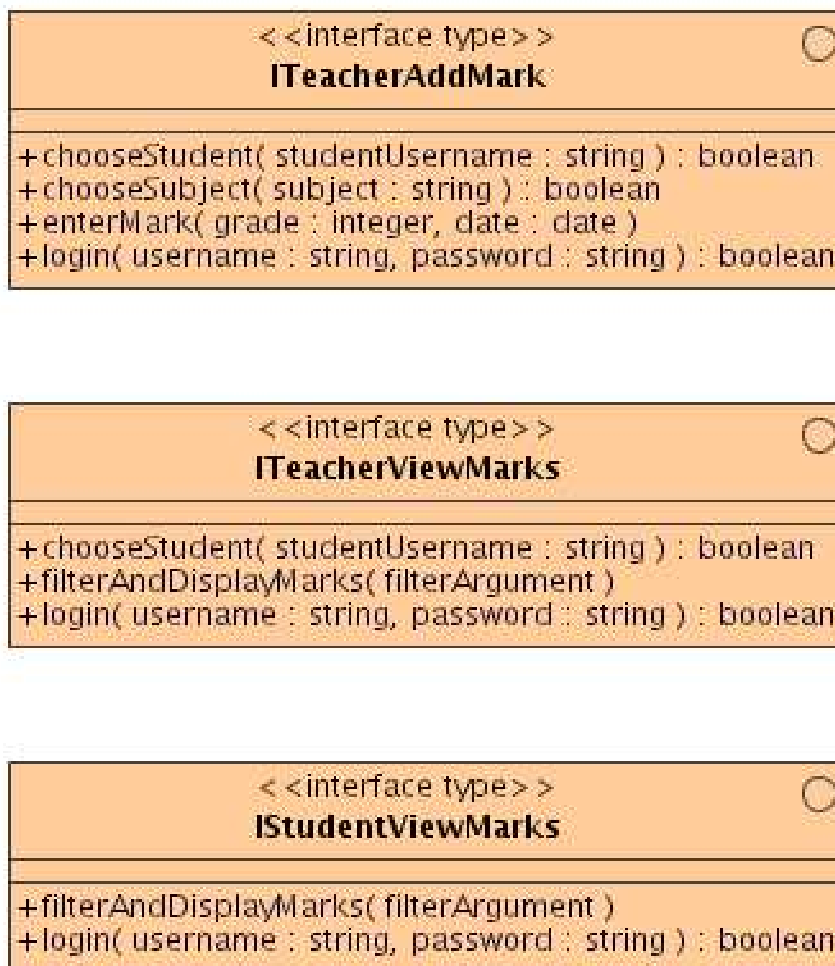

Figure 2.22: System Interfaces of the Student Mark Registration System

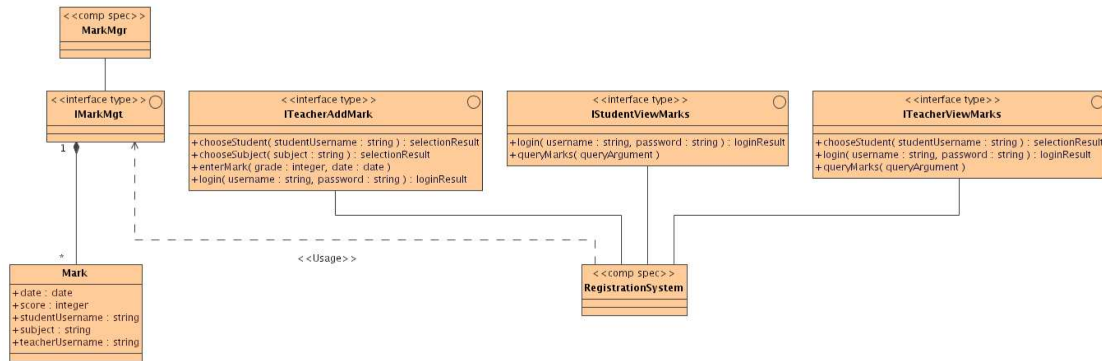

Figure 2.23: Component and System Architecture of the Student Mark Registration System

  - $QueryMarks(queryArgument)$.
    Retrieves and displays the marks of the specified student using the query specified in $queryArgument$.

- $ITeacherAddMark$

  - $chooseSubject(subject : string) : boolean$.
    Try to choose the subject specified in $subjectName$. If it exists, choose it and return $true$, otherwise return $false$.
  - $enterMark(grade : integer, date : date)$.
    Enter a mark for the subject specified with $chooseSubject()$. The grade is specified in $grade$. The date of the test is specified in $date$.

- $ITeacherViewMarks$. The methods of this interface are already discussed.

- $IStudentViewMarks$. The methods of this interface are already discussed.

In this example the component and system architecture is shown in figure 2.23. The component specifications are $MarkMgr$ (for the business interface) and $ReservationSystem$ (for the system interfaces). The information base belonging to the business interface is represented by the data type $Mark$. It should be no surprise that this data type is closely related to the class $Mark$. Three fields have been added to the data type representing the information base:

- $studentUsername$ provides the username of the student who has received a particular mark.

- $teacherUsername$ provides the username of the teacher who has entered a particular mark.

The last two fields are derived from the relations present in the business type model.

2.4 Transformation Types for the UML Component Method

This section gives a summary of transformation types required to implement the UML Component Method as functions. The required transformation types as obtained from this chapter are:

- (a1.1) Add an interface, package, or class with a given name, stereotype(s), and tag(s) to a package or namespace. The added elements are either new or derived from another class, actor, use case, or interface.

- (a1.2) Add attributes or operations with a given name, type, multiplicity, and visibility to a class. These attributes or operations are either new or copied from another class.

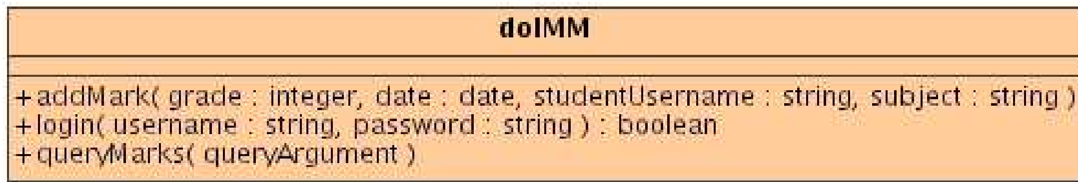

Figure 2.24: Class *doIMM* before Weaving

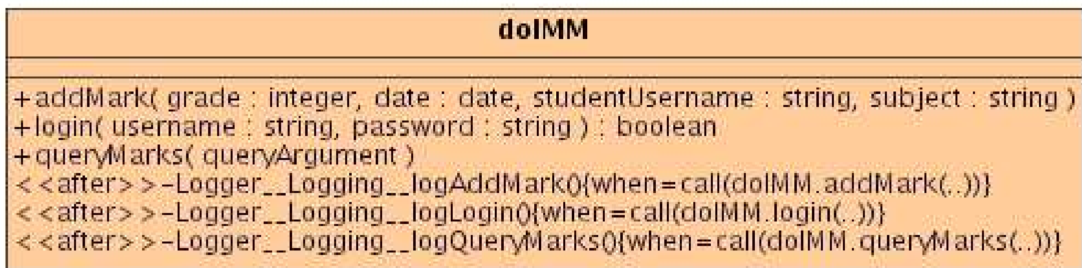

Figure 2.25: Class *doIMM* after Weaving

- (a1.3) Transform messages of a sequence diagram into interface methods with a given name, parameter(s), visibility, and return type. The parameters of these methods have a given name and type.
- (a1.4) Add an association, dependency, or composition.
- (a1.5) Change the name of the actual parameter.
- (a1.6) Make it possible to combine different diagram types in a transformation.

2.5 Case for AOSD

A concrete example of an AOSD application is logging the actions of the users of the Student Mark Registration System. For this system it means that viewing and adding marks is logged as well as logging on to the system. This case can be used to aid in deriving UML functions for AOSD, especially for aspect weaving. Figure 2.4 shows the use case slice which implements these logging capabilities. The methods *queryMarks*, *addMark*, and *login* of the class(es) implementing the interface *IMarkMgt* are respectively extended with the extensions *logQueryMarks*, *logAddMark*, and *logLogin*. These extensions are called after their "base" methods. The new extensions are woven into the class(es) of their "base" method. Figure 2.24 shows the class *doIMM* before weaving. This class implements the interface *IMarkMgt*. Figure 2.25 shows class *doIMM* after weaving. Note that the extensions are added as private methods and that their complete original namespace is prefixed. This prefixing is necessary to avoid name clashes when weaving method extensions with the same name from different aspects or different use case slices. Also note that the behavioral pointcuts are preserved as these are resolved at runtime. The collaboration *log* only serves to realize the use case. It is not part of the implementation or of the weaving result.

Non-use-case-specific slices are stereotyped as <<non-usecase-specific-slice>>. These slices normally belong to multiple use cases. They do not contain aspects.

Slices can contain relations to indicate which relations are specific for a certain use case and which relations are non-use-case-specific. If such a relation is connected to an aspect, then this relation needs to relate any class affected by this aspect to the other end of this relation. Like other UML elements, use case slices and non-use-case-specific slices can be related to each other. The resulting structure of slices is called a use case structure. This structure indicates how the slices are to be overlaid onto each other. This

structure is normally indicated as a class diagram, in which dependencies indicate which slices are required by other slices. So if slice *a* requires slice *b*, then slice *b* is to be overlaid onto slice *a*.

2.6 Transformation Types for AOSD

This section gives a summary of the transformation types required to implement AOSD as functions. All transformations are centered around use case/non-use-case-specific slices. The name of each class, method, pointcut, or aspect can be parametrized by surrounding it with angle brackets.

Several transformations reference a set *C*. This set consists of class names. These class names are either static (parametrized) or dynamic. In the latter case, the class names are specified as a structural pointcut predicate. If *C* contains predicates, then these predicates must be surrounded by square brackets. The required transformation types are:

- (a2.1) Add a class with specified contents which is contained in a slice to a namespace.
- (a2.2) Merge attributes or operations into a specified set of classes *C*.
- (a2.3) Merge operation extensions as specified by the structural pointcuts into a specified set of classes *C*.
- (a2.4) Tag operations affected by behavioral pointcuts with these pointcuts.
- (a2.5) Remove <<aspect>>, <<usecase-slice>>, and <<non-usecase-specific-slice>> packages once their contents is processed.
- (a2.6) Add relations from within a slice to a namespace.

Apart from the above tagging, behavioral pointcuts are ignored by the transformations as they are runtime issues.

Structural join points together with set *C* indicate which classes need to be extended with operations. Figure 2.4 has only one set of classes *C*. In general slices can have multiple sets of these classes.

2.7 Answered Research Questions

This chapter answered two research questions. Section 2.4 listed the transformation types needed for the UML Component Method. This provided the answer for research question 1. Research question 2 is answered by section 2.6. This section listed the transformation types needed for AOSD.

Chapter 3

An Analysis of Possible Solutions

"If a problem is not completely understood, it is probably best to provide no solution at all."
—FreeBSD Developer's Handbook, chapter 1.3

This chapter discusses the possible solutions which I found worth considering for the research. The chapter mentions the selection criteria, a list of possible solutions, and the final solution chosen from this list.

3.1 Selection Criteria

This section sets out the criteria used to select the possible solutions. The criteria are selected from both the view of the programmer and the view of the end user. They should be satisfiable for both groups. The criteria are for the proposed solutions are:

- (b1) Suitability in the production line of GPR. This criterion tells how well the solution cooperates with other tools of GPR. It also focuses on things like speed and robustness, since these are factors of competitive interest.
- (b2) Specification. This criterion tells how concise the specification is. It also gives an indication of the maturity of the solution as a whole.
- (b3) Ease of understanding. This criterion is somewhat subjective. It tells how well I personally understand the solution. It also gives a measure of how well the solution fits in from a common sense point of view.
- (b4) Availability. This criterion tests if the solution is publicly and freely available.
- Ease of use. This criterion is split into three subcriteria:
  - (b5.1) End user usability. This criterion is somewhat subjective. It is rated by me imagining an end user actually using the final solution.
  - (b5.2) Ease of programming. This criterion is somewhat subjective. It reflects my familiarity and adaptability with the considered programming languages. It also indicates the rate between figuring out the solution itself and how to implement the solution in the programming language. For a well chosen programming language the main activity should be the former.
  - (b5.3) Abstraction level. This criterion reflects the abstraction level of the solution: does it work at an XML or a UML level? How does the programming language fit in?

3.2 Existing Methods

This section discusses some methods which handle related subjects. Some of them are fully implemented while others are only proposals.

3.2.1 XMI.difference

This proposal comes from Annika Wagner ([20]). In this proposal models and rules to transform models are written in UML. These UML entities can be specified using any UML tool which can export to XMI. After the entities are modeled they are passed to a transformation preparer and finally to a transformation machine.

The transformation preparer has three tasks:

1. Find occurrences of the rule pattern in the model. This pattern describes when and where to apply the rule.
2. Copy the effect of the rule to the model. This effect is specified as a $XMI.difference$ entity. XMI.difference is a part of the XMI standard which specifies differences between XMI models using add, replace, and delete operators.
3. Substitute the links to the rule pattern with the links to the model.

This approach is limited to UML models which has the advantage that the semantic checks are already built in into the meta-model itself. Some reasons to avoid this method are:

- It is only a proposal and no progress information seems to be available.
- The description is open-ended.
- Support for XMI.difference in tools is still lacking as of November 2005.

3.2.2 XMI-based Model Transformations

This proposal comes from Jernej Kovse and Theo Härder ([11], section 2.2). It uses an XSLT template (T) to transform an XMI model (m_i) into another XMI model (m_j). The template has parameters which have to be instantiated to create T_k, which can be applied to m_i. The template is specified by the designer. They define how a transformation generator generates T_k from T.

The generator can use the following three mechanisms for the above generation:

- Static parametrization ($< xsl : template >$) uses parameters from T to generate concrete properties of differential elements used in T_k. For example, these properties can be names or multiplicities.
- Iteration ($< xsl : for-each >$): the parameters of T enable the agent to influence how often a segment of T will be used in T_k.
- Conditional include ($< xsl : if >$, $< xsl : choose >$) is analogous to iteration, except that the agent decides whether a segment will be used.

If an XSLT implementation is chosen, our system could use some ideas from the above system.

3.2.3 Proposals of DSTC

This section discusses some proposals to transform models by the DSTC University of Australia. Their research focused on transforming an EDOC Business Process Model into a Breeze Workflow Model, which are both XMI models.

Text

Text-based transformations could only be used for small transformations. This was because no abstraction of a parse tree existed and because the tools (like awk or Perl) operate on instance level instead of model level.

The suggestion that Perl lacks a XML parser was already false when the paper was written in 2002, as $XML :: Parser$ already existed in September 1998. This option will not be considered further when using an awk or Perl implementation.

XSLT

XSLT as a transformation language looks like a good choice, because both the input and output model are written in XMI and XSLT is especially written for XML transformations. However, some scaling problems exist:

- XSLT is quite verbose, so the transformations can become unreadable.
- XSLT is hard to debug.
- Because XSLT passes arguments by value: it becomes inefficient with large transformations.

Another problem of XSLT transformations is the lack of abstraction: it requires the syntactic details of the target model. However XSLT has become a de facto method for XML transformations, so many tools should support it.

Mercury

Mercury is a purely declarative language. This style provides unification for the pattern rules of the source instance graphs. It also has an efficient implementation and a CORBA IDL interface for a possible connection to MOF repositories.

The disadvantage of Mercury is that it cannot capture inheritance in models. This leads to a fragmented ruleset which becomes hard to maintain.

All models need to be represented explicitly to make their semantics available when writing the rules and to be able to define multiple targets in a single rule. The latter possibility should lead to more compact rules.

I do not feel particular strong about this option because of the fragmented ruleset syndrome.

F-Logic

F-Logic, which stands for Frame Logic, is a deductive object orientated language based on Prolog.

Apart from the fact that a production quality implementation was unavailable at the time the paper ([7]) was written, the language seems to have a lot of advantages:

- a flexible, compact syntax for defining rules.
- rules at both model and instance level.
- the possibility to define multiple targets in one rule. This leads to compact rules and avoids repetition.
- Matches can be on constraints, types, and associations.
- The rules can be written in a white-board like manner instead of as a mapping.

Normally rules are grouped for readability. F-Logic encourages to combine them instead.

A quite extended implementation of F-Logic is the Flora2 language which comes bundled with XSB ([24]). While this implementation is publicly available from SourceForge, the underlying language is not yet finalized. Apart from F-Logic it also implements HiLog to support meta-programming and Transaction Logic to support logical updates. Another implementation comes from Ontoprise. This implementation seems more mature but is not publicly available. The fact that the language is not yet mature makes it inadvisable to choose F-Logic as the implementation language, since the implementation needs to be modified each time the language is changed.

3.2.4 Python

Python comes with a DOM implementation, which makes it suitable for working with XML-based languages like XMI. In contrast to XSLT, which is a functional language, Python is an imperative language. Its imperativeness could be a disadvantage for Python, because the ruleset would have to be specified at a lower level than when using XSLT. On the other hand Python looks more familiar than XSLT and it can be learned in a shorter time than XSLT.

3.3 Comparison of Methods

This section gives a tabular overview of the discussed solutions and selects the most appropriate one. The scale used to measure the criteria consists of five discrete values: −− (very low), - (low), 0 (average), + (good), ++ (excellent). A question mark indicates that the rating is unknown.

Table 3.1: Summary of Possible Solutions

Solution	b1	b2	b3	b4	b5.1	b5.2	b5.3	Remarks
XMI.difference	−−	0	+	−	0	n/a	+	no tool support, proposal only
XMI-based	−−	-	0	−−	-	+	+	based on XMI.difference
Perl	0	+	+	++	+	0	0	
XSLT 2	++	+	+	++	+	0	+	Saxon 8 required
Mercury	-	+	0	+	+	?	0	on Windows: .NET (alpha) or Cygwin required
F-Logic	-	0	0	++	+	?	+	language in development, high learning-curve
Python	0	+	+	++	+	+	0	with pyxml plugin

The criteria were measured as follows:

- (b1): How fast and mature is the solution?
- (b2): How complete and concise is the specification?
- (b3): This criterion is somewhat person-dependent.
- (b4): This criterion is measured by the availability and price of the tool.
- (b5.1): The first two methods require that the end user performs multiple steps, the other methods only require one step: running a single tool.
- (b5.2): This criterion is completely person-dependent.
- (b5.3): How abstract is the model support and the language itself?

A first implementation was being written in XSLT 2.0. Work on this implementation was stopped because of a misunderstood requirement which XSLT cannot handle: that functions could specify a different location for their output than the current location in the model. The tool was reimplemented in Python, based on the XSLT code. Python made it possible to accommodate for the misunderstood requirement. The new implementation worked well until the memory requirements became too much to fit into an internal Python stack. From that moment, the tool was again reimplemented, this time based on the Python code. This time, it was clear that the misunderstood requirement was bogus, allowing a clean XSLT implementation. Although the new XSLT code is based on the old XSLT code, it is much cleaner. The new XSLT implementation has three advantages with respect to the Python implementation:

- It does not run out of memory.
- It is much faster.
- The implementation is smaller, which should make it easier to maintain.

The new XSLT implementation is the final one.

3.4 Transformations: UML 1.X or UML 2.0 ?

This section explains which UML version is best suited for specifying model transformations as used in this project.

UML 1.1 was published by the OMG in November 1997. This was the first official version of UML, version 1.0 was only a draft ([23]). In UML 1.X each concrete syntax (a diagram type) has its own abstract syntax ([5]), totaling to 14 syntaxes. On July 4th, 2005 the OMG published the UML 2.0 specification ([14]). UML 2.0 only has 2 abstract syntaxes (”Structure” and ”Behavior”) for 13 concrete syntaxes.

UML 1.X has certain limits for modeling specific software systems, such as realtime systems or component systems ([3]). It also lacks support for model driven development and model execution. UML 2.0 integrates activities and actions. UML 1.X, like UML 2.0, has an action model. This model is insufficient for the specification of precise semantics (at a programming language level) and is therefore replaced by a new action model which can specify these semantics.

The main difference between UML 1.X and UML 2.0 regarding templates is that the possibilities for templateable elements have been restricted to those for which it is, according to the OMG, meaningful to have template parameters, whereas in UML 1.X any classifier, package, or operation template was allowed. The UML specification unfortunately does not elaborate on this restriction.

These restrictions of the OMG can further restrict the modeling of templates. With UML 1.X, it seems already impossible to draw two different diagram types into one diagram, which limits the possibilities to express dependencies between these diagram types. Another difference is that in UML 2.0 templates are bundled in a package.

Some tools support both UML 1.X and UML 2.0, while others only support UML 1.X. The current version of ArcStyler, which is used as the MDA tool within GPR, only supports UML up to version 1.4 ([12]).

The above arguments suggest that UML 1.X is better suited for this project than UML 2.0, which means that UML 1.4 will be used.

3.5 Answered Research Questions

This chapter answered two research questions. Section 3.4 answered the question which UML version to use by choosing UML 1.4 as the modeling language. Section 3.3 answered research question 9 by choosing XSLT 2.0 as the implementation language for the tool.

Chapter 4

Theoretic Foundations

"If the facts don't fit the theory, change the facts."
—Albert Einstein

This chapter describes the various theoretical parts of the project. These include the UML function meta-model, the definitions used in the final solution, and rules for these definitions.

4.1 Model Transformation

This section describes a method to transform UML models into other UML models. This is done by schemes, the legend for these schemes is given in figure 4.1. The aim is to transform UML model *a* into UML model *b*. This is shown in figure 4.2.

The transformation is done in two major steps. One step transforms UML model *a* into UML model *b* using a generated script, the other step transforms a set of UML functions into the aforementioned script. Most UML tools use XMI to represent the UML models. XMI is a method to represent a UML model as one or more XML files. Appendix B provides an introduction to XML. Thus UML models *a* is represented as XMI model *a*. The conversion step is now reduced to convert XMI model *a* to XMI model *b*. This is shown in figure 4.3.

The transformation itself is given by some ruleset, which is applied on the input (XMI model *a*) by some program. In figure 4.4 this program is represented by an XSLT script. The script is generated. The script could also be programmed manually, but then each new UML transformation would have to be translated manually into an XSLT script. A generator, shown in figure 4.5, can perform this tedious task for us. Appendix D contains a short introduction into XSLT.

The input for the generator, a set of UML functions represented as XMI, is shown in figure 4.6 as XMI model f. Section 4.3 will explain these functions in detail.

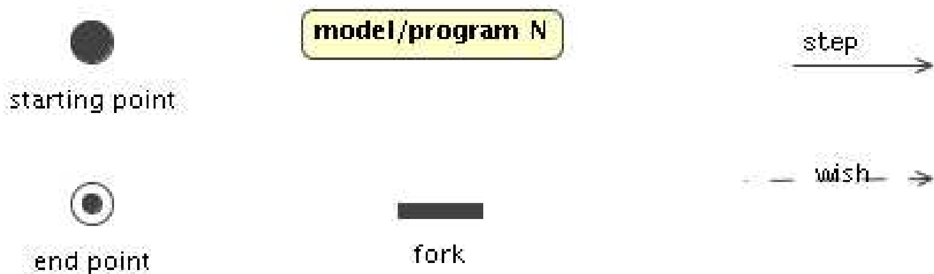

Figure 4.1: Transformation Goal Legend

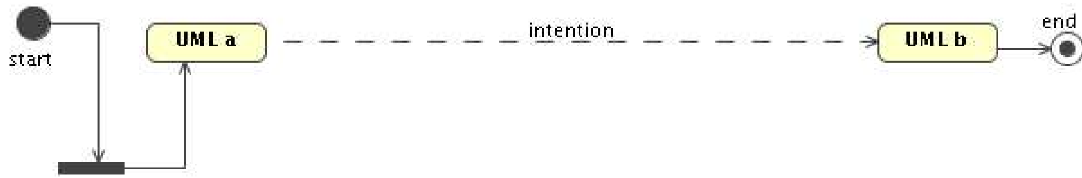

Figure 4.2: Transformation Goal, Part 1

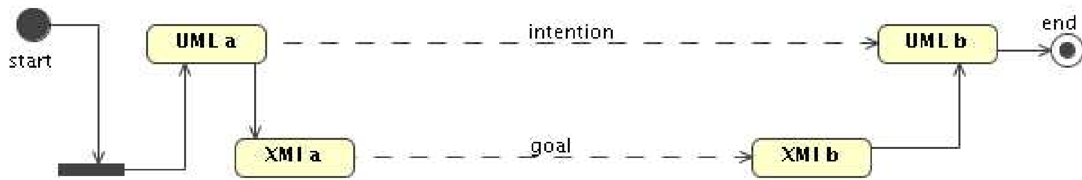

Figure 4.3: Transformation Goal, Part 2

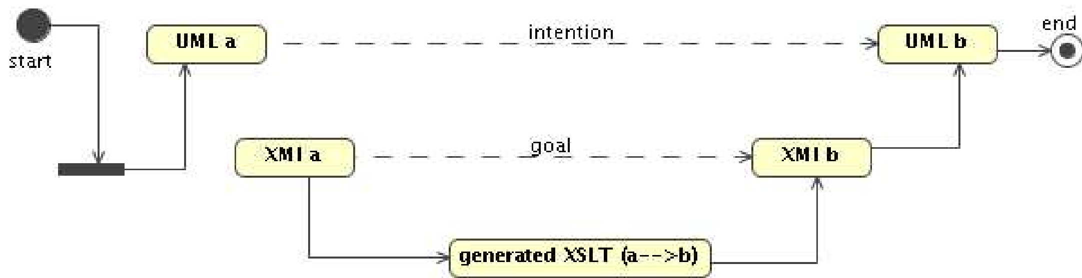

Figure 4.4: Transformation Goal, Part 3

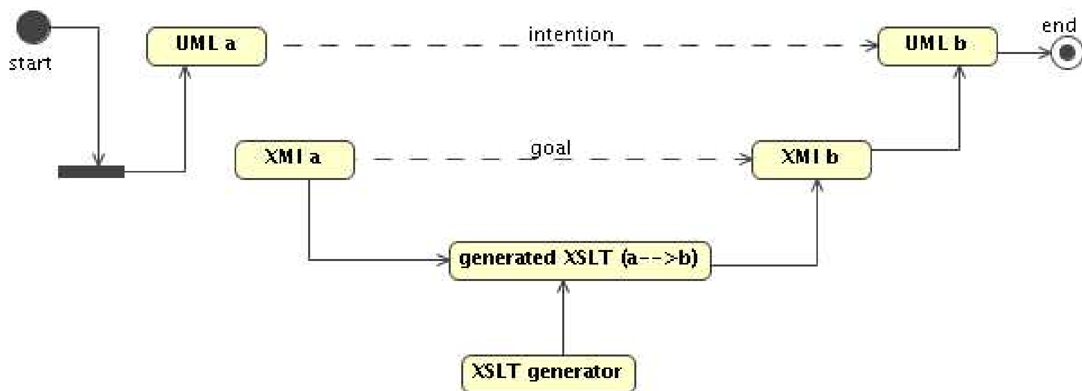

Figure 4.5: Transformation Goal, Part 4

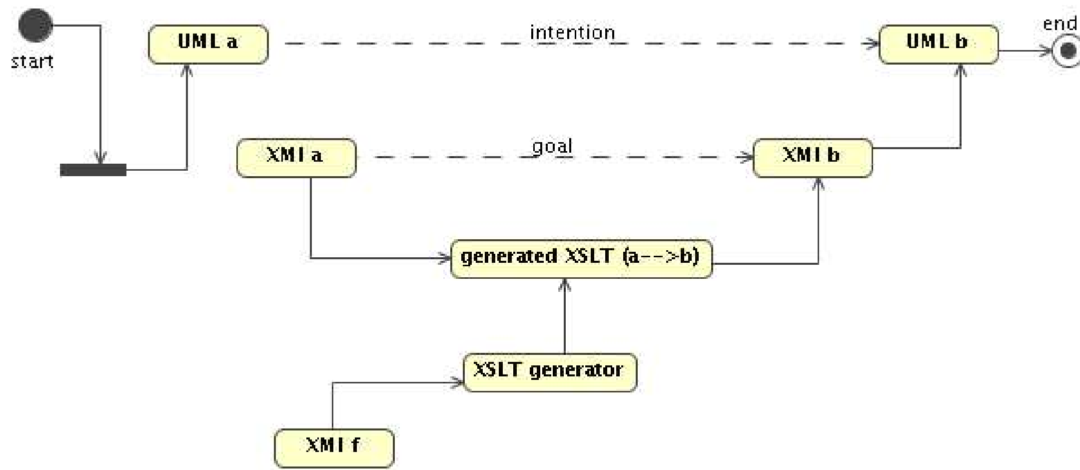

Figure 4.6: Transformation Goal, Part 5

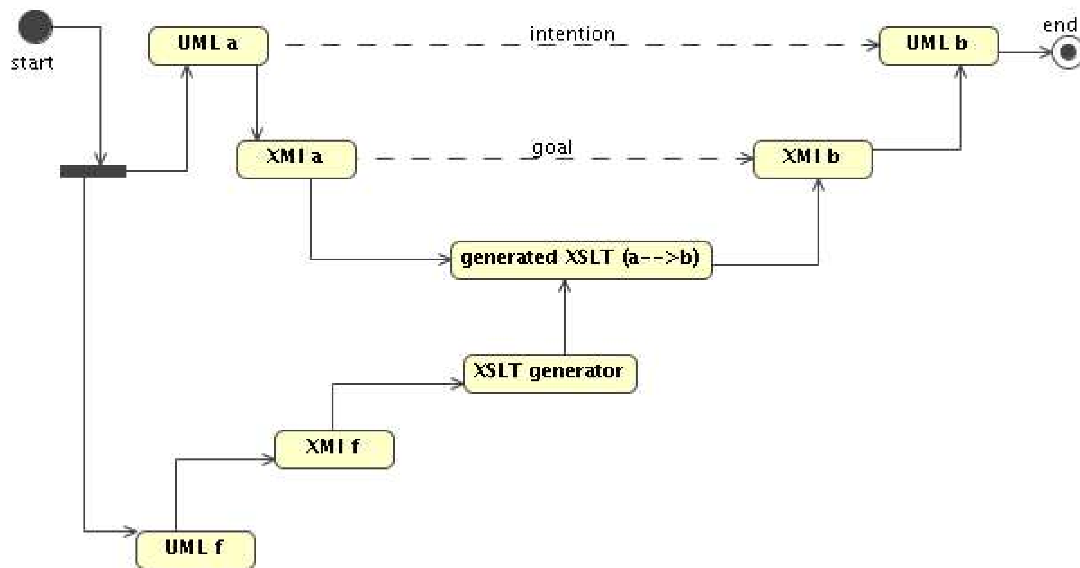

Figure 4.7: Transformation Goal

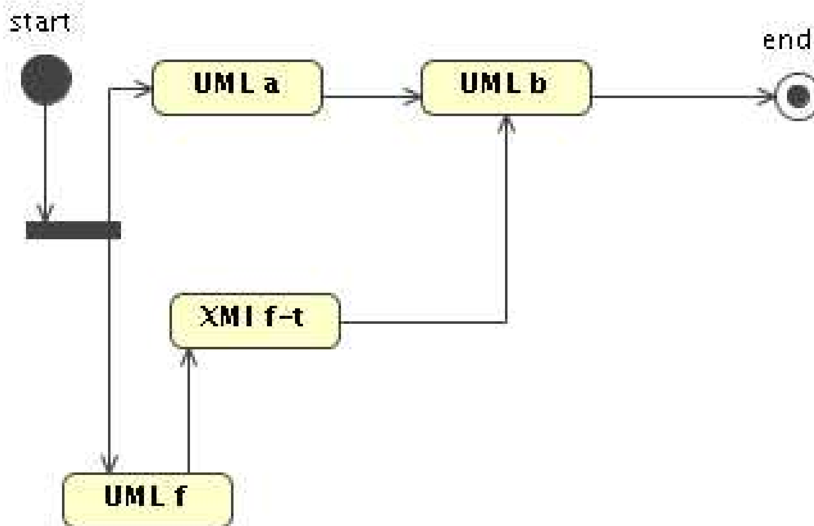

Figure 4.8: The Ideal World

The complete transformation process uses the UML version of the functions, represented by UML model f. Figure 4.7 shows the completed process. It has two inputs, the UML source model a and the UML function model f, and one output, the converted UML model b. The generator needs both input models to perform the transformation.

In this project UML functions are denoted using UML profiles. These profiles consist of UML stereotypes and UML tags. A function is modeled as a UML package having a stereotype <<function>>. These functions describe how to transform one UML model into another UML model. This mechanism should be usable in existing UML tools such as ArcStyler (`http://www.interactive-objects.com/`).

The node $XMI\ f-t$ in figure 4.8 is the XMI version of the profile that would be used by a tool which understands UML templates.

4.2 UML Function Model

This section introduces a formal model on which the UML functions are based. The model is largely based on $\lambda\omega$. A specification of this calculus can be found at chapter 5 of [2].

This model was chosen for the following reasons:

- it is a natural formalization of the model which grew out of the modified UML template model as defined by the OMG;
- it supports the parameter model as defined in chapter 4.3;
- it supports proxies as defined in chapter 4.3.3.

Note that the second reason is the result of the bad support of current UML tools for template parameters as defined by the OMG: MagicDraw 10.5 and higher versions have reasonable support, MagicDraw 9.0 does not support template parameters for packages, and ArgoUML 0.20 completely lacks support for template parameters. Although MagicDraw 10.5 and later support template parameters, they does not support XMI 1.2 which is required by requirements (c3) (listed in chapter 5.2). The bad tool support was the main reason to develop the function model as defined in chapter 4.3.

UML models can be considered as a set of nested namespaces. The namespaces are represented by packages and usually contain other elements to make up the model. The contents of each namespace can be represented as a list, because each set can be represented as a list. The notation introduced in chapter 6

of [6] to represent lists is adopted: the empty list is represented by *nil* and elements are prepended to a list using the *cons* constructor.

Variables which represent lists are typed as *list*. Other types which are used are *int*, *string*, *bool*, and *UML_X*. The *X* in *UML_X* is a placeholder for names of UML elements. All model elements from the UML domain are prefixed with *UML_* to distinguish them from other elements.

The constant definitions are:

- $TRUE : bool = \lambda x.\lambda y.x$
- $FALSE : bool = \lambda x.\lambda y.y$
- $nil : list = \lambda x.TRUE$
- $0 : int = \lambda x.(TRUE)$
- $n : int = \lambda x.x^n(TRUE)$

eq is a Boolean function which returns whether two terms are equal. It cannot be defined in λ-calculus. This is a result of Gödels first incompleteness theorem. It is detailed in section "Undecidability of equivalence" of [22]. The consequence of this is that the intuitive meaning of *eq* has to suffice.

Variables can be untyped. In the expression ($\lambda l : list.e.$expression with l and e) the variable l is of type list while variable e is untyped. This is because in the declarations l is followed by : *list* while e is not followed by any type.

The other functions are defined as:

- $cons : list = \lambda h.\lambda t : list.\lambda l : list.l\ h\ t$ (preprends h to l)
- $ITE = \lambda c : bool.\lambda t.\lambda e.c\ t\ e$ (if-then-else)
- $head = \lambda l : list.l\ TRUE$
- $tail = \lambda l : list.l\ FALSE$
- $Y = \lambda h.(\lambda x.h(x\,x))(\lambda x.h(x\,x))$ (the fixpoint operator)
- $SUCC : int = \lambda n : int.cons\ 0\ (cons\ n\ nil)$
- $AND : bool = \lambda x : bool.\lambda y : bool.x\ y\ FALSE$
- $OR : bool = \lambda x : bool.\lambda y : bool.(x\ TRUE)\ y$
- $NOT : bool = \lambda x : bool.x\ FALSE\ TRUE$
- $IMPLY : bool = \lambda x : bool.\lambda y : bool.((x\ FALSE\ TRUE)\ TRUE)\ y$
- $\exists : bool = NOT \forall b : bool.NOT\ b$
- $isEmpty : bool = \lambda l : list.l(\lambda h.\lambda t.FALSE)$

In this model strings are sugared versions of lists. They are defined in table 4.1.

Table 4.1: Sugared Versions of Lists

String	List representation
$''$	nil
$'e'$	$cons\ e\ nil$
$'e' + +S$	$cons\ e\ (cons\ S\ nil)$

In this model lambda expressions will be sugared as defined in table 4.2. The functions defining some of these expressions are defined after the table. Furthermore all expressions will be written in infix notation

instead of prefix notation.

Table 4.2: Sugared Versions of Lambda Expressions

Lambda notation	Sugared notation
$Y\lambda_f : t.expr(_f)$	$f : t.expr(f)$
$AND, OR, NOT, IMPLY$	$\wedge, \vee, \neg, \Rightarrow$
$f\ a1 \ldots\ an$	$f(a1, \ldots, an)$
eq	$=$
in	$\in$
$NOT\ eq$	$\neq$
$NOT\ in$	$\notin$
$AND\ (eq\ a[i]\ 'c1') \ldots (eq\ a[i+j]\ 'cj')$	$a[i \ldots i+j] =' c1 \ldots cj'$
$ITE\ i\ t\ e$	if i then t else e
$OP\ a : t.OP\ b : t \ldots OP\ n : t$	$OP\ a, b, \ldots n : t$
$cons\ list\ nil$	$list$
$cons\ list\ (cons\ element\ nil)$	$list + +element$
$listcons\ list1\ list2$	$list1 + +list2$

The remaining functions are defined as:

- $listcons : list = \lambda x, y : list.$if $isEmpty(x)$ then y else $head(x) + +tail(x) + +y$
- $in : bool = \lambda e.\lambda l : list.$if $l = nil$ then $FALSE$ else (if $e = head(l)$ then $TRUE$ else $e \in tail(l))$
- $listin : bool = \lambda x, y : list.\exists i, j : list.y = i + +x + +j$
- $len : int = \lambda l : list.$if $isEmpty(l)$ then 0 else $SUCC(len(tail(l)))$
- $delete : list = \lambda l : list.\lambda e.$if $head(l) = e$ then $tail(l)$ else $head(l) + +delete(tail(l), e)$
- $listdelete : list = \lambda x, y : list.$if $isEmpty(y)$ then x else $listdelete(delete(x, head(y)), tail(y))$ (subtracts list y from list x)
- $nonlast : list = \lambda l : list.$if $isEmpty(l)$ then nil else(if $isEmpty(head(l))$ then nil else $head(l) + +nonlast(tail(l)))$ (returns every element of l except the last one)

UML elements have properties. Property p of element e is denoted as $e.p$. The UML element properties are:

- $name : string$ returns the name of the element
- $type : string$ returns the type of the element (for attributes and operations)
- $stereotypes : [string]$ returns the stereotypes of the element
- $attributes : []$ returns the attributes of the element
- $operations : []$ returns the operations of the element
- $children : [UML_Element]$ returns a list containing the content of the package
- $kind : [UML_Element]$ returns the kind (class, message, association, ...) of the element as a singleton list
- $tag(tagname : string) : string$ returns the contents of a specified tag of the element
- $parameters : [string]$ returns the set of parameters of the element

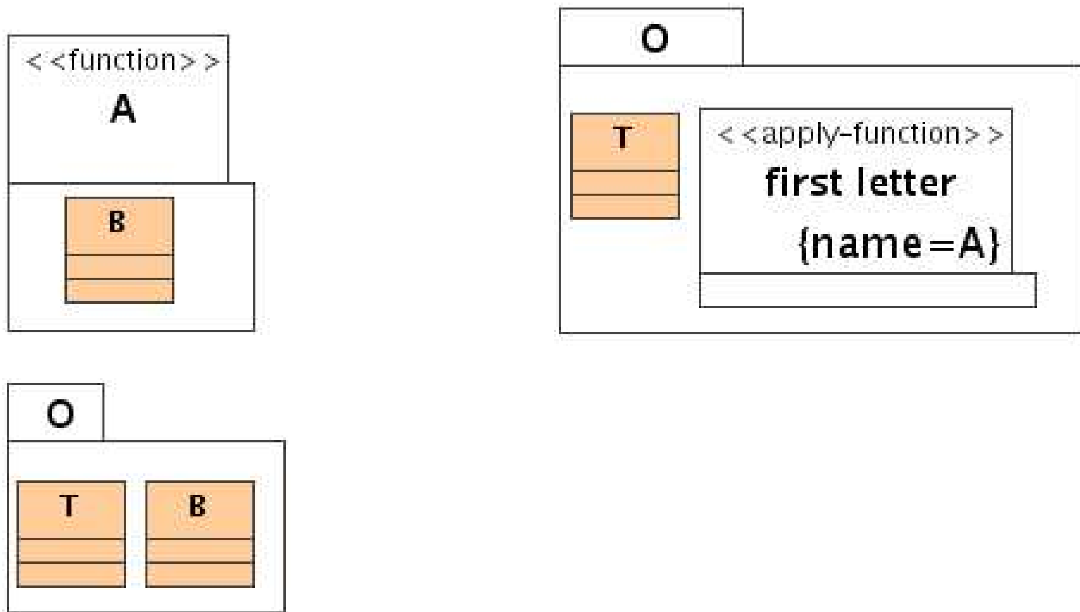

Figure 4.9: Transforming Package *O* with Function *A*

Function A, which adds a package B to the current namespace, can be defined as: $\lambda N : list.c : UML_Package$ $[c.name =' B'] ++N$, where c is another "reserved" variable. In this example, c is a "reserved" variable: it is essentially a free variable but it should not be used outside function definitions. Other "reserved" variables can be used too. These variables belong to the namespace specified in the UML functions. Constraints on variables are put between square brackets.

Functions can also be applied in a nested way. For example, function A can be applied to itself using $A(A(N))$ which normalizes to $B++(B++N)$.

4.3 UML Functions

This section will explain how UML models can be transformed using UML functions. This section also lists the rules which the UML function transformations must obey.

UML functions are normal mathematical functions which operate in the domain of UML models instead of the domain of numbers. They are written in UML and are used to describe transformations. UML functions are free of side effects: they only alter the model as specified. They can be present along with ordinary "static" UML models, which are not altered by the tool. This way it is possible to change an existing UML model.

Note that the resulting elements of a function application always appear in the same layer as where the function was applied. It is not possible nor desirable to change this location. The function application itself is removed from the resulting model.

UML functions are modeled as packages. They have a stereotype <<function>> to distinguish them from ordinary packages. An example is given for package O which applies function A. In figure 4.9 package O contains a class T and applies function A to add another class B. Function applications are indicated by a package which has stereotype <<apply-function>>. Function applications can be unnamed. The name is only used to be able to apply the same function multiple times in the same namespace. The name of the referenced function itself is given in the tag *name* belonging to the stereotype <<apply-function>>.

The rules to which the functions must comply form the precondition for applying these functions. The rules are self-derived. They are derived by starting with an empty ruleset and testing which conditions are needed to eliminate unwanted results when applying functions: some rules are "obvious", other are more

delicate and were found by creatively testing transformations on hypothetical models and analyzing their results. The ruleset should be complete. During this project some rules were added while others were deleted. Each rule is given in English and in a formal way. The formal descriptions are derived from the English descriptions and are given in the sugared $\lambda\omega$ notation described above.

4.3.1 General UML Functions

The general rules for UML functions are given first. The rules use functions in their definitions. These functions serve both as a shorthand and as an abstraction.

The following abstract functions are used in the rules:

- An element which connects to elements: $\rightarrow: UML_Element = \lambda p, q : UML_Element.p$ points to q
  The connecting element is the result of this function. The result of the function can also be used as a boolean value. In that case the function indicates that p must be connected to q but the connecting element is irrelevant.

- $multiplicity : string = \lambda r, e : UML_Element.$ multiplicity of the relation r regarding element e.

The following concrete functions are used in the rules:

- A directed relation between two elements:
  $dirRelation : UML_Element = \lambda p, q : UML_Element.\exists r : UML_Element.$if $r = \rightarrow (p, q)$ then r else nil

- A client of a relation is the depending part of a relation:
  $isClient : bool = \lambda c, s, r : UML_Element.(r = dirRelation(s, c)) \wedge (r.kind = ['UML_Dependency'])$

- A leaf package is a package which does not contain packages of its own:
  $isLeaf : bool = \lambda p : UML_Package.\forall e : UML_Element.(e \in p) \wedge (e.kind \neq ['UML_Package'])$

- A function is a package with has a stereotype <<function>>:
  $isFunction : bool = \lambda p : UML_Package.('function' \in p.stereotypes)$

- A supplier of a relation is the master part of a relation:
  $isSupplier : bool = \lambda s, c, r : UML_Element.isClient(c, s, r)$

Furthermore, $apply$ is used as a shorthand to denote the application of $apply - function$:
$apply \equiv \lambda f : UML_Package.apply - function(f[isFunction(f)])$
The following rules must hold for any package that has a stereotype <<apply-function>>. In these rules the referenced function is called q.

1. The referenced package, which is referenced by the tag *name* of the referrer, has to exist:
   $\forall p : UML_Package.\exists q : UML_Package.(q.name = p.tag('name'))$
   This rule is needed to avoid applying a non-existent function.

2. The stereotype of q is <<function>> and nothing else:
   $\forall q : UML_Package.('function' \in q.stereotypes) \Rightarrow (q.stereotypes = ['function'])$
   If other stereotypes would be allowed, then they are probably part of the model and not of the function application. In that case it would not be clear how –if at all –to apply the function and what to do with the model stereotypes.

3. All elements in the same leaf package and the elements which are referenced by the same leaf package must be unique regarding name and kind:
   $\forall e : UML_Element.\exists p : UML_Package.(isLeaf(p) \wedge ((e \in p) \vee (p \rightarrow e))) \Rightarrow (\neg\exists e' : UML_Element.(e.name = e'.name) \wedge (e.kind = e'.kind) \wedge ((e' \in p) \vee (p \rightarrow e')))$
   This rule is actually a UML rule which states that two elements in the same leaf package which have the same name have to be of a different kind (class, collaboration, etcetera) and that two elements in the same leaf package which are of the same kind must have a different name. Elements which are

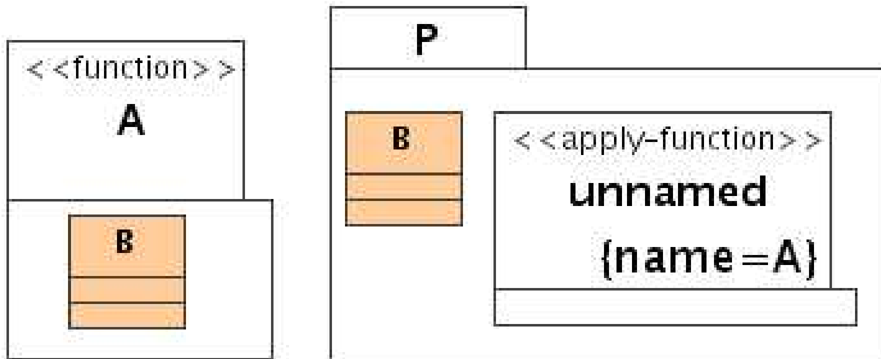

Figure 4.10: A Package which Violates Rule 3

referenced by a leaf package are elements which reside inside a function, which could eventually be placed inside the referencing package. Package *p* in figure 4.10 violates rule 3 because it has its own class *B* and because it references a class *B* via function *A*, which it applies. Therefore, the function cannot be instantiated here.

4. <<apply-function>> behaves like a function, so it is an exclusive stereotype:
$\forall p : UML_Package.('apply - function' \in p.stereotypes) \Rightarrow (p.stereotype = ['apply - function'])$

4.3.2 Parametrized Elements

This section explains function parameters and lists the rules to be obeyed when applying function with parameters.

The function model uses UML dependencies inside <<function>> packages to specify how to substitute actual parameter elements with their resulting elements. UML functions can also be used to describe additions to the input model. Thus UML functions model the relation between the parameter part and the resulting part of a <<function>> package. The dependency element was chosen because this element can be related to more UML elements than other connectors such as associations or generalizations.

Inside the function packages are formal parameters elements, resulting elements, and dependencies connecting the formal parameter elements to the resulting elements. The formal parameters of UML functions are indicated by elements whose name is surrounded by angle brackets and which do not have an outgoing dependency arrow. Formal parameters are never part of the result.

The substitution of formal parameter elements by their corresponding resulting elements can be seen as β-reduction. This is shown in figure 4.11. In this example, x is the formal parameter, $2x$ is the formal resulting element, 10 is the actual parameter, and 20 is the actual resulting element. The transformations which UML functions can perform consist of two basic kinds:

- A transformation can add parts to the model by including elements in its definition which are not connected to any of its formal parameters. Figure 4.9 gives an example of a function which adds a class *B* to the model.

- A transformation can change the model by including elements in its definition which are connected to at least one of its formal parameters. The elements which are connected to the formal parameters are added to the output model. Figure 4.12 gives an example in which class *C* is renamed to $res1$. Class $res1$ is empty, regardless of the features (attributes and operations) of the input class. If the features must be preserved, then this must be indicated explicitly by letting the resulting features depend on the input features. This is done in the function definition. Attributes and operations can be preserved independently; which can be useful if only one of these sets must be preserved. Both features of classes and interfaces can be preserved this way. It is even possible to convert a class

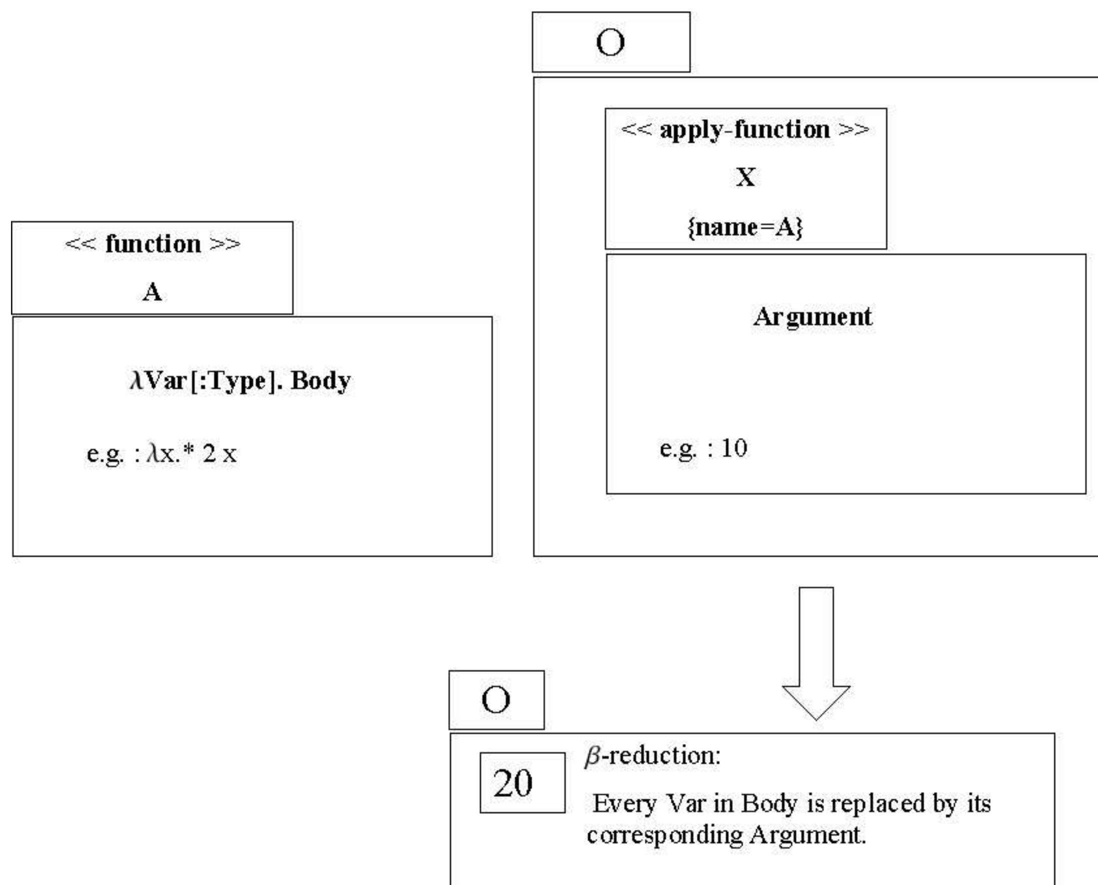

Figure 4.11: Transformations Use β-reduction

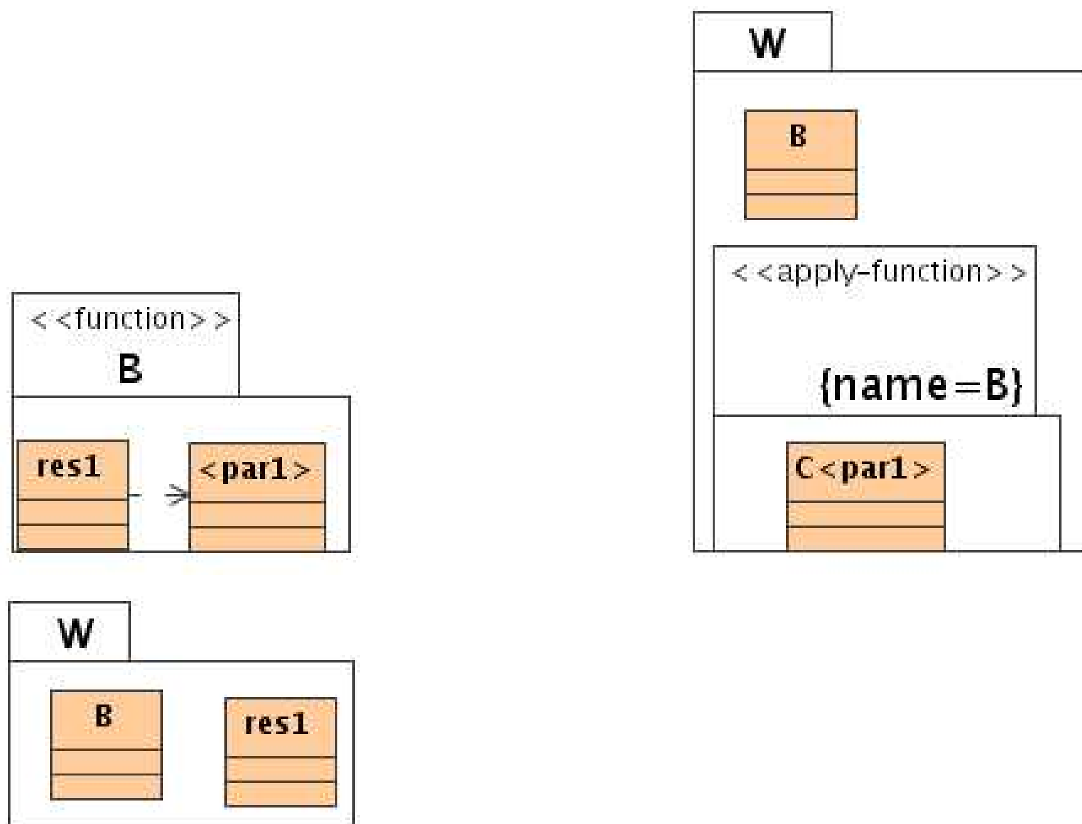

Figure 4.12: Class *C* Gets Renamed to *res*1 by Function *B*

to an interface (or vice versa) and still preserve its features. Note that the unity of transformation consists of two identical model parts connected by a dependency.

If function B must generate multiple resulting elements from formal parameter $par1$, then all these resulting elements must be connected to $par1$ using one dependency per resulting element. On the other hand, if multiple parameter elements must somehow be merged into one resulting element, then that resulting element must be connected to all these parameter elements. The above two situations can also be combined to generate m resulting elements from n parameter elements.

Functions are not limited to one conversion of n parameter elements to m resulting elements. Multiple of these conversions and additions (see previous item) can be defined in one function.

The format of actual input names is $actual < formal >$ (the angle brackets are literal). The format shows that formal parameter names are quoted with a start and an end quote. The start quote is always $<$, the end quote can be one of $>$, $?>$, $*>$, or $+>$. The different end quotes are explained below:

- $<$ and $>$
  In this case the actual parameter is compulsory, exactly one actual parameter must be given.
- $<$ and $?>$
  In this case the actual parameter is optional. If it is given, the actual resulting element as specified in the function is generated. If it is omitted, no actual resulting elements is generated.
- $<$ and $*>$
  In this case the actual parameter is also optional. The difference with the above case is that multiple actual parameters can be instantiated if the actual part of their actual names is different. In that case the number of resulting elements is the number of actual parameters multiplied by the number of resulting elements for that actual parameter.
- $<$ and $+>$
  This case is the same as the previous case. The only difference is that the actual parameter must be instantiated at least once.

The formal part (the part between the angle brackets) of the actual parameter name must include the same ?, *, or + if the formal parameter name contains a ?, *, or +.

The formal part of the actual input name is needed to deduce which formal parameter belongs to which actual parameter when multiple formal parameters of the same kind exist. When multiple actual parameters are to be matched with multiple formal parameters, and all these parameters are of the same kind, then no possibility exists for the tool to tell the actual parameters apart if the formal part of the actual parameter name was omitted.

Function $B3$ (figure 4.14) shows that a formal parameter name can be referenced by a formal resulting element by including the name of the formal parameter in the name of the formal resulting element. In this case the name of the formal parameter must be quoted with angle brackets. This quoting results in the name of the actual parameter being copied into the name of the actual resulting element. The parts of the formal resulting element which are not quoted with angle brackets are copied verbatim into the name of the actual resulting element. Furthermore, it is possible to specify that only a part of the actual parameter name must be copied into the name of the actual resulting element by including a start and end position between square brackets. A negative position means that the position should be counted from the right. For formal parameters $par1$ and $par2$ with actual parameters a and b respectively this leads to the overview in table 4.3. In this table A represents the name of the actual parameter, F represents the name of the formal resulting element, and R represents the name of the actual resulting element. A dot represents string concatenation.

Table 4.3: Parameter Names

A	F	R
a	<par1>	a
a	res1	res1
a	P<par1>	P.a
a	<par1>S	a.S
a, a	<par1><par1>	$a.a$
b	<par2>	b
a, b	<par1><par2>	$a.b$
parameter	<par1>[1..-5]	param

Formally, the relation between the name of the actual resulting element ar and the name of the actual parameter ap is as follows:
$ar = ap((< p_i > / < a_i >)[b_i..e_i]) \wedge$
$\forall_i : \mathbb{N}.(0 \leq i \leq n) \wedge (1 \leq b_i \leq l_i) \wedge (b_i \leq e_i \leq l_i \vee -l_i \leq e_i \leq -1)$
The variables are defined as follows:

Table 4.4: Parameter Variable Definitions

Name	Type	Description
n	integer	Number of formal parameters of the function
p_i	string	Name of formal parameter i
a_i	string	Name of actual parameter belonging to p_i
b_i	integer	Starting position within p_i
e_i	integer	Ending position within p_i
l_i	integer	Length of p_i

b_i and e_i can be negative meaning that the values have to be added to $n_i + 1$, as they then count from the end. The notation A/B denotes substitution: all occurrences of A have to be substituted simultaneously by B.

New functions are introduced:

- Parameter of f:
  $isParameter : bool = \lambda p : UML_Package.(p.name[1] = '<') \wedge (p.name[-1] = '>') \wedge (\neg \exists q, r : UML_Element.isClient(p, q, r))$

- $actualParameters$ returns the actual parameters of function f:
  $actualParameters : [UML_Element] = \lambda f : UML_Package.(apply(f).parameters)$

- $formalParameters$ returns the formal parameters of function f:
  $formalParameters : [UML_Element] = \lambda f : UML_Package.[c : UML_Element[(c \in f.children) \wedge isParameter(c)]]$

- Range of f:
  $range : [UML : Element] = \lambda f : UML_Package.\forall e, p : UML_Element.isParameter(p) \wedge (e \in f) \wedge (p \in f) \wedge \exists r : UML_Element.isClient(e, p, r)$

- Relationship between two elements:
  $relation : UML_Element = \lambda e, f : UML_Element.\exists r : UML : Element.$if $(r = dirRelation(e, f) \vee r = dirRelation(f, e))$ then r else nil

As in the previous section all rules are given in English and in a formal way. The rules are:

1. All formal parameters inside the function can be connected to a depending element, which is the resulting element. If a formal parameter is not connected to any other element, it is left out of the

result:
$\forall f: UML_Package.\forall fp: UML_Element.(fp \in formalParameters(f)) \wedge (\exists e: UML_Element.((fp \rightarrow e) \wedge (e \in range(apply(f)))) \vee \neg\exists e: UML_Element.(fp \rightarrow e))$

2. All formal parameters must be instantiated by placing the appropriate elements within the function package:

   (a) Each actual parameter kind must match its formal parameter kind:
   $\forall t,s: UML_Package.\forall fp: UML_Element.(fp \in formalParameters(t)) \wedge \exists ap: UML_Element.(ap \in actualParameters(s)) \wedge (fp.kind = ap.kind)$
   This rule states the common requirement that the argument type of a function must match. A function of a strongly typed programming language will not take an integer if it expects a string argument, similarly, UML functions will not take a use case actor if they expect a class.

   (b) If a formal parameter has stereotypes, then its actual parameter must include all these stereotypes:
   $\forall t,s: UML_Package.\exists fp: UML_Element.((fp \in formalParameters(t)) \wedge \neg isEmpty(fp.stereotypes)) \Rightarrow (\exists ap: UML_Element.(ap \in actualParameters(s)) \wedge (\forall x: string.(x \in fp.stereotypes) \Rightarrow (x \in ap.stereotypes)))$
   This rule is similar to the previous one, except that it now lives in the stereotype domain and that the stereotype set of the actual parameter only has to be a superset of the stereotype set of its formal counterpart. The extra stereotypes are ignored for the resulting element.

   (c) If the stereotype set of a formal parameter and the stereotype set of its resulting element are incomparable, then the stereotype set of the actual parameter must be a superset of the stereotype set of the formal parameter to which it belongs:
   $\forall t,s: UML_Package.\forall r: UML_Element.((r \in range(apply(f))) \wedge \exists fp: UML_Element.(fp \in formalParameters(t)) \wedge \neg(fp.stereotypes = r.stereotypes)) \Rightarrow \exists ap: UML_Element.(ap \in actualParameters(s)) \wedge \forall x: UML_Element.(x \in fp.stereotypes) \Rightarrow (x \in ap.stereotypes)$
   An example of this rule is given in figure 4.13. The class *actual* is a correct actual parameter for the function *demo* because its stereotype set is a superset of the stereotype set of the function. The class $actual - wrong$ is an incorrect actual parameter because its stereotype set {<<collection>>,<<compspec>>} is incomparable with the stereotype set {<<collection>>,<<core>>} of the function.

3. Each package with stereotype <<apply-function>> can only contain the input for one transformation at a time. This is necessary to be able to differentiate between the different parameters of different transformations. This means that the number of actual parameters has to match the number of formal parameters:
   $\forall f: UML_Package.isFunction(f) \Rightarrow (len(actualParameters(f)) = len(formalParameters(f)))$
   An example of when this rule is needed is given in figure 4.14. In this figure class C and D are transformed into class *resC* and object *DRes* respectively using one invocation of function $B3$. The same function is used to transform class E and F into class *resE* and object *FRes* respectively. If all input classes were put in the same package, then parameters C and E could not be distinguished from each other, as they both have $par1$ as their formal counterpart. The same holds for parameters D and F with respect to their formal counterpart $par2$.

4. All actual parameters names of a function must have the format $actual < formal >$:
   $\forall t,s: UML_Package.\forall n: string.(n \in formalParameters(t)) \Rightarrow \exists m: string.(m \in actualParameters(s)) \wedge listin('<' + + n.name + +' >', m.name)$
   An example of this rule is given in figure 4.14. Without this rule the function can not distinguish between the actual counterpart of $par1$ and $par2$, which leaves *resC* and *DRes* but also *resD* and *CRes* as valid results.

5. If a child named $< n* >$ for any n of a client depends on a child named $< n* >$ of the corresponding supplier, then all children of the supplier are copied to the client:
   $\forall f: UML_Package.\forall c,s,cc,sc: UML_Element.((c \in f) \wedge (s \in f) \wedge (sc \in s.children) \wedge (cc \in$

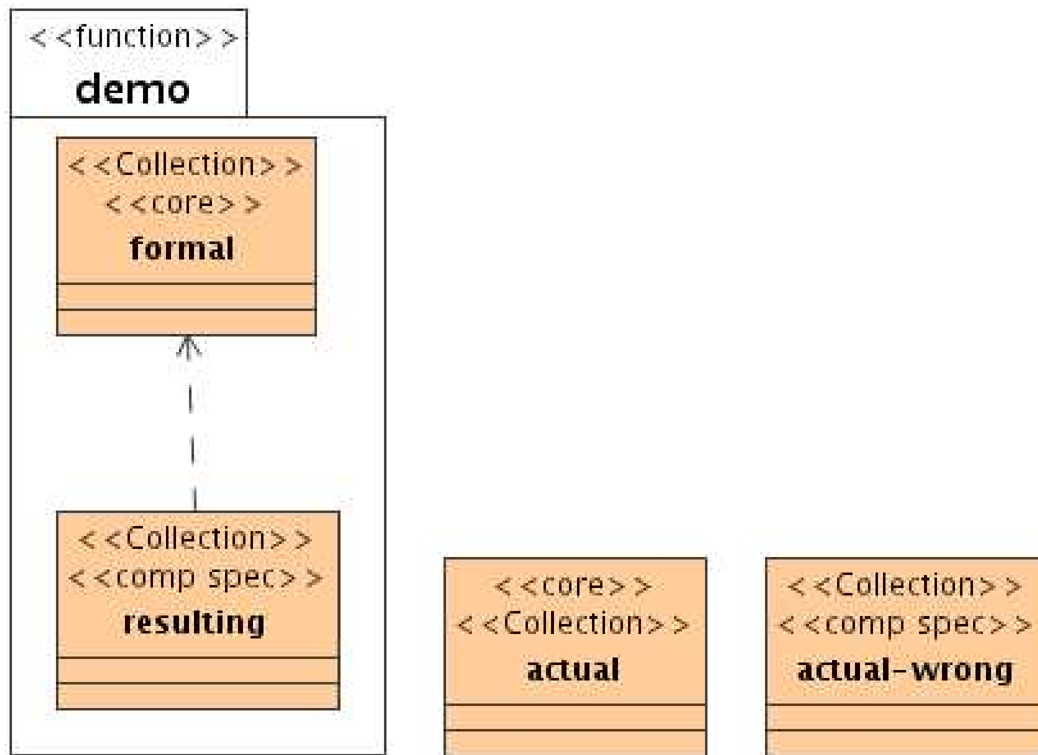

Figure 4.13: A Function with a Correct Actual Parameter and a Wrong Actual Parameter

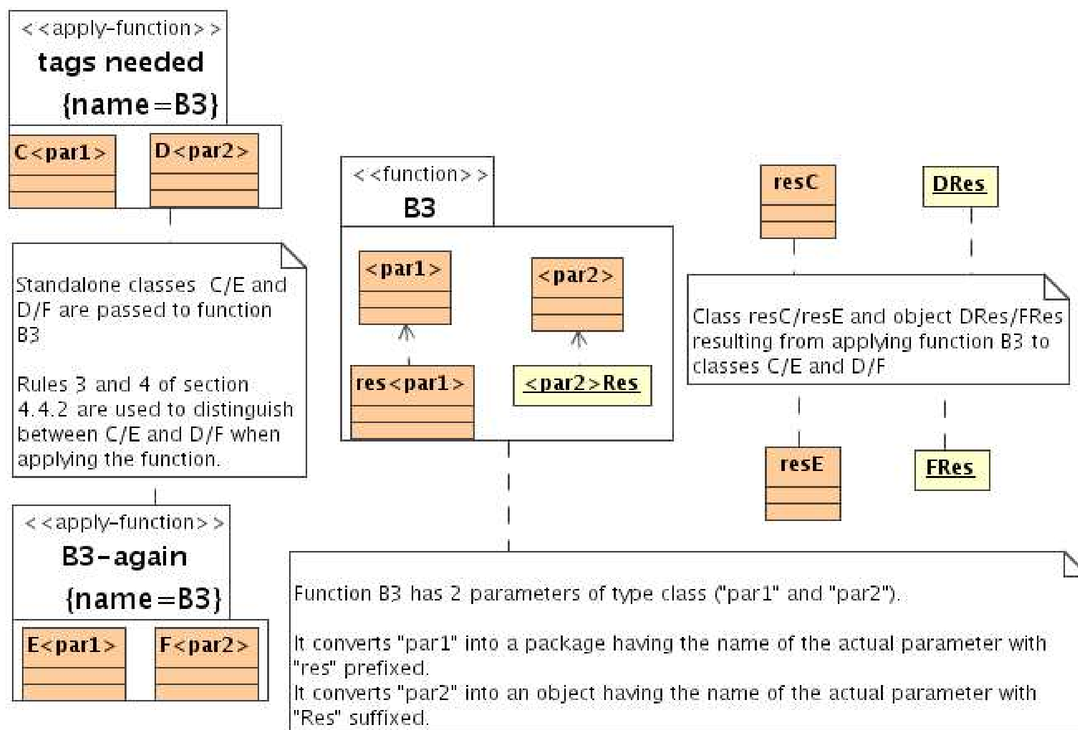

Figure 4.14: Different Transformations Require Separate Invocations

$c.children) \wedge (sc.name[1] ='<') \wedge (sc.name[-2..-1] =' * >') \wedge (cc.name[1] ='<') \wedge (cc.name[-2..-1] =' * >') \wedge \exists r : UML_Element.r = relation(sc,cc)) \Rightarrow (\forall e : UML_Element.e \in range(apply(f)) \wedge \exists ap : UML_Element.(ap \in actualParameters(f) \wedge \forall x : UML_Element.(x \in e.children) \Rightarrow (x \in ap.children)))$

6. Copying children not copied by $< n* >$ must be indicated explicitly:
   $\forall f : UML_Package.\forall c,s,cc,sc : UML_Element.((c \in f) \wedge (s \in f) \wedge (sc \in s.children) \wedge (cc \in c.children) \wedge \exists r : UML_Element.(isClient(cc,sc,r) \wedge isSupplier(sc,cc,r))) \Rightarrow (\exists e : UML_Element.(e \in sc) \wedge (e \in range(apply(f))))$

7. The stereotype set of the actual resulting element is that of the actual parameter minus that of the formal parameter plus that of the formal resulting element:
   $\forall f : UML_Package.\forall c,s : UML_Element.((c \in f) \wedge (s \in f) \wedge \exists r : UML_Element.(isClient(c,s,r) \wedge isSupplier(s,c,r))) \Rightarrow (\forall v,w : UML_Element.(v \in actualParameters(f)) \wedge (w \in range(apply(f))) \wedge (w.stereotypes = listdelete(v.stereotypes, s.stereotypes) + +c.stereotypes))$

8. Combinations like $<< n >>$ are not allowed:
   $\forall e : UML_Element.('<' \notin tail(e.name)) \wedge ('>' \notin nonlast(e.name))$

9. For formal parameters which name contains a "?", "*", or "+", the name of the connected resulting element can contain at most one such formal name:
   $\forall f : UML_Package.\forall e : UML_Element.((e \in formalParameters(f)) \wedge (('?' \in e.name) \vee ('*' \in e.name) \vee ('+' \in e.name))) \Rightarrow \forall a : UML_Package.\forall r : UML_Element.((r \in actualParameters(a)) \wedge \exists x : UML_Element.isSupplier(e,r,x) \wedge (\exists p : string.name(r) = p + +' <' + +e.name + +' >' \wedge (p[1..-2] \neq '? >') \wedge (p[1..-2] \neq ' * >') \wedge (p[1..-2] \neq ' + >')))$

4.3.3 Combining UML Diagrams

For some transformations the need arises to combine different types of UML diagrams. An example of this is a function which generates methods in a class diagram from messages in a sequence diagram. It is not possible to draw the sequence chart and the resulting class in one diagram and relate the methods to the messages using dependencies as done in e.g. figure 4.12 for function *B*.

One solution to this is to define proxy elements in the diagram which contains the resulting elements. These proxy elements are related to the resulting elements using dependencies. Proxy elements are defined as normal elements with a stereotype <<proxy>> and a tag *ref*. The tag contains the name of the referenced element. The stereotype can be applied to any element which fits in the resulting diagram. This gives the best opportunity to keep the intended semantics. The elements referenced by proxies have a stereotype <<proxied>>. The most likely diagram types which will be using proxies are class, state, and activity diagrams.

An example of this solution is shown in figure 4.15. In this figure the message name is $< op1* >$. The example also shows the method to transform sequence messages into operations contained in a class or an interface:

1. Define the visibility of the resulting operation;

2. Obey rules 1 to 3 defined below.

A new function is introduced:

- $proxy : UML_Element = \lambda e : UML_Element.$ "dereference of" e

The rules for proxied elements are:

1. The name of the formal resulting operation must be quoted with $<$ and $* >$, and the parameter set (a generic parameter) and the type of this resulting operation must be quoted with $<$ and $? >$. The supplier of the operation must be an object with stereotype <<proxy>> which references a sequence message:

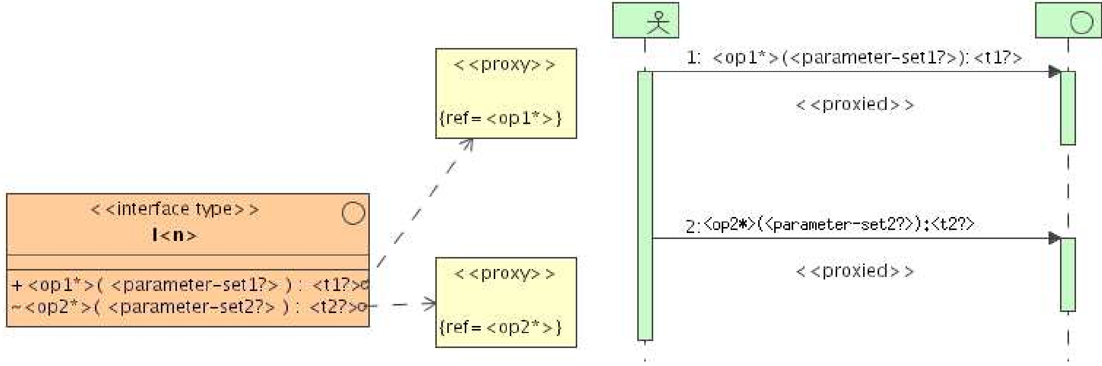

Figure 4.15: Combining Different Diagram Types

$\forall f : UML_Package.\forall e : UML_Element.\forall o.((e \in formalParameters(f)) \wedge (o \in e.operations) \wedge (o.stereotypes = ['proxy']) \wedge \exists p, r : UML_Element.(proxy(p).kind = ['UML_Message'] \wedge isSupplier(p, o, r))) \Rightarrow ((o.name[1] = '<') \wedge (o.name[-2..-1] = '*>') \wedge (o.type[1][1] = '<') \wedge (o.type[1][-2..-1] = '?>') \wedge (len(o.parameters) = 1) \wedge (o.parameters[1] = '<') \wedge (o.parameters[-2..-1] = '?>'))$

2. The name of the referenced sequence message must be $< o* > (< ps? >) :< t >$, where o is the operation name, ps the name of the parameter set, and t the name of the resulting type. The referenced message must be a 'call' message:
$\forall f : UML_Package.\forall e : UML_Element.\forall o.(e \in formalParameters(f)) \wedge (o \in e.operations) \wedge (\exists p, r, m : UML_Element.isSupplier(p, o, r) \wedge (proxy(p) = m) \wedge (m.kind = ['UML_Message']) \wedge (m.type = 'call') \wedge (m.name = o.name ++ o.parameters ++ o.type))$

3. If an operation returns a type t, then t must be the name of an existing DataType:
$\forall f : UML_Package.\forall e : UML_Element.\forall o.((e \in formalParameters(f)) \wedge (o \in e.operations) \wedge (o.type \neq nil)) \Rightarrow \exists t : UML_DataType.(t.name = o.type)$

4. For each element with stereotype <<proxy>>, exactly one element with stereotype <<proxied>> must exist, and vice versa:
   - $\forall e : UML_Element.('proxy' \in e.stereotypes) \Rightarrow \exists e' : UML_Element.(e' = proxied(e)) \wedge ('proxied' \in e'.stereotypes)$
   - $\forall e : UML_Element.('proxied' \in e.stereotypes) \Rightarrow \exists e' : UML_Element.(e = proxied(e')) \wedge ('proxy' \in e'.stereotypes)$

5. The name of a dereferenced element must match the name in the tag ref of the referencing proxy element:
$\forall e, e' : UML_Element.(('proxy' \in e.stereotypes) \wedge ('proxied' \in e'.stereotypes) \wedge (e' = proxied(e))) \Rightarrow (e.tag('ref') = e'.name)$

Note that the stereotypes <<proxy>> and <<proxied>> are never part of the stereotype set of an actual resulting element.

4.3.4 AOSD Extensions

Some new definitions and rules are needed to support AOSD.

First, a new function is introduced:

- The abstract function $random : string$ returns a unique random string.

The rules specific to AOSD are:

1. The character ˆ denotes a random string:
   $\forall e : UML_Element.\forall o.\exists x : string.(o \in e.operations) \wedge ((o.name = '\hat{}' ++x) \Rightarrow (o.name = random++x))$
   The random string is used as a unique prefix for the woven versions of the original AOSD classes. In [9] this prefix is constructed from the namespace of the AOSD class. That solution requires a more difficult function than the first solution. Therefore the first solution is chosen.

2. The formal stereotype <<behavioral>> results in one of the actual stereotypes <<after>>, <<afterReturning>>, <<around>>, or <<before>> when used with operations in a function application:
   $\forall fp : UML_Package.\forall fe : UML_Element.\forall fo.((fe \in formalParameters(fp)) \wedge (fo \in fe.operations) \wedge ('behavioral' \in fo.stereotypes)) \Rightarrow \forall ap : UML_Package.\forall ae : UML_Element.\forall ao.((ae \in actualParameters(ap)) \wedge (ao \in ae.operations) \wedge (('after' \in ao.stereotypes) \vee ('around' \in ao.stereotypes) \vee ('afterReturning' \in ao.stereotypes) \vee ('before' \in ao.stereotypes)))$
   This rule allows the function definitions to use unspecified behavioral stereotypes. They only need to be specified when the function is applied.

4.3.5 Nested Function Application

When working on chapter 5.3 the need arised to apply one function inside another function, i.e. to nest function applications. To support this, the stereotype <<apply-function>> is extended with the tag *var*. A new rule regarding this tag is introduced:

1. The tag 'var' contains all the actual variable names which result from the nested function application. The names are separated by spaces and each name must consist of an actual and a formal part. This is denoted as $actual < formal >$:
   $\forall p,q : UML_Package.(('apply-function' \in p.stereotypes) \wedge ('apply-function' \in q.stereotypes) \wedge (p \in q.children)) \Rightarrow (\forall e : UML_Element.(e \in range(apply(p))) \Rightarrow \exists s : string.(s \in tokenize(q.tag('var'),'\backslash s+')) \wedge \exists k : UML_Element.(k \in q) \wedge (s = e.name ++ '<' ++ k.name ++ '>'))$

The function $tokenize(s,r)$ splits a string s into a list of strings by the given regular expression r.

An example of nested function application is given below. Two functions are defined: $add-a$ and $add-b$. Figure 4.16 shows functions $add-a$ and $add-b$. Function $add-a$ is a function which converts a package $< n >$ into a package $a < n >$. Function $add-b$ is a function which converts a package $< n >$ into a class $b < n >$. Function $add-a$ can be applied to itself. This is shown in figure 4.17. In this figure the first function application ($l3$) converts package p to package ap. The second function application ($l2$) converts package ap to package aap. The last function application ($l1$) converts package aap to class $baap$.

4.4 Answered Research Questions

This chapter gave an answer to five research questions. Section 4.1 answered research question 3 by giving a scheme which states how UML models can be transformed. Section 4.2 answered two related research questions. It answered research question 4 by giving a semantic meta-model for the model transformations. It answered research question 5 by explaining why this meta-model had to be chosen. Section 4.3.3 gave an answer to research question 6 by explaining the UML proxy profile. Section 4.3.5 gave an answer to research question 11 by explaining a method to nest function applications.

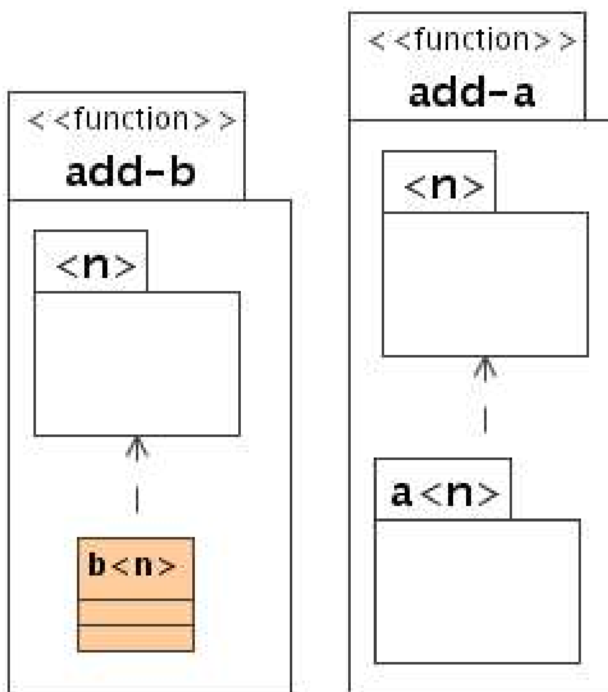

Figure 4.16: Functions $add-a$ and $add-b$

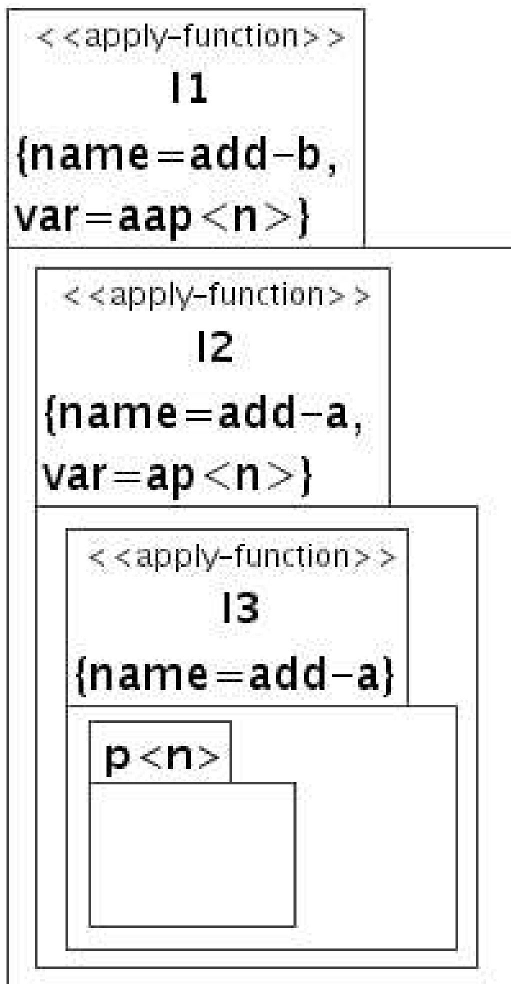

Figure 4.17: Applying Function $add - a$ in a Nested Way

Chapter 5

A Proof of Concept

"If you can get 90 percent of the desired effect for 10 percent of the work, use the simpler solution."
—FreeBSD Developer's Handbook, chapter 1.3

This chapter describes an approach to transform models which consists of two major parts. The first part consists of the UML functions describing the transformation functions. The transformation functions are expressed in UML themselves. The second part consists of the script generator which translates the UML functions of the first part into functionally equivalent XSLT scripts. An example case is described in chapter 5.3 which uses both parts. This case serves as an example of how to use the functions and the generator in practice.

5.1 Transformation Functions

The transformation functions which are described and modeled here are derived from the cases shown in chapter 2. They describe the transformations for supporting both the UML Component Method and for AOSD.

5.1.1 Functions for the UML Component Method

The functions for the UML Component Method are handled first. Each function is accompanied by a textual description.

Business Interfaces

For each data type with stereotype <<core>> in the business type model a business interface can be derived. Two similar functions are defined for this purpose.

The first function, *btm_to_bizint*, is the simpler one. This function is shown in figure 5.1. It expands a <<core>> class from the business type model into an interface. Its operational semantics are:

- Create a new UML interface $I < m > Mgt$, where $< m >$ is the name of the given <<core>> class.
- Set the stereotype of this interface to <<interface type>>.

A more extended version of the above function, *btm_to_bi*, is given in figure 5.2. This function expands a <<core>> class from the business type model into a business interface package which manages the <<infotype>> instances of the <<core>> class. Its operational semantics are:

- Create a new package $I < m > Mgt$, where $< m >$ is the name of a <<core>> class in the business type model.
- Add a new interface $I < m > Mgt$ to the package, where m is again the name of a <<core>> class in the business type model.

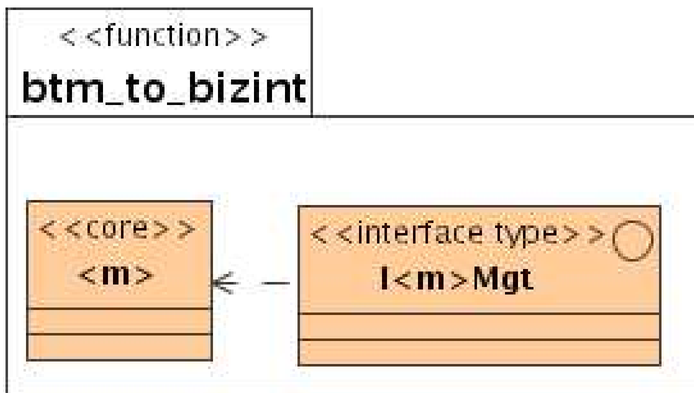

Figure 5.1: Function for the Business Interface

- Set its stereotype to <<interface type>>.
- Add a new class $< m >$ with the stereotype <<infotype>> to the package. This class, which represents a data type, is directly copied from the business type model. This means that it also inherits all attributes and operations from its counterpart from the business type model.
- Add all the attributes and operations from all <<type>> classes which have a compositional relationship to the <<core>> class to the <<infotype>> class.
- The data type and the interface are related to each other using a 1:* composition, where each interface can have many data types.

System Interfaces

For each sequence diagram in the use case model a system interface can be derived. This interface provides an API specification for the use case. Again two similar functions fulfill this goal.

The first function for this transformation is given in figures 5.3 and 5.4. It operational semantics are:

- Create a new UML interface $I < a >< n >$, where $< a >$ is the name of the actor and $< n >$ is the name of the use case.
- Set its stereotype to <<interface type>>.
- Add all messages of type 'call' which appear in the proxied sequence diagram as operations to this interface:
  - the name, specified before <op*>.
  - the optional parameter set (names and types), specified between parentheses.
  - the optional return type, specified after the final colon in the message name, outside any parentheses.

An extended version of this function is given in figures 5.5 and 5.6. In the extended function the generated interface can also contain methods which have visibility <<package>>. Its operational semantics are:

- Create a new package $I < n >$, where $< n >$ is the name of the use case.
- Create a new interface $I < n >$ inside this package, where $< n >$ is the name of the use case.
- Set the stereotype of this interface to <<interface type>>.

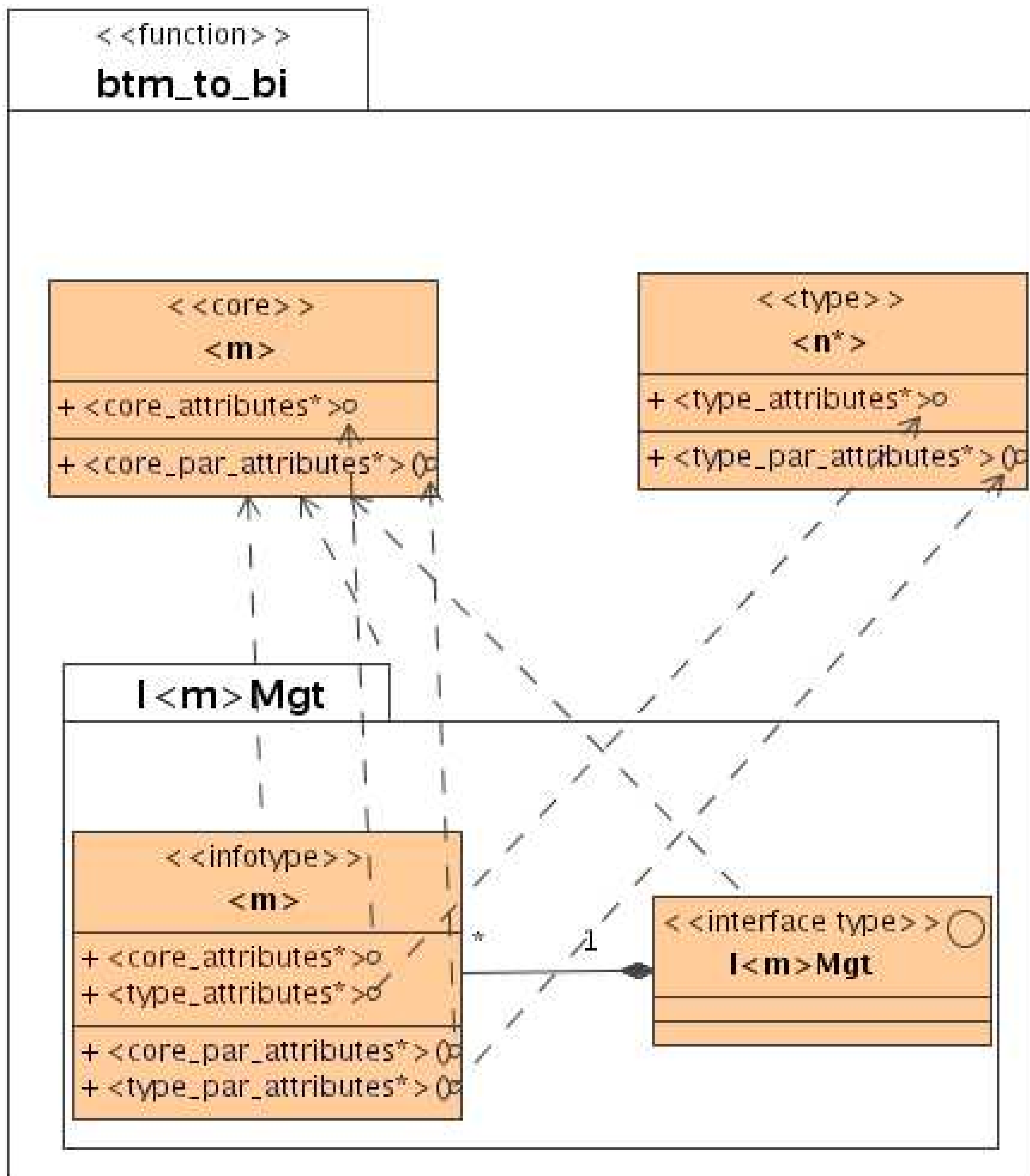

Figure 5.2: Extended Function for the Business Interface

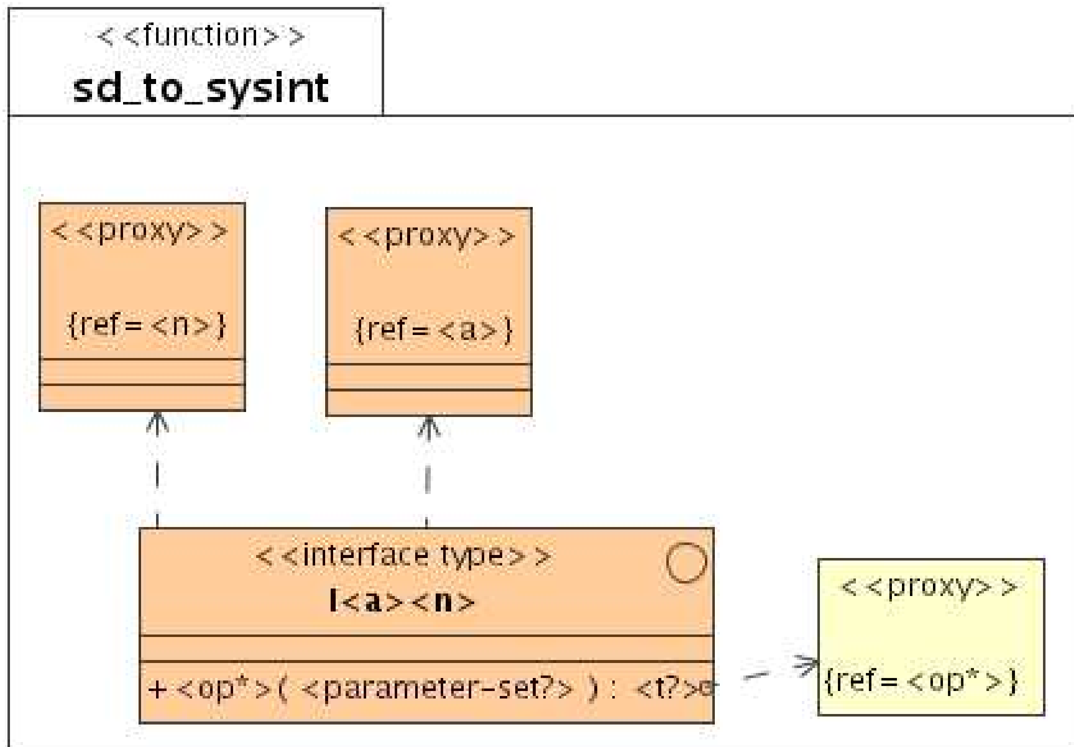

Figure 5.3: Function for the System Interfaces

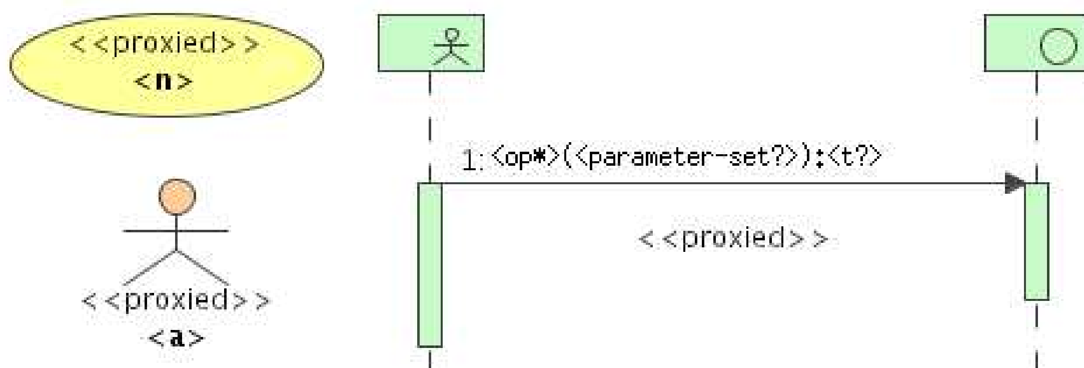

Figure 5.4: Proxied Elements of the System Interfaces Function

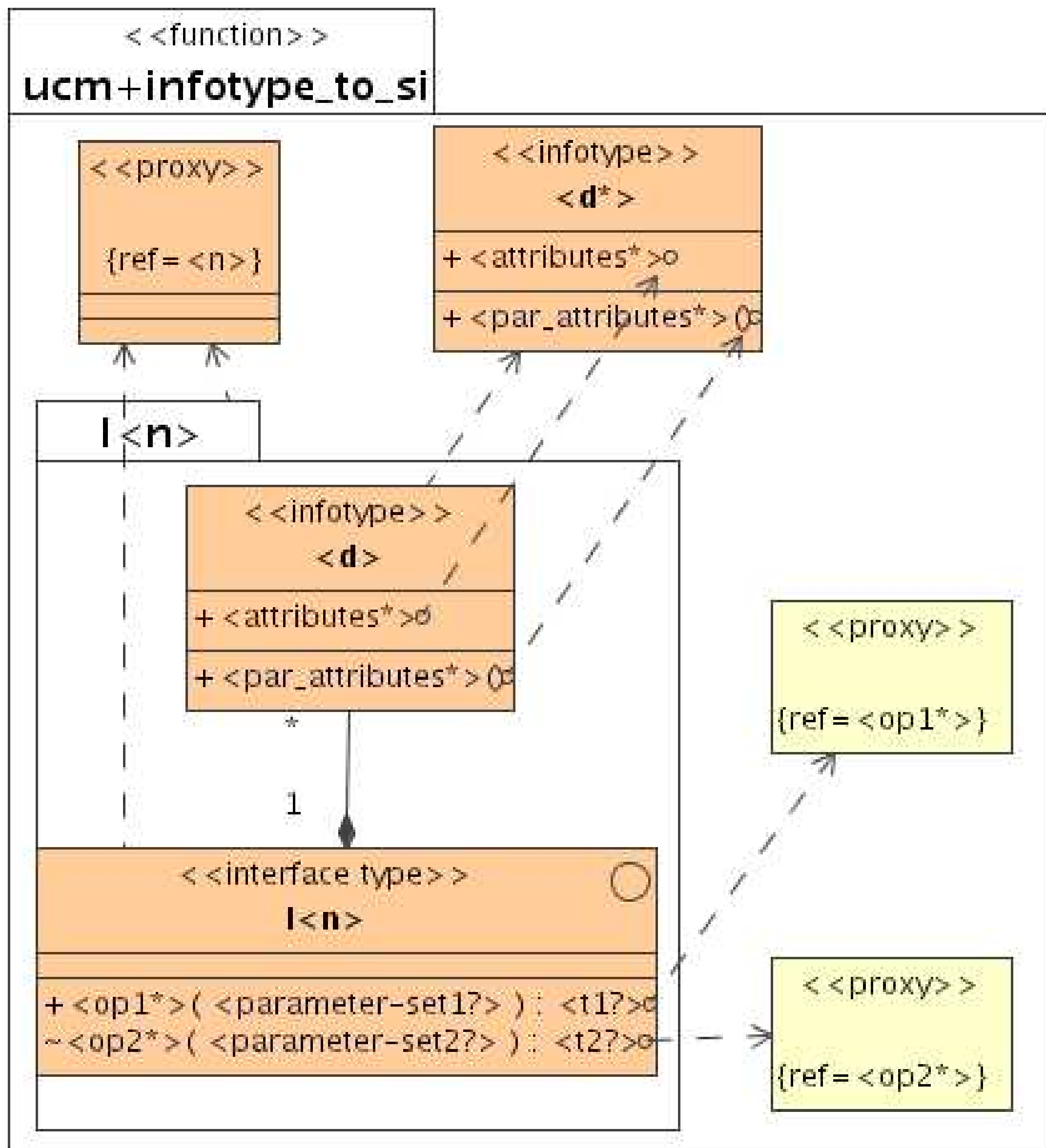

Figure 5.5: Extended Function for the System Interfaces

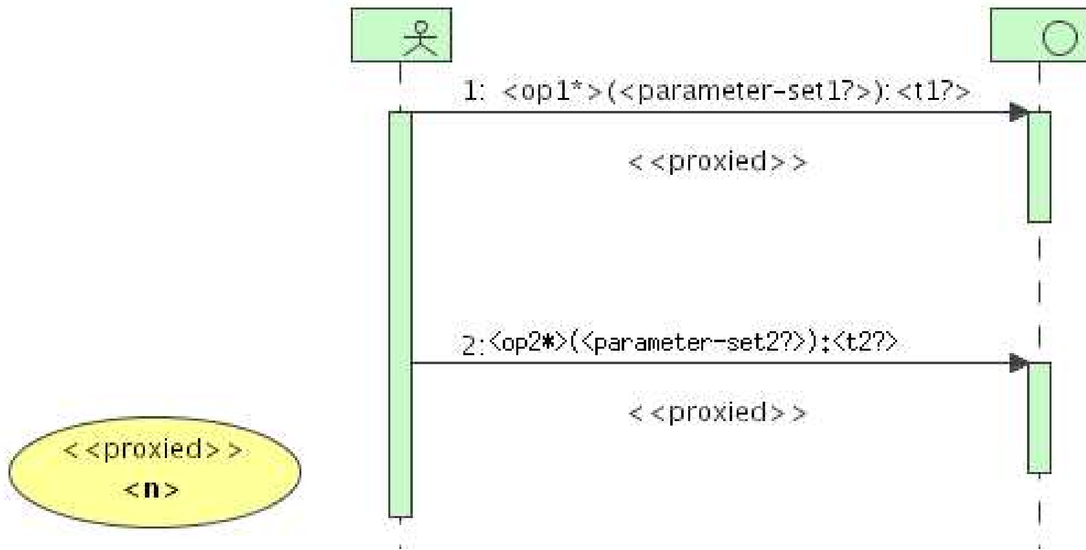

Figure 5.6: Proxied Elements of the Extended System Interfaces Function

- Add an operation to the interface for each 'call' message in the sequence diagram belonging to the use case:
  - The specified parameters (names and types) are included in the method signature. They are given in the message name between parentheses.
  - The return type of the operation is given in the message name after the last colon which is not between parentheses.
  - The visibility of the operation is either public when bound to <op1*> or package when bound to <op2*>.
- Copy all parameters which are <<infotype>> classes into the resulting package.
- Connect all copied classes to the resulting interface using a 1:* composition from the interface to the data types.

Component and System Architecture

Each use case model or business type model can be transformed into a component specification. This section discusses some functions for these transformations.

One way to construct the component and system architecture is to use the function *make_csa*. It joins the various architectural layers into one model. Its operational semantics are:

- Copy all business interfaces given by the parameter $< bi* >$.
- Create a new component specification class for each business interface. These classes have the name of the core class of which the business interfaces were originally created suffixed with 'Mgr'. Set the stereotype of these component specification classes to <<comp spec>>.
- Pairwise connect each business interface to each <<comp spec>> class using an association.
- Copy all <<core>> classes given by the parameter $< btm_core* >$.
- Pairwise connect each business interface to each <<core>> class using a 1:* composition.
- Create a new component specification class for each classifier role in the system interfaces diagram. Set the stereotype of these component specification classes to <<comp spec>>.

- Pairwise connect each business interface to each classifier <<comp spec>> class using a <<use>> dependency.

- Copy all system interfaces given by the parameter $< si* >$.

- Pairwise connect each system interface to each classifier <<comp spec>> class using an association.

Note that all attributes and operations of the interfaces and <<core>> classes are also copied. The function for this transformation is given in figure 5.7 and 5.8.

Another way to specify the component architecture is given by the functions *ucm_to_ca*, *ucm_to_cs*, and *btm_to_cs*. Figure 5.9 and 5.10 show the function *ucm_to_ca*. The operational semantics of this function are:

- Create a package $< n >$, where $< n >$ is the name of the given use case.

- Create a class $< n >$ *Form* inside package $< n >$, where $< n >$ is the name of the given use case.

- Analogous, create a class $< n >$ *Controller* inside package $< n >$.

- Set the stereotype of the class $< n >$ *Form* to <<boundary>>.

- Set the stereotype of the class $< n >$ *Controller* to <<control>>.

- Connect the created classes by means of an association.

Figure 5.11 and 5.12 show the function *ucm_to_cs*. This function creates a component specification from a <<system boundary>> package $< s >$. Its operational semantics are:

- Create a new class $< s >$, where $< s >$ is the name of the above <<system boundary>> package.

- Set the stereotype of this class to <<comp spec>>.

Figure 5.13 shows the function *btm_to_cs*. This function transforms a <<core>> class $< m >$ into a component specification. Its operational semantics are:

- Create a new class $< m >$ *Mgr*, where $< m >$ is the name of the given <<core>> class.

- Set the stereotype of this class to <<comp spec>>.

The final function defined for the UML Component Method is the helper function <<extract-infotype>>. Figure 5.14 shows this function. It extracts an <<infotype>> class from a package. Its operational semantics are:

- Create a new class $< m >$, where $< m >$ is the name of the given <<infotype>> class. The given class resides within a package which can be anonymous.

- Set the stereotype of this class to <<infotype>>.

- Copy all attributes and operations from the given class to the resulting class.

5.1.2 Functions for AOSD

The functions for AOSD are described below. Each function is again accompanied by a textual description.

<<function>>
make_csa

<<interface type>>
<bi*>
+ <attributes*>
+ <operations*>

<<comp spec>>
<bi>[2..-4]Mgr

<<proxy>>
{ref=<sds*>}

<bi>
+ <attributes*>
+ <operations*>

<<core>>
<btm_core*>
+ <attributes*>
+ <par_attributes*>

<<use>>

*

<<comp spec>>
<sds>

<<interface type>>
<si*>
+ <attributes*>
+ <operations*>

<btm_core>
+ <attributes*>
+ <par_attributes*>

<si>
+ <attributes*>
+ <operations*>

Figure 5.7: Function for the Component and System Architecture

Figure 5.8: Proxied Elements of the Component and System Architecture Function

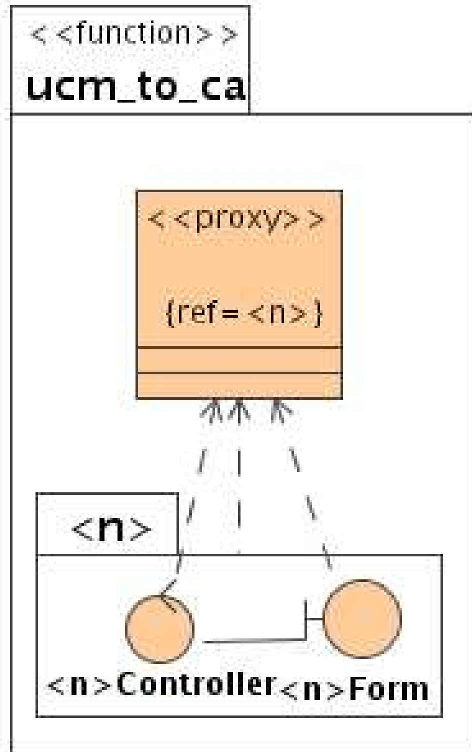

Figure 5.9: Function *ucm_to_ca* for the Alternative Component and System Architecture

Figure 5.10: Proxied Elements for Function *ucm_to_ca*

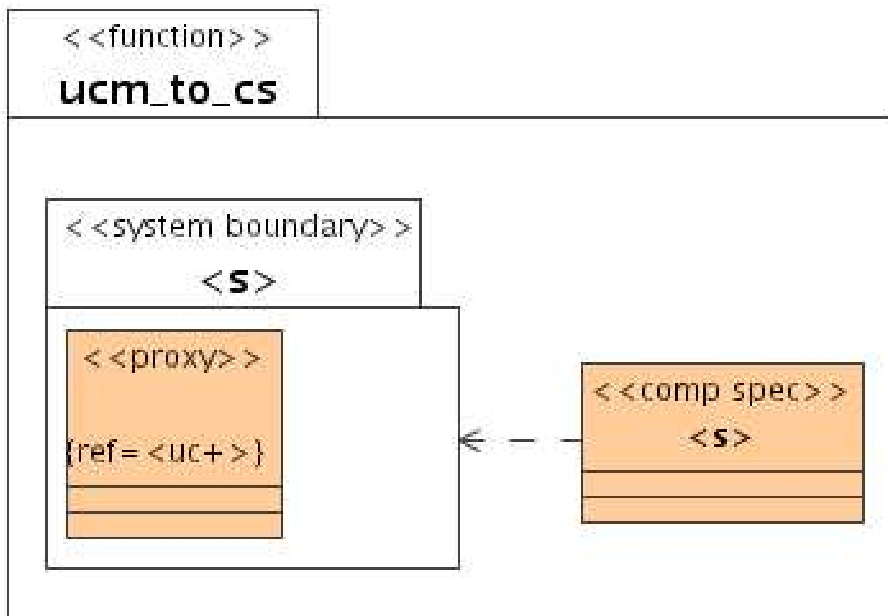

Figure 5.11: Function *ucm_to_cs* for the Alternative Component and System Architecture

Figure 5.12: Proxied Elements for Function *ucm_to_cs*

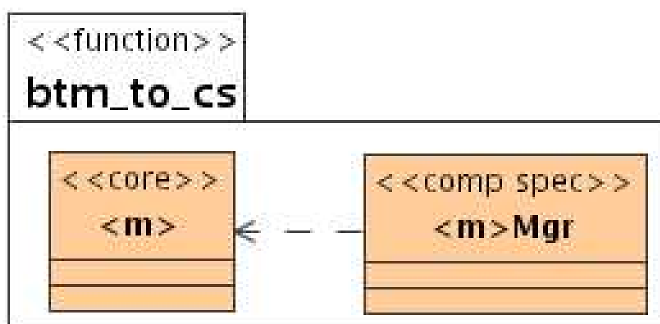

Figure 5.13: Function *btm_to_cs* for the Alternative Component and System Architecture

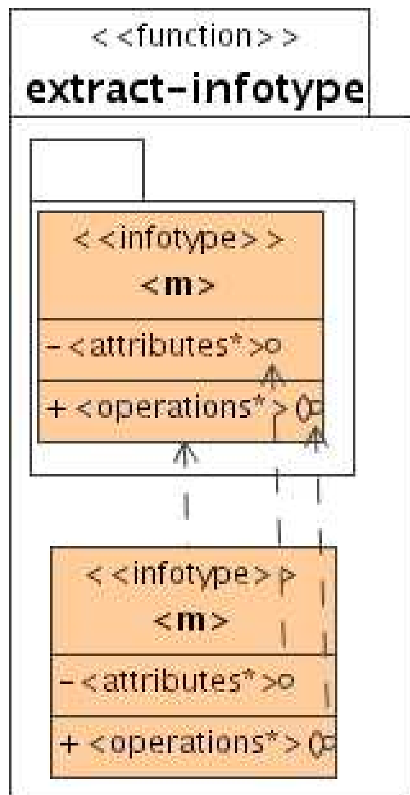

Figure 5.14: Function $extract - infotype$

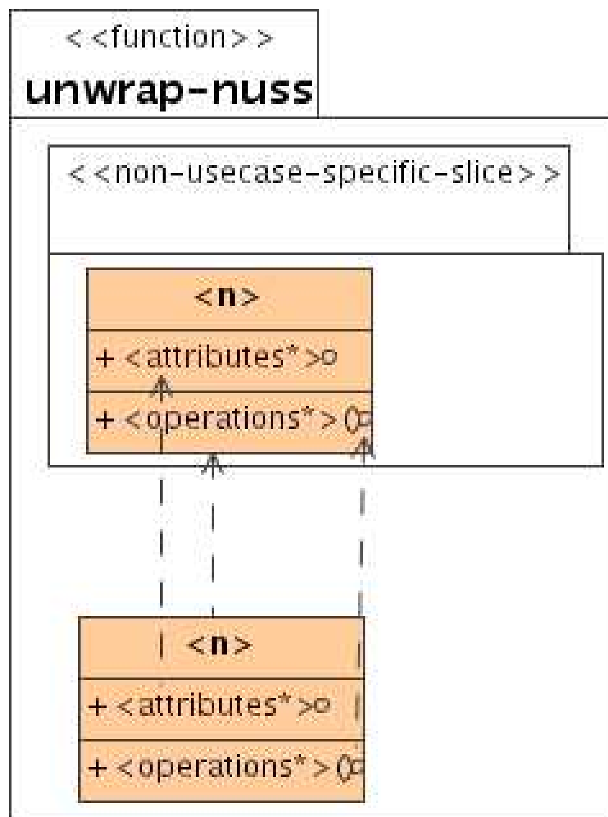

Figure 5.15: Function to Extract a Class from a Non Use Case Specific Slice

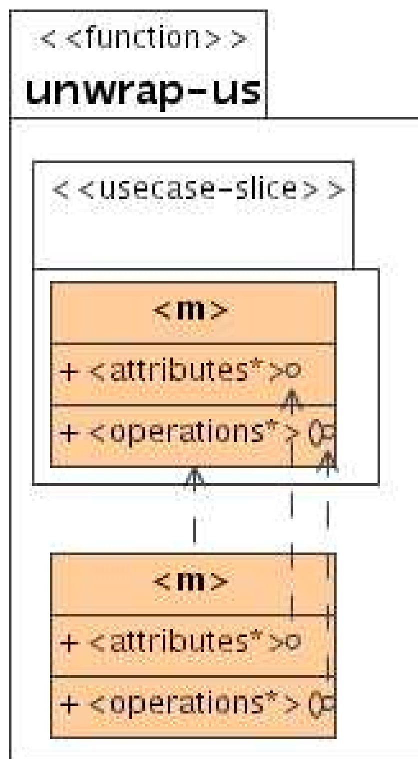

Figure 5.16: Function to Extract a Class from a Use Case Specific Slice

Class Extraction

The first two functions, $unwrap - nuss$ and $unwrap - us$, extract a class from its package. This package has stereotype <<non-usecase-specific-slice>> respectively <<usecase-slice>>. These are stereotypes specific to AOSD to indicate that the package is a slice (see also chapter 2.2.1). The functions are shown in figure 5.15 and 5.16.

The operational semantics of these functions are:

- Create an empty class $< n >$. where $< n >$ is the name of the actual class parameter. This class must be contained in a package which has either stereotype <<non-usecase-specific-slice>> or stereotype <<usecase-slice>>.
- Copy the attributes from the actual class parameter into $< n >$.
- Copy the operations from the actual class parameter into $< n >$.

Pointcuts and Extension Classes

Two more functions exist. They deal with pointcuts and extension classes. The function given in figure 5.17, $substitutePointcuts$, substitutes pointcut definitions into extension classes. The operational semantics of $substitutePointcuts$ are:

- Create a new class $< n >$, where $< n >$ is the name of the actual extension class parameter.
- Stereotype the new class as <<extensionClass>>.
- Copy the attributes from the actual class parameter into $< n >$.
- Copy the operations from the actual class parameter into $< n >$.
- Create an operation *extname*, where *extname* is specified in the tag *name* of the actual behavioral stereotype belonging to the pointcut operation of parameter $< n >$. The name of this operation is prefixed with a random string.
- Set the visibility of *extname* to private.
- Set the stereotype of *extname* to the behavioral stereotype of the actual class parameter $< n >$.
- Copy the tag *when* from the actual behavioral stereotype.
- Copy the tag $struct$ from the tag defined in the actual pointcut definition $< m >$.

Step 5 through 9 are repeated for each pointcut operation.

The function $mergeContents$, given in 5.18, merges the contents of the extension class $< m >$ into a normal class $< n >$ and renames it to $r < n >$. The extension class is contained in an aspect. The operational semantics of $mergeContents$ are:

- Create an empty class r$< n >$ for each normal actual class parameter $< n >$. At least one actual class $< n >$ must exist.
- Copy the attributes from each class $< n >$ into class r$< n >$.
- Copy the operations from each class $< n >$ into class r$< n >$.
- Copy the attributes from each class $< m >$ into class r$< n >$.
- Copy the operations from each class $< m >$ into class r$< n >$.

Here, $< m >$ is a set of actual class parameters. At least one actual class $< m >$ must exist. All classes $< m >$ must be contained in a package with stereotype <<aspect>>.

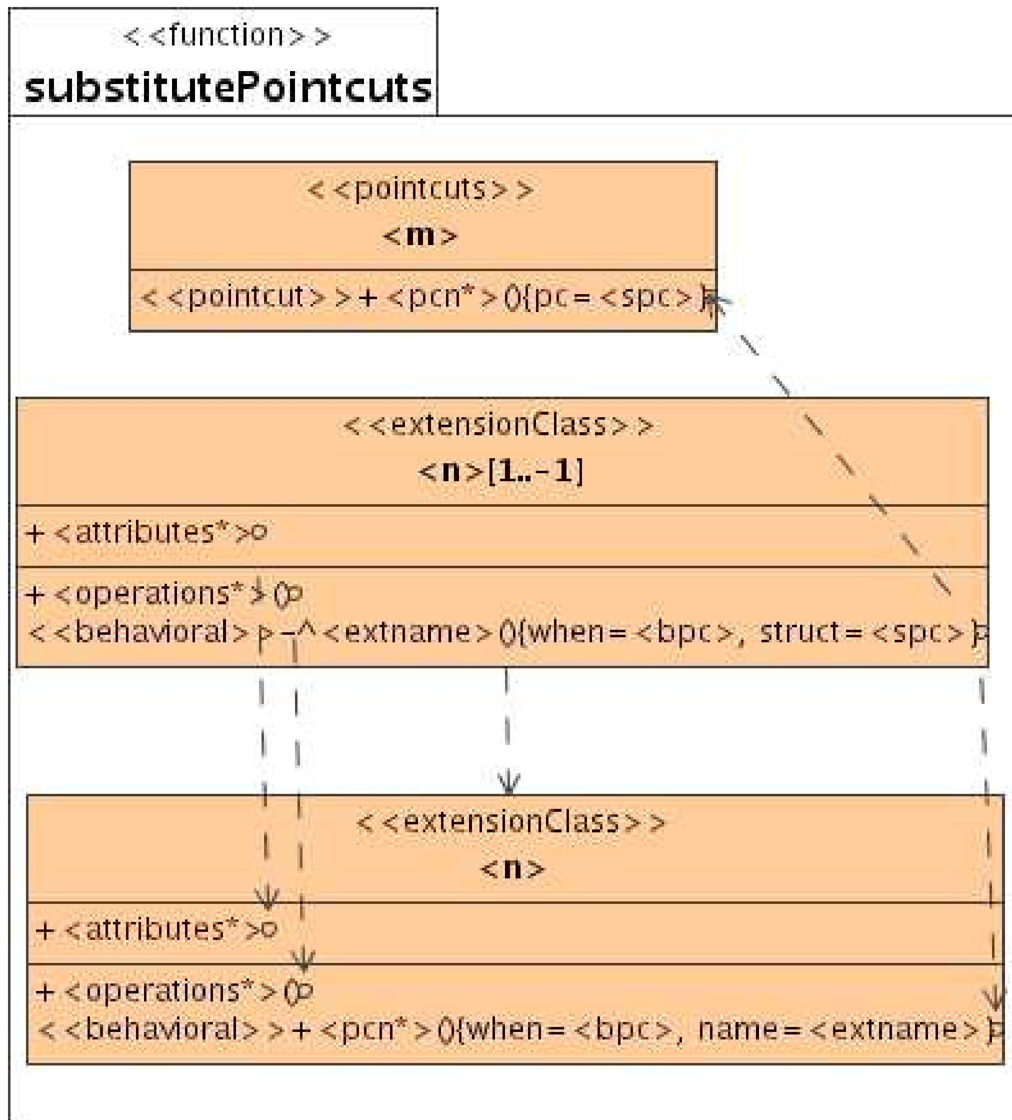

Figure 5.17: Function to Substitute Pointcut Definitions

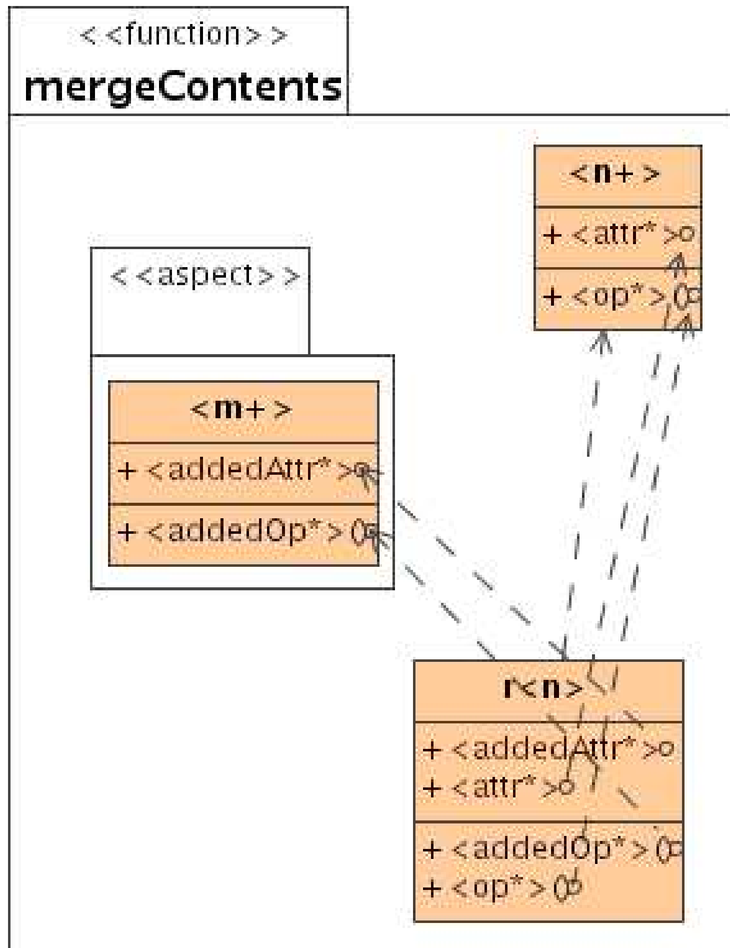

Figure 5.18: Function to Merge an Extension Class into a Normal Class

5.2 Design of TPUPT

This section describes the design of TPUPT. TPUPT stands for *Two Phase UML Phunction Transformer*. It will also state some requirements for this system.

The requirements for the conversion system are:

- (c1) The system is two pass.
- (c1.1) Pass 1 transforms the UML function definitions into a script.
- (c1.2) Pass 2 applies the UML functions.
- (c1.3) Pass 2 is generated by pass 1.
- (c1.4) Both passes are defined in the same programming language.
- (c2) The rules of chapter 4 are obeyed by the scripts.
- (c3) The input and output of the scripts is valid XMI 1.2
- (c4) ArcStyler 5 can be used to define the UML functions.

The two phase model which the tool uses has the advantage that the functions only need to be defined once. Once they are defined, the functions can *theoretically* be applied to any model. However, due to the way UML models are represented, the functions can only be applied to the model in which they are defined.

These tool itself is implemented in XSLT and XPath. An introduction to XSLT and XPath is given in appendix D. The tool itself can be freely downloaded from [17].

The tool consists of three files. The design is given per file, as each file represents a logical part of the tool:

- tpupt.xsl is the program that implements pass 1;
- base.xsl is a module that contains all the parts of the generated XSLT templates which remain constant;
- libxmi.xsl is a library for both phases.

The XSLT templates defined in each file are given as classes. If the name of the class is quoted, then this class represents a named template which is invoked with an xsl:call-template instruction. Otherwise the name indicates the XMI nodes for which an xsl:apply-templates instruction invokes this template. Numbers must be ignored, they are only present to keep the UML editor happy.

The name and type of XSLT template parameters are denoted using UML template parameters. UML constraints represent additional XSLT conditions such as an XPath test or an XSLT mode. The XPath functions defined in each file are given in an anonymous interface.

The associations between the templates represent the flow between these templates. This flow is normally bidirectional, but an end containing an arrow forces the direction of the flow. To further clarify the template invocations all associations which represent an invocation via a call-template instruction are stereotyped as <<call>>.

The goal of each function and template is described in a per-file table. In this table:

- the first column specifies the name or the match expression of the item. The | character specifies alternatives;
- a rule-template is an XSLT template which is invoked using the xsl:apply-templates mechanism, while a call-template is an XSLT template which is invoked using the xsl:call-template instruction.

The flow diagram describing the flow between the XSLT templates of tpupt.xsl is given in figure 5.19.

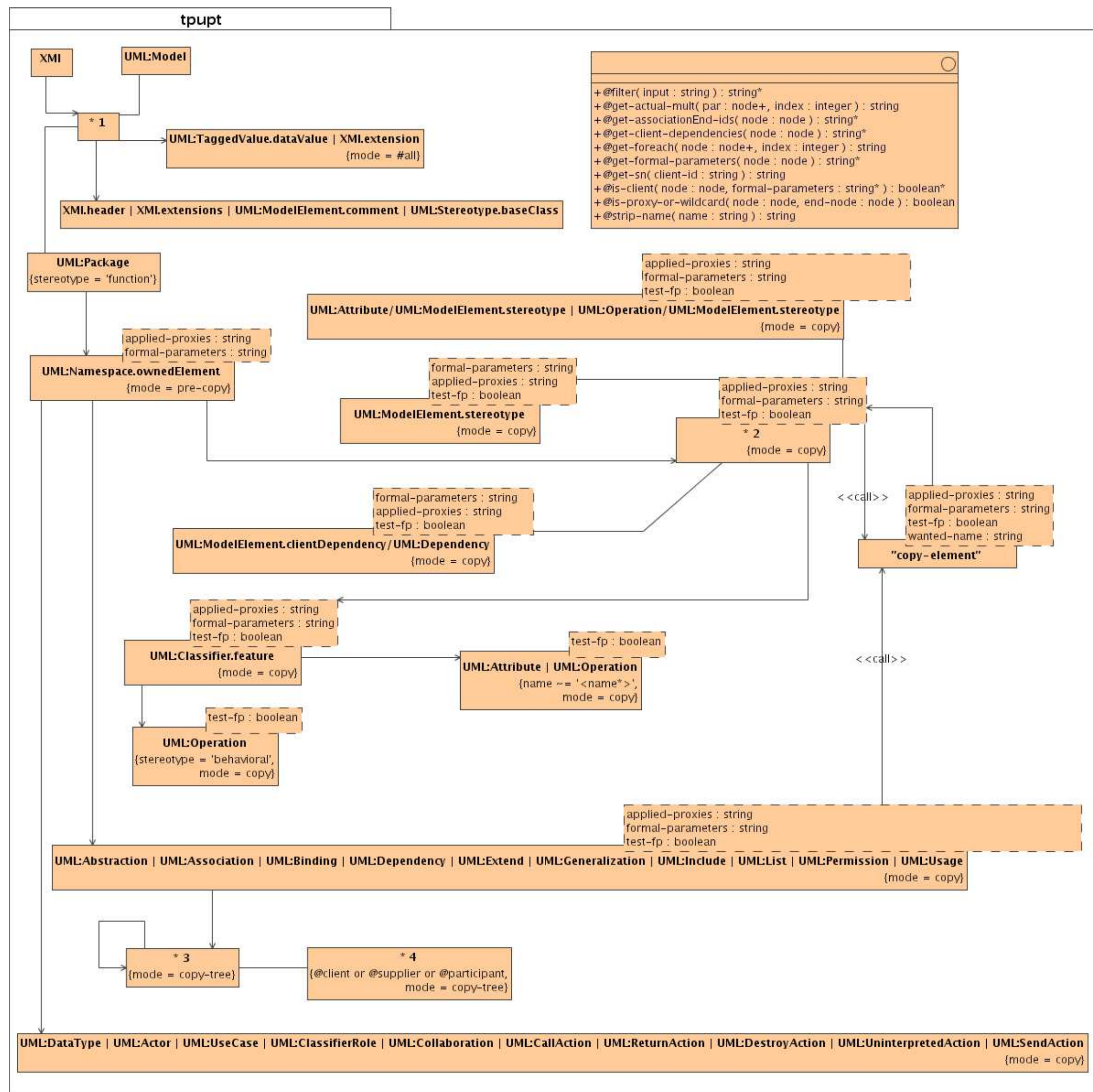

Figure 5.19: XSLT Template Flow Diagram of tpupt.xsl

Table 5.1: Tabular Overview of tpupt.xsl

Item	Kind	Goal
XMI	rule-template	Checks if the input file is an XMI 1.2 file.
UML:Model	rule-template	This element contains the XMI tree.
UML:Package	rule-template	Process packages which define a UML function.
* 1	rule-template	Search for XMI elements while skipping XMI extensions.
XMI.header | XMI.extensions | UML:ModelElement.comment | UML:Stereotype.baseClass	rule-template	Skip these elements.
UML:Attribute | UML:Operation	rule-template	If the name matches $< name* >$, copy the actual attributes or operations, or convert actual sequence messages into operations.
UML:Operation	rule-template	Process <<behavioral>> operations.
UML:ModelElement.stereotype	rule-template	Copy stereotype definitions, except for outermost resulting elements (a package, class, interface, or object) which name has a reference to a formal element.
UML:Abstraction | UML:Association | UML:Binding | UML:Dependency | UML:Extend | UML:Generalization | UML:Include | UML:List | UML:Permission | UML:Usage	rule-template	Process relational elements (copy resulting elements)
UML:ModelElement.clientDependency / UML:Dependency	rule-template	Copy dependency references if they do not point to a formal parameter, a proxy, or a <<behavioral>> operation.
* 2	rule-template	Process the contents of a UML function (copy resulting elements).
UML:DataType | UML:Actor | UML:UseCase | UML:ClassifierRole | UML:Collaboration | UML:CallAction | UML:ReturnAction | UML:DestroyAction | UML:UninterpretedAction | UML:SendAction	rule-template	These elements are not processed.
copy-element	call-template	Recursively copy the current element and all its children verbatim into the resulting UML model. All identifiers and references to an element are prefixed with $caller-id.
* 3	rule-template	Copy the elements of a temporary result tree verbatim, except for the XMI attributes *participant*, *client*, and *supplier*.
* 4	rule-template	Process elements of a temporary result tree which have *participant*, *client*, or *supplier* attributes.
UML:TaggedValue.dataValue | XMI.extension	rule-template	These elements are not processed.

Continued on next page

Table 5.1—continued from previous page		
Item	Kind	Goal
filter	XPath function	Return the list of formal parameter names.
get-actual-mult	XPath function	Calculate the multiplication factor of the resulting relation(s) between the resulting elements.
get-associationEnd-ids	XPath function	Return the *associationEnd* identifiers of this association.
get-client-dependencies	XPath function	Return the pointer(s) to the client part of a dependency if the formal parameter has a dependency.
get-foreach	XPath function	Return the name of the referenced formal parameter.
get-formal-parameters	XPath function	Extract the formal parameter identifiers if the parameter is not a resulting element.
get-sn	XPath function	Obtain the name of the supplier.
is-client	XPath function	Determine if the parameter refers to a formal parameter or a proxy.
is-proxy-or-wildcard	XPath function	Determine if the parameter is a proxy or has a name which matches $< name* >$.
strip-name	XPath function	Extract the part of the parameter name which is suitable for XSLT variable names.

The functional overview of tpupt.xsl is given in table 5.1. The flow diagram describing the flow between the XSLT templates of the generated script is given in figure 5.20. The generated template UML:Package and the flow related to that template is stereotyped as <<generated>>.

Table 5.2: Tabular Overview of the Generated Script

Item	Kind	Goal
XMI	rule-template	Start processing the input document and update the timestamp.
UML:Comment	rule-template	Write copyright notice to the model.
UML:Package	rule-template	Generate the resulting elements.
* 1	rule-template	Recursively copy this element to the output and modify attributes when needed.
* 2	rule-template	Process the result of the nested application. The result of this processing serves as an actual parameter for the parent function application.
fix-stereotypes	call-template	Ensure that the stereotype set of the resulting element is that of the actual input parameter minus that of the formal input parameter plus that of the formal output element, minus <<proxy>> and <<proxied>>
convert-messages	call-template	Convert sequence messages to operations.
handle-behavioral	call-template	Substitute the pointcuts definitions of all actual <<extensionClass>> classes.
get-actual-parameters	XPath function	Obtain the formally passed parameters.
get-actual-var	XPath function	Determine the value for the $actual - *$ variables.
get-ancestor-var	XPath function	Return this node or the nearest <<apply-function>> package which has a tag ’var’.
Continued on next page		

Table 5.2—continued from previous page		
Item	Kind	Goal
get-kind	XPath function	Determine the kind of the element which name is in the list of actual parameters.
get-operation-tag	XPath function	Return the specified tag of an Operation node.
get-subst-name	XPath function	Determine the element name after all $[a..b]$ sections are expanded (left to right).
get-tag-name	XPath function	Return the item out of the tag 'var' which contains the given name.
nest-actual	XPath function	Evaluate the parameter, which is an <<apply-function>> package, and change the result such that it can be used as an actual parameter to the parent function application package.
show-name	XPath function	Display and return the given name verbatim.

The functional overview of the generated script is given in table 5.2. libxmi.xsl is given in figure 5.21. It does not contain any XSLT template. This file serves as a library of XPath functions which are used by both tpupt.xsl and the generated script.

Table 5.3: Tabular Overview of libxmi.xsl

Item	Kind	Goal
root	variable	Reference to the root element of the document.
stereotype-id	XSLT key	Matches stereotypes on their XMI identifier.
get-ancestor	XPath function	Return the node or the nearest ancestor of the node whose name is in a list.
get-stereotype-ids	XPath function	Return the XMI identifiers of all stereotypes of the current node.
get-stereotype-names	XPath function	Return the names of all stereotypes of the current node.
get-supplier	XPath-function	Obtain the supplying element of the dependency belonging to this client.
get-tagged-value	XPath function	Return the name of the tag contained in the node.
xmi-version-supported	XPath function	Check if the XMI version of this document is supported, i.e. version 1.2.

The interface of this library is given in table 5.3.

5.3 An Actual Conversion

This section describes an actual conversion of the hotel reservation system as described in chapter 2.3.1. The functions used in this conversion are described in chapter 5.1.1. As in chapter 5.1.1, this section is split up into a business interfaces layer, a system interfaces layer, and a component and system architecture layer. This section describes the input for the automatic conversion and the functions used to realize this. It also shows the output generated by TPUPT resulting from the input. The pictures are manually drawn from the generated XMI tree.

5.3.1 Business Interfaces

This section describes the functions used to generate the business interfaces. Figure 5.22 shows the $make-Customer-Mgt$ package. This package generates the interface for the <<core>> class *Customer* at the business level. This interface will consist of a package containing an <<infotype>> class and an <<interface type>> interface. The class will contain the attributes of both the classes *Customer* and *Reservation*. Figure 5.23 shows the package $make-Room-Mgt$. The goal of this function application is

generated

+@get-actual-parameters(node : node) : node*
+@get-subst-name(name : string) : string
+@get-operation-tag(node : node, tag : string) : node?
+@get-actual-var(fp-name : string, actual-parameters : node*, func-name : string, fp-kind : string) : string*
+@nest-actual(node : node) : node*
+@get-tag-name(tag-var : string*, name : string) : string
+@get-ancestor-var(node : node) : node?
+@show-name(node : node) : node
+@get-kind(actual-parameters : node*, actual-name : string) : string

XMI

caller-id : string
index : integer = 1
owner : string

UML:Comment
{name(../..) = 'UML:Model'}

caller-id : string
index : integer = 1
modelElement : string
owner : string
tagName : string
type : string

* 1

parameters-wanted : boolean
type-wanted : boolean
formal-feature-id : string
formal-message-name : string
feature-visibility : string
caller-id : string
actual-parameters : node
index : integer

"convert-message"

<<generated>>

<<call>>
<<generated>>

<<generated>>
UML:Package
{stereotype = 'apply-function',
get-tagged-value(., 'name') = $function-name}

<<generated>>

tag-var : string

* 2
{mode = process-nested-result}

<<call>>
<<generated>>

<<call>>
<<generated>>

ec-name : string
pc-name : string
visibility : string
caller-id : string
actual-parameters : node
res-formal-name : string

"handle-behavioral"

node : node
formal-in-stereotypes : string
formal-out-stereotypes : string

"fix-stereotypes"

Figure 5.20: XSLT Template Flow of the Generated Script

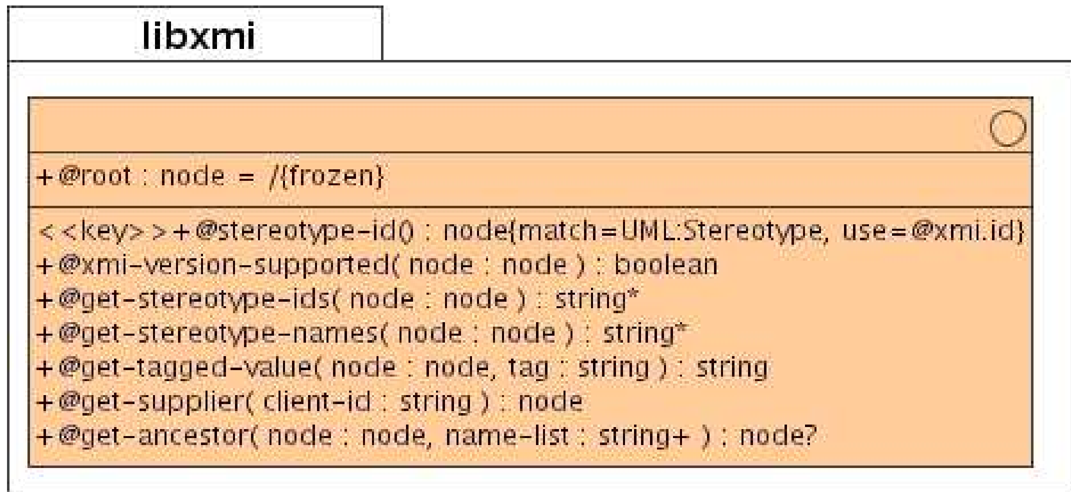

Figure 5.21: Interface Diagram of libxmi.xsl

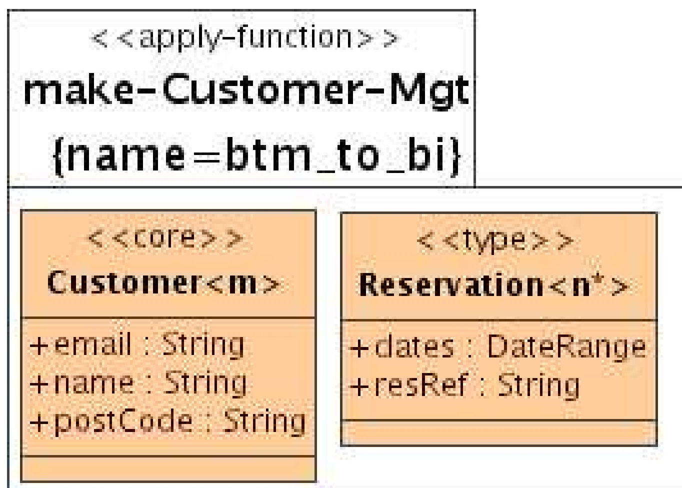

Figure 5.22: Generating the Business Interface for the *Customer* Class

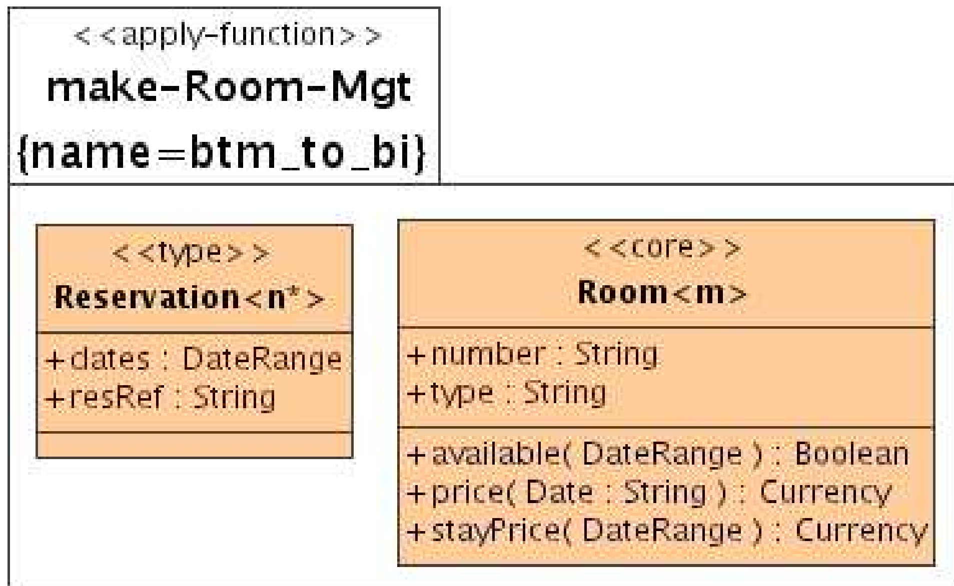

Figure 5.23: Generating the Business Interface for the *Room* Class

similar to that of $make-Customer-Mgt$. It will also create a package containing a class and an interface. Here, the class will contain the attributes of the *Room* and *Reservation* classes.

Note that the features *available*, *price*, and *stayPrice* of class *Room* are attributes, not operations. This is explained in [4].

Figure 5.24 and 5.25 respectively show the packages *ICustomerMgt* and *IRoomMgt*. These packages are generated by the packages $make-Customer-Mgt$ and $make-Room-Mgt$ respectively.

The <<infotype>> classes contain all attributes from the specified <<core>> classes (*Customer* and *Room*) and <<type>> classes (*Reservation*). The generated interfaces which are associated with the <<infotype>> classes remain empty. They must be completed manually.

5.3.2 System Interfaces

This section describes the functions applied to generate the system interfaces. The system interface is generated by the package $make-Hotel-interface$. This package is shown in figure 5.26. The proxied elements for this package are shown in figure 5.27. This package creates a package named after the use case (here *TakeUpReservation*). The package will contain an interface named after the use case. The interface will contain methods which are derived from the sequence messages pointed to by $< op1* >$ and $< op2* >$. The package will also contain the <<infotype>> class generated by the nested application of *btm_to_bi* as specified in $extract-Room$.

Figure 5.28 shows the generated system interface for the Take Up Reservation use case. The interface *ITakeUpReservation* is the main element. Figure 5.26 shows that the name of this interface is constructed from the use case passed as the actual parameter. The operations are constructed from the actual sequence messages shown in the same figure. The figure also contains an auxiliary class *Room*. This class is present because the interface uses this class in its signature. This class is constructed using the function *btm_to_bi*. Only <<infotype>> classes are added to the interface.

The function *btm_to_bi* generates a package containing an interface and a class. The generated class is extracted from this package using the function $extract-infotype$ and passed as actual parameter for the formal parameter $< d* >$.

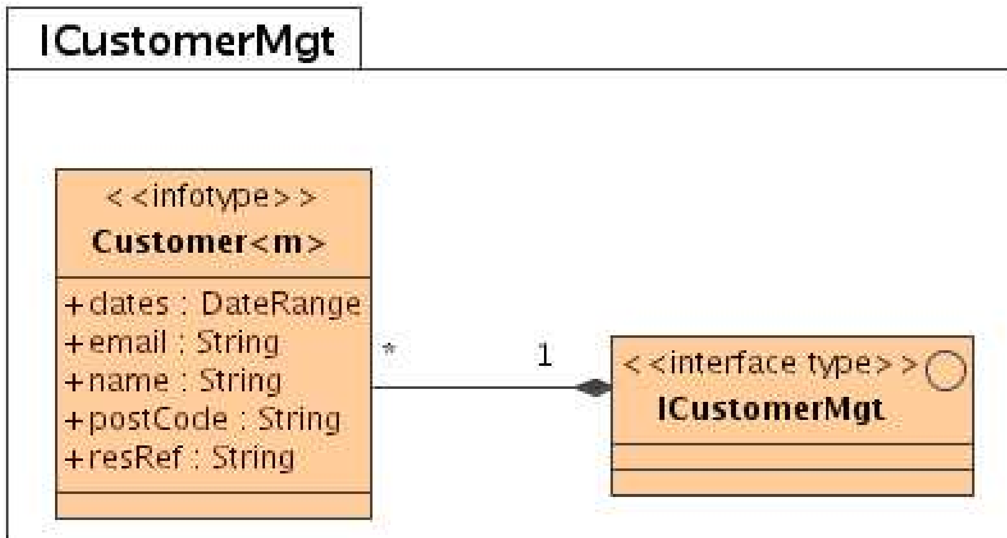

Figure 5.24: Generated Business Interface for the *Customer* Class

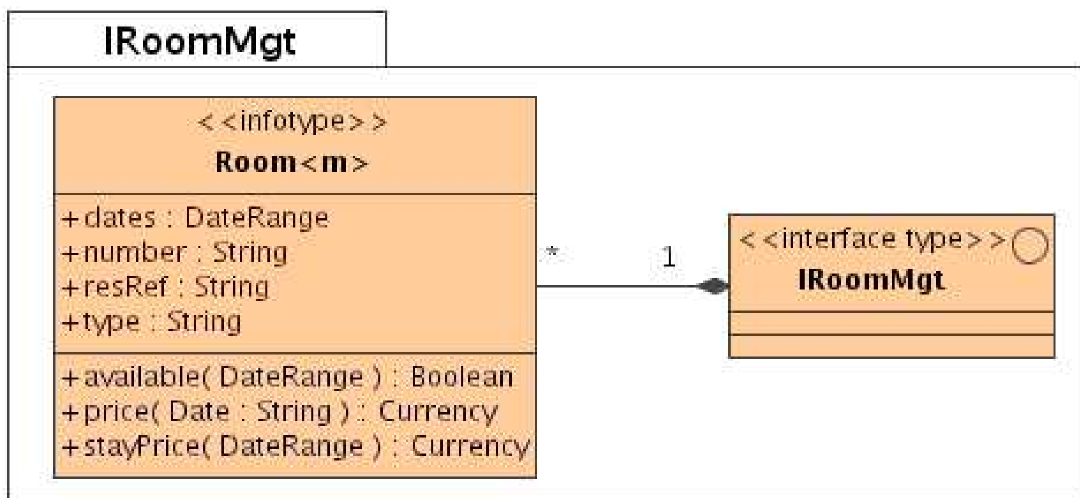

Figure 5.25: Generated Business Interface for the *Room* Class

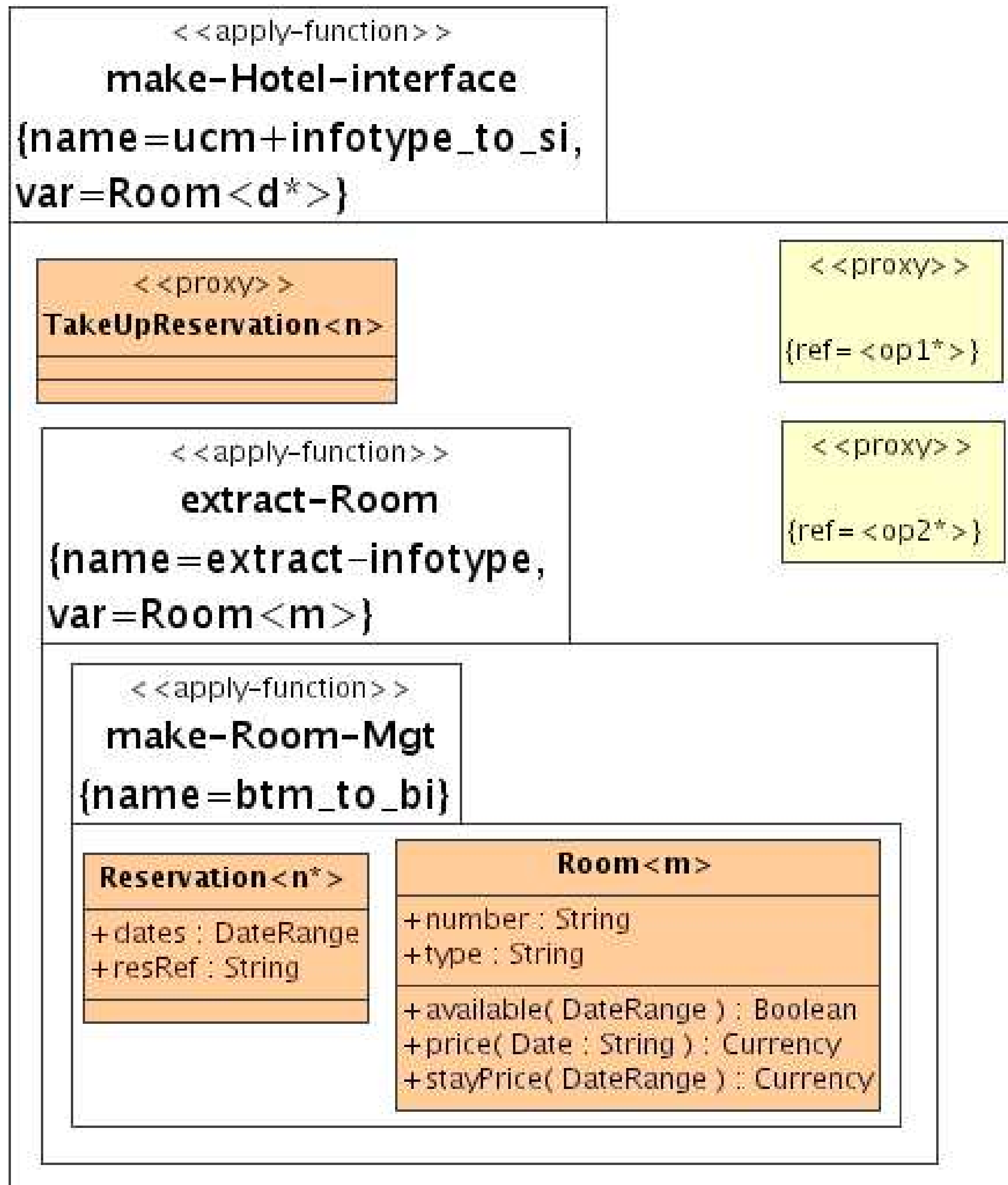

Figure 5.26: Generating the System Interface for the Hotel

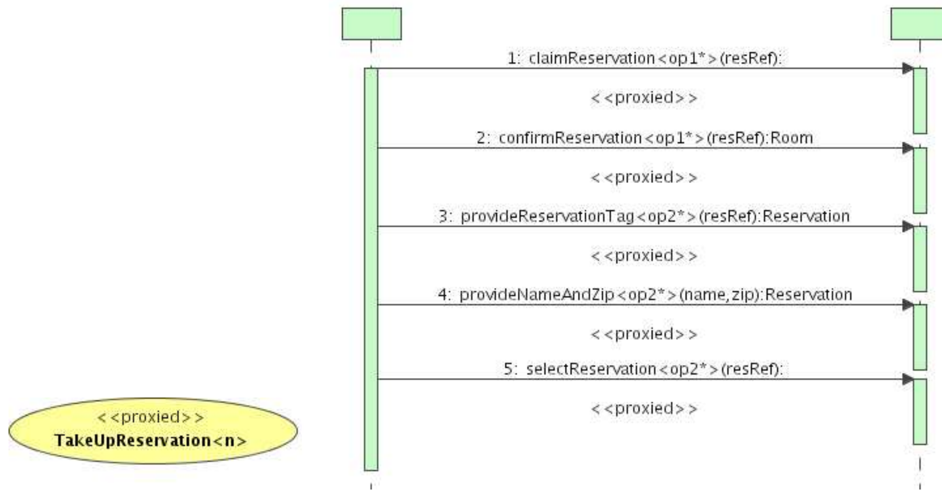

Figure 5.27: Proxied Elements for Generating the System Interface

5.3.3 Component and System Architecture

This section describes the functions applied to generate the component and system architecture. Figure 5.29 shows the package $make-Customer-Mgr$. This package generates a component specification class for the <<core>> class $Customer$. Figure 5.30 shows the package $make-Room-Mgr$. This package generates a component specification class for the <<core>> class $Room$. Figure 5.31 and 5.32 show the package $make-IdentifyReservation-arch$. This package generates a component architecture for the Identify Reservation use case. This architecture will consist of a package containing a control class and a boundary class. All generated elements will be named after the use case. The two classes will be connected using an association. Figure 5.33 and figure 5.34 show the package $make-TakeUpReservation-arch$. This package generates a component architecture for the Take Up Reservation use case. The architecture will be analogous to that of the Identify Reservation use case. Figure 5.35 and 5.36 show the package $make-Reservation-compspec$. This package generates a component specification class from the supplied <<system boundary>> package (here $ReservationSystem$). The use cases are not used for the conversion proper; they are only used to ensure that the <<system boundary>> package is not empty.

Figures 5.37 and 5.38 show the generated component architectures for the two use cases in the hotel reservation case: Identify Reservation and Take Up Reservation. Both architectures contain of a package having the same name as the use case. These packages both contain a control class and a boundary class which are connected using an association. The name of the control class ends in Controller, while the name of the boundary class ends in Form. Figure 5.39 shows three component specification classes. Class $ReservationSystem$ is generated by the package $make-Reservation-compspec$ (figure 5.35), while the other two classes are generated by the packages $make-Customer-Mgr$ (5.29) and $make-Room-Mgr$ (figure 5.30). All component specification classes remain empty. They must be completed manually.

5.4 Answered Research Questions

This chapter answered the final three research questions. Section 5.1.1 defined the functions required for the UML Component Method. This gave an answer to research question 7. Section 5.1.2 defined the functions needed for AOSD. This answered research question 8. The last research question, number 10, was answered in section 5.2. This section provided the design for the implementation of the tool.

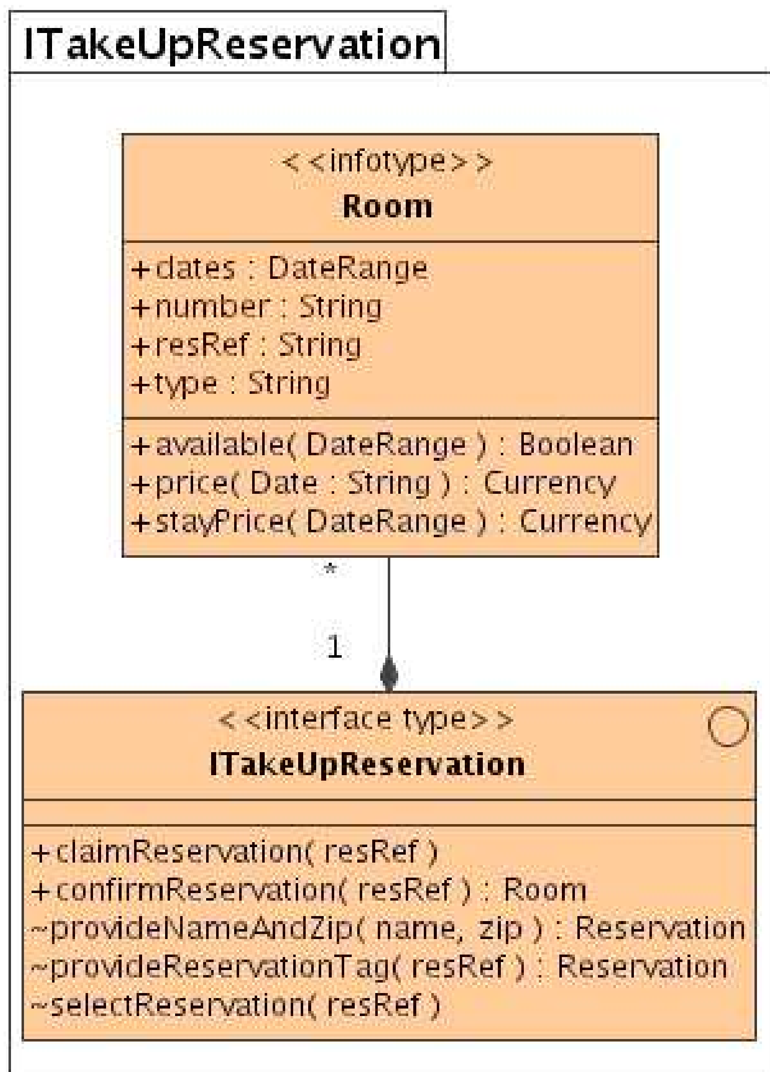

Figure 5.28: Generated System Interface for the *Take Up Reservation* Use Case

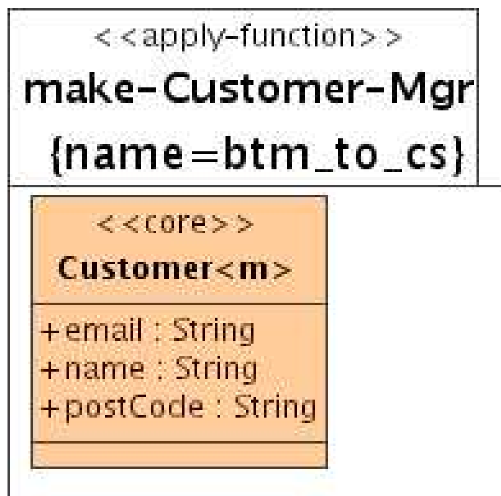

Figure 5.29: Generating the Component Specification for the *Customer* Class

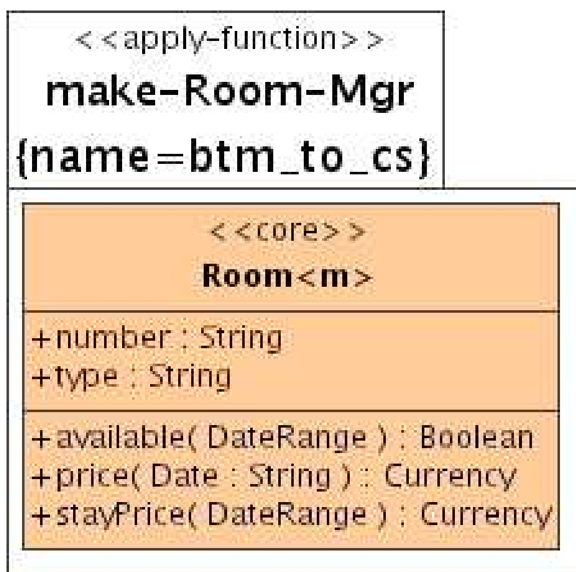

Figure 5.30: Generating the Component Specification for the *Room* Class

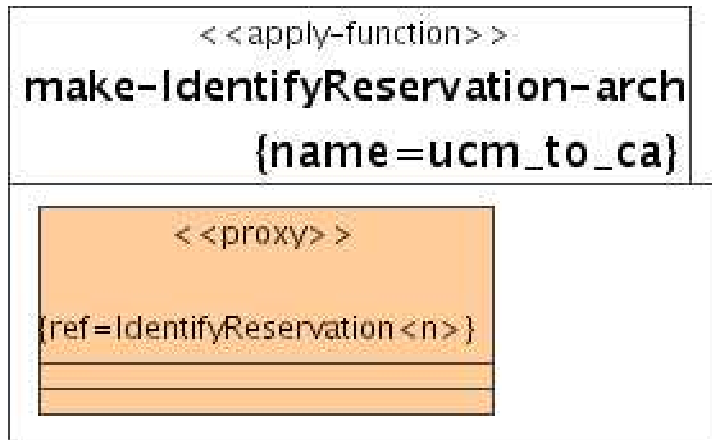

Figure 5.31: Generating the Component Architecture for the *Identify Reservation* Use Case

Figure 5.32: Proxied Elements for the *Identify Reservation* Use Case

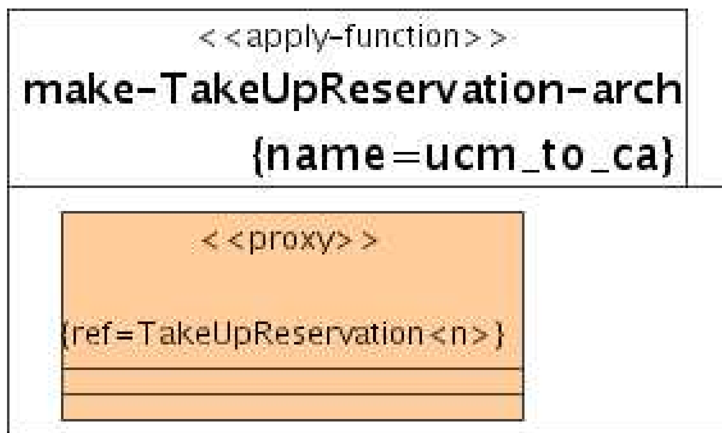

Figure 5.33: Generating the Component Architecture for the *Take Up Reservation* Use Case

Figure 5.34: Proxied Elements for the *Take Up Reservation* Use Case

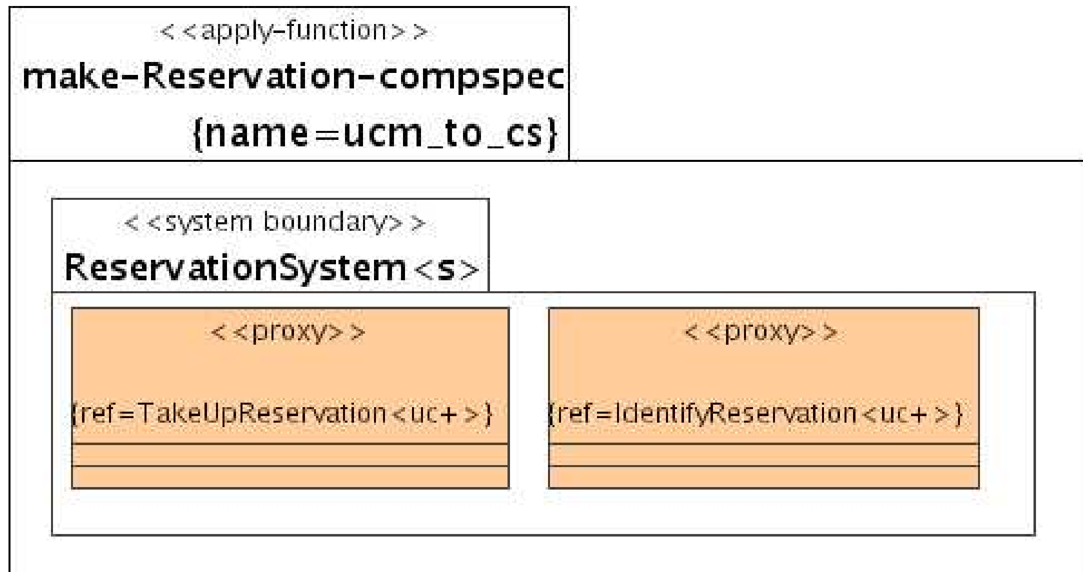

Figure 5.35: Generating the Component Specification for the *ReservationSystem* Package

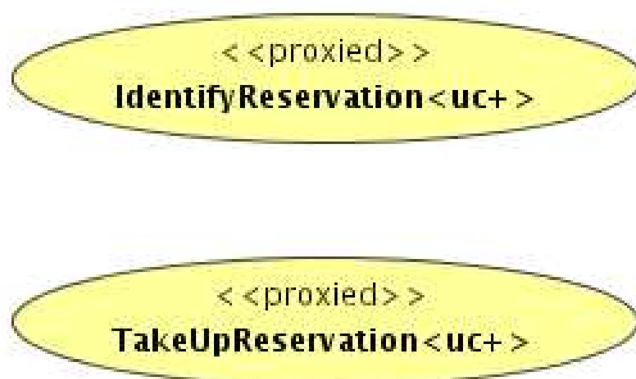

Figure 5.36: Proxied Elements for the Component Specification

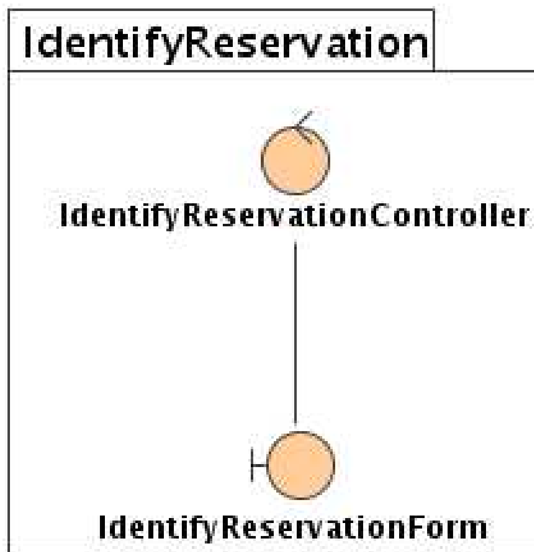

Figure 5.37: Generated Component Architecture for the *Identify Reservation* Use Case

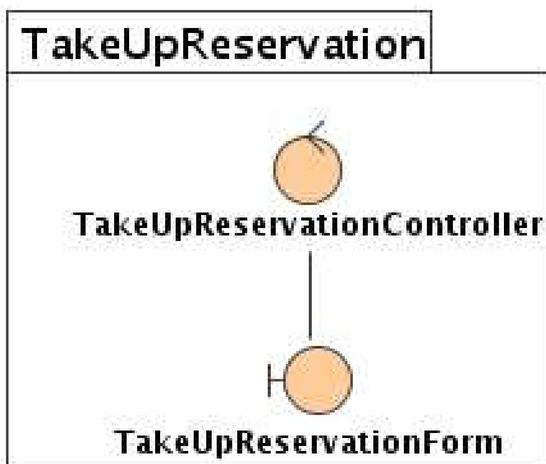

Figure 5.38: Generated Component Architecture for the *Take Up Reservation* Use Case

Figure 5.39: Generated Component Specifications

Chapter 6

Conclusions

"Don't let it end like this. Tell them I said something."
—Pancho Villa

This chapter draws some conclusions from the project by giving a summary, an overview of the things I learned during this project, and stating some possible future directions. First it gives a list stating which requirements have been fulfilled.

6.1 Final Conformance to the Requirements

This section gives an overview of the requirements. It explains the requirements and it show in how far they have been fulfilled.

Chapter 2 defined requirements (a1.1) to (a1.6) for the UML Component Method:

- (a1.1) Add an interface, package, or class with a given name, stereotype(s), and tag(s) to a package or namespace. The added elements are either new or derived from another class, actor, use case, or interface. This requirement is completely fulfilled. All functions create a resulting class, interface, and/or package.
- (a1.2) Add attributes or operations with a given name, type, multiplicity, and visibility to a class. These attributes or operations are either new or copied from another class. This requirement is completely implemented by the functions btm_to_bi, $ucm + infotype_to_si$, $make_csa$, $extract - infotype$, $unwrap - nuss$, $unwrap - us$, $substitutePointcuts$, and $mergeContents$.
- (a1.3) Transform messages of a sequence diagram into interface methods with a given name, parameter(s), visibility, and return type. The parameters of these methods have a given name and type. This requirement is completely implemented by the functions $ucm + infotype_to_si$ and sd_to_sysint.
- (a1.4) Add an association, dependency, or composition. This requirement is completely implemented by the functions btm_to_bi, $ucm + infotype_to_si$, $make_csa$, and ucm_to_ca.
- (a1.5) Change the name of the actual parameter. This requirement is completely fulfilled. All functions support name editing, the name changes in table 4.3 are fully supported.
- (a1.6) Make it possible to combine different diagram types in a transformation. This requirement is completely implemented by the functions sd_to_sysint, $ucm + infotype_to_si$, $make_csa$, ucm_to_ca, and $ucm_to_c s$.

The above list shows that all requirements for the UML Component Method are fulfilled, which was a main requirement for this project.

Chapter 2 also defined requirements (a2.1) to (a2.6) for AOSD. Some of these requirements reference a set C. This set contains static or dynamic class names.

- (a2.1) Add a class with specified contents which is contained in a slice to a namespace. This requirement is completely implemented by the functions $unwrap - nuss$ and $unwrap - us$.
- (a2.2) Merge attributes or operations into a specified set of classes C. This requirement is partially implemented by the function *mergeContents*. Merging into dynamic members of C is not supported.
- (a2.3) Merge operation extensions as specified by the structural pointcuts into a specified set of classes C. This requirement is partially implemented by the function *mergeContents* when applied to the result of the function *substitutePointcuts*. Merging into dynamic members of C is not supported.
- (a2.4) Tag operations affected by behavioral pointcuts with these pointcuts. This requirement is completely implemented by the function *substitutePointcuts*.
- (a2.5) Remove <<aspect>>, <<usecase-slice>>, and <<non-usecase-specific-slice>> packages once their contents is processed. This requirement is completely implemented by the function *mergeContents* for <<aspect>> packages, by the function $unwrap - us$ for <<usecase-slice>> packages, and by the function $unwrap - nuss$ for <<non-usecase-specific-slice>> packages.
- (a2.6) Add relations from within a slice to a namespace. This requirement is not fulfilled at all.

The above list shows that the requirements for AOSD are only partially fulfilled.

Chapter 3 defined requirements (b1) to (b5.3) regarding the proposed solution:

- (b1) Suitability in the production line of GPR. This requirement is completely fulfilled. Saxon is a fast and the most conforming XSLT processor according to [15].
- (b2) Specification. This requirement is not a requirement for the tool or the transformation functions.
- (b3) Ease of understanding. This requirement is not a requirement for the tool or the transformation functions.
- (b4) Availability. This requirement is completely fulfilled. ArcStyler 5.1 and XSLT are already included in the production line of GPR.
- (b5.1) End user usability. This requirement is not a requirement for the tool or the transformation functions.
- (b5.2) Ease of programming. This requirement is not a requirement for the tool or the transformation functions.
- (b5.3) Abstraction level. This requirement is not a requirement for the tool or the transformation functions.

The above list shows that the two requirements, (b1) and (b4), are fulfilled.

Chapter 5 defined requirements (c1) to (c4) regarding the final solution:

- (c1) The system is two pass. This requirement and its subrequirements are completely fulfilled, see the remarks of requirements (c1.1) through (c1.4). out-M.xsl is a generated file which consists of XSLT 2.0 translations of the UML functions.
- (c1.1) Pass 1 transforms the UML function definitions into a script. Pass 1 is executed by tpupt.xsl which creates out-M.xsl
- (c1.2) Pass 2 applies the UML functions. Pass 2 is executed by out-M.xsl
- (c1.3) Pass 2 is generated by pass 1. tpupt.xsl generates out-M.xsl, where M is the name of the UML model.
- (c1.4) Both passes are defined in the same programming language. All program files (tpupt.xsl, base.xml, libxmi.xsl, and out-M.xsl) are written in XSLT 2.0

- (c2) The rules of chapter 4 are obeyed by the scripts. This requirement is completely fulfilled. The rules are respected but not enforced.
- (c3) The input and output of the scripts is valid XMI 1.2 This requirement is completely fulfilled. The generated output can be read successfully with ArcStyler 5.1
- (c4) ArcStyler 5 can be used to define the UML functions. This requirement is completely fulfilled. The supplemental files aosd.xml, reservation.xml, and student.xml are written in ArcStyler 5.1

The above list shows that all requirements regarding the tool are completely fulfilled.

6.2 Summary

This section gives a short summary of the main part of this thesis: chapter 4.2, 4.3, 5.1, and 5.2.

Chapter 4.2 defines the formal model which is used in chapter 4.3. This model consists solely of lists which are described in $\lambda\omega$. It was necessary to create this model because the template model as defined in [14] is only partially implemented in contemporary UML tools. Another reason to create this model is that this model is much smaller than the UML template model.

Chapter 4.3 defines the foundations for the UML functions defined in chapter 5.1. They can be invoked using a package having stereotype <<apply-function>>. This package contains the actual parameters for that function. These functions can use the results of other UML function calls as actual parameters. However, function definitions cannot be recursive. UML functions can be and most of the time are defined in an existing UML model. This way they are able to change that model.

The functions are defined as packages having stereotype <<function>>. UML functions behave like normal functions, which means that they are free of side effects. Each function changes a group of n parameter elements into a group of m resulting elements. Both these groups consist of zero or more elements.

If the parameter elements spawn multiple diagram types, then the diagram which contains the resulting elements must contain proxies to the elements in the other diagram types. The proxy elements are often classes or objects. The proxied elements are often use cases, actors, and sequence messages.

Chapter 4.3 also defines several rules. These rules serve as conditions for both defining and applying the functions.

Chapter 5.1 defines all the functions necessary to fulfill the requirements in chapter 5.1.1 and 5.1.2. It defines 7 functions for the UML Component Method and 4 functions for AOSD.

Chapter 5.2 described the design of the tool itself. The tool is a two pass script: the first pass creates an XSLT script of the UML functions, the second pass applies the functions to the original model. The tool respects, but not necessarily enforces, the rules defined in chapter 4.3. The design of the tool, which is written in XSLT, is given as UML class diagrams accompanied by a textual tables.

6.3 Things Learned

This section lists some things which I have learned since starting this project.

6.3.1 XSLT

GPR had a preference for the implementation language of the tool: XSLT. This forced me to learn XSLT from scratch, and gain a high enough level to be able to implement the tool. XSLT is a language designed to convert documents. UML models are also documents and are quite suitable to be processed by XSLT.

Chapter 3.2.3 mentions some problems when using XSLT for larger programs, but these problems did not affect the tool (around 58 kB of code). The code of the initial XSLT implementation became quite cluttered due to a lack of experience. The intermediate Python implementation, although never finished, was a great help in understanding all the architectural requirements of the tool. Since I was already familiar with Python I could concentrate on the tool itself without having to concentrate on the implementation language. This proved to be quite a help with the final XSLT implementation.

6.3.2 Writing User-oriented Documents

Writing documents itself is not really a problem. Writing them in a way such that a (novice) user understands them is a lot harder, because a lot of ”obvious” things have to be explained. These things are either ”obvious” out of familiarity or because they seem ”unimportant details”.

This is the point where reviewers who bug me about all these ”obvious” things come in. Their constructive criticism result in a clearer document.

6.3.3 XMI 1.2

XMI 1.2 is a standard to textually describe UML models. An introduction is given in appendix C. The full specification can be found at [13]. It took me about a week to figure out how XMI 1.2 actually works. The hard part to understand were the crossreferences which link together the various parts of the model. The rest was much easier to understand. XMI 1.2 looked familiar for the most part since I already had some experience with the XMI 1.0 standard. XMI 1.2 is like XMI 1.0, but more compact, making the model elements easier to recognize.

6.3.4 AOSD

AOSD is still a relatively new method to develop software. It has not yet been included in the UML standard. The essence of AOSD is described in chapter 2.2.

Although AOSD support was not a main requirement for the project, it was certainly an interesting concept to explore. The nice thing about AOSD is that an application can be developed from independent parts. which are woven together at the end of the development. Although this is also possible with more conventional methods like module based development, AOSD allows e.g. all error handling or all logging functionality to be in one module instead of having it scattered all over the application source.

The fact that AOSD is not yet standardized gave a wider degree of freedom to implement the support. It is for the most part still based on [9].

6.3.5 UML Component Method

The UML Component Method is the main pillar of this project. Because a number of operations in the various steps are more or less duplicates of earlier operations, some people at GPR thought that the method could be automated. The outcome of this project has proved that this automation is indeed possible.

The method itself is not hard to learn. To quickly grasp the method, it is best to read some introduction and to experiment with some examples from [4].

6.3.6 λ Calculus

The calculus $\lambda\omega$ is used in chapter 4.2 and 4.3. Although this calculus was not new to me, writing this chapter forced me to reread some material on the calculus.

This calculus is set up in a quite minimalistic way, which makes it easy to prove the correctness of its applications. On the other hand, this minimalism also causes expressions to grow hairy quite fast. This disadvantage lead me to invent the sugaring rules described in chapter 4.2.

6.4 Future Directions

The following lists contains some possible future directions, based on customer interest:

- Models which are transformed by TPUPT contain corrupt picture information. ArcStyler generates a bunch of ignorable warnings as a result of this corruption. The cause of this corruption is that the picture information is stored in a part specific to each UML tool. TPUPT does not bother about these parts as they are not standardized.

- TPUPT applies few checks to both phases of the model transformation. These checks could be added, thereby possibly increasing the execution time. The checks are described in section 4.3.
- TPUPT has no support for the Object Constraint Language (OCL). Adding this support makes it possible to include constraints in the functions. These constraints can then serve as a user-defined extension to the above checks.
- As an alternative to XSLT, one could have a look at F-Logic (see section 3.2.3). Parts of TPUPT might be rewritten in F-Logic.
- The AOSD support as specified by requirements (a2.1) through (a2.6) could be completed. This probably requires modifications to both the AOSD functions and the tool.

Bibliography

[1] The AspectJ Project at Eclipse.org.
http://www.eclipse.org/aspectj/.

[2] Henk P. Barendregt. Lambda Calculi with Types. *Handbook of Logic in Computer Science*, 2, 1992.
ftp://ftp.cs.ru.nl/pub/CompMath.Found/HBK.ps.

[3] Morgan Björkander and Cris Kobryn. Architecting Systems with UML 2.0. pages 57–61, July/August 2003.
http://www.uml-forum.com/docs/papers/IEEE_SW_Jul03_p57_Kobryn.pdf.

[4] John Cheesman and John Daniels. *UML Components –A Simple Process for Specifying Component-Based Software*. Addison-Wesley, 2001. ISBN 0-201-70851-5.

[5] Andy Evans, Paul Sammut, James S. Willans, and Alan Moore Girish Maskeri. A Unified Superstructure for UML. *Journal of Object Technology*, 4(1):165–181, January-February 2005.
http://www.jot.fm/issues/issue_2005_01/article6.pdf.

[6] Anthony J. Field and Peter G. Harrison. *Functional Programming*. Addison-Wesley, 1988. ISBN 0-201-19249-7.

[7] Anna Gerber, Micheal Lawley, Kerry Raymond, Jim Steel, and Andrew Wood. Transformation: The Missing Link of MDA. Technical report, Distributed Systems Technology Centre (DSTC), 2002.
http://www.dstc.edu.au/Research/Projects/Pegamento/publications/icgt2002.pdf.

[8] Ivar Jacobson. Use Cases and Aspects –Working Seamlessly Together. *Journal of Object Technology*, 2(4):7–26, July-August 2003.
http://www.jot.fm/issues/issue_2003_07/column1.pdf.

[9] Ivar Jacobson and Pan-Wei Ng. *Aspect-Oriented Software Development with Use Cases*. Pearson Education Inc., 2005. ISBN 0-321-26888-1.

[10] Micheal Kay. *XSLT 2.0 Programmer's Reference*. Wrox Press, 3rd edition, 2004. ISBN 0-764-56909-0.

[11] Jernej Kovse and Theo Härder. Generic XMI-based UML Model Transformations. Technical report, University of Kaiserslauten, Department of Computer Science.
http://wwwdvs.informatik.uni-kl.de/pubs/papers/KH02.OOIS.pdf.

[12] OMG. *Unified Modeling Language Specification 1.4*, September 2001.
http://www.omg.org/docs/formal/01-09-67.pdf.

[13] OMG. *XML Metadata Interchange (XMI) Specification 1.2*, January 2002.
http://www.omg.org/cgi-bin/apps/doc?formal/02-01-01.pdf.

[14] OMG. *Unified Modeling Language: Superstructure 2.0*, August 2005.
http://www.omg.org/cgi-bin/apps/doc?formal/05-07-04.pdf.

[15] Support for the Saxon XQuery Processor.
http://www.stylusstudio.com/saxon_xquery_processor.html.

[16] Mario Südholt. Language Support for AOP –AspectJ and Beyond. Technical report, INRIA and École des Mines de Nantes, March 2005.
http://www.emn.fr/x-info/sudholt/talks/sudholt-boston-aspect-days-050308.pdf.

[17] TPUPT.
http://home.tiscali.nl/rladan/tpupt/.

[18] XMI DTD.
MDROOT/data/dtd_xsd/uml14xmi12.dtd, MDROOT is the root folder of a MagicDraw installation.

[19] W3C. *Extensibible Markup Language (XML) 1.0*, 4th edition, August 2006.
http://www.w3.org/TR/xml/.

[20] Annika Wagner. A Pragmatical Approach to Rule-Based Transformations within UML using XMI.difference. Technical report, University of Paderborn, Department of Computer Science.
http://ctp.di.fct.unl.pt/~ja/wituml/wagner.pdf.

[21] Aspect-Oriented Programming –Wikipedia.
http://en.wikipedia.org/wiki/Aspect-oriented_programming#Implementations.

[22] Lambda Calculus –Wikipedia.
http://en.wikipedia.org/wiki/Lambda_calculus.

[23] Unified Modeling Language –Wikipedia.
http://en.wikipedia.org/wiki/Unified_Modeling_Language.

[24] XSB.
http://xsb.sourceforge.net/.

Appendix A

UML Legend

This appendix contains a small UML legend which covers the UML usage in this project. It also covers AOSD and the functions used in the final solution. The four general legends are covered first.

A.1 Use Case Diagram

Figure A.1 contains the legend for use case diagrams. This diagram type specifies which actions each user (actor) can take. The basic elements of each use case diagram are actors and normal use cases connected by associations. Multiple actors can be connected to multiple use cases. The inclusion and extension use cases are mostly used within AOSD. They respectively model the inclusion of one use case into another and use cases used as a library.

A use case diagram is in fact a specialized class diagram; both actor and use case elements are classifiers.

A.2 Sequence Diagram

Figure A.2 contains the legend for sequence diagrams. This diagram type details a use case from the use case diagram by specifying the precise actions between an actor and a system. This is done by sending messages, which is denoted by arrows between the two classifier roles. The only message types used in this project are call, return, and destroy messages. Call and return messages behave much like normal messages, destroy messages specify the destruction of the classifier it points to. This is indicated by a cross. Call messages are able to denote the interface method to invoke, in this case s and o. Call and return messages can be split to model alternatives.

A.3 Class Diagram

Figure A.3 contains a legend for class diagrams. This diagram type is used to model the static parts of software systems.

The upper part contains a class *Class* connected to a class <<control>> *Class* by a composition. Class *Class* has a private attribute a and a public operation o. Attribute a and operation o have no specified type, while parameter p of operation o has type int. Class <<control>> *Class* has a stereotype <<control>>, which is denoted graphically by the arrowed circle. It has a protected attribute a of arrayed type int, meaning this attribute can only be seen by the class itself and by derived classes. The composition between the two classes specifies that class *Class* cannot exist without class <<control>> *Class*, as it is a part of it.

The second part contains a package *Package* which has a stereotype <<stereotype>>. This is denoted by placing the name of the stereotype between guillemots. Packages are designed to contain other elements. This package contains a class <<boundary>> *Class* and an interface $Interface$, which are connected

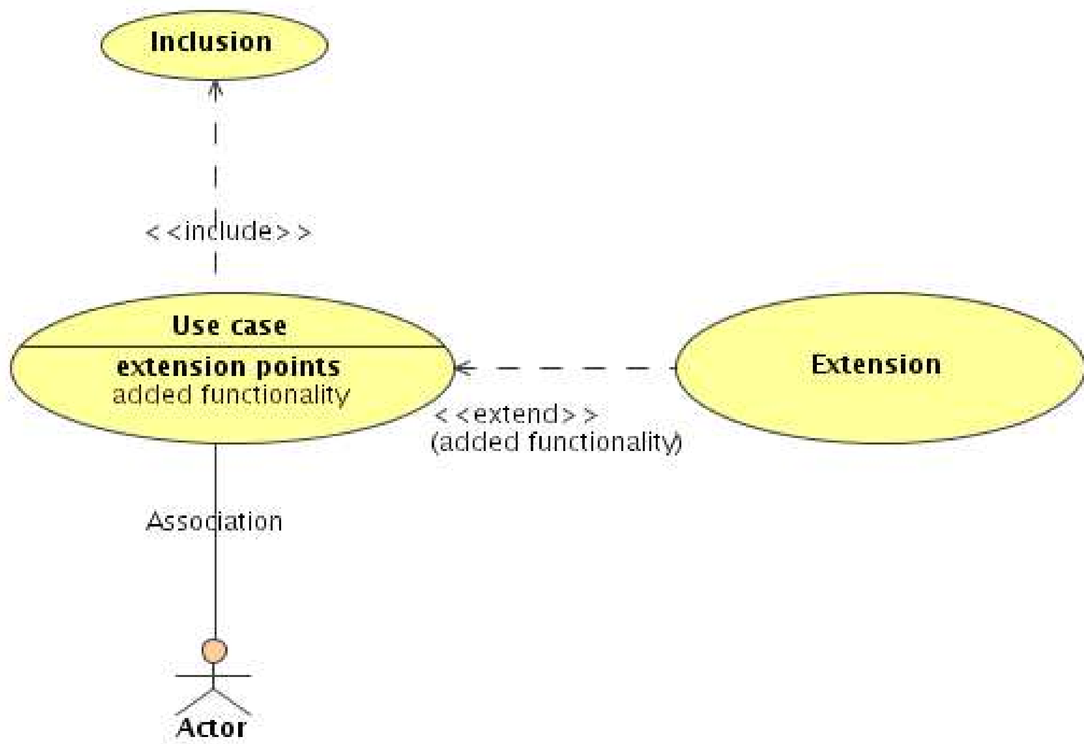

Figure A.1: Legend for Use Cases Diagrams

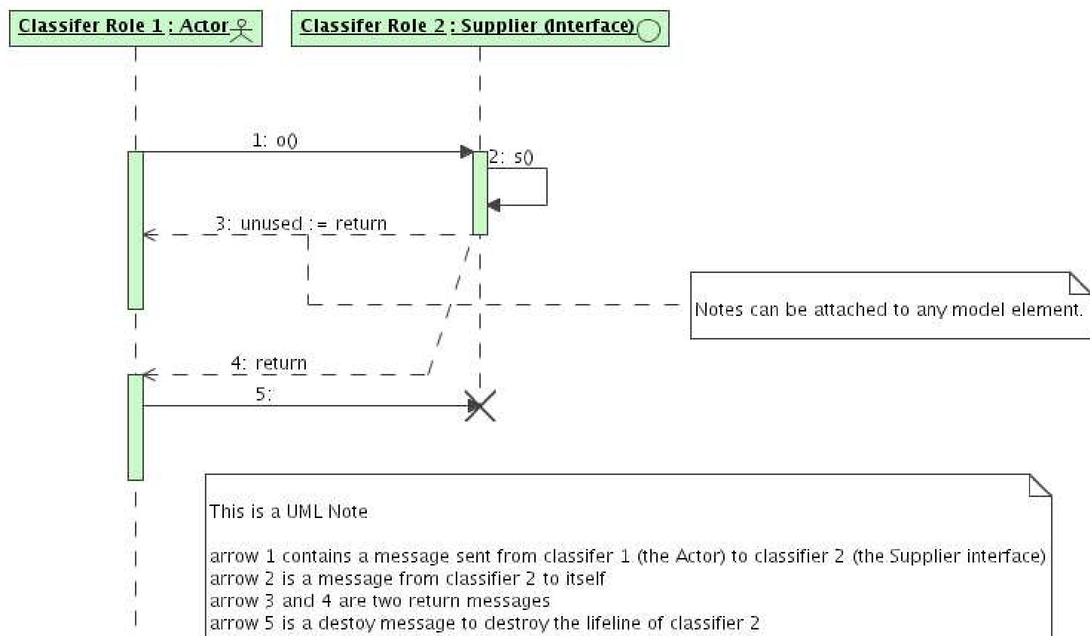

Figure A.2: Legend for Sequence Diagrams

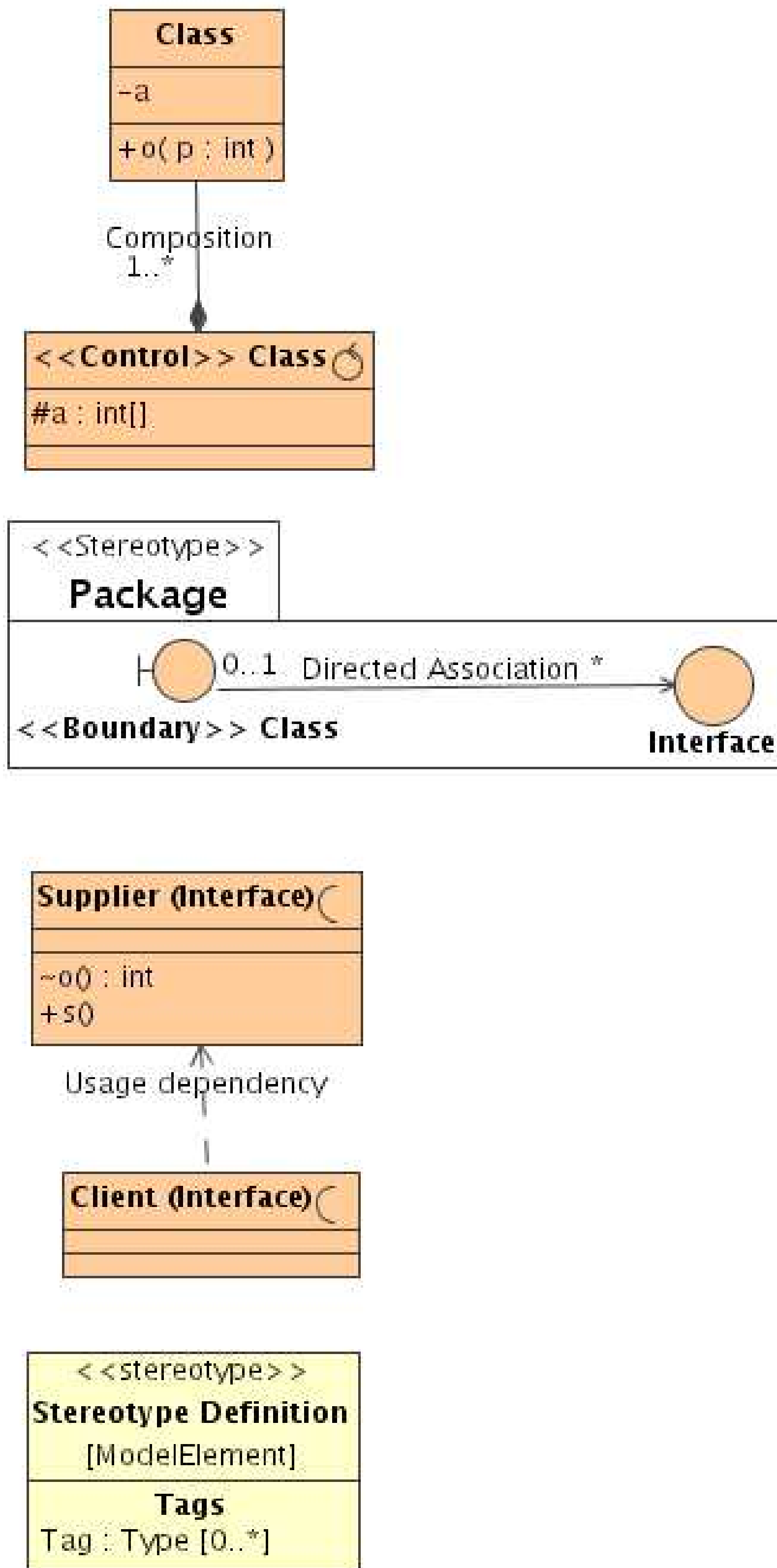

Figure A.3: Legend for Class Diagrams

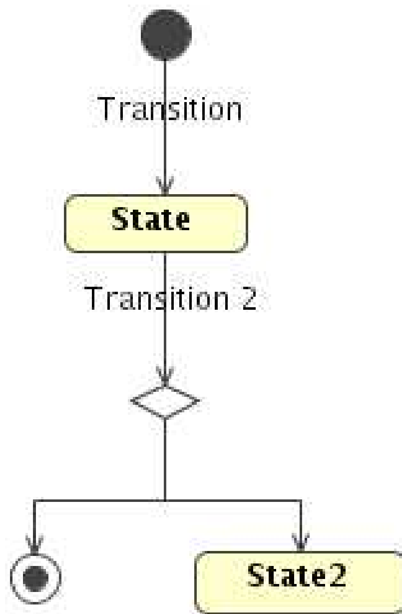

Figure A.4: Legend for Activity Diagrams

using a directed association *Directed Association*. The class has stereotype <<boundary>>, which is denoted graphically by turning the rectangle into a circle with a "T". It has a multiplicity 0..1, meaning that each class is related to at most one interface. The interface has multiplicity $*$, meaning that each interface is related to any number of classes. The association is directed towards the interface, meaning that the class can see the interface but not vice versa.

The third part contains two interfaces connected by a usage dependency. The interface *Client* uses elements from the interface *Supplier*. The latter has a method o which returns an integer and which has visibility "package". This means that all elements in the package and all elements in subpackages of that package can see o. It also has a public method s.

The fourth part denotes a stereotype definition. This definition contains the name of the stereotype, here <<stereotype definition>>, and the type of model element for which it is defined, here *ModelElement*. Stereotypes can have tags, which act as name/value extension pairs to the stereotype.

A.4 Activity Diagram

Transition diagrams are not used in this project. An exception is the Student Registration example. A transition diagram models the different states in which a system can be. In this diagram the states are one of:

- The initial state denoted by the filled circle.
- The final state denoted by the half-filled circle. A transition diagram can have more than one final state.
- Intermediate states denoted by rectangles. In this diagram, these are *State* and *State*2.

Transition arrows can be split and joined by pseudo states, which are modeled as diamonds. Figure A.5 contains the legend for AOSD accompanied by a representation in AspectJ. Use case slices and aspects are modeled as packages with appropriate stereotypes.

A.5 AOSD Class Diagram

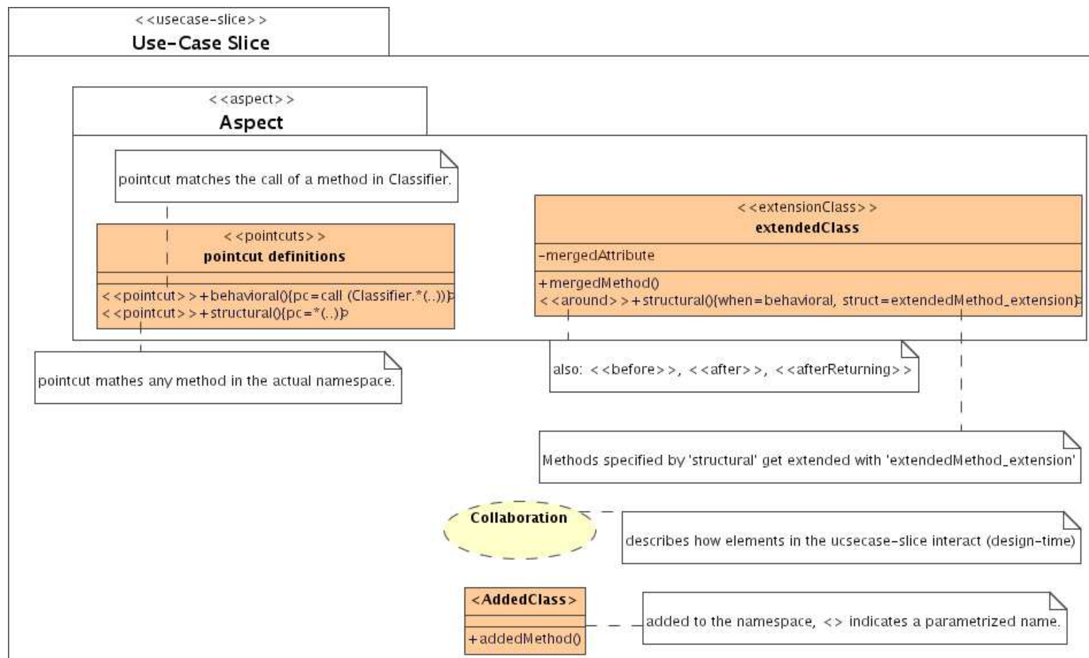

Figure A.5: Legend for AOSD

A use case contains one collaboration, zero or more added classes ($< AddedClass >$ in the example), and one or more aspects (*Aspect* in the example). Added classes get added to the namespace verbatim after parameter substitution. Collaborations are not added as they are design-time items.

Aspects get woven into the classes they specify by the <<extensionClass>> stereotype. In figure A.5 the class *extendedClass* gets extended with an attribute *mergedAttribute*. All methods which match the pointcut *structural* get extended at the points where they call a method matched by the pointcut *behavioral*. At these points the calls are encapsulated by the code given by *extendedMethod_extension*.

Pointcuts are modeled as methods in a class with stereotype <<pointcuts>>. These methods must be stereotyped as <<pointcut>>. Their match is specified by the tag *pc*. In this tag wildcards can be used: $*$ to specify any (part of a) method name and (..) to indicate any method signature.

Methods can also be added statically to the extended class (just like attributes). In this case the specifications of *structural* and *behavioral* are omitted. Note that the tool only processes structural pointcuts in <<pointcuts>> classes.

A.6 UML Function Diagram

The diagram in figure A.6 models the functions and their application using UML. The function itself is shown as the package *Function*. This package has a stereotype <<function>> to specify that it is a UML function. This function can have zero or more parameters. The function converts an interface *formal parameter* into an empty class *result* and "eats" a collaboration *eaten*. It also adds an empty package *added* to its output environment.

The second package, which is colored brown for recognition, is an application of the function. The application of a function is denoted by the stereotype <<apply-function>>. This stereotype has a tag *name* which contains the name of the function to apply. The name of the package itself is not considered. Any parameters which the function requires are modeled as elements inside the applying package. The result of this application would be an empty class *result* and an empty package *added*.

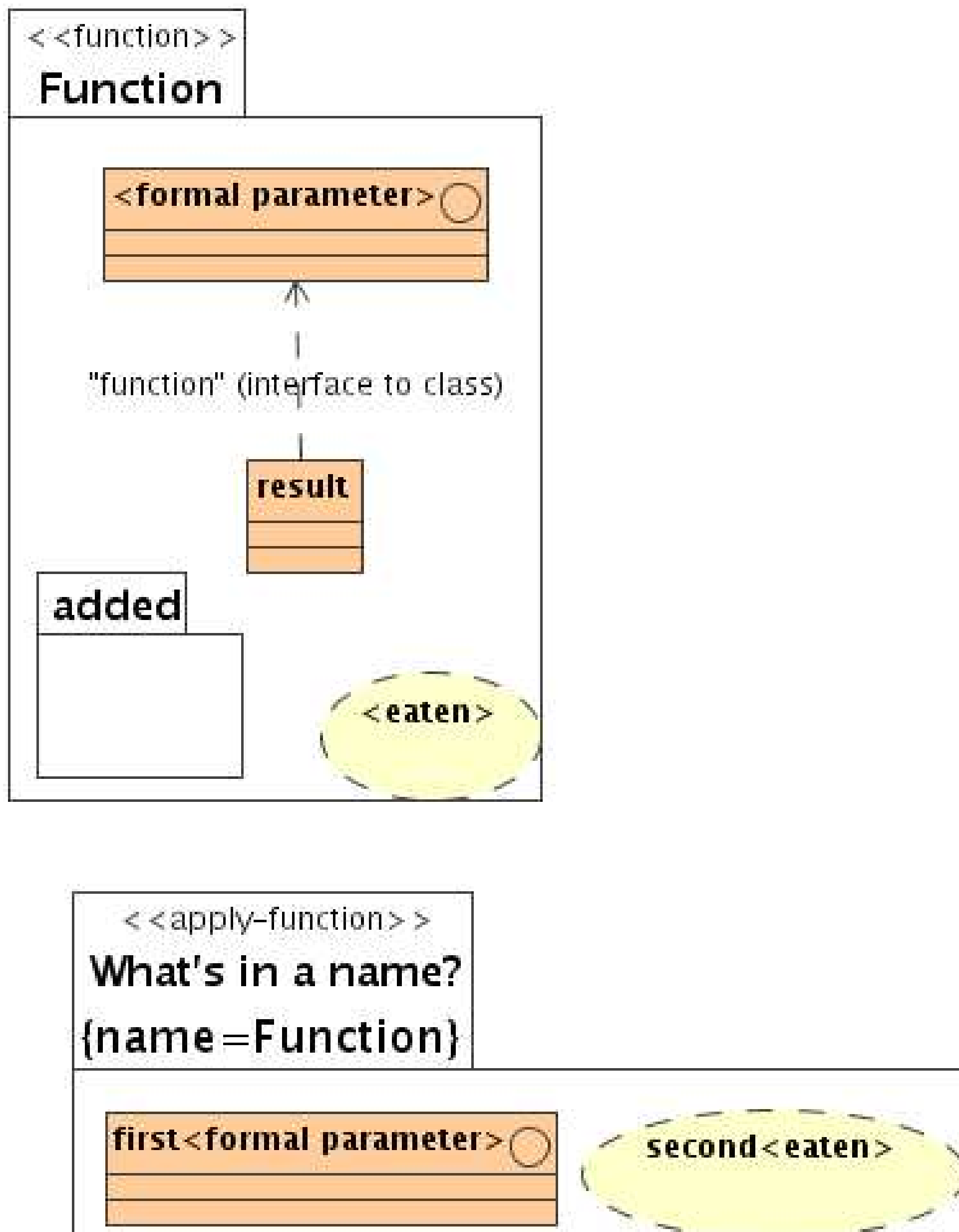

Figure A.6: Legend for Our Self-defined Functions

Functions can preserve parameter names by mentioning the name of the formal parameter, including the angular brackets, in the name of the resulting element. The name of the actual parameter can also be preserved partially by suffixing it with $[a..b]$. Here, a indicates the starting position within the name and b indicates the end position. A negative value indicates that the the position is to be counted backwards from the end.

Appendix B

Introduction to XML

This appendix gives a quick introduction to the eXtensible Meta Language (XML). It only describes the parts of XML which are relevant to the project. Refer to [19] for a complete description of XML. XML is used for both XMI (discussed in appendix C) and XSLT (discussed in appendix D).

XML is not a real language, but a method to denote objects. An example XML file follows:

```
<?xml version="1.0" encoding="iso-8859-15"?>
<main_object>
  <child_1/>
  <child_2 attribute="i_have_no_children"/>
  <child_3 attribute="me_neither">
  </child_3>
  <child-4>
    this text is part of child-4
  </child-4>
  <child-5 attr1="my first attribute" attr2="123.45"/>
  <child-6>
</main_object>
```

The first line is a processing instruction. It declares the XML version (either 1.0 or 1.1) and the encoding used in the rest of the document. Comments must be enclosed between $<!--$ and $-->$. White space is ignored unless it is part of a text node or a comment.

Each XML file contains one main object. This main object can have properties and child objects. Each object must be enclosed in angle brackets. Attributes are used to describe properties of objects and are always denoted as $attribute-name=$ ”$attribute-value$” (the quotes are compulsory). Both the quotes ” and ’ can be used as long as they are correctly paired.

Child elements can either be text nodes, comments, or other XML elements. An element can have any number of children. Child elements can also be nested arbitrarily deep.

Object *child_1* shows that an XML element can close itself if it has no child elements. Note that no semantic difference between $<child1/>$ and $<child1></child1>$ exists.

Certain characters in element and attribute names and in text nodes must be escaped using ampersand entities. These include & for &, < for $<$, and > for $>$. This escaping is necessary because it would otherwise confuse the XML parser.

Appendix C

Mapping UML Elements to XMI

This section gives a formal mapping from the UML elements used in the UML functions to XMI.

XMI (XML Metadata Interchange) is an XML-based language designed to represent UML models. Each model element has a unique identifier (*xmi.id* or *xmi.uuid*) which makes it possible to refer to other model elements via e.g. *xmi.idref* attributes. This way relations between model elements (like arrows and inclusion) can be represented. It is often used in XMI documents. Because XML does not support inheritance, XMI uses a copy-down mechanism to simulate inheritance.

Apart from UML, XMI can be used to represent any kind of metadata that can be expressed using MOF (see [13]). It can also include tool-specific data extensions. XMI also supports exchanging metadata in differential form, but this is not widely implemented by the tool developers. XMI supports general data types. This makes it possible to map data types from UML or another language (e.g. Java) to XMI.

Parametrized values of XMI elements are placed inside parentheses. For some parameters the generator always generates the same value. Optional attributes or elements are prefixed with a '*'. Referenced element parts are placed inside square brackets together with their appropriate parameters.

The XMI descriptions are based both on sample output from ArcStyler 5.1 and on [18]. Almost all elements have three flags set to *false*:

- *isRoot* denotes if this element is a root element. If so, it may not have ancestors.
- *isLeaf* denotes if this element is a leaf element. If so, it may not have descendants.
- *isAbstract* denotes if this element is abstract. Abstract element can only be indirectly instantiated, i.e. via a child element.

Other flags and settings are:

- *isActive* denotes if the class maintains it own thread of control. If not, it is controlled by the active object which controls the caller.
- *changeability* denotes the changeability of an attribute. This is normally set to changeable, but can also be frozen (no changes) or add-only (new values may be added to a multi-attribute).
- *concurrency* specifies the concurrency policy of operations and can be one of sequential (no concurrency allowed), guarded (one-by-one concurrency with blocking semantics), or concurrent (full concurrency).
- *isNavigable* denotes if this association end is navigable.
- *isQuery* specifies that this operation returns a value and has no side effects.
- *ordering* denotes if the element is ordered or unordered.
- *ownerScope* denotes if the owner is an ordinary instance (flag set to instance) or an classifier (flag set to classifier). Instance elements contain a reference to an instance of the target classifier. Classifier elements contain a reference to the target classifier itself.

- *targetScope* is similar to *ownerScope*.

Some common parametrized attributes are:

- *xmi.id*: this attribute provides the unique identifier of its element within this document.
- *name*: a string value denoting the name of its element.
- *owner*: the class identifier that owns this attribute or operation.
- *stereotype*: a string containing a reference to a stereotype definition. This string defines the stereotype of this element.
- *visibility*: the visibility of this element, one of *private*, *public*, *protected*, or *package*. The default visibility for attributes is *private*, the default visibility for classes, interfaces, and operations is *public*.

C.1 Package

Define a package with optional content, stereotypes, and tagged values.

```
<UML:Package
    xmi.id="(id)"
    name="(name)"
    isRoot="false"
    isLeaf="false"
    isAbstract="false"
   *stereotype="(stereotype-ref)">
   *<UML:Namespace.ownedElement>
   *   <!-- package content -->
   *</UML:Namespace.ownedElement>
   *[TaggedValue
        (tv-name)
        (id)
        (tv-id)
        (tv-data)]
</UML:Package>
```

C.2 Interface

Define an interface with optional stereotypes, attributes, and operations.

```
<UML:Interface
    xmi.id="(id)"
    name="(name)"
    visibility="(visibility)"
    isRoot="false"
    isLeaf="false"
    isAbstract="false"
   *stereotype="(stereotype-ref)">
   *<UML:Classifier.feature>
       *[Operation
             (operation-name)
             (id)]
       *[Attribute
```

```
            (attribute-name)
            (id)]
   *</UML:Classifier.feature>
</UML:Interface>
```

C.3 Class

Define a class with optional stereotype, attributes, operations, and tagged values.

```
<UML:Class
    xmi.id="(id)"
    name="(name)"
    visibility="(visibility)"
    isRoot="false"
    isLeaf="false"
    isAbstract="false"
    isActive="false"
   *stereotype="(stereotype-ref)">
   *<UML:Classifier.feature>
       *[Operation
            (operation-name)
            (id)]
       *[Attribute
            (attribute-name)
            (id)]
   *</UML:Classifier.feature>
   *[TaggedValue
        (tv-name)
        (id)
        (tv-id)
        (tv-data)]
</UML:Class>
```

C.4 Operation

Define an operation of a class or interface with optional stereotype, tagged values, and parameters.

```
<UML:Operation
    xmi.id="(id)"
    name="(name)"
    visibility="(visibility)"
    ownerScope="instance"
    isQuery="false"
    concurrency="sequential"
    isRoot="false"
    isLeaf="false"
    isAbstract="false"
    owner="(owner)"
   *stereotype="(stereotype-ref)">
   *<UML:BehavioralFeature.parameter>
   *    <UML:Parameter
           xmi.id="(param-id)"
           kind="(param-kind)"
```

```
           *name="(param-name)"
           *type="(param-type)"/>
   *</UML:BehavioralFeature.parameter>
   *[TaggedValue
        (tv-name)
        (id)
        (tv-id)
        (tv-data)]
</UML:Operation>
```

Parameters:

- *param − kind*: the kind of this parameter, one of *in*, *inout*, *out*, or *return*.
- *param − name*: the name of this parameter, absent if *param − kind* is set to *return*.
- *param − type*: a reference to the type of this parameter, only present if *param − kind* is set to *return*.

C.5 Attribute

Define an attribute of a class or interface with optional stereotype.

```
<UML:Attribute
    xmi.id="(id)"
    name="(name)"
    visibility="(visibility)"
    ownerScope="instance"
    changeability="changeable"
    targetScope="instance"
    ordering="unordered"
    owner="(owner)"
   *stereotype="(stereotype-ref)"/>
```

C.6 Object

Define an object with optional stereotype and tagged value.

```
<UML:Object
     xmi.id="(id)"
    *stereotype="(stereotype-ref)">
    *[TaggedValue
        (tv-name)
        (id)
        (tv-id)
        (tv-data)]
</UML:Object>
```

C.7 Association

Define an association between two elements.

```
<UML:Association
    xmi.id="(id)"
    isRoot="false"
```

```
    isLeaf="false"
    isAbstract="false">
    <UML:Association.connection>
        [AssociationEnd
            (aggregation1)
            (id)
            (participant1)]
        [AssociationEnd
            (aggregation2)
            (id)
            (participant2)]
    </UML:Association.connection>
</UML:Association>
```

C.8 AssociationEnd

Define an end of an association.

```
<UML:AssociationEnd
    xmi.id="(id)"
    visibility="private"
    isNavigable="true"
    ordering="unordered"
    aggregation="(aggregation)"
    targetScope="instance"
    changeability="changeable"
    association="(association)"
    participant="(participant)">
   *<UML:AssociationEnd.multiplicity>
   *    <UML:Multiplicity
   *        xmi.id="(id1)">
   *        <UML:Multiplicity.range>
   *            <UML:MultiplicityRange
   *                xmi.id="(id2)"
   *                lower="(lower)"
   *                upper="(upper)"/>
   *        </UML:Multiplicity.range>
   *    </UML:Multiplicity>
   *</UML:AssociationEnd.multiplicity>
</UML:AssociationEnd>
```

Parameters:

- *aggregation*: a string value denoting the aggregation kind of this association end, one of *none*, *aggregate*, or *composite*.
- *association*: the identifier of the parent association.
- *participant*: the identifier of the participant connected to this association end.
- *lower*: the optional lower bound of this multiplicity.
- *upper*: the optional upper bound of this multiplicity.

A value of -1 for *lower* or *upper* indicates infinity.

C.9 Dependency

```
<UML:Dependency
    xmi.id="(id)"
    client="(client-ref)"
    supplier="(supplier-ref)"/>
```

If the stereotype of this dependency is *use*, then the XMI element name is $< UML : Usage >$ instead of $< UML : Dependency >$. Parameters:

- $client - ref$: the unique identifier of the referred client of this dependency. This is the element at the tail of the arrow.
- $supplier - ref$: the unique identifier of the referred supplier of this dependency. this is the element at the head of the arrow.

The client element of the dependency has an attribute *clientDependency*, which has the value id.

C.10 UseCase

Define a use case bubble.

```
<UML:UseCase
    xmi.id="(id)"
    name="(name)"
    visibility="public"
    isRoot="false"
    isLeaf="false"
    isAbstract="false"/>
```

C.11 Actor

```
<UML:Actor
    xmi.id="(id)"
    visibility="public"
    isRoot="false"
    isLeaf="false"
    isAbstract="false"
    *name="(name)"/>
```

C.12 ClassifierRole

Define a classifier role in a sequence diagram.

```
<UML:ClassifierRole
    xmi.id="(id)"
    name="(name)"
    base="(base)"
    visibility="public"
    isRoot="false"
    isLeaf="false"
    isAbstract="false"/>
```

Parameters:

- *base*: refers to the identifier of the type of this classifier role (the part after the colon).

C.13 Message

Define a message in a sequence diagram.

```
<UML:Message
    xmi.id="(id)"
    sender="(sender-ref)"
    receiver="(receiver-ref)"
   *action="(action-ref)"
   *name="(name)"
   *activator="(activator-ref)"/>
```

Parameters:

- *sender*: the unique identifier of the sender of this message.
- *receiver*: the unique identifier of the receiver of this message.
- *action*: a reference to the type of this message. Message of type *uninterpreted* do not have this attribute.
- *activator*: a reference to a message of type *ReturnAction* (see below).

The referenced message type is one of *CallAction*, *DestroyAction*, or *ReturnAction*.

C.14 CallAction

This defines a call message.

```
<UML:CallAction
    xmi.id="(id)"
    isAsynchronous="false"/>
```

C.15 TaggedValue

This defines a tagged value of a stereotyped element.

```
<UML:ModelElement.taggedValue>
    <UML:TaggedValue
        *name="(name)"
        modelElement="(modelElement)"
        xmi.id="(id)"
        type="(tag-definition)">
        <UML:TaggedValue.dataValue>
            (data)
        </UML:TaggedValue.dataValue>
    </UML:TaggedValue>
</UML:ModelElement.taggedValue>
```

Parameters:

- *modelElement*: the unique identifier of the element for which this tagged value is defined.
- $tag-definition$: the unique identifier of the definition of this tagged value.
- *data*: a string which defines the contents of this tagged value.

Appendix D

Introduction to XSLT and XPath

This appendix gives a quick introduction to the XSLT and XPath language. Within the XSLT world, XPath is used as an embedded language to express queries on the input model. Refer to a book like [10] for a more thorough introduction and a complete reference.

XSLT is mainly used to convert XML documents to other XML documents (which is what TPUPT does), but it can be used to convert between arbitrary document types. XSLT programs are executed using an XSLT processor (e.g. Saxon). The processor takes an input document and the XSLT program and generates an output document.

The XSLT program, also called a script, is processed using rule evaluation. The script consists of a set of rules which are activated when the right conditions are met. Such a condition consists of the name of the current input element, an optional test on other content of the document which must hold for the current element, and the current mode of the XSLT processor. The other content is normally used as an additional requirement on the current element. Scripts can both read and set the mode of the processor. This mode can be used to repeat some processing, to conditionally process parts of the document, etcetera.

The rules in the script have the following structure:

```
<xsl:template match="N:E[C]" mode="M">
  <output elements / variable declarations / logic>
</xsl:template>
```

This structure consist of:

- N: the namespace of the input document. This namespace is ”UML” for UML documents.
- E: the element name for which the body of the rule must be activated. The element name can also contain the required ancestors by prepending them to the element, using the ’/’ symbol as separator: $A1/A2/E$ denotes an element E with parent $A2$ and grandparent $A1$.
- C: an XPath expression which refers to other content of the document, or children or attributes of the current element. This expression is optional. If it is empty, $N : E$ suffices as the match expression.
- M: the optional processor mode. If the mode is not specified, then the rule is evaluated in the default mode. The name of the mode is a constant string value.

The match expression can consist of multiple $N : E[C]$ parts separated by ’—’ symbols. These multiple parts form a range of alternatives for the match expression. This notation avoids the need to repeat the template body for each alternative.

When the template body is empty, then this rule cuts off processing of all child elements of the current element. After the current elements and optionally its child elements are processed, the next sibling of the current element is processed.

The most distinctive XSLT instruction is <xsl:apply-templates>. This instruction tells the processor to find an XSLT template for the specified selection S and switch to the specified mode M. The value for

S can be omitted, in which case the children of the current element are processed. This processing takes place in document order. The mode can be omitted, in which case the mode is reset to the default mode.

XSLT scripts can contain XPath function definitions, definitions for global variables and keys, and named templates. They can also import other XSLT scripts.

A named template is an XSLT template, but it is invoked using an <xsl:call-template> instruction instead of an <xsl:apply-templates> instruction. Therefore named templates do not have an attribute *match*, but an attribute *name*. A named template is used as a shorthand. Templates can also have both a *match* and a *name* attribute. Such templates can be invoked using both mechanisms.

XSLT templates are mostly used to construct new elements, while XPath functions are mostly used to query existing elements. XPath functions are denoted using the <xsl:function> element. This element calculates a value which is returned to the caller. This value is known as a result sequence. The expression used to calculate this value can be a pure XPath expression or the resulting value of an XSLT template invocation.

Key definitions are used to define XSLT keys. These keys are used as a syntactic sugar for looking up elements with certain attribute values.