Tampereen teknillinen yliopisto. Julkaisu 531
Tampere University of Technology. Publication 531

Petri Selonen

Model Processing Operations for the Unified Modeling Language

Thesis for the degree of Doctor of Technology to be presented with due permission for public examination and criticism in Tietotalo Building, Auditorium TB104, at Tampere University of Technology, on the 29th of April 2005, at 14.00 o´clock.

Tampereen teknillinen yliopisto - Tampere University of Technology
Tampere 2005

ISBN 952-15-1346-2
ISSN 1459-2045

Abstract

This thesis proposes a set of model processing operations for manipulating architecture and design level software engineering models. The approach draws from well-established and widely used software modeling paradigms like class diagrams, statecharts, and interaction diagrams. The operations are based on the usually implicit dependencies that exist between models describing the same system from different perspectives, at different levels of abstraction or at different phases of evolution. The Unified Modeling Language (UML), as a widely adopted industrial standard providing a common design vocabulary, is chosen as the target modeling language.

The thesis outlines categories for model processing operations and defines them based on the UML metamodel. The thesis also studies how to combine the operations to form high-level model processing tasks. The presented operation categories are transformation operations, set operations, projection operations, and conformance operations. The main targets for applying the operations are assumed to be merging, slicing, synthesis, and checking of models. The presented approach aims towards supporting incremental model development, a faster and easier creation of models, improved model consistency and comprehension, and a better customization of model processing tools.

The thesis gives example usage scenarios for applying the operations and shows how they can be exploited in practice in the maintenance of real-life product platform architecture. Further, it shows that the operations can be implemented, integrated with a computer aided software engineering environment, and succesfully used during software engineering. The tools and techniques have been implemented and deployed in industrial settings.

Preface

They call me Mr. Knowitall, I am so eloquent;
perfection is my middle name and whatever rhymes with 'eloquent'.

– Primus – "Mr. Knowitall"

This research was made possible by several colleagues I had the privilege to work with. First and foremost, I want to thank my supervisor, Professor Kai Koskimies, for all his guidance, support, and sense of humour throughout my research. I would also like to thank Professor Tarja Systä for providing a scientific shoulder to lean on, Jari Peltonen for his endless optimism, and Jianli Xu for providing excellent opportunities for industrial research collaboration.

Further, I would like to thank all the people who made this thesis possible: Jani Airaksinen for his endurance while implementing the tools; Jomppa Koskinen and Mika Siikarla for their work on the research platform development; Claudio Riva, Markku Sakkinen, Antti-Pekka Tuovinen and Yaojin Yang as my co-authors; Ilkka Haikala, Joni Helin, Ludwik Kuzniarz, Tommi Mikkonen, and Albert Zündorf for their useful comments on the thesis; Stephen King for proof reading the thesis; and Laph Roaig whose influence on this thesis cannot be overestimated. My sincere thanks are also due to all the other members of the PRACTISE team, past and present, for providing a pleasant working environment, and to all the other colleagues whom I have worked with during the research.

Finally, I would like to express my gratitude to my family and friends, those few who stayed loyal and supported me through the hard times. You know who you are. Thank you!

Contents

List of Included Publications

[I] P. Selonen, K. Koskimies, and M. Sakkinen. Transformations Between UML Diagrams. *Journal of Database Management*, 3(14):37–55, 2003.

[II] C. Riva, P. Selonen, T. Systä, and J. Xu. UML-based Reverse Engineering and Model Analysis Approaches for Software Architecture Maintenance. In *Proceedings of the International Conference on Software Maintenance (ICSM'04)*, pages 50–59. IEEE CS Press, September 2004.

[III] C. Riva, P. Selonen, T. Systä, A.-P. Tuovinen, J. Xu, and Y. Yang. Establishing a Software Architecting Environment. In *Proceedings of the 4th Working IEEE/IFIP Conference on Software Architecture (WICSA'04)*, pages 188–200. IEEE CS Press, June 2004.

[IV] P. Selonen and J. Xu. Validating UML Models Against Architectural Profiles. In *Proceedings of the 9th European Software Engineering Conference held jointly with 10th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE 2003)*, pages 58–67. ACM Press, 2003.

[V] J. Peltonen and P. Selonen. Processing UML Models with Visual Scripts. In *Proceedings of the 2001 Human-Centric Computing Languages and Environments (HCC'01)*, pages 264–271. IEEE CS Press, September 2001.

[VI] P. Selonen, T. Systä, and K. Koskimies. Generating Structured Implementation Schemes from UML Sequence Diagrams. In L. QiaYun, R. Riehle, G. Pour, and B. Meyer, editors, *Proceedings of the 39th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS USA 2001)*, pages 317–328. IEEE CS Press, July-August 2001.

[VII] J. Peltonen and P. Selonen An Approach and a Platform for Building UML Model Processing Tools. In *Proceedings of the ICSE'04 Workshop on Directions of Software Engineering Environments (WoDiSEE'04)*, pages 51–57, May 2004.

[VIII] P. Selonen. Set Operations for the Unified Modeling Language. In P. Kilpeläinen and N. Päivinen, editors, *Proceedings of the 8th Symposium on Programming Languages and Tools (SPLST'03)*, pages 70–81. University of Kuopio, June 2003.

Chapter 1

Introduction

> As the strategic value of software increases for many companies, the industry looks for techniques to automate the production of software. We look for techniques to improve quality and reduce cost and time-to-market ... We also seek for techniques to manage the complexity of systems as they increase in scope and scale. In particular, we recognize the need to solve recurring architectural problems ... One of the key motivations ... was to ... adequately address all scales of architectural complexity, across all domains.
>
> – The Unified Modeling Language Specification version 1.5

> ... and when the Ball game world within one's own head has no contact with the outside world, this subjective vision must be respected until someone proves it untrue.
>
> – R. Nelimarkka-Seeck, *Self Portrait*, PhD thesis, Helsinki University of Arts and Design, 2000

1.1 Motivation

Typical software systems of today are inherently large and complex. The importance of modeling is well recognized among the practitioners and addressed by a number of modeling languages and methods available. While models are becoming the first-class citizens of software engineering, the available modeling tools have not developed at the same pace, or reached the same level of maturity as, say, programming tools. The currently available computer aided software engineering tools—CASE tools—are mostly visual

editors that let the user draw diagrams. At best, the tools maintain a model repository and support code generation and round-trip engineering to some degree. Although there is no common agreement on the characteristics environment, there obviously exists a gap between the needs of software engineers and the available tool support. There is a need for adequate high-level tool support for the various modeling tasks occurring during conventional software engineering.

Such tool support can be based on, for example, the dependencies between different models describing the same system. As models can be used to observe the system from different perspectives, at different levels of abstraction, and at different stages of a design process, they typically share information. For instance, models produced at successive phases of a software development process become dependent on each other. Similarly, during the lifetime of a system it may undergo several modifications, implying dependencies between the original models and the modified ones. In addition, models describing members of the same product family often have mutual dependencies.

At the time of writing, the ability of the CASE tools, both academic and commercial, to exploit the dependencies is quite modest. The lack of tool support leads to a situation where new models are created manually. The dependencies between the existing models have to be taken into account by the designer, which requires tedious and error-prone maintenance work and increases the probability of introducing inconsistencies. As the number of implicit dependencies between the models increase, the designer also has to produce new partial models to maintain comprehension of the system model.

1.2 Approach of the Thesis

To address the abovementioned problems, this thesis proposes a set of *model processing operations* for manipulating architecture and design level software engineering models. The approach draws from well-established and widely used software specification paradigms like class diagrams, statecharts, and interaction diagrams. The operations are built exploiting the dependencies between models and they are combined to form *model processing tasks* of higher level of functionality to perform various software engineering tasks. It is assumed that most of the tasks occurring during conventional software engineering can be built as such model processing tasks. The presented research futher aims to provide tool support for automating the tasks. Instead of creating several high-level tools for dedicated purposes, most model manipulation tasks occurring during conventional software development are expected to be built as combinations of the model processing operations.

The presented operation categories are *conformance operations*, *set operations*, *transformation operations*, and *projection operations*. Conformance operations are used together with *profiles*, a special model type stating the legality criterion, to define and enforce domain-specific constraints and conventions. Set operations produce a new model based on union, intersection, or difference of existing models. Transformation operations produce a new model based on an existing model of another type. Projection operations produce a new model based on an existing model of the same type. The operations categories are discussed in more detail in Chapter 2. The main usage scenarios are expected to be the *synthesis*, *merging*, *slicing*, and *checking* of models. The thesis questions, presented later in this chapter, are drawn along these scenarios. The presented approach aims at supporting incremental model development, faster and easier creation of models, improved model consistency and comprehension, and better customizability of model processing tools.

To establish a common defininition for the operations using a concrete modeling language, this language is expected to meet the following prerequisites: 1) it must have a well-defined metamodel, including an abstract syntax and a set of well-formedness rules, and 2) it must have sufficiently defined semantics to map its modeling concepts to the concepts of the adopted paradigms. In the context of this thesis, the Unified Modeling Language (UML) version 1.5 [38] is chosen as the target modeling language. UML has become an industrial standard for the presentation of various design artifacts in object-oriented software development. It provides different diagram types supporting the development process from requirements specification to implementation. Essentially, UML is not a single modeling language but a set of design languages, each represented by a particular diagram type, and its metamodel is structured accordingly.

The terminology used in this thesis has been aligned with the one provided by the UML specification. A model is defined as a description of a system, given in terms of the abstract syntax of the selected modeling language (e.g. a UML model is a UML metamodel instance). A diagram is a visualization of a model, emphasizing a particular modeling paradigm. The type of model or diagram implies the used metamodel subset. Further, the terms model fragment and system model are used when it is necessary to distinguish between an incomplete model and a complete model describing a system at a given point in time. This division is not absolute, however, as the models are always tied to a particular context.

While a model may be presented with an arbitrary number of diagrams, a diagram is assumed to imply a particular model fragment. Each diagram type has specific semantics and notation, and provides a mapping between

notational elements and metamodel elements. As the presented model processing operations are defined in terms of model element types, the terms model and diagram can be used interchangeably in the context of this thesis unless otherwise stated. Nevertheless, the motivation and rationale behind the presented approach lies heavily in the role of diagrams as a means for communicating software specifications between designers. The relationship between models and diagrams in the UML context is described in Chapter 2.

1.3 Thesis Questions

To elaborate the approach, a set of example usage scenarios occurring during software engineering work are given as thesis questions. The questions are divided along the lines of synthesis, merging and slicing, and checking of models. The hypothesis is that the the presented model processing approach can be used to address the questions.

Synthesis of models is used for producing new models on the basis of existing models. In the context of this thesis, synthesis is used for both generating projections of existing models and for transforming models of one type into models of another type. The questions to be answered include the following:

- How to describe the information implied by an existing model using another modeling paradigm to express a different point of view (e.g. a structural model implied by a behavioral model)?
- How to compose a model according to a given composition criteria in order to emphasize an alternative point of view (e.g. architectural concern)?

Merging and slicing of models is used for composing and decomposing models. The former is used for adding the information contained by a model fragment to another model fragment or a system model. The main motivation for merging of models stems from the needs of incremental and iterative model development. Model slicing is used for creating partial views of models, focusing on a particular viewpoint, possibly presented by other models of different types. It is a useful technique for emphasizing certain aspects of the model while suppressing others to improve model comprehension. The slicing criterion is often presented by another model— sometimes of a different type

than the model to be sliced. The questions to be answered include the following:

- How to allow different stakeholders to introduce model increments to the system model?
- How to support model comprehension by comparing models describing the system from different perspectives, possibly using different modeling paradigms (e.g. static and dynamic views)?

Checking of models is used for examining if two models are in agreement with each other. In this context it focuses on providing the user with means for checking individual models for mutual consistency, and for conformance to domain, product platform, or project specific architecture and design level constraints and conventions. The questions to be answered include the following:

- How to ensure that the concepts and their relationships in a system model are in agreement with domain specific conventions, rules, and restrictions?
- How to confirm that architecture or design level models at (e.g. different stages of evolution) are in agreement with each other?

The questions will be put in the context of a case study on establishing an architecture maintenance process and returned to while drawing the conclusions in Chapter 8.

1.4 Thesis Contributions

This thesis presents an approach to define and implement a set of primitive model processing operations based on a metamodel describing the selected modeling language. It is shown that these operations can be combined to form meaningful high-level software engineering tasks and that these tasks can be exploited during real-life software engineering. It is further shown that the model processing operations can be used as a basis of tool support in computer aided software engineering environments. The approach also supports customizing and parametrizing the tasks for different domains. The thesis outlines categories for the model processing operations and specifies them in terms of the UML metamodel. It presents a novel approach to

using the UML extension mechanism to express and enforce domain-specific modeling conventions, constraints, and rules. The thesis gives example usage scenarios for applying the operations and shows how they can be exploited in practice in the maintenance of a real-life product platform architecture.

1.5 Context of the Thesis

The research work presented in this thesis has been carried out in the PRACTISE research group[1] at the Institute of Software Systems of Tampere University of Technology. The research started with the ATOS (Advanced Tools for Object-oriented Software Development) project during 1999–2002. The project was funded by the National Technology Agency of Finland (TEKES grant 40908/98) together with Nokia, Metso Automation, Sensor Software Consulting, Ebsolut and Plenware. The main areas of research were UML-integrated visual scripting mechanisms, UML model transformation, abstraction, synthesis, and checking techniques, and reverse engineering techniques producing UML models. In particular, the project investigated new kinds of automated tool support that could be integrated with CASE tools.

The research work was continued with the UML++ project (Techniques for UML Based Software Development) during 2001–2004, funded by the Academy of Finland. The project aimed at investigating new techniques facilitating and exploiting the use of UML in software development. The work particularly concentrated on automated model synthesis in UML, analyzing software systems with UML, and developing infrastructures for UML-based model processing. The project studied techniques to support both forward and reverse engineering and approaches to analyze and understand software systems using UML. The project was carried out in co-operation with the University of Helsinki and the University of Tampere.

The research was further continued with ART (UML Tool Kit for Software Architecture Modeling and Analysis Introduction) financed by, and carried out in close co-operation with, the Nokia Research Center in Helsinki during 2002–2004. The project aimed at developing general infrastructure for UML based software architecture modeling to be used in a prototype environment for modeling, analyzing, communicating, documenting, maintaining, and monitoring of architecture design artifacts. During the course of the project, an additional goal was set to apply the techniques and tools on reverse engineered software architecture models provided by the customer and to use during case studies to validate the ART approach.

[1] http://practise.cs.tut.fi

1.6 Organization of the Thesis

The introductory part of this thesis is organized as follows. Chapter 2 discusses model processing and UML in general, outlines the operation categories and discusses how the model processing operations can be used to compose high-level model processing tasks. Chapter 3 outlines specification techniques and gives example definitions for each operation category. In Chapter 4, the implementation of the operations is discussed, along with the implementation platform, and other related issues. Chapter 5 describes how the operations are used in a concrete case study. Related research is presented in Chapter 6. The included publications are introduced in Chapter 7. Finally, thesis questions are revisited with concluding remarks in Chapter 8.

Chapter 2

Model Processing and UML

This chapter discusses how the relationships between different modeling paradigms, and dependencies between model fragments, are used to define the model processing operations and how they are combined to form model processing tasks. UML and its role as the modeling language of choice is discussed in the second half of the chapter.

2.1 Model Processing Operation Categories

Models describing the same system typically share information and are mutually dependent. Following this rationale, the following types of dependencies can be identified, among others (Koskinen *et al* [29]):

- *view dependencies* occur between the logically related elements in different model fragments of the same system model;
- *process dependencies* occur between model fragments produced at different phases of a software development process;
- *evolution dependencies* occur between model fragments produced at different stages of system evolution; and
- *family dependencies* occur between model fragments describing members of a software product family sharing similar structure and functionality.

The dependencies are present and strongly intertwined in software development processes, and they vary between different organizations, processes, and domains. Figure 2.1 illustrates the different kinds of dependencies. Typically, when moving from one phase to the next, new design or implementation

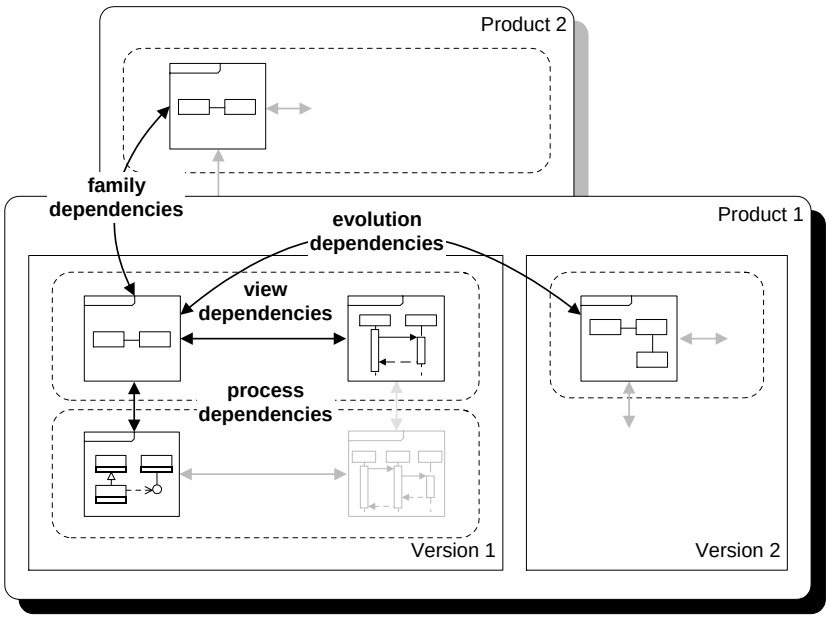

Figure 2.1: Examples of dependency types

level information is added to the models, yielding processwise dependent diagrams. The added information is then taken into account in the other models constructed in that phase, yielding view dependent diagrams. These diagrams are modified during maintenance, leading to evolutional dependencies. The presented set of dependency types is not exhaustive; rather, it provides a useful vechile for reasoning with the usage scenarios for different operation types.

Figure 2.2 illustrates a set of view dependencies between different diagrams describing a simplified car rental system. The figure shows four types of diagrams. The class diagram shows a `Customer` and a `RentalCenter` managing a set of `Vehicle` entities (i.e. a `Car` and `Truck`). The interaction diagram shows an unnamed instance of a `Customer` renting a car from `RentalCenter`. The `RentalCenter` sets a `Car` instance reserved and returns it to the `Customer`. The component diagram shows components `RentalCenter` and `Customer` with an interface `Customer Services`. The statechart diagram shows three states for `Car`. The dependent parts among the diagrams are shown in black while the independent parts are in gray.

To exploit the dependencies, four major categories of model processing operations are introduced together with characterizing signatures. The operands are given in form D^t where D is a diagram of type t. P denotes a profile which is a special diagram type stating legality criterion. A profile can be

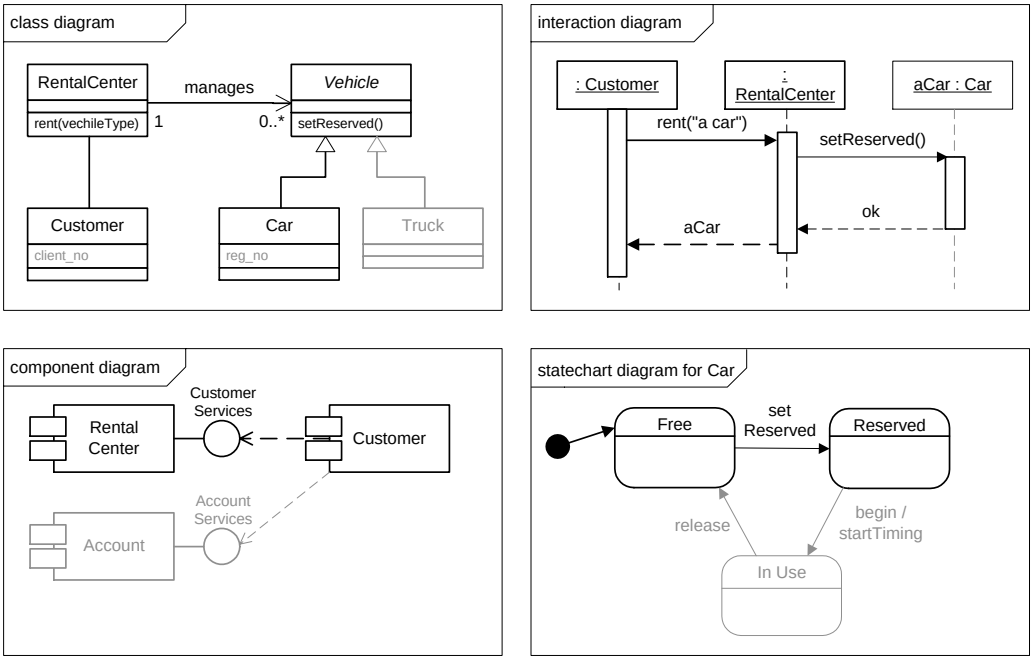

Figure 2.2: Examples of dependencies between diagrams

seen to represent metalevel information that a model must be in agreement with. The term is borrowed from the UML vocabulary. Profile variants are described in more detail later in this chapter.

For simplicity, the operations are either presented as unary or binary. The connectivity properties of the operations allow them to be combined to form more complex expressions. The presented model processing approach relies heavily on the ability to build arbitrary complex functionality out of primitive operations. In cases where a single diagram is not a sufficient starting point, the input diagram can be composed of several diagrams of the same type using other model processing operations, or the operations can be applied in an incremental manner.

Conformance operations ($\mathbf{P}\times\mathbf{D}^t\longrightarrow\mathbf{Boolean}$) offer a way to validate whether a given model *conforms* to a profile. P is a profile describing the legality criterion and D^t is the model to be checked. The result is a boolean value indicating whether or not the model is in agreement with the profile. An illustration of a profile and a conforming model is shown in Figure 2.3. A conformance operation gives the interpretation for this profile. Typically, the profile explicitly states the allowed structure the model must be analogous with. In the example, the profile can be interpreted to state the

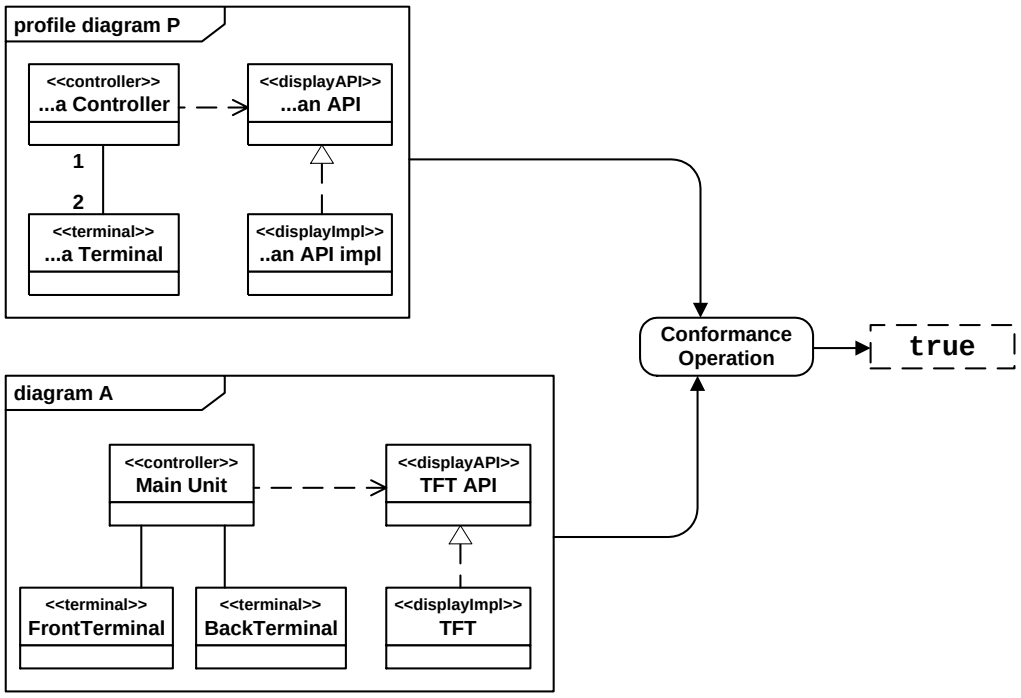

Figure 2.3: Example of a conformance operation

allowed relationships between the concepts. The particular concepts of interest are marked with matching guillemets: ≪`controller`≫, ≪`terminal`≫, ≪`displayAPI`≫, and ≪`displayImpl`≫. The profile further states that a ≪`controller`≫ must have exactly two associations to different ≪`terminal`≫ elements, a ≪`controller`≫ can have a dependency to a ≪`displayAPI`≫ element, and a ≪`displayImpl`≫ can realize the interface ≪`displayAPI`≫. In this particular example, the relationships between the model elements are family dependencies. The profiles state conventions and constraints on the way the elements are to be used in the context of e.g. a given domain, product family, or design style.

Set operations ($\mathbf{D}^t \times \mathbf{D}^t \longrightarrow \mathbf{D}^t$) generate a new model based on two existing models of a particular type. The operation assumes there is a way of deriving a *correspondence* relationship between the elements belonging to the input models representing the same semantic concept. The set operations described in this thesis are variants of union, intersection, and difference. Typically, a union can be used for merging models, intersection for detecting the commonalities of models, and difference for detecting the differences of models. The operations are expected to provide support for analyzing the

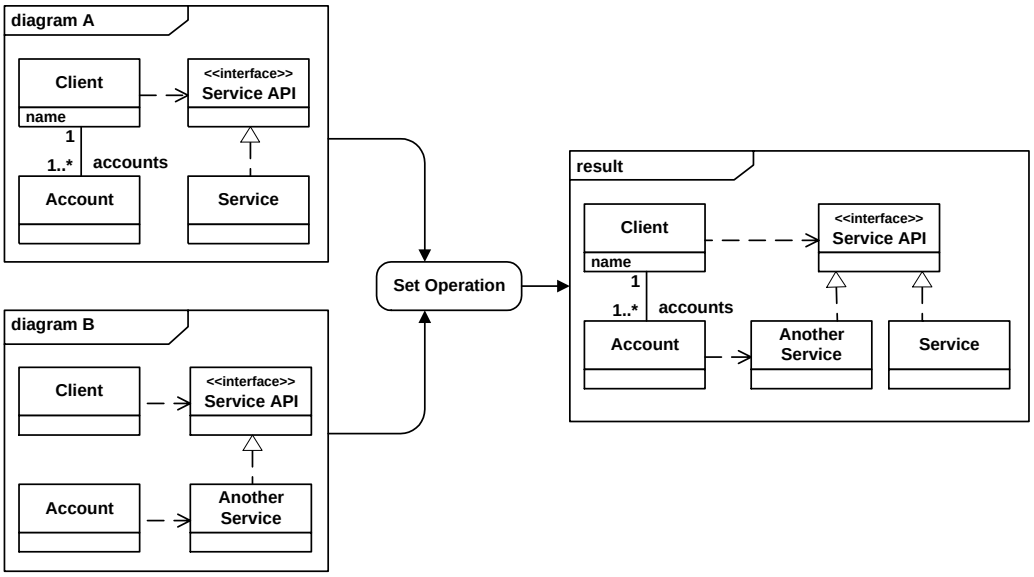

Figure 2.4: Example of a set operation

models, for improving model comprehension, and for incremental and iterative software development. They are also a key mechanism when combining the model processing operations.

An illustration of applying a set operation is given in Figure 2.4. The left-hand side of the figure shows two models describing a system for different stakeholders. Assuming sound engineering practices, the elements with the same name should denote the same semantic concept (e.g. `Client`, `Service API`, `Account`). By exploiting this information, the model elements assumed to represent the same semantic concepts can be merged. The result of a union operation is shown on the right-hand side of the figure.

The types of dependencies present in the figure can be view dependencies, but they may also result from the software engineering process used, hence being process or evolution dependencies. If the two input models represent the design of two different products of a common product family, they can also be classified as family dependencies.

Transformation operations ($\mathbf{D^s \longrightarrow D^t}$, $\mathbf{s \neq t}$) offer a way of synthesizing a new model based on an existing model of another type. One example of a transformation operation is the synthesis of a statechart model from an interaction model where partial sequences are merged to form a behavioral specification. Another example is the synthesis of a class model from an interaction model.

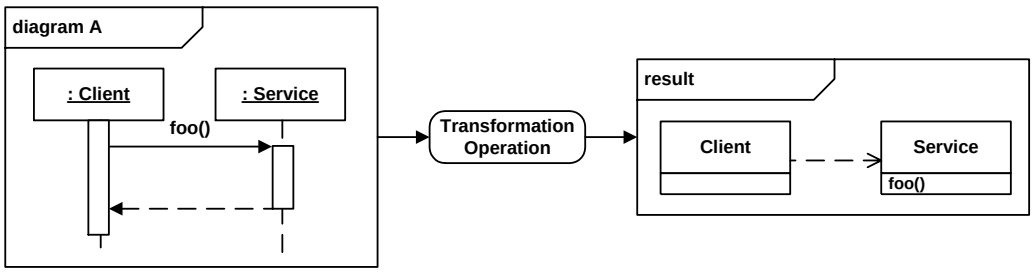

Figure 2.5: Example of a transformation operation

An illustration of a transformation operation is given in Figure 2.5. The source model, shown on the left-hand side of the figure, describes a simple interaction model with two unnamed participanting instances of (`Client` and `Server`), one message with operation call `foo()`, and one return message. The right-hand side of the figure shows the resulting structure model as implied by the source model. It contains a class specification for `Client` and `Server` together with an operation specification for `foo()`. In addition, to facilitate the communication between the classifier instances, a dependency is shown between the two classes.

The dependencies in Figure 2.5 can be view dependencies. The transformation operation may also be used to perform tasks related to the software engineering process used, thus implying process or evolution dependencies. For example, a set of sequence diagrams, describing the realization of a use case, can act as a starting point for a design level class diagram.

Projection operations ($\mathbf{D}^t \longrightarrow \mathbf{D}^t$) offer a way of generating a new model of an existing model of the same type. An illustration of a projection operation is given in Figure 2.6. The left-hand side of the figure shows the source structure model for the operation. The model has four classes, one composition association, one dependency, and one generalization (inheritance). The right-hand side of the figure shows the resulting projection of the source model. In the result, the whole-part relationship between classes `Main Unit` and `Controller` is collapsed, as is the inheritance hierarchy of classes `Display` and `TFT monitor`. The original `controls` dependency between classes `Controller` and `TFT monitor` remains intact, but now it is between the compressed classes `Main Unit` and `Display`. The example projection operation effectively implements abstraction of the class diagram using a relationship–based compression.

Again, the dependencies between the source and target models can be view dependencies. Dependending on the software engineering process used,

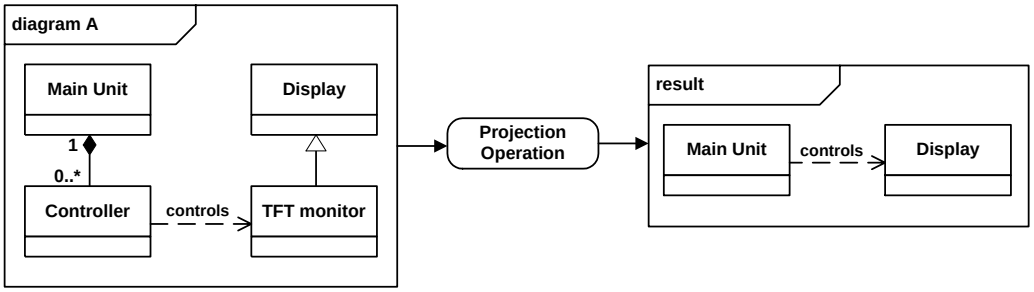

Figure 2.6: Example of a projection operation

the projection operation can be a refactoring operation, or perhaps the appliance of, say, a design pattern. In such a case, the projection operation implies process or evolution dependencies between the source and the target models.

The model processing approach provides support for typical incremental software engineering processes: rather than producing one complete model, it allows the use of individual, although conceptually overlapping, model fragments. A simple example of combining model processing operations to form higher level model processing tasks is shown in Figure 2.7. The figure illustrates how the structure implied by an interaction diagram is added to an existing structure diagram: the sequence diagram is transformed into a class diagram and the resulting class diagram is merged with the original class diagram. The presented task can be used, for example, to support incremental model development where the structure implied by the sequence diagram is merged to the existing structure model presented by the class diagram. By highlighting the common and individual parts of the original diagrams on the resulting diagram, the task can be used to support model comprehension.

2.2 UML Metamodel Architecture

UML is a "graphical language for visualizing, specifying, constructing and documenting the artifacts of a software-intensive system" (OMG [38]). UML models are presented, constructed, and manipulated as diagrams that can view a system from different perspectives, making them mutually dependent and overlapping. UML version 1.5 presents nine different diagram types, each emphasizing a particular concern: class diagrams, sequence diagrams, collab-

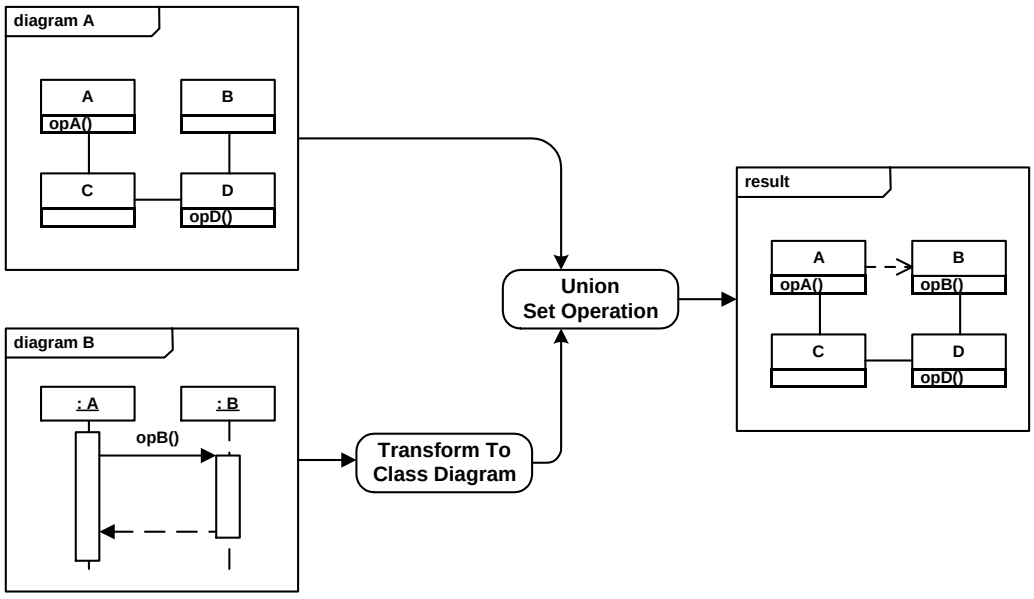

Figure 2.7: Example of combining model processing operations

oration diagrams, statechart diagrams, activity diagrams, object diagrams, component diagrams, deployment diagrams, and use case diagrams.

The model processing approach is, in principle, independent of the software engineering process used and the details of the particular software modeling language. However, UML brings different specification paradigms together by adopting them as its sublanguages, each represented by a particular diagram type, and thus provides a common platform for defining and implementing the operations in practice. Consequently, UML is a key enabling factor for the approach presented in this thesis. UML is widely adopted by the software industry and UML-based modeling is supported by a number of CASE tools. Providing a common design vocabulary with a standardized metamodel is one of UML's most important contributions to software engineering practices.

Models are, first and foremost, used for communication. One way to facilitate a common understanding of models is to use metamodels. Metamodels are models that describe other models: the allowed metaelements, their relationships and properties. UML is defined by its metamodel, including an abstract syntax, well-formedness rules, informal semantics, and notation specification. The abstract syntax is given as a MOF model, expressed using the UML class diagram notation. The well-formedness rules are expressed using the Object Constraint Language (OCL) (OMG [38] Chap. 6).

UML builds on the OMG Meta-Object Facility (MOF) [35], a metadata

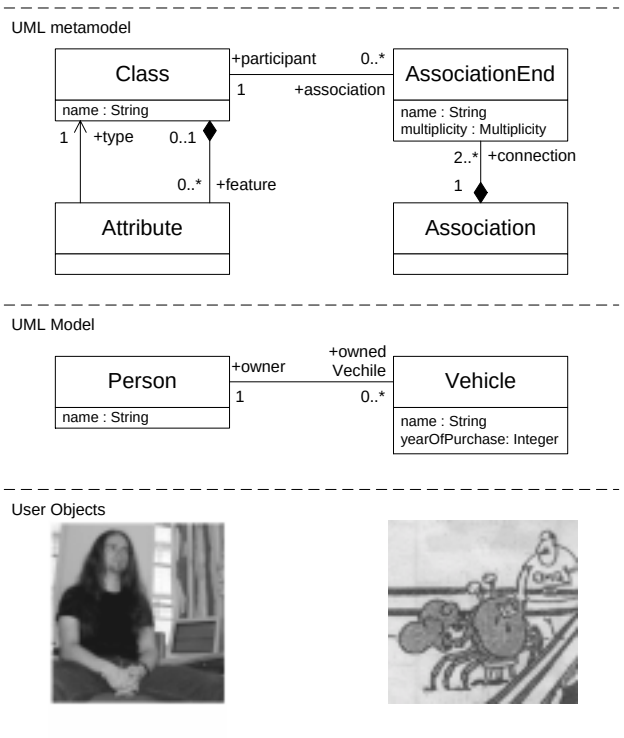

Figure 2.8: Illustration of the UML metamodeling architecture

architecture designed to support the construction of metamodels. MOF is a meta-metamodel for defining metamodels for various domains and modeling languages. It is a self-describing, object-oriented metamodeling framework aligned with the UML class diagram constructs. The architecture of UML is based on a four-layer metamodel structure comprising of the following layers: user objects, models, a metamodel (a MOF model) and a meta-metamodel (MOF metamodel). The meta-metamodel layer is the infrastructure for a metamodeling architecture and it defines the language for specifying metamodels. The metamodel layer is an instance of the meta-metamodel and defines the language for specifying a model. An example of the UML architecture is shown in Figure 2.8, the meta-metamodel is omitted for simplicity. The metamodel excerpt shows metaclasses connected by meta-associations: a `Class` can contain `Attributes` and participate in `Associations` through `AssociationEnds`. An example UML model is shown in the middle of the figure, showing classes `Person` and `Vehicle`. Finally, some user objects are shown in the bottom of the figure.

While the abstract syntax and well-formedness rules are formally defined, UML semantics are given in natural language. This is both a factor behind

UML's rapid adoptation by the software industry and also the reason why UML models suffer from being ambiguous. Efforts have made to formalize parts of the current UML metamodel semantics (e.g. the precise UML group[1], Evans and Kent [16]). However, the contradictory requirements for precision and generality seem to restrict these attempts to very limited domains.

Although UML offers a rich variety of modeling constructs, it is sometimes desirable to introduce new domain-specific extensions to the modeling language. The Extension Mechanisms package specifies how UML model elements are customized and extended with new semantics using *stereotypes*, *constraints*, *tag definitions*, and *tagged values*. A coherent set of extensions constitutes a *UML profile* (OMG [38], Sec. 2.6). A stereotype effectively specializes a metaclass and extends its semantics for a particular purpose (see e.g. OMG [38] Chap. 4). The extensions must be strictly additive to the standard UML semantics, i.e., they are a means for specializing UML. The UML profiles can be seen as a UML-specific implementation of the general profile concept introduced in Section 2.1.

UML profiles are a lightweight built-in extension mechanism for UML. Although it is outside the scope and intent of UML specification, it is also possible to extend UML metamodel by explicitly adding new metaclasses and other metaconstructs. However, such a heavyweight mechanism loses the major asset of having a common and standardized metamodel. In the context of this thesis, only profiles are used.

2.3 UML and Architectural Profiles

This thesis works with architecture and design level UML models. While UML has established itself as a software design language, it has significant problems when documenting software architectures as pointed out by e.g. Ivers *et al* [20], Medvidovic *et al* [30], and Riva *et al* [47]. The problems are related not only to the inadequacies of UML modeling concepts when representing architectures, but also to the lack of methodological support for UML-level architecture design. The latter can be addressed with *architectural profiles* [IV], another variant of the profile concept previously described in this chapter.

Architectural profiles concretize the work context of software architects. They are used for defining domain-specific architectural constraints and conventions. They define the structural constraints and rules of the domain (e.g. product platform) in question and are used to drive, check, and automate the software architecture design process and the creation of architec-

[1] http://www.cs.york.ac.uk/puml/

tural and design views. They should be followed in order to guarantee that resulting architecture design has necessary properties and lacks any undesirable ones. It is argued that they are on an appropriate abstraction level to express architectural constraints.

Figure 2.9 shows the architectural model structure used in this work, adopted and developed from the conceptual, module interconnection, execution, and code view model of Soni, Nord and Hofmeister [54]. It is comprised of three main parts: architectural profiles, a *system context model*, and *architectural views*. Architectural profiles contain a *conceptual profile* and a set of *view profiles*. The conceptual profile should specify the architectural style and the validation rules. It defines the nature of the system, as all architectural views should conform to it. The domain model is included for the sake of model completeness; how the mapping is done is up to the particular process used and outside the scope of this thesis. A view profile adds new concepts that are specific to a particular view and emphasizes a particular concern or viewpoint. Conceptual and view profiles can be further divided into *stereotype definition profiles* and *constraint definition profiles*. The former define the concepts relevant to the particular viewpoint in the form of stereotypes, while the latter constrain how the stereotypes can be used.

Concrete software architecture is described with a system context model and several architectural views. The former defines the system border and deployment environment, typically showing how the system or subsystem connects to other (sub)systems through interfaces it implements or depends on. The latter contain all UML models describing the software system itself. A view should use only the subset of UML elements and extensions defined in the corresponding view profiles, and follow the rules and constraints they impose. What views are needed in the system description may vary from case to case, depending on what main architectural concerns the architects are going to tackle.

The profile-based approach for architecture validation is applied in the case study presented in Chapter 5. Conformance operations are the main vechile for realizing the approach.

2.4 Relationships between UML Diagram Types

Most widely adopted general-purpose software modeling languages preceding UML were presented in the context of respective design methodologies like OMT [48], the Booch Method [6], OOSE [21], and Structured Analysis [61], to name a few. Each design methodology introduced its own notation, relying

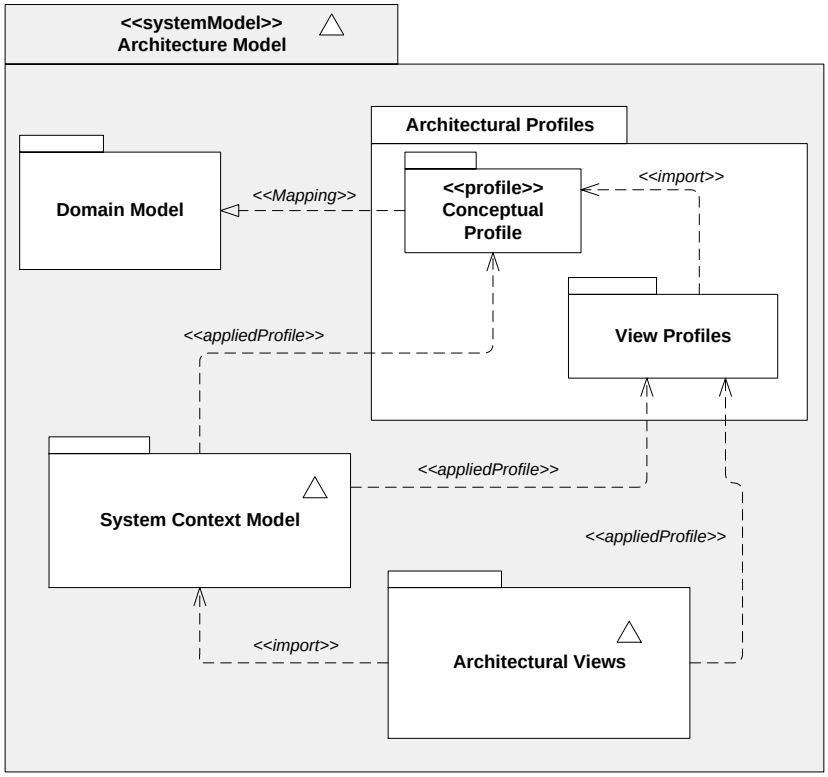

Figure 2.9: The structure of an architectural model ([IV], Fig. 1)

solely on giving example figures on how to draw the diagrams. UML brings the different modeling languages together and aims to provide a common, although loose, metamodel level representation for the diagrams. This facilitates the exchange of information between tools and the writing of generic model processing operations.

It is often assumed that there exists a single, complete, and well-formed system model expressed in UML. While appealing, this is very rarely the case in practice. The individual diagrams are often independent specifications, although they are conceptually describing the same elements. To make matters worse, although UML is a visual language by definition, the Notation Guide ([38], part 3) does not provide a well-defined mapping between UML diagrams and UML models or model fragments. Given that UML is a visual language meant for communicating, it is surprising that the relationship between models and diagrams have been omitted as a technical question to be addressed by tool vendors. This has also led to serious compatibility issues between different CASE tools.

As described previously in this chapter, a transformation operation aims

at representing the modeling information of its source diagram with the modeling concepts of its target diagram. To support defining the operations, the transformation operations can be classified as *full transformations*, *strong transformations*, *supported transformations*, and *weak transformations* [I]. The categorization is based on the information content an operation preserves, relative to the expressiveness of the target diagram. The categories are implied by the varying strength of relations between different pairs of diagram types: the weaker the relation, the more user interaction and guidance is required to make the operation useful.

Full transformations convey the information present in the source diagram adequately to the target diagram. These diagram pairs share a large common metamodel subset, making the interpretation of the transformation an identity relationship. In UML, there are two semantically close diagram pairs: sequence and collaboration diagrams, and statechart and activity diagrams. For example, while a collaboration diagram emphasizes the distribution of objects, the interaction itself can be equally expressed using a sequence diagram; both diagram types are called interaction diagrams in UML.

Strong transformations are based on semantic relationships between diagram types having different metamodels. Used together with a set of heuristic rules, a significant amount of information from the source diagram can be conveyed. For example, the behavior described by a sequence diagram also implies structure, which can be expressed using a class diagram [VI].

Supported transformations are based on conventions and user interaction. For example, a class diagram can be transformed to a component diagram. Without additional guidance from the user, supported transformations generally become weak.

Weak transformations typically produce only a diagram skeleton as a starting point for design. Categorization for the operations is given in Figure 2.10, together with the abbreviations for different diagram types. The figure emphasizes the central role of sequence diagrams, statechart diagrams, and class diagrams.

The given categories for the transformations are first and foremost used for suggesting the appealing transformations. Strong transformations are considered to be the most interesting category of transformation operations.

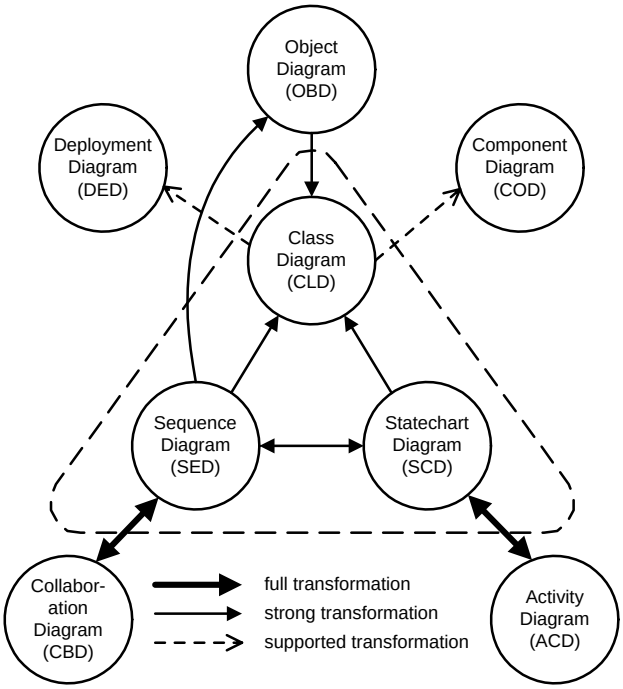

Figure 2.10: Categorization for the transformation operations ([I], Fig. 5)

Figure 2.10 displays, in total, six strong transformations. The figure underlines the fact that behavioral diagrams contain more information than structural diagrams.

2.5 Notes on UML 2.0

While UML 1.5 [38] is the latest official UML specification issued by the OMG, the next major revision, UML 2.0 [39], has reached its finalization phase and is expected to be released during 2005. In particular, it will introduce major changes to the structure of the UML specification. It will align UML with the other OMG standards and address issues not present in UML 1.5. Moreover, it will introduce a set of new diagram types, make some other types obsolete, and present a completely renewed metamodel specification. In particular, the activity diagrams of UML 2.0 are based on Petri-net like semantics, and the sequence diagram notation has been strenghtened with new new features and improved notation.

While UML 2.0 will significantly differ from UML 1.5, the underlying principles will remain relatively stable. As stated by the OMG UML 2.0 Superstructure Request for Proposal document ([34], Sec. 6.3), the new spe-

cification should maintain backward-compatibility with UML 1.x specifications. Further, most modeling techniques introduced by UML were well-established before UML emerged. This follows directly from the nature of UML: it aims at providing a unified modeling language that offers a common base for expressing the modeling artifacts present in software engineering. Consequently, while the technical definitions given in this thesis will change in some parts, the underlying principles will not. Of the presented model processing operation categories, the changes will mainly affect the transformation operations, also opening up new possibilities for stronger tool support.

Similarly, OMG will issue a major revision of the Meta Object Facility, MOF 2.0 [37]. As the specification states (Chap. 16) that "MOF 1.4 metamodels can be translated to MOF 2.0 models based on a straigthforward mapping that can be fully automated", the effects of the migration to the approach presented in this thesis are expected to remain negligable.

Chapter 3

Example Operation Definitions

This chapter demonstrates how to define the model processing operations by giving example definitions for each operation category. The sections describing conformance operations, set operations and transformation operations are extended from the included publications [IV], [VIII] and [I], respectively. The operations that are presented in more detail have been used in the case study described later in Chapter 5.

3.1 Conformance Operations

As previously shown in Figure 2.3 (Section 2.1), the architectural profile notation somewhat resembles that of design patterns: both present example-like structures that the user models should conform to. The profiles explicitly manifest the allowed structures, making them analogous with the conforming models. UML profiles, on the other hand, are used for specializing existing metaclasses with user-defined stereotypes and for introducing additional constraints upon them. The profiles describe the allowed structures at metalevel, thus making them more obscure and harder to understand for a designer not familiar with the UML metamodel. In that sense, architectural profiles are more intuitive when compared to standard UML profile notation.

3.1.1 Extended UML Profiles

To preserve the intuitive nature of the architectural profiles while still benefiting from the standard UML profile mechanism, the conformance operations can be interpreted as translations between the two notations. Following this rationale, the architectural profiles are presented as UML profiles, and the outcome of the conformance operations is determined by the well-formedness

of the UML model in respect to the UML profiles. However, UML profiles can only contain tag definitions, stereotypes, constraints, and data types ([38], Part 2, pp. 190), preventing the designer from explicitly constraining the inherited meta-associations among user-defined stereotypes. The derived connections cannot be explicitly expressed but are lost in localized OCL constraints.

This shortcoming is addressed by using an extended form of the UML profile mechanism as presented by Selonen *et al* [51]. The extended profiles contain two parts: a standard UML metamodel part, showing the subset of the metamodel that is being extended, and an extension part, showing the user-defined stereotypes and the inherited meta-associations with additional constraints. The mechanism follows the spirit of UML, as the standard explicitly states that a profile is not a first-class extension mechanism: it can not modify or restrict the existing UML metamodel ([38], Sec. 2.6.1). All restrictions presented in an extended profile target only the user-defined stereotypes and therefore do not affect standard UML models. Furthermore, the extended profiles can be normalized into standard UML profiles with additional OCL constraints.

Figure 3.1 shows an example of an extended UML profile. The upper side of the figure shows the UML metamodel subset that is being extended, consisting of `Class` and `Dependency` metaclasses, and two meta-associations between them. The lower side shows three user-defined stereotypes: ≪`MyClient`≫, ≪`MySupplier`≫, and ≪`MyDependency`≫. The base classes of stereotypes are shown using filled generalization arrows (as suggested by UML 2.0, [39] Sec. 18.3.1) and the inherited meta-associations with hollow-headed generalization arrows. The multiplicities on the inherited meta-associations show additional constraints on the stereotypes. A `{strict}` constraint at a meta-association end implies that the other end of the meta-association is always required to point to a given stereotype ([51], pp. 5) at an instance level.

When necessary, the profile can be "normalized" to contain only the metaclass definitions. The additional constraints implied by the inherited meta-associations with the `{strict}` constraint and restricted multiplicities can be expressed using OCL as follows:

```
context MyDependency inv:
  -- interpretation for {strict} constraints:
  self.supplier->forall(stereotype->includes(MySupplier)) and
  self.client->forall(stereotype->includes(MyClient)) and
  -- interpretation for restricted multiplicities:
  self.supplier->size = 1 and self.client->size = 1

context MySupplier inv:
```

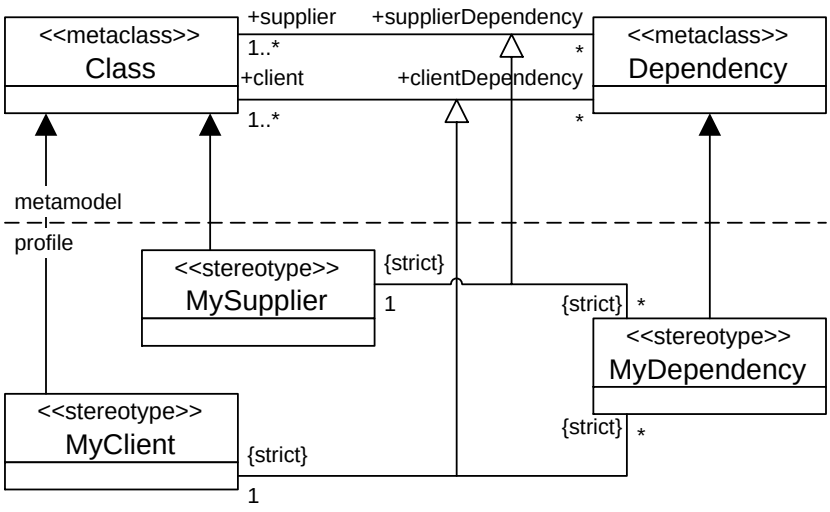

Figure 3.1: Example of an extended profile

```
  self.supplierDependency->forall(stereotype->includes(MyDependency))

context MyClient inv:
  self.clientDependency->forall(stereotype->includes(MyDependency))
```

For clarity, the above invariants use the name of the stereotype as their context instead of the extended metaclass. When working with several architectural profiles, the profiles can be translated individually to the extended profile representation, and these profiles can be then merged together. The inherited meta-associations, together with the `{strict}` association ends, result in a disjunction of constraints.

While the extended profile mechanism is useful when introducing constraints on individual stereotypes, it is sometimes necessary to place constraints on an entire model. To make this easier, two assumptions are made: (1) there is a pointer to a universal namespace, i.e., a root namespace from which it is possible to navigate to all model elements belonging to the particular UML model, and (2) such a navigation mechanims exists, i.e., there is a way to traverse the transitive closure of the model. The former assumption is realized by assuming a root namespace element for the model and profiles, `modelRoot` and `profileRoot`, respectively. A mechanism satisfying the latter assumption (the *find* operator) is discussed by e.g. Siikarla, Peltonen, and Selonen ([53], pp. 179). In practice, the assumptions are typically realized by any reasonable model processing environment.

3.1.2 Stereotype Conformance

Description

> `stereotype_conformance(P: Profile, M: Model): Boolean;`
> Every non-standard stereotype in model `M` must be defined in a stereotype definition profile `P`. Every classifier in the model must have a properly defined stereotype.

Specification

In principle, the former part of the definition follows directly from the UML specification: for a stereotype to exist, it must be properly defined. A formalization for the latter part can be given as an OCL invariant as follows:

```
context Classifier inv:
  stereotype->exists( s | P.find( t | s=t )->notEmpty() )
```

The constraint states that each Classifier in the model must have a stereotype defined in the stereotype definition profiles. See "UML Notes" below for further discussion.

Example

An example of applying the stereotype conformance operation is given in Figure 3.2. The left-hand side of the figure shows a stereotype definition profile with two user-defined stereotypes ≪`OwnCategory1`≫ and ≪`OwnCategory2`≫. The notation follows the one suggested by the UML specification ([38], Fig. 4-1). The right-hand side shows an example model with the upmost class `Foobar` conforming to the stereotype conformance definition, while the bottom class `Foo` has an undefined stereotype ≪`OwnSubCategory`≫.

Pragmatics

In essence, this rule requires that only the properly introduced user-defined concepts, expressed with stereotypes, are allowed to be used in the models.

Variants

The given specification for stereotype conformance is rather restrictive. When necessary, it can be relaxed to allow the use of classifiers without stereotypes, and also the use of user-defined stereotypes not present in the stereotype definition profiles.

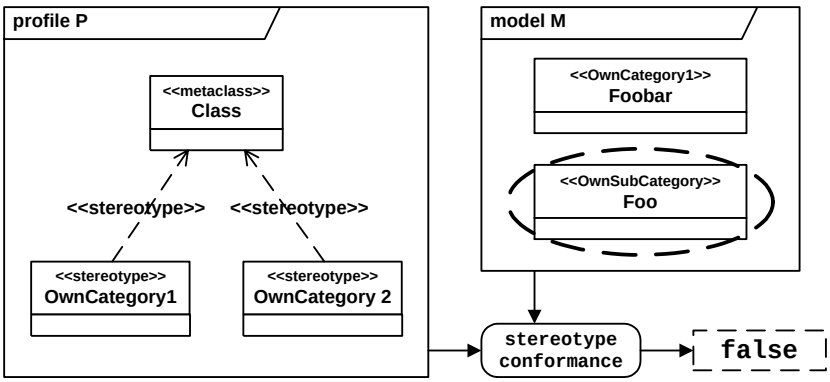

Figure 3.2: Example of stereotype conformance operation.

3.1.3 Relationship Conformance

Description

`relationship_conformance(P: Profile, M: Model): Boolean;`
Every relationship (i.e. association, dependency) in model `M` is required to have a corresponding relationship in profile `P`.

Specification

Figure 3.3 shows how a dependency in an architectural profile shown on the left-hand side is interpreted as an extended UML profile, shown on the right-hand side. The latter has already been described in Figure 3.1. Figure 3.4 shows the analogous interpretation for an association. The left-hand side of the figure shows two classes with stereotypes ≪`MyClass`≫ and ≪`MyOtherClass`≫ connected to association ≪`MyAssociation`≫. The right-hand side shows the corresponding extended UML profile with multiplicities `a1...a2` and `b1...b2` attached to the association end stereotypes.

Example

Figure 3.5 shows examples of applying relationship conformance on a simple model. Profile `P` requires that a ≪`client`≫ always connects to two ≪`proxy`≫ elements with an `Association`. Further, a dependency is allowed between ≪`client`≫ and ≪`API`≫. The realization relationship between ≪`API Impl`≫ and ≪`API`≫, shown in model `M`, however, is not allowed by the particular profile.

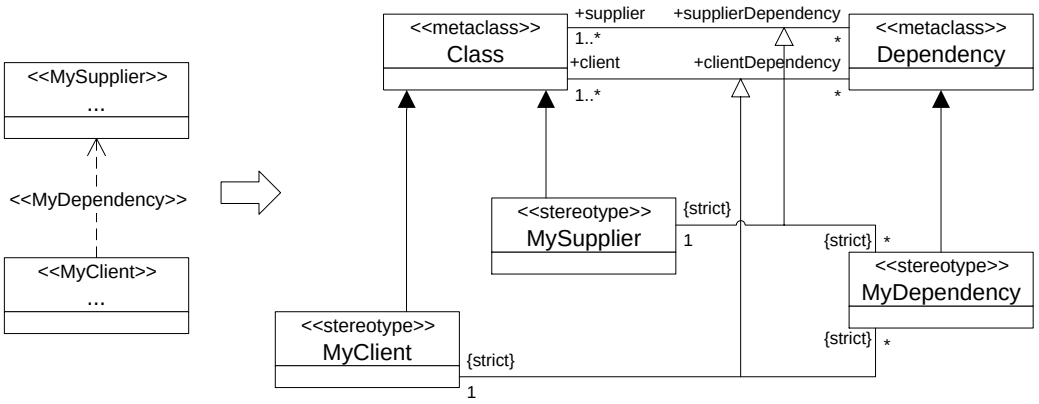

Figure 3.3: Interpretation of a dependency

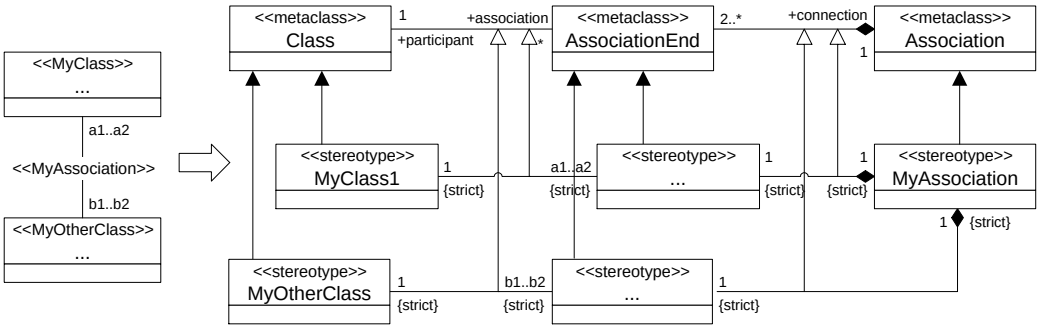

Figure 3.4: Interpretation of an association

Pragmatics

The relationship conformance operation can be used for ensuring that only allowed relationships between architectural concepts are being used and that the number of relationships between the concepts is correct.

Variants

As with stereotype conformance, the given specification for relationship conformance is also rather strict. When necessary, it can be relaxed in a similar manner to allow the use of unstereotyped relationships between unstereotyped model elements, or even between model elements with user-defined stereotypes.

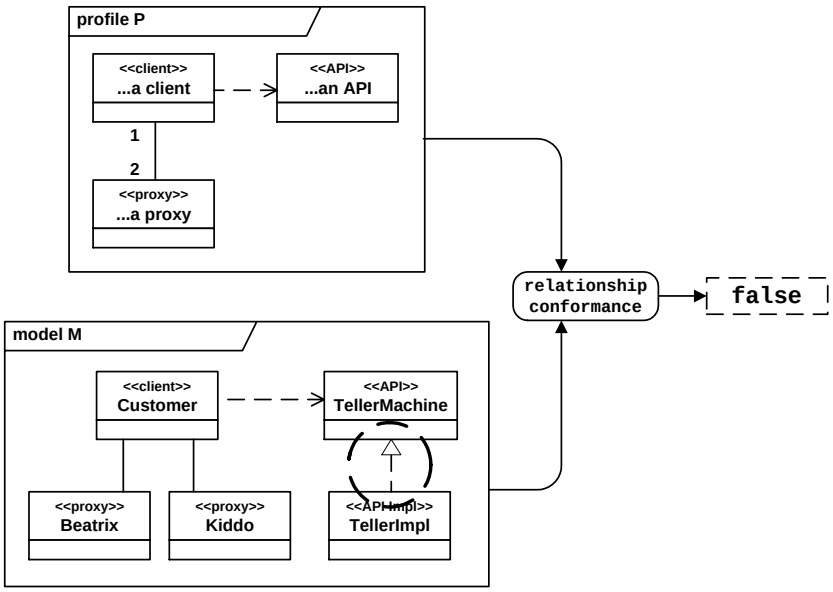

Figure 3.5: Example of relationship conformance operation

UML Notes

Figure 3.3 is identical with the extended profile example given in Figure 3.1. While relationship conformance is defined for dependencies, associations, and their subtypes, the same mechanism can be analogously defined for the remaining UML relationships as well (e.g. generalization).

3.2 Set Operations

The set operations are based on deriving a correspondence relationship among elements that are seen to represent the same modeling concept. Correspondence is used as the basis for resolving the possible conflicts between the model elements (e.g. inconsistent multiplicities on association ends), and for performing the actual operation. Obvious correspondence criteria include the type (metaclass) and name of the model elements, and a repository identifier when available. The context of a model element can be used as an additional correspondence criteria. Such context can comprise, for example, the end elements of a relationship or a composite element owning part elements (e.g. a class owning attributes).

The basic definitions offer a starting point for applying whatever rules and heuristics might be available for the given domain. While the correspondence definitions can be arbitrarily complex, in practice the way humans

use identifiers makes names and types of the model elements a very useful correspondence criterion. The described approach works on static models, and static structure models in particular. Depending on the naming scheme used, it can be also used on instance-level structure diagrams.

3.2.1 Deriving Correspondence

Name and type based correspondence is useful when targeting UML classifiers, but it is insufficient for describing correspondence between relationships or composite model elements. Instead of defining correspondence separately for each UML metaclass of interest, it is beneficial to construct a general correspondence derivation scheme as a basis for the approach. The definition exploits the MOF specification [35] and assumes that the abstract syntax for a given modeling language—typically, but not necessarily, UML—is given as a MOF model. The main assumption is that the meta-associations present in an abstract syntax define a *context* for each metaclass that can be used as a starting point for reasoning about correspondence.

The context includes the structure necessary for deriving correspondence for a model element. For example, the context of a UML `Attribute` comprises its owning `Namespace` and type `Classifier`, and the context of an `Association` comprises its owning `Namespace`, and its `AssociationEnd` elements and their participant `Classifiers`. To establish a specification for context, some fundamental definitions are given first:

- A *parent* element connects to its *child* element via a metalink that instantiates a composition meta-association between the corresponding metaclasses (types) of the elements, the type of the parent element being at the composite end, and the type of the child element being at the other end. For each child element, there can exist at most one parent element. This is guaranteed by MOF composition semantics ([35], Sec. 4.10.2).

- Model element e_d is *descendant* of model element e_a if e_a is the parent of e_d, or if the parent of e_d is a descendant of e_a. Similarly, e_a is an *ancestor* of e_d.

- A *mandatory neighbour* of a model element is connected to the element by a metalink that instantiates a meta-association between the corresponding types of the elements, where the lower multiplicity bounds on the mandatory neighbour side are greater than zero. A *mandatory child* element is a mandatory neighbour of its parent element.

- Model element e_{md} is a *mandatory descendant* of model element e_a if e_{md} is a mandatory child of e_a, or if e_{md} is a mandatory child of a mandatory descendant of e_a.

The parent–child and ancestor–descendant dichotomies contribute to a model namespace hierarchy. The mandatory neighbours state the context of the model element. The *primary context* of a model element is a set of model elements consisting of

1. its parent element,
2. its mandatory neighbours,
3. its mandatory descendants, and
4. the mandatory neighbours of its mandatory descendants.

The difference between a context defined by the neighbours of an element and the primary context is that the latter constitutes the structure that is required to exist for the model element to be properly defined in terms of the abstract syntax. Following from the definition, a unique primary context always exists. Exploiting the definition of context, the correspondence criterion assumed in this thesis is *name-type-context-correspondence*:

> Given models A and B, model elements e_A belonging to A and e_B belonging to B are said to name-type-context-correspond if they have the same metaclass, the same name, and a name-type-context-correspondent primary context.

Two elements are *uniquely corresponding*, if they only correspond to each other. In the case of several corresponding *candidate* elements, the default functionality is to ignore them. When necessary, these ambiguities can be dealt with e.g. as suggested by Egyed ([12], Sec. 4.2).

To further illustrate the apporach, Figure 3.6 shows two simple example UML class diagrams. Both diagrams share the `Contract` and `Client` classes and an association between them, and further `Firm` class specializing `Client`. The diagrams are used as input for the union, intersection, and difference variants later in this section. The UML metamodel instances of the class diagrams are shown in Figure 3.7 with the non-corresponding model elements drawn with a dashed line. In addition, the figure shows the parent, mandatory child, and mandatory neighbour relationships between the model elements.

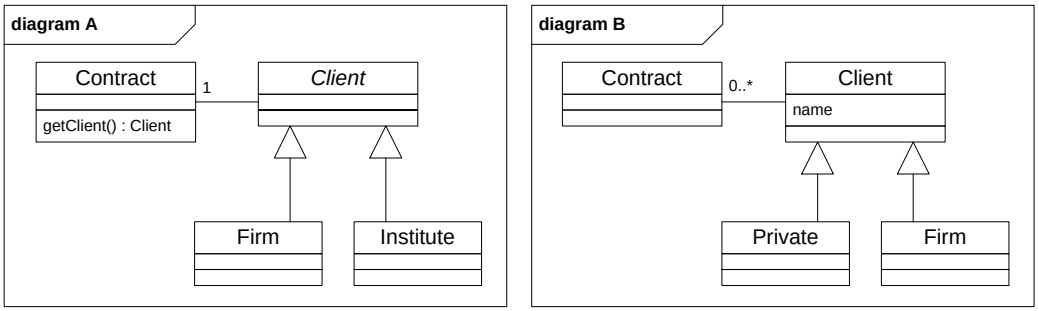

Figure 3.6: Example class diagrams

It is possible for a number of definitions for correspondence to exist. The collection of model element properties can be arbitrary complex. However, given the way humans use identifiers (as names) when making specifications, there is a strong reason to give the name property a special emphasis. This is a reasonable assumption and supported by the fact that a vast majority of the similar approaches rely on these techniques, as pointed out by Clarke ([9] pp. 58). The principles for deriving the correspondence relationship are straigthforward and rely heavily on the sensibility of the input diagrams.

It is sometimes reasonable to relax the requirement of having identical metaclasses, making it possible to associate model elements whose metaclasses share a common superclass in the UML metaclass hierarchy (e.g. allowing a `Dependency` and an `Abstraction` to correspond). When the primary context definition makes correspondence derivation unnecessarily strict, it can be relaxed: for example, two `Operations` with different return types can then be considered correspondent. Similarly, in cases where the two models are composed differently, it can be reasonable to allow e.g. `Classifiers` to have different parent elements. On the other hand, while a `Stereotype` does not belong to the primary context of a UML model element, it is still sometimes sensible to require that corresponding elements have the same stereotypes.

The presented correspondence derivation technique has been instantiated in the context of the UML metamodel. So far, the instantiation has been achieved manually and, consequently, the implementation is defined in terms of the UML metaclasses.

3.2.2 Union, Intersection and Difference

Description

```
union(A: Model, B: Model):  Model;
```

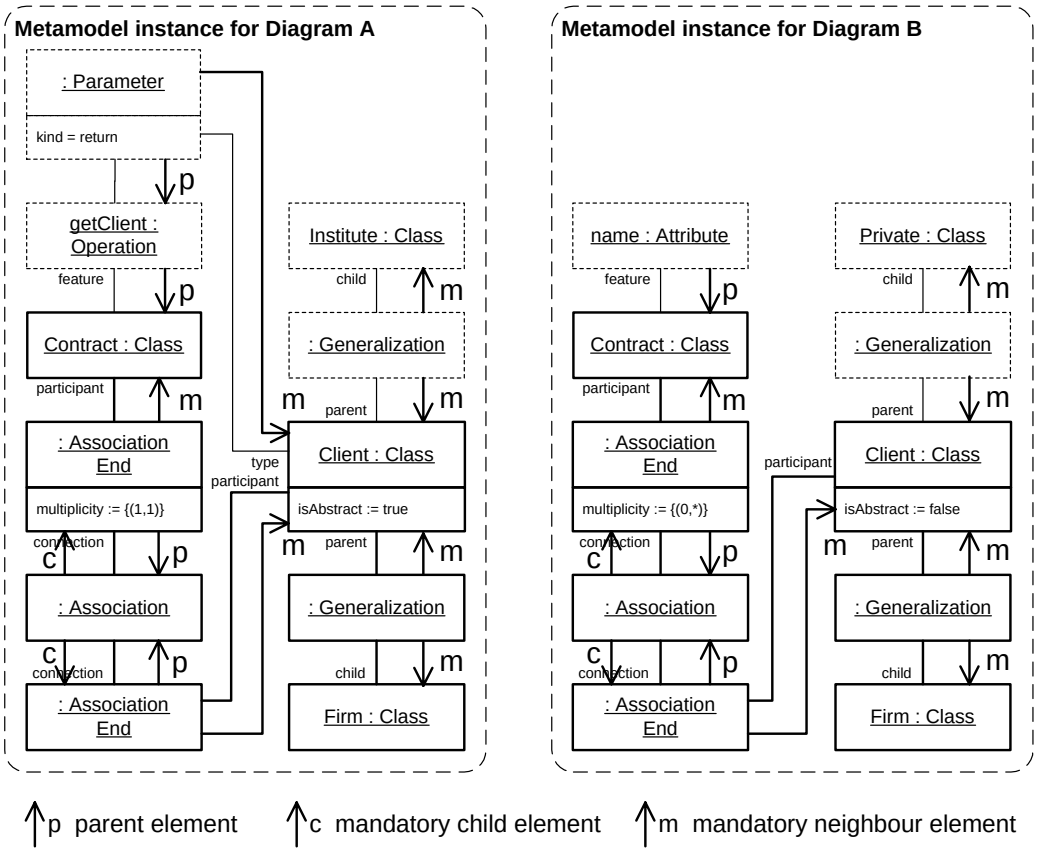

Figure 3.7: Example class diagrams as annotated UML metamodel instances

Union(A,B) contains all model elements from models A and B, with uniquely corresponding elements merged and their meta-association instances redirected to the merged model elements.

`intersection(A: Model, B: Model): Model;`
Intersection(A,B) contains only the uniquely corresponding model elements.

`difference(A: Model, B: Model): Model;`
Difference(A,B) contains all elements in A that do not have a corresponding element in B, and whose ancestors belong to Difference(A,B).

Specification

For the MOF-level specification, see the correspondence definition given in the previous subsection (3.2.1).

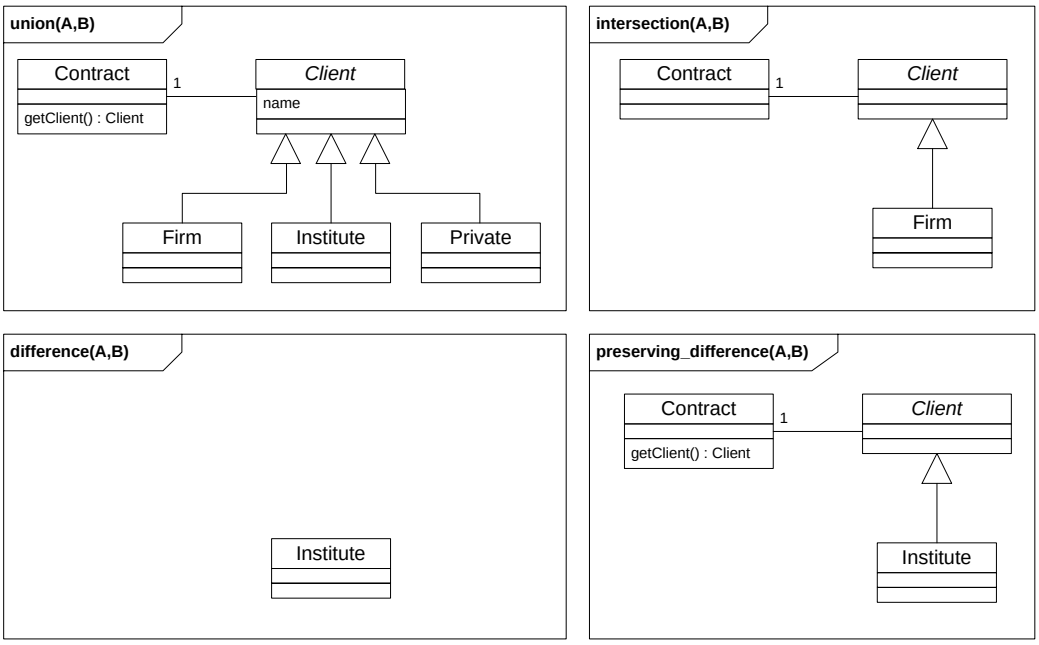

Figure 3.8: Examples of applying UML set operations

Example

Figure 3.8 shows the results of applying the union, intersection, and difference set operations on the example diagrams introduced in Figure 3.6. In addition, it shows the result of applying a difference invariant (preserving difference) discussed below ("Variants").

Pragmatics

The operations play a key role in the model processing approach presented in this thesis. They offer a way of composing and decomposing models, and thus support the suggested model fragment and diagram based model processing.

How the conflicts among the states of corresponding model elements (e.g. an active versus a passive class) should be interpreted depends on their origin, i.e., how their models relate to each other. For example, if one model is produced at a later phase of evolution, it might be considered more accurate. A straigthforward approach to address such an asymmetric situation is *asymmetry*: to favor one operand over another. When necessary, the details of the selected modeling language—e.g. how should inconsistent multiplicity ranges of corresponding association ends be merged—can be taken into account to refine the approach.

Variants

The given form of difference can result in an unnecessarily large number of model elements being removed. For example, if both models A and B contain package P, Difference(A,B) will result in the loss of all model elements under P, even if they are only present in model A. A difference variant leaving the context of non-corresponding model elements intact is shown in Figure 3.8. It can be defined as follows:

> `preserving_difference(A: Model, B: Model): Model;`
> Preserving difference(A,B) contains all elements in A that do not have a corresponding element in B, and all elements that belong to the primary context of an element belonging to Preserving difference(A,B).

The preserving difference variant presented here is more conservative than the one presented in the included publication [VIII]. While the example diagrams shown in Figures 3.6 and 3.8 are taken from [VIII] (Figures 2 and 4, respectively), the association between `Contract` and `Client` has been left intact in Figure 3.8.

UML Notes

The operation definitions themselves do not guarantee that the resulting UML diagrams conform to the UML abstract syntax or its well-formedness rules. The former case occurs frequently in practice with diagrams that are not complete specifications. For example, the model fragment implied by the diagram in Figure 3.6 is invalid in the sense that it does not specify a type for attribute `name`. An example of the latter case is a union resulting with circular inheritance hierarchies.

The set operations are primarily suggested to be used on structure diagrams (i.e. class, component, deployment, and use case diagrams). In addition, they are applicable on instance-level structure diagrams (i.e. object diagrams and the instance-level variants of component and deployment diagrams) if the objects have unique names. The operations can also work on very restricted behavioral specifications (i.e. statechart and activity diagrams), but as they do not take behavioral semantics into account, the results are rarely useful in practice. With sequence and collaboration diagrams, the approach produces a target diagram containing the source diagrams expressed as disjointed models migrated under a common namespace, unless the input diagrams are identical. For discussion on alternative strategies for merging such models, see included publication [VIII] (Sec. 4.1 and 4.2).

3.3 Transformation Operations

The generality of the UML specification allows the designer to generate almost arbitrary models, but as a trade-off, it is very complex and its semantics are ambiguous, leaving it up to the modeler to decide how certain structures should be described using UML. When defining a transformation operation, some assumptions must be made about the characteristics of the source and target models. Consequently, while there is a need for building bridges between the metamodels of different UML diagram types, there is no single and obvious way of building them, so one must make assumptions about the source and target diagrams when defining a particular transformation operation.

There are several levels of confidence for a transformation operation. In the strictest form (the *minimum principle*) all information in the target diagram is required to be implied by the source diagram. In the most relaxed form (*maximum principle*) no information in the target diagram is in conflict with the source diagram. The derivation of the transformation rules can be based on a *push approach* or a *pull approach*. In the former, the implication of each type of modeling concept (i.e. metaclass) of the source diagram type in the target diagram is considered. In the latter, possible reasons for each feature of the target diagram is sought from the source diagram. While strong transformations are based on applying the minimum principle and the push approach, additional heuristics can be introduced to take advantage of the maximum principle and pull approach. The supported transformations rely on the use of the maximum principle and the pull approach.

3.3.1 Sequence Diagram to Class Diagram Transformation

Description

`sed_to_cld(D: Model): Model;`
The operation takes a sequence diagram as its input operand and produces a class diagram as a result. The structure implied by the interaction of the sequence diagram is represented by the modeling concepts of the class diagram.

Specification

The specification is adapted from the definitions given by Selonen, Koskimies, and Sakkinen ([50], Sec. 5) as follows:

- Generate a class for each classifier role with a distinctive class name. Interpret stereotypes and special constraints (e.g. {active}, [38] Sec. 3.71.2) accordingly.
- For each message, if a relationship between the base classes of the sending and receiving classifier roles does not exist, generate a corresponding dependency (or alternatively, an association) representing the communication connection for the message in question.
- For each class receiving a message, if there does not already exist an operation with the same name, generate a new operation for the class. If there are arguments attached to the message, generate new parameters for the operation with corresponding UML basic types and type expressions.
- For a message marked with stereotype ≪`create`≫, attach the dependency with the stereotype .
- For a self-message marked with stereotype ≪`destroy`≫, attach the stereotype to the generated operation.
- If the message is marked with a stereotype ≪`signal`≫ and there does not exist a signal with the same name, generate a new signal. Associate the corresponding class with this signal.
- If the message is marked with a stereotype ≪`become`≫, the sending and receiving classifier roles are equated (this rule assumes that an object cannot dynamically change its type).
- If the message has an object appearing in the sequence diagram as an argument, add a dependency between the corresponding classes of the sending object and argument object, if one does not already exist.
- If the message is a return message, containing only a return value of some type, the return type of the operation of the preceding message is concluded.

Depending on the source model, some parts of the specification might be disregarded and others included.

Example

Figure 3.9 shows a simplified example of applying the sequence diagram to the class diagram transformation (SED⟶CLD) operation in the manner

presented in [50], including additional heuristics for generating interface hierarchies, composition relationships, and explicit unbound multiplicities (see "Variants" below).

Pragmatics

The sequence diagram may contain contradictory information, for example an active and a passive object of the same class. In this case, the result of the transformation operation is undefined.

Variants

In addition to the straightforward mapping of structural concepts, a set of heuristics suggesting more elaborate constructs to the user can be introduced. Three examples of such rules are given in [50] (pp. 8–9): *simple interface heuristics*, encapsulating the generated operations into a set of interfaces and realization relationships, *broadcast heuristics*, generating zero-to-many multiplicities on association ends, and *composition heuristics*, that generate composition associations. In addition, the interfaces can be arranged into an interface hierarchy as shown in Figure 3.9 ([50], pp. 9). The heuristics aim at using the dynamic information and call patterns contained by a sequence diagram to generate additional structural information that might otherwise be lost during the transformation operation.

A variant of the sequence diagram to class diagram transformation operation is introduced in the included publication [VI]: synthesis of annotated structure diagrams. In addition to the basic transformation, the approach synthesizes a state machine from the set of sequence diagrams and further generates a pseudocode presentation based on the state machine that is attached to the respective classes. The techniques are expected to provide help for the designer in the early phases of the design process. The synthesized class diagram and operation descriptions do not aim to specify the system to be designed comprehensively; they are constructed from the incomplete information given as sequence diagrams and thus reflect the current state of the design.

UML Notes

The transformation operation creates a metalevel mapping from a sequence diagram metamodel to a class diagram metamodel. The nature of the metamodel subsets depends on the assumptions made based on the domain and the concrete environment realizing the operations. The transformation can be seen as a way of further completing a model, following the meta-associations of

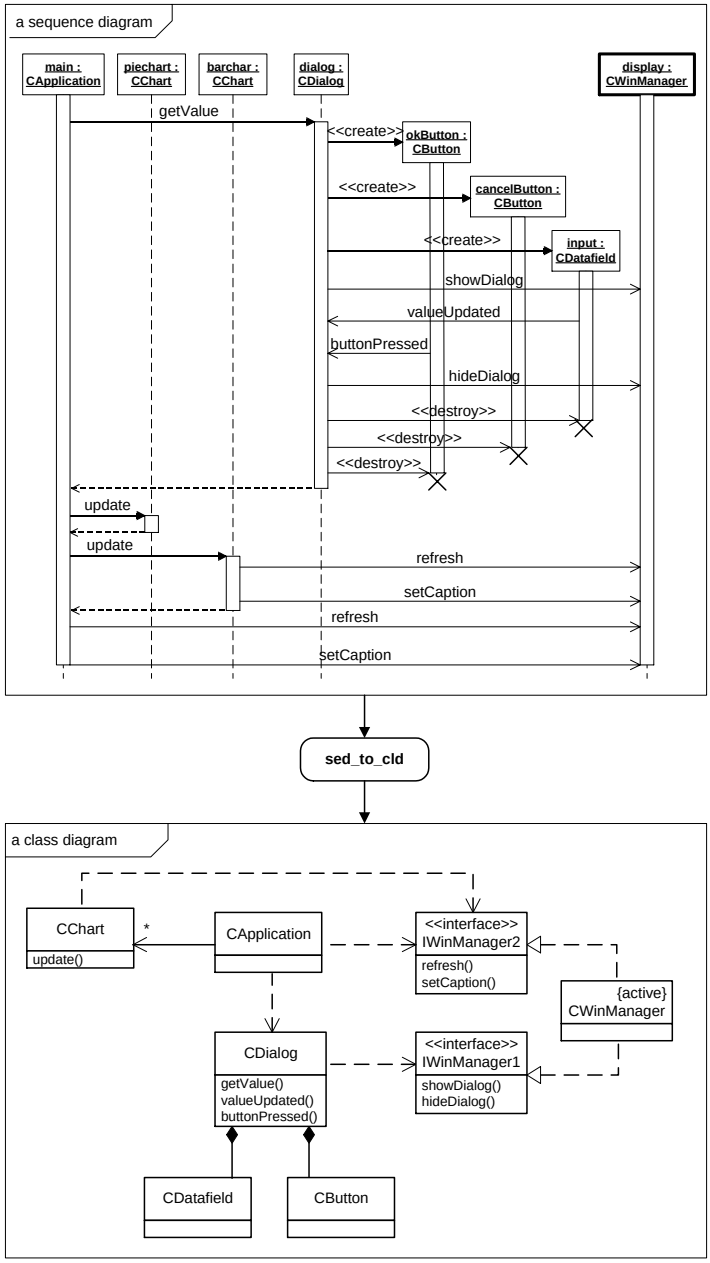

Figure 3.9: Example SED⟶CLD transformation ([50], Fig. 4 and Fig. 10)

the metamodel. The information is "pushed" via the meta-associations and spread further through the system, with meta-associations acting as bridges between metamodel fragments describing different types of modeling information.

3.4 Projection Operations

In principle, projection operations can have arbitrary definitions provided the source and target diagram types remain the same. Projection operations can be based on e.g. abstraction, compression and refactoring of models. In what follows, two example projection operation definitions are given. The operations are selected based on their role in the case study presented in Chapter 5.

3.4.1 Context Diagram Generation

Description

> `context_diagram_generation(A: Model): Model;`
> Abstract a detailed structure diagram to a context diagram showing only the highest-level subsystems, interfaces, and their mutual dependencies.

The structure diagram is expected to include packages, subsystems, classes, interfaces, realizations[1], dependencies, and operations. The resulting diagram shows only the highest-level subsystems, together with the interfaces realized by elements contained by their namespace hierarchies, and the implied dependencies.

Specification

To produce the target context diagram, a series of transformation steps are performed on the source diagram as follows:

- Collect all dependencies having interfaces as their suppliers.
- For each such dependency, redirect its client side to the highest-level subsystem containing its client, if such a subsystem exists.
- Migrate the interfaces and the subsystems to the root level.

[1] A realization is a `Dependency` with stereotype ≪`realize`≫ ([38], pp. 2-17).

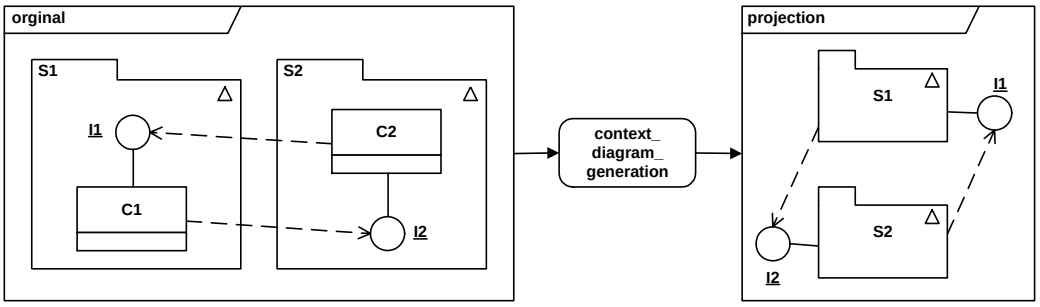

Figure 3.10: Illustration of a context diagram projection operation

- Remove all model elements not connected to an interface through a dependency. Leave subsystems intact.

Example

An example of applying the operation is given in Figure 3.10. The figure shows how the interfaces are raised to the top level, and how their dependencies are redirected to their root namespace subsystems.

Pragmatics

The operation can be used for abstracting the details of lower level structure diagrams and only reveal the high-level subsystems, interfaces, and their inter-relationships.

Variants

A variant of this operation further suppresses details by abstracting away the interfaces and only showing the root namespace subsystems and their dependencies.

UML Notes

Because UML does not define a unique mapping between a diagram and a model fragment it represents, the practical implementation must make several assumptions about e.g. where the dependencies are located in the UML model (under the client namespace, supplier namespace, the namespace containing either two, etc.). In practice, such details are left for tool vendors to decide. Similarly, issues like preserving namespace hierarchies, updating bidirectional metalinks, and maintaining model well-formedness in partial

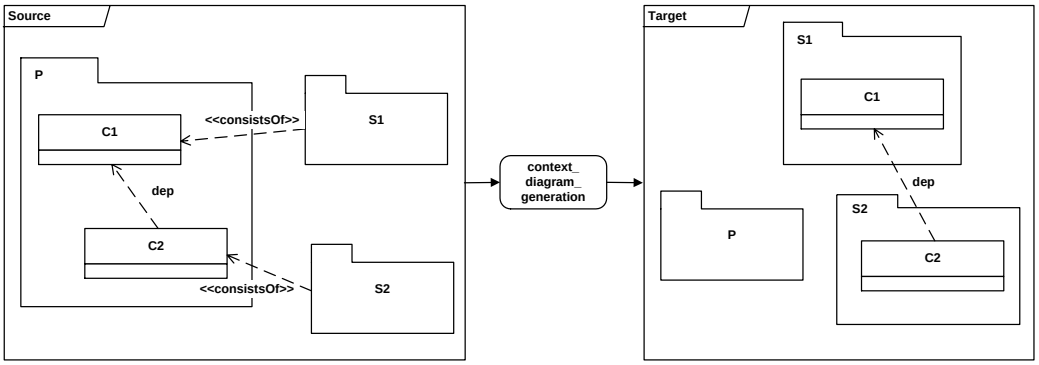

Figure 3.11: Example of applying namespace migration

models become implementation nuisances, hindering the core functionality of the operation to be defined.

3.4.2 Namespace Migration

Description

`namespace_migration(A: Model): Model;`
Recompose a model by migrating selected model elements under a designated set of namespaces according to a given criteria.

Specification

This operation can have numerous variants according to the migration criteria used. This particular example uses special stereotyped (≪`consistsOf`≫) binary dependencies between namespaces to guide the migration.

- For each dependency with stereotype ≪`consistsOf`≫, migrate the supplier under the client namespace.
- Remove ≪`consistsOf`≫ dependencies.

Example

An example of applying the operation is given in Figure 3.11. The figure shows how two classes under package `P` are migrated under new packages, `S1` and `S2`, and the ≪`consists of`≫ dependencies are removed from the model.

Pragmatics

The operation effectively describes a method for re-arranging namespace hierarchies. For example, it can be used as an alternative way of composing a model to emphasize a different concern.

UML Notes

The notes given in the previous subsection (3.4.1) also apply to this subsection.

3.5 Summary

It is not feasible to give general, yet formal definitions for the operations. First, UML itself is not formally defined. Second, OCL has limited expression power for manipulating metamodels and requires a fixed application model, making it hard to reason about models and the metamodel at the same time. To address this issue, the Object Management Group has issued a Request for Proposal [36] for a general-purpose transformation language for MOF-based metamodels as a part of the Model Driven Architecture movement. This language, named QVT (for Queries-Views-Transformation), will emerge at earliest by the end of 2005.

This is not only a technological issue, though. Finding the right abstraction level for specifying the model processing operations is not straigthforward. Regardless of the selected formalism, the generality and the degrees of freedom of the UML specification make this impractical. A one-to-one translation schema forces the operations to be tied to a particular context, makes them lose their generality, and clutters their definitions with modeling language specific details.

Tool support and customizability are key goals of the presented model processing approach, and they have been achieved. It is the belief of the author that communicating the underlying principles and ideology should be the key contribution of this chapter.

The conformance operations introduced in the included publications are summarized in Table 3.1. The introduced set operations are summarized in Table 3.2. Both operation categories allow an arbitrary set of operation definitions: conformance operations only dictate the metamodel–model relationship between its source models, while the set operations rely on establishing a correspondence relationship between the model elements representing the same concepts. The number of different transformation operation types, on the other hand, is bound to the number of different diagram types (modeling

Table 3.1: Summary of Conformance Operations

Operation	Definition
Stereotype conformance	Every stereotype must be defined in a stereotype definition profile. Every classifier must have a stereotype.
Relationship conformance	Every relationship (i.e., association, dependency) is required to have a corresponding relationship in a profile.
Multiplicity conformance	The number of associations each classifier participates in must fall in the range of the multiplicities defined by the corresponding association in a profile. The rule follows directly from using the relationship conformance variant introduced previously (i.e., mapping the multiplicity definitions a1..a2 and b1..b2 in Figure 3.4).
Interface conformance	For every interface, if there exists a corresponding interface and a (set of) realizing class(es) in a profile, there must also exist a corresponding realizing class.
Link conformance	Every link must have a corresponding link in a profile.
Concrete composition conformance	Each classifier must be contained by a legal parent namespace defined in a profile.

paradigms). However, the interpretation of individual operations can vary across different domains and purposes. Strong transformations are summarized in Table 3.3. The descriptions given for the operations are not the only possible ones as there might be alternative ways of defining them. For example, the transformation from a statechart diagram to a class diagram could be done using the State pattern (Gamma *et al* [18]). Finally, the projection operations can have arbitrary definitions provided the source and target diagram types remain the same. Different projection operation categories are summarized in Table 3.4.

Table 3.2: Summary of Set Operations

Operation	**Explanation**
Union	Merging of models.
Intersection	Preserving the common parts of the model.
Difference	Preserving only the parts of model unique to the first operand.
Preserving Difference	Preserving only the parts of the model unique to the first operand, and their owning elements.

Table 3.3: Summary of Transformation Operations

Operation	**Explanation**
SED⟶CLD	As defined in Subsection 3.3.1.
SED⟶SCD	The transformation can be interpreted as an algorithm for synthesizing a minimal state machine from a set of sequence diagrams automatically. First, a trace from the sequence diagram is extracted from the point of view of the classifier role of interest. The items in the message trace are mapped to transitions and states of a state machine. Sent messages are regarded as primitive actions associated with states. Each received message is mapped to a transition.
SCD⟶SED	The transformation generates a sequence diagram by simulating a set of statechart diagrams. Actions are messages sent or actions performed by the object described by the statechart diagram. Event triggers of transitions are interpreted as messages received by the object. Actions are transformed to sent messages of an interaction. This operation uses the same interpretations as the previous one.
SCD⟶CLD	The transformation reveals the static class structure implied by a statechart diagram. Signal events occurring at transitions and states are mapped into signals, and call events are mapped into operations. The context of the state machine is mapped into the classifier.
SED⟶OBD	The operation effectively results in transforming the sequence diagram into a collaboration diagram, and then by removing their interaction.
OBD⟶CLD	The objects and links are transformed into their respective classes and associations, and the states of the objects are transformed into their structural features.

Table 3.4: Summary of Projection Operation Categories

Category	**Explanation**
Abstraction	Abstraction of models is a process of deriving concepts at a higher level of abstraction from concepts at a lower level of abstraction, an example being a class diagram to component diagram transformation. When abstraction is achieved within a same modeling paradigm (i.e. a UML diagram type), it can be seen as a projection operation. For example, the context diagram generation operation presented in this chapter is an abstraction operation.
Compression	Compression of models aims at hiding details irrelevant to the selected viewpoint. For example, inheritance hierarchies, composition relationships, and classifier relationship chains can be collapsed. Conceptually, compressed model elements typically reside at the same level of abstraction than the elements that are suppressed as opposed to abstracted model elements.
Refactoring	With the increasing need for re-engineering software systems, mechanisms for applying refactoring patterns have become more important.
Patterns	Design and architecture patterns are a well-established mechanism for collecting the industry best practices and common know how. Although this bends the nature of the projection operations, the additional modeling constructs can be generated (semi)automatically by a projection operation after the roles of the pattern have been tied to concrete model elements.

Chapter 4

Implementing the Model Processing Approach

To support the model processing approach in practice, mechanisms are needed for implementing the model processing operations, for combining the operations together, and for integrating them with existing UML CASE tools. This chapter introduces a UML model processing platform as such a mechanism and addresses the implementation of individual operations. Further, it discusses the role of the implemented operations in the context of a larger architecting environment. The chapter summarizes the included publications [III] and [V].

4.1 The xUMLi Model Processing Platform

xUMLi is a UML model processing platform for building small model processing tasks, and for combining them to gain high-level functionality and eventually complete tools (e.g. Airaksinen *et al* [2]). Model processing operations are combined using *VISIOME* (Peltonen [41], Siikarla [52]), a high-level visual scripting language. VISIOME is discussed in more detail in the Section 4.2. The platform has been developed by the PRACTISE research group since 2001. Individual model processing operations are implemented as components on top of the platform.

The platform consists of a UML specialization layer, a set of standard components for importing and exporting models between xUMLi and other tools, and a visual scripting language for combining user components to build high-level scripts. The specialization layer provides a data model compliant to a subset of the UML version 1.5 metamodel and a high-level API for UML model processing. In general terms, xUMLi can be seen as model processing

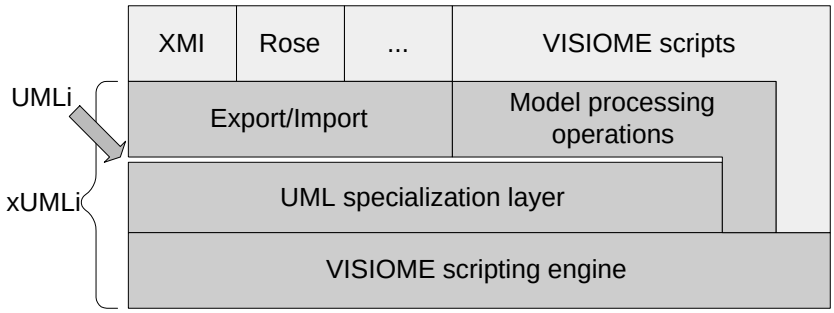

Figure 4.1: The architecture of xUMLi

middleware. The environment is not dependent of any specific CASE tool, but offers a plug-in interface for components that transfer models between a tool repository and the data model. It is therefore possible to support several different CASE tools or UML model repositories. Such import/export plug-ins have been built for IBM Rational Rose[1], XMI (OMG [38] Sec. 5), and some proprietary file formats. Figure 4.1 shows the high-level architecture of xUMLi.

The UML interface offers relatively high-level support for Python and other COM-compliant programming languages by providing access to the UML models through getters, setters, and OCL-based queries, aimed at allowing the user to concentrate on the UML-related problem at hand on a high conceptual level. The individual model processing operations are implemented using such a programming language (e.g. Python, C++) and then combined together with VISIOME.

A simple example of using the xUMLi interface is shown in Figure 4.2 (Siikarla, Peltonen and Selonen [53], Fig. 9). The first part of the example queries for all properly defined stereotypes in stereotype definition profiles. It recursively collects the user-defined stereotypes contained by `profile` to collection `st` using the `find` operator. The OCL expression is true for all classes which have a dependency marked with stereotype ≪`stereotype`≫ and pointing at another class with stereotype ≪`metaclass`≫, effectively introducing a new user-defined stereotype in UML (see e.g. [38], Fig. 4–1). The second part of the example iterates over all classes in `model` and checks whether they have a valid stereotype included in `st`. The operation implements a variant of the stereotype conformance operation described in Subsection 3.1.2.

[1] http://www.rational.com/products/rose/

```
st = profile.find( \
     "element.clientDependency->exists(cd |" \
     "cd.stereotype.name->includes('stereotype')" \
     "and cd.supplier.stereotype.name->includes('metaclass'))")

for cls in model.find("element.metaclass->includes('Class')").ToList():
  for cst in cls.Get("stereotype"):
    if st.select("element.name->includes("" +cst.name+ "").Length==0:
        # handle class with wrong stereotype
```

Figure 4.2: Python implementation excerpt of stereotype conformance operation

4.2 The VISIOME Visual Scripting Mechanism

VISIOME notation is a variant of the UML activity diagram notation ([38], Sec. 3.84) with extended semantics. It emphasizes the use of OCL with a set of elementary programming constructs. The scripts are build using *activities*. Activities can be either other VISIOME scripts or executables, i.e., COM-components[2] realizing a special `IVisiomeExecutable` interface. The scripts are built using *objects* and activities, and put together with *object flows* and *control flows*.

Figure 4.3 shows an example VISIOME script, "Paint Difference". The script can be used for highlighting the differences between a base model and a set of reference models. The script takes two or more diagrams as its input operands. Given the asymmetric nature of the described operation, the reference input is marked with '1'. The script makes a *decision*, marked as a diamond with *guard conditions*, based on the diagram type and converts all input diagrams to class diagrams using suitable transformation operations. Similarly to UML activity diagrams, decisions indicate alternative paths of control. *Synchronization* is marked with a line. All concurrent branches have to be completed before the difference is applied to the diagrams. The difference can be constructed using set operations. The differences to the original diagram are highlighted in the resulting diagram. While the script has a simple structure, it is suitable for several interesting software engineering

[2] http://www.microsoft.com/com/

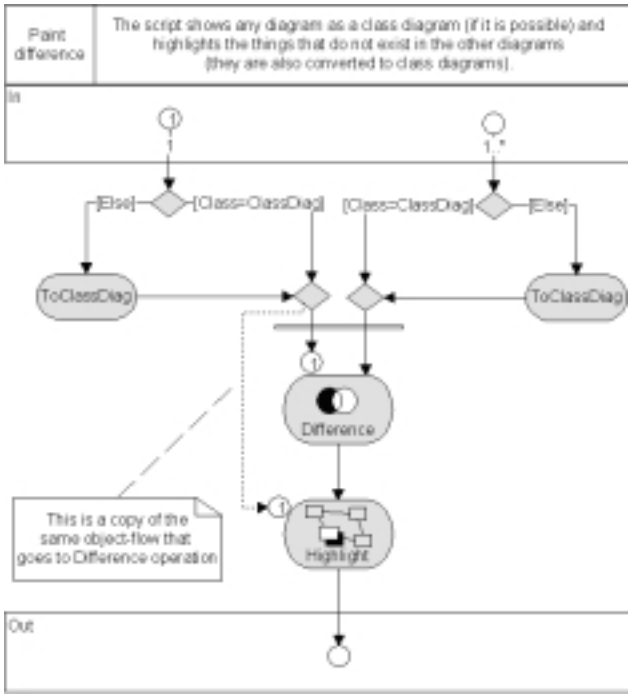

Figure 4.3: Paint Difference VISIOME script

tasks with only a small variation. Other example scripts are presented in the included publication [V].

4.3 The artDECO Architecture Validation Tool

The profile-based architecture model validation approach described in Section 2.3 is realized by *artDECO* (e.g. Airaksinen [1]), a tool for validating architectural models against profiles in UML. It constitutes a set of conformance operations, implemented as xUMLi Python components, and domain specific VISIOME scripts describing validation configurations of the conformance operations. An artDECO validation script imports the architectural profiles and views from a CASE tool, performs a series of model manipulations on them (using e.g. projection operations), executes a set of conformance operations, and outputs the results from the validation in XML format. Alternatively, the user can choose to browse the reported incidents with an xUMLi error browser component integrated with Rational Rose. In practice it is often more feasible to annotate the non-conforming model elements with information on the encountered problems than to return a single

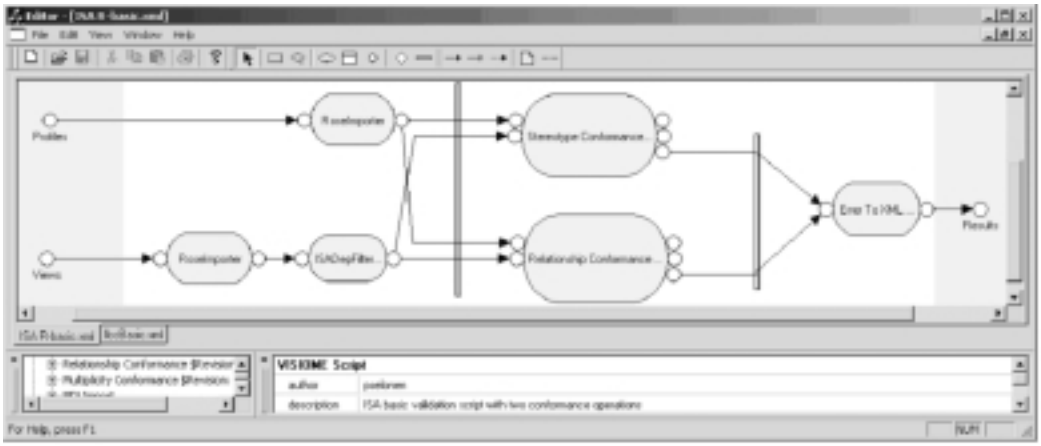

Figure 4.4: Screenshot of an artDECO script opened in VISIOME editor

boolean value indicating whether the model was conformant to the profile or not as a whole.

Figure 4.4 shows a screenshot of an artDECO configuration opened in the VISIOME editor (Tolvanen [57]). The script has two inputs, one for architectural profiles and one for architectural views. Both models are imported from a selected Rational Rose repository by a xUMLi component `RoseImporter`. The architectural views are filtered by an `ISADepFilter` component (i.e., unwanted dependency types are removed) before `Stereotype Conformance` (Section 3.1.2) and `Relationship Conformance` (Section 3.1.3) components are executed. After both operations are completed, their results—incident data describing the non-conforming structures—are forwarded to an `Error To XML` component that produces a summary incident report.

In addition to the conformance operations, implemented as a part of the artDECO tool, other model processing operations described in Chapter 3 have been implemented during the course of research. The first versions were realized on top of *TED*, a Nokia proprietary UML CASE tool (Wikman [60]), as described by Koskinen *et al* [28]. The operations were used e.g. in a case study evaluating the differences between reverse engineering tools (Kollman *et al* [26]). A subset of the operations were migrated on top of xUMLi as described by Airaksinen *et al* [2]. The case study described in Chapter 5 exploits transformation operations, projection operations, and set operations implemented on top of xUMLi. The implementation of the set operations is described by van der Ven [58]. A case study exploiting the set operations on comparing different WSDL descriptions is presented by Lipponen *et al* [22].

4.4 Integrating the Techniques

Some of the implemented tools have been integrated with an experimental software architecting environment [III]. The architecting environment, using UML as the architecture modeling language, facilitates the software architecture design, architecture model analysis and processing, architecture model reconstruction and maintenance, during the entire life-cycle of a software product-line. The environment comprises tools for architecture model validation, architecture model analysis and processing, and reverse-architecting. It fits the current software development process inside Nokia, and is integrated with the design and documentation tools that have already been used by Nokia software architects. The environment has been used in the architecture design and maintenance of a main product-line of Nokia mobile terminal products, and has also been partly applied in another product-line as described in Chapter 5.

The implementation work has aimed at a flexible tool environment that is customizable and adoptable for different domains. For instance, the profiles influence the model validation rules, model manipulation methods, and reverse engineering methods. Therefore, the tools used in this environment are designed and implemented to be customizable and modifiable, and new tools can be integrated in the environment.

The experiences gained during the establishment, deployment, and application of the environment have demonstrated that methods used for architecture-centric software development and maintenance are heavily influenced by the particular context they are applied to. It is necessary that the tools belonging to the environment are reconfigurable and modifiable so that they could be conveniently adapted to a new domain. To be able to support software development and maintenance tasks in different domains, it should also be easy to integrate new model manipulation and validation tools to the environment.

4.5 Summary

The purpose of this thesis is to describe the foundations of the model processing approach. Consequently, the underlying principles of the individual model processing operations are more important than their exact definitions. While they aim to be precise enough, the given definitions are not executable *per se*. They have been implemented on top of the xUMLi platform, integrated with a concrete CASE tool, and during implementation fitted into the

context they are applied in. The real-life case studies act as a proof of the concept for the approach.

Chapter 5

Case Study and Evaluation

> We took four cardboard tubes—the kind of tube you'd find in a regular brand of household toilet tissue—and then proceeded to place them on the floor, making four columns equidistantly, thus. We wanted to test if these cardboard tubes would support the average body weight of a human man.
> ... no.
>
> – Professor Denzil Dexter, University of Southern California[1]

This chapter describes a case study utilizing the model processing approach in practice in the larger context of maintaining a large mobile terminal product platform. The chapter outlines the problem domain, presents the suggested solution, and discusses the obtained results. The presentation is extended from the included publication [II].

5.1 Context of the Study

Software systems, and telecommunication systems in particular, are typically characterized by frequent introduction of new requirements and change of existing features, forcing the products and product platforms to be under constant development. The evolution of a platform is affected by a significant time-to-market pressure and rapid release cycles. Consequently, the architecture of the platform is inevitably decaying over time, making it harder to comprehend and maintain by the designers, thus leading to increased maintenance costs and reduced quality of the products. To complicate issues further, the architecture models produced during forward engineering are rarely

[1] a character played by John Thompson on the BBC's "Fast Show"

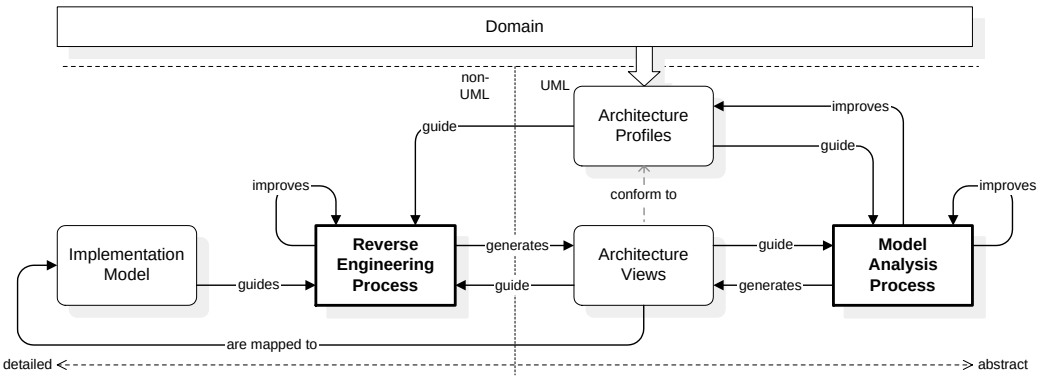

Figure 5.1: A general maintenance process

kept synchronized with the actual implementation model (i.e. source code). Both techniques and tools must be provided to ensure that the product platform architecture includes a set of desired properties, while lacking undesired ones.

To address the abovementioned problems, a UML-based reverse engineering and model analysis approach for software architecture maintenance is introduced. The overall process consists of two subprocesses: a *reverse engineering process* (RE-process) and a *model analysis process* (MA-process) [II]. The former is used for generating a high-level architecture model describing the selected platform release, while the latter is used for analyzing the model and giving feedback for the design of the next platform version.

The maintenance process is illustrated in Figure 5.1. The RE-process combines a traditional bottom-up reverse engineering approach with a top-down approach relying on the domain knowledge captured by the platform architectural profiles. The profiles describe the vocabulary of the domain and how the model should be composed, and guide the RE-process to reach the desired goal. The architectural views and profiles are analyzed by the MA-process, together with the possibly existing reference architecture models, like forward engineering models and models of previous platform versions, and other existing information. The results achieved while applying the MA-process are reflected back to the platform architecture and used for tuning the RE-process and MA-process further.

The case study presented in this chapter instantiatiates the maintenance approach in the context of a real-life software system and exploits the model processing operations to build tools for analyzing the target system. The case study has two main objectives; First, it aims at profile-based validation of architectural concerns, that is, analyzing individual product platform architecture models to ascertain if they are in agreement with the conventions,

restrictions, and rules defined by the system architects. Second, it aims at the comparison of architecture models, that is, comparing the architecture models of different platform releases to monitor the evolution of the system.

5.2 The Target System

The target system of the case study is a large-scale mobile terminal product platform of Nokia Mobile Phones, hereafter referred to as *the ISA platform*. The ISA platform has significant financial value through the various product lines relying on its services. The platform is constantly developed and maintained, and new releases are produced regularly.

The maintenance process, instantiated in the ISA context, is shown in Figure 5.2. As the forward engineering architecture model of the ISA platform is only referential in nature, the actual design decisions and rationale laying in the product itself, the source code (implementation model) of an ISA release must transformed by the RE-process to a reconstructed architecture model (the R-model).

While the reverse engineering subprocess, initially reported by Riva [46], is far from being trivial, it is outside the scope of this thesis. The starting point for the case study is an existing ISA platform architecture model given in UML.

5.3 The Target Architecture Model

The ISA R-model produced during the RE-process has a relatively simple structure. The main architecture model consists of high-level subsystems, subpackages, and components. The components can be servers, applications, delegates, or common applications, and they can connect to each other through interfaces using messages or invocations. In addition, the ISA architecture models can contain information about e.g. the design teams responsible for implementing the components. The size of the R-model is roughly 150 subsystems, 1000 components, and 15000 inter-component dependencies.

Figure 5.3 shows the overall profile hierarchy for the ISA architectural profiles, specialized from the general concept introduced in Section 2.3. The `Conceptual Profile` defines the fundamental architectural concepts for the ISA domain and the basic architecture style that should be followed by the design. The `Package Structure View Profile` imports the concepts of the `Conceptual Profile` and adds information on how the architecture model itself is organized into namespace hierarchies. The `Layer View Profile` im-

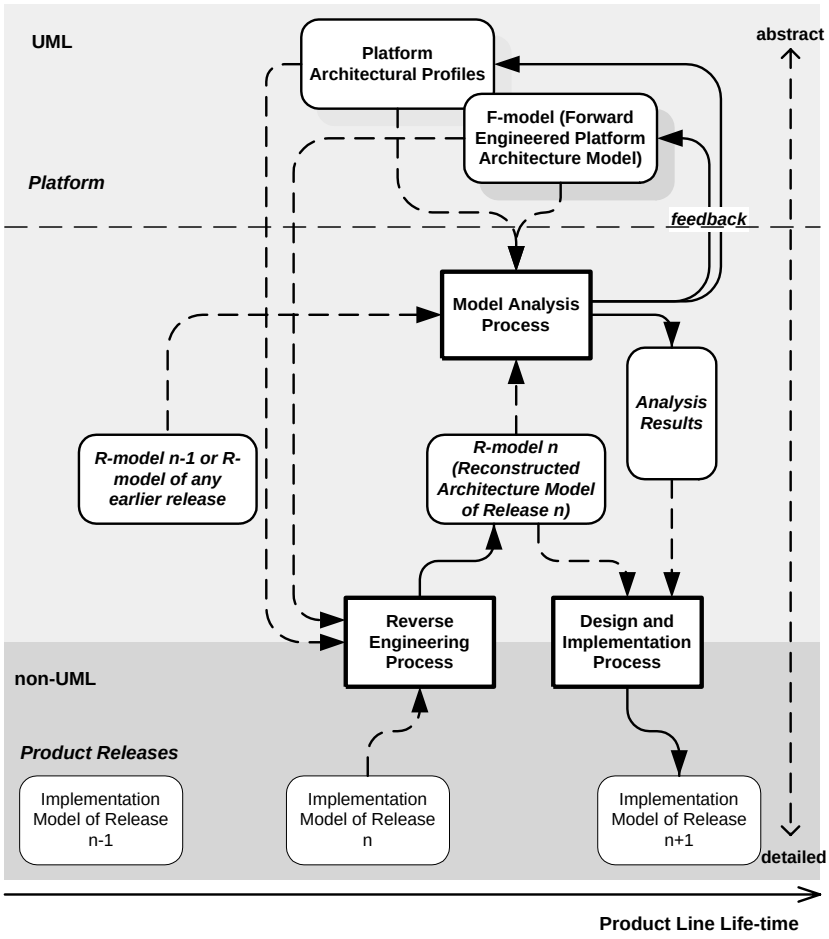

Figure 5.2: The ISA maintenance process ([II], Fig. 10)

ports both profiles and describes an additional architectural style the R-model should follow.

As the synthesis of the ISA R-model during the RE-process is guided by the `Package Structure View Profile` profile, it is obvious that the architectural concepts and model composition are known to be correct *a priori*. Consequently, the profile is not discussed further. For more details, see included publication [II].

Figure 5.4 shows an example of a stereotype definition part of the ISA `Conceptual Profile`. The profile defines a set of concepts relevant to the architecture style used. Stereotypes ≪`Server`≫, ≪`Application`≫, ≪`Delegate`≫, and ≪`CommonApp`≫ represent logical components in the architecture, while ≪`message`≫ and ≪`invocation`≫ represent logical dependencies between these concepts. Figure 5.5 shows a subset of the constraint definition part for the

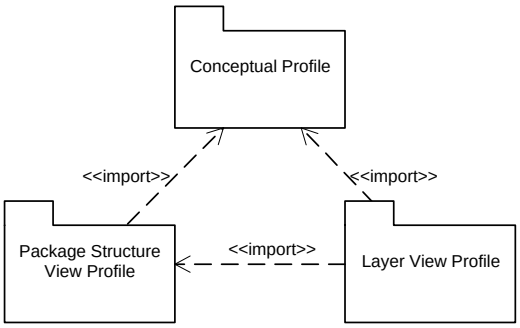

Figure 5.3: ISA R-model Profile Hierarchy

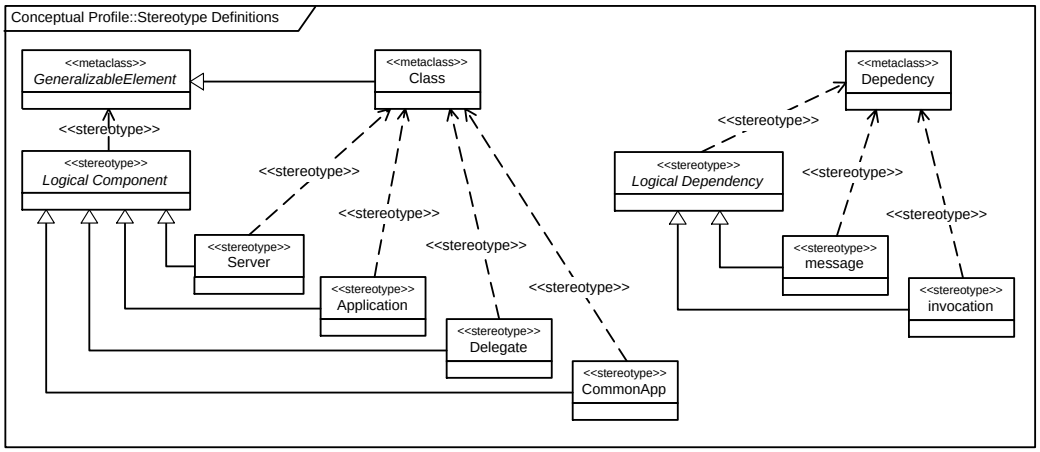

Figure 5.4: ISA conceptual profile stereotype definition example

`Conceptual Profile`. The profile defines the allowed relationships between the architectural concepts and the types of these relationships. The UML well-formedness rules require classes to have unique names. The adopted naming convention is to begin with the name of a class with three dots (...) followed by the name of its stereotype. The names themselves have no semantic meaning. The particular example profile defines a set of interfaces, their legal realizers, and the components depending on these interfaces. For example, ≪`Server`≫ can realize ≪`Server IF`≫, and ≪`Application`≫ can connect to ≪`Server IF`≫ with a ≪`message`≫ dependency. The profile effectively introduces a client–server architecture style.

The RE-process utilizes both the `Conceptual Profile` and the `Package Structure View Profile` while building the R-model. The latter also defines the *primary decomposition* (e.g. Tarr *et al* [56]) of the model. In contrast, the `Layer View Profile` describes an alternative, parallel model decomposition.

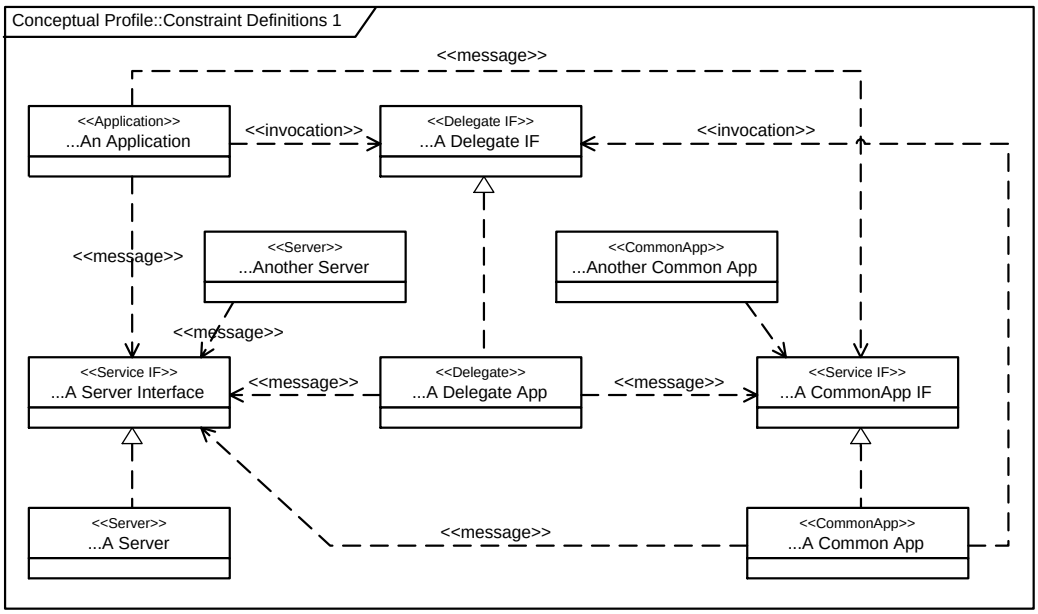

Figure 5.5: ISA conceptual profile constraint definition example

As the RE-process does not take this profile into account when constructing the architecture model, it is discussed later in this chapter.

5.4 Profile-Based Validation of Architectural Concerns

This section discusses the profile-based validation of the ISA architecture model. It sets the goals, describes the applied method, presents the achieved results and analyzes them. This section is based on the included publication [II].

5.4.1 Goals and Applied Method

The ISA profiles describe the UML interpretation for the architectural concerns. The goal is to ensure that the following architectural concerns are met:

1. Only the architectural concepts allowed by the domain should be used.
2. Only the allowed relationships between the architectural concepts should be used and that the number of relationships between architectural concepts should be correct.

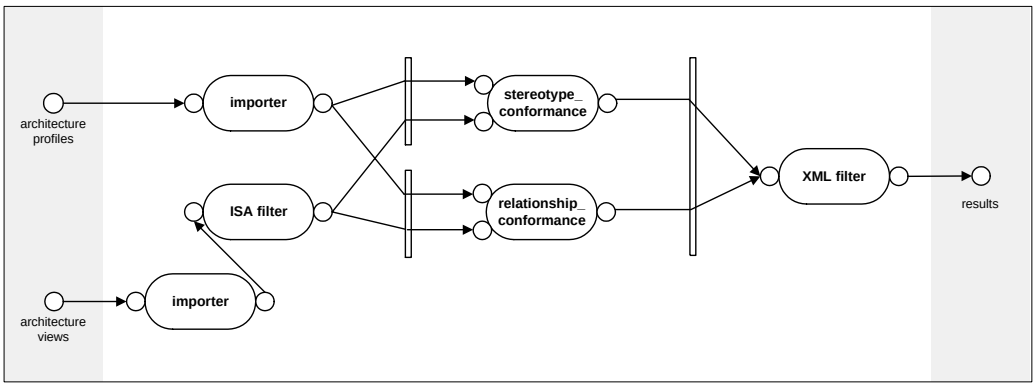

Figure 5.6: A script for ISA basic architecture validation

3. The architecture model should conform to a three-layer architecture style.

The conformance checking is divided into two parts: basic conformance checking and checking of conformance to the layered architecture style. While the former is established using the conformance operations directly, the latter requires some restructuring of the architecture model using suitable projection operations.

Basic conformance checking

Figure 5.6 shows a VISIOME script describing the basic architecture validation task. The stereotype conformance operation (`stereotype_conformance`) is used for validating that only the allowed architectural concepts are used and that the classifiers have proper stereotypes. The relationship conformance operation (`relationship_conformance`) is used for validating that only allowed relationships between architectural concepts are used. The script further shows two input interfaces for architecture profiles and architecture views, two importers for CASE tools (`importer`), a filtering component (`ISA filter`) for filtering out uninteresting information, a synchronization bar before and after the set of validation components, and finally an XML filter (`XML filter`) for filtering the results.

Conformance to layered architecture style

In addition to the selected physical system decomposition, the ISA platform architecture should conform to a layered architecture style. The three layers are defined in the left-hand side of Figure 5.7, introducing ≪`Arch Layer`≫ and its three subtypes: ≪`UI&App Layer`≫, ≪`Service&Resource Layer`≫, and

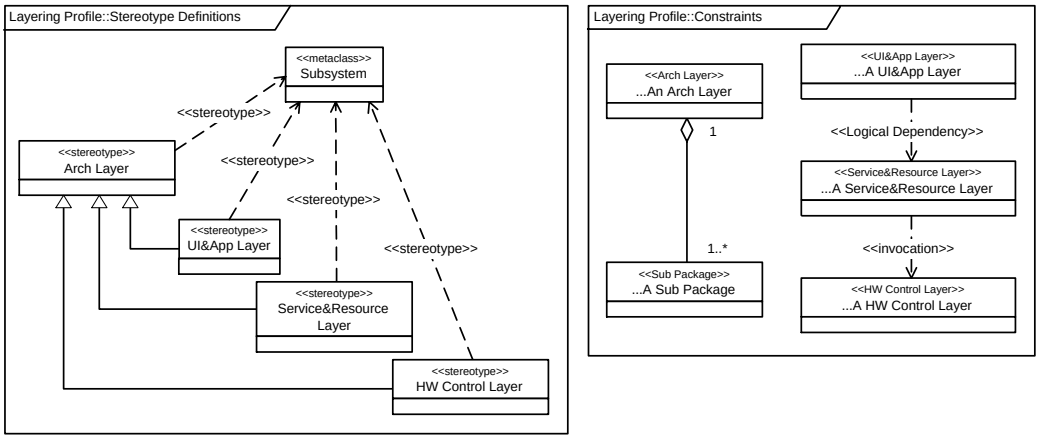

Figure 5.7: ISA layering profile

≪`HW Control Layer`≫. The right-hand side of the figure shows the level of composition granularity for the architecture layers: ≪`Arch Layer`≫ must be composed of ≪`Sub Package`≫ subsystems. The relationship between the three layers is given in the rightmost structure, effectively introducing a three-layer architecture style. The top layer (≪`UI&App Layer`≫) can only communicate with the middle layer (≪`Service&Resource Layer`≫), which in turn can only communicate with the bottom layer (≪`HW Control Layer`≫). Lower-level layers are not aware of upper-level layers, and the top layer is not allowed to communicate directly with the bottom layer by bypassing the middle layer.

To validate that the architecture model conforms to the specified layered architecture style, the model is recomposed according to the rules specified by the profile and *instantiation criteria*. An example of such criterion is shown in Figure 5.8. The diagram ties two ≪`Sub Package`≫ subsystems to `ISA Service&Resource Layer`.

Figure 5.9 shows a VISIOME script describing the layered architecture validation task. The architecture profiles, architecture views, and the instantiation model are given as script inputs and imported by importer components. After the instantiation model is merged with the architecture models (`union`), it is restructured using `namespace_migration` and `context_diagram_generation` components, implementing the projection operations presented in Section 3.4. The logical recomposition is used for restructuring the architecture model to reflect the division into the architecture layers. The resulting model shows the layers and their inter-relationships. This model is then validated against the constraint profile using the relationship conformance

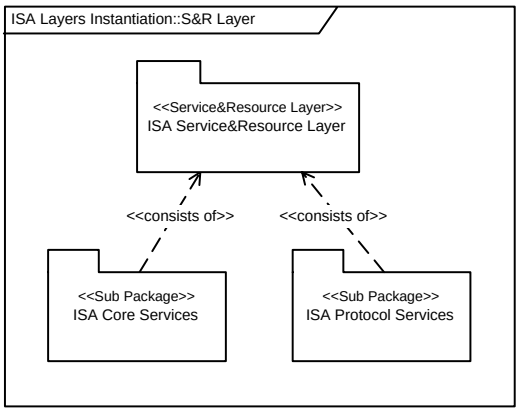

Figure 5.8: An example of recomposition instantiation criteria

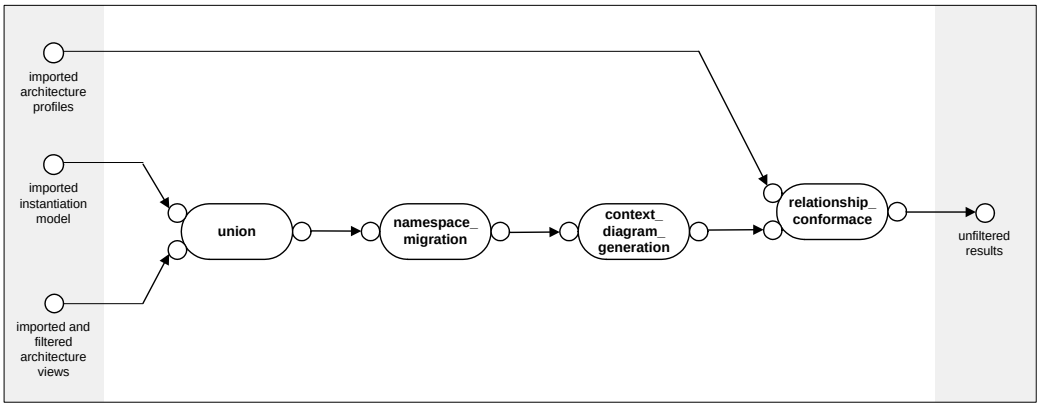

Figure 5.9: A script for ISA layered architecture validation

operation (`relationship_conformance`). The results are finally filtered using the XML filter component.

Summary

The main focus when working with the R-model is the checking of relationships and conformance to the layered architecture style. With manually composed F-models, the other conformance checks are also of interest—see Table 5.1 for additional concerns. The table summmarizes the architectural concerns, their UML interpretations, and the applied model processing operations.

Table 5.1: Architectural concerns, UML interpretations and applied model processing operations

Architectural concern	**UML interpretation**	**Applied MPO's**
Validate that only the architectural concepts allowed by the domain are being used.	Check that all classes, packages, and relationships (dependencies, realizations, etc.) have their stereotypes defined in the stereotype definition profiles. Verify that each Classifier has a designated stereotype.	Use stereotype conformance rule with approriate profiles.
Validate that only allowed relationships between the architectural concepts are being used and that the number of relationships between architectural concepts is correct.	Check that all used dependencies have been defined in the constraint profiles.	Use relationship conformance rule with appropriate profiles.
Validate that the interfaces have correct realizers in the design.	Check that the realizer relationships connecting to the interfaces have proper clients as defined in the constraint profiles.	Use the interface conformance rule together with the relationship conformance rule with appropriate profiles.
Validate that the physical composition hierarchy of the architecture conforms to the domain-specific conventions.	Check that the namespace containment hierarchy follows the parent–child relationships defined in the packaging profiles.	Use the concrete composition conformance rule with appropriate profiles.
Validate that the system conforms to a layered architecture style.	Check that all the dependencies between classes belonging to layers defined in the layering profiles are directed from a higher level layer to a lower level layer.	Recompose the model and compress the relationships using a logical composition projection and relationship conformance rule with appropriate profiles.

5.4.2 Analysis of the Results

The reported incidents for the first ISA R-model validated (`03w39`) were initially reported in the included publication [II]. They included approximately 300 stereotype violations, 5000 relationship violations, and 650 layering violations.

The results were presented to, discussed with, and analyzed by, the chief architects responsible for managing the ISA archictecture model. Obviously, many of the violations originated from "quick and dirty" solutions under the time-to-market pressure. Investigation of the results revealed the following problems ([II], Sec. 6):

1. *Evolving and incomplete profiles.* When new stereotypes were introduced, the constraints of the dependencies regarding them could not be completely specified, and some of them were even left open. The dependencies that were not covered by the profiles were reported illegal.

2. *Components having inappropriate stereotypes.* Some components played a different role than was originally planned. This discovery showed that the rules used by the RE-process for mapping the concepts should be improved, and that the design and implementation of these components should be reviewed.

3. *Sharing of code by the same development team.* Many illegal dependencies occurred between components implemented by the same team or several closely co-operating groups working on the same feature or feature set. As an example, a component can share a function from another component with a function call, which can easily create an illegal dependency regarding the design principles.

4. *Dependencies introduced due to performance and other non-functional requirements.* Some of the dependencies short-cutting across layers involved top-layer components directly invoking the functions of the bottom layer components. These dependencies fall into a special category of dependencies allowed for etc. performance reasons. However, they should be closely monitored and controlled within a very limited scale.

5. *Inappropriate function allocation.* Investigation showed that several components controlling global data had been placed in the top layer, and several components on the middle layer depended on them. Further analysis showed that just a few components were causing a majority of all the reported incidents.

Table 5.2: Results of architecture validation

	ISA 04w07	**ISA 04w21**	**ISA 04w35**
Subsystems	201	230	235
Components	1165	789	816
Relationships	9141	9892	10246
Stereotype violations	42	4	3
Total relationship violations	2343	2170	2081
Client-server relationship violations	306	212	241
Total layer violations	447	582	623
Layer violations (client-server)	132	121	99

Table 5.2 summarizes the architecture validation results obtained while running the maintenance process on three more recent ISA platform releases. After each run, the results were analyzed and both the process and the platform were modified accordingly. The number of incidents caused by the inadequacies of the maintenance process itself has been significantly reduced. Special attention has been paid on the layering violations. While the figures are anecdotal by nature, they suggest that after a few iterations the number of stereotype violations has decreased and the number of total relationship violations has also declined. Among the total relationship violations the number of client–server relationships (first-class dependency of the system) violations has gone down and stabilized, and the number of illegal cross-layer client–server dependencies has steadily decreased.

5.5 Comparison of Architecture Models

This section discusses the comparison of the architecture models of different ISA releases. It also presents an example of slicing the architecture model based on the behavioral information gained during execution of high-level use cases. It sets the goals, describes the applied method, presents the achieved results and analyzes them.

5.5.1 Goals and Applied Method

The goals for comparing architecture models are as follows:

1. Gain a better understanding of the evolution of the platform by observing the changes in the ISA platform architecture between different releases.

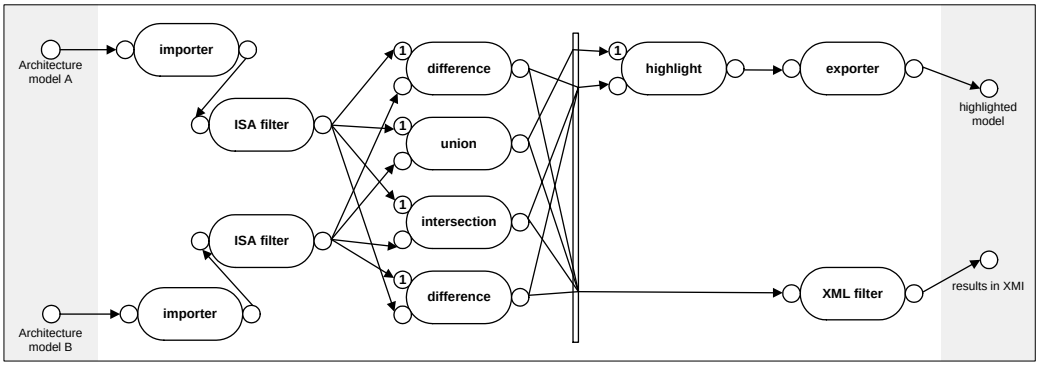

Figure 5.10: A script for ISA architecture comparison

2. Gain a better understanding of the effects of changes in user requirements to the architecure subset involved in executing high-level use cases.

Comparison of different releases

The architecture models are used as input operands for union, intersection, and difference set operations. The resulting commonalities and differences are reported using XML and visualized as a symmetric difference of the models. Particular attention is placed on the addition, removal, migration, and changes of architectural elements. Figure 5.10 shows a VISIOME script describing implementation of the comparison task. The script shows the importing and filtering of two architecture models. The models are then being compared against each other using `union`, `intersection`, and `difference` components. The commonalities and differences are highlighted against the merged model (`highlight`), exported to a CASE tool (`exporter`), and reported using XML.

Comparison of architecture model and execution traces

The set of execution traces is gathered from an actual mobile terminal device running instrumented software built on top of the ISA platform. A series of high-level use cases were performed on the device, and the resulting traces were produced during the RE-process. The abstracted component-to-component communication traces were mapped to UML interaction diagrams.

To compare the structure implied by the gathered execution traces to the architecture model, the sequence diagrams are transformed into class diagrams and merged using the union set operation. The resulting structure model is then used as a slicing criterion for the ISA R-model architecture.

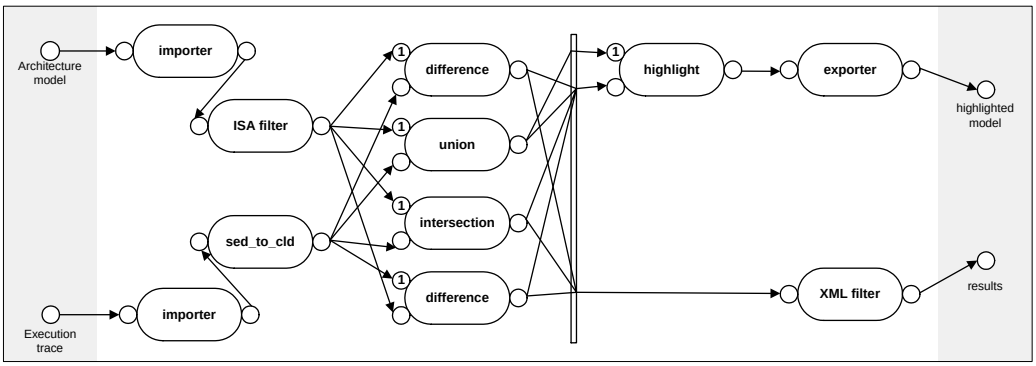

Figure 5.11: A script for slicing ISA architecture against traces

By using the intersection and difference set operations, the commonalities and differences between the models are compared. The results are reported using XML and visualized as a symmetric difference of the models.

Figure 5.11 shows a VISIOME script describing implementation of the architecture slicing task. Apart from transforming the execution trace into a class diagram using `sed_to_cld` component, the script has the same structure as the previous script in Figure 5.10. They both implement the model comparison task first described in Figure 2.7.

Summary

The architectural concerns, their UML interpretations, and the applied model processing operations are summarized in Table 5.3.

5.5.2 Analysis of the Results

The results of comparing three ISA release R-models are presented in Table 5.4. The table shows number of the common, added, and removed elements between two pairs of ISA R-model releases, and it also states the number of changed and migrated elements. With relationships, the presented figure includes the relationships between common components and the figure in paranthesis is the total number of added or removed relationships. Particular attention is placed on the following issues:

1. *Added and removed subsystems and components.* In general, addition and removal of elements represent evolution of the target software system. In the context of the ISA maintenance process, however, the large number of added and removed elements reflects changes in the

Table 5.3: Architectural concerns, UML interpretations and model processing operations for architecture model comparison

Architectural concern	**UML interpretation**	**Applied MPO's**
Compare the R-model against the R-model of the previous ISA release.	Compare the class diagrams and visualize their commonalities and differences.	Compose a symmetric difference by highlighting the differences and intersection against the reference model.
Compare a reference architecture model against the parts involved in executing individual use cases to detect whether the violate the former.	Transform the trace, represented as a set of sequence diagrams, into a class diagram and slice the original architecture model against it.	Use SED⟶CLD, and highlight the intersection against the original class diagram.

RE-process: the parts of the platform selected for architecture reconstruction and changes in the profiles guiding the process. Nevertheless, the set of added and removed elements can also represent feature reconfiguration. After observing the added components and subsystems, such features as enhanced language support and energy management services were detected.

2. *Added and removed relationships.* As the relationship correspondence is defined through their end elements, the focus is placed on the added and removed relationships between common components. While the configuration of included components varies, this restricted set of relationships gives the system architects the opportunity to monitor the changes in the component dependencies.

3. *Migrated subsystems and components.* In general, migrated subsystems and components represent re-engineering performed on the platform architecture for e.g. the purpose of achieving a better separation of concerns. In the context of the ISA maintenance process, migration can also result from the changes in the RE-process. The detected changes were related to refining the component responsibilities by migrating components to dedicated subsystems.

4. *Changed subsystems and components.* In general, the changes in the model elements reflect potential conflicts between their properties that should be reconciled. In the context of the ISA architecture models,

Table 5.4: Results of model comparison

	Subsystems	**Components**	**Relationships**
|**04w21** ∩ **04w07**| (common)	200	668	7191
|**04w21** \ **04w07**| (added)	30	121	805 (2701)
|**04w07** \ **04w21**| (removed)	1	497	343 (1950)
Changed	0	28	N/A
Migrated	0	5	N/A
|**04w35** ∩ **04w21**| (common)	219	752	9019
|**04w35** \ **04w21**| (added)	16	64	493 (1227)
|**04w21** \ **04w35**| (removed)	12	34	290 (873)
Changed	0	0	N/A
Migrated	0	0	N/A

Table 5.5: Summary of trace based architecture slicing

	Components	**Relationships**
|2003w39 ∩ Phone startup|	46	48
|Phone startup \ 2003w39|	18	108

the changes are defined in terms of changed stereotypes. Such changes typically resulted from the restructuring of the platform architecture through improvements on the profiles and the RE-process.

The comparison of architecture model and execution traces was performed on three execution scenarios, one of which is discussed here in more detail. The selected traces describe the activities involved when starting up a mobile terminal device. The characteristics of the subsystem slice are shown in Table 5.5.

The results revealed that over one quarter of the components taking part in the execution trace were not present in the actual architecture model. Most of these components were not mapped due to incomplete mapping tables during abstraction and mapping procedures involved in producing the high-level execution scenarios from recorded low-level activities. However, there still were five mapped components not present in the R-model.

Of the 100 non-correspondent relationships, around 30 involved an unmapped component as a client or a supplier and 20 an abovementioned completely mapped but unpresent components. The rest of the dependencies represented, however, communication channels not present in the architecture model itself. The main reason for this was concluded by the Nokia architects to be that the traces did not take into account method invocations

and interfaces. The obtained results were not as comprehensive or useful as with the profile-based architecture validation or model comparison between different releases. However, they do suggest that the approach has potential for helping to further increase model comprehension.

5.6 Summary

To summarize, the goals of the case study were achieved using the model processing approach. It was shown that model analysis subprocess of the maintenance process can be established using a set of model processing operations. More importantly, the tools to support the approach were built and tried out in real-life architecture models. The profile-based validation of platform architecture and the comparison of the architectures of two platform releases were both found to be useful. According to the feedback gained from the Nokia architects involved, the results obtained by the tools were found to be significant for the further development of the product platform. Most notably, the environment was set up and used also on site by the architects themselves. The anecdotal evidence gathered during the case study suggested that the tasks helped the software architects to pinpoint the potential problem areas of the target architecture model and to act upon them accordingly, as well as tuning the profiles and RE-process further.

The experience gained during the establishement, deployment, and application of the environment demonstrates that methods used for architecture-centric software development and maintenance are heavily influenced by the particular context they are applied to. Therefore, it is necessary that the tools belonging to the environment are reconfigurable and modifiable so that they can be conveniently adapted to a new domain. The model processing approach with a domain-independent set of operations was found to be a feasible solution for achieving this degree of customizability.

Chapter 6

Related Research

Model transformations in the general sense have been a long-term research topic. However, the research has mainly focused on individual model transformations rather than on providing a complete framework, and typically downplayed the importance of diagrams and compliance to the selected modeling language. Perhaps the closest to the presented approach comes the research carried out in the area of academic UML CASE tool suites, like FUJABA[1] (see e.g. Burmester *et al* [7] and Köhler *et al* [25]), and SMW[2] (Porres [43]) and the CORAL[3] modeling framework built on top of SMW (see e.g. Alanen and Porres [3]). In what follows, some related research on different model processing categories is presented.

Conformance operations. UML has been widely used for designing and describing software models. However, effective approaches and tool support for UML-based architecture modeling has been lacking, as pointed out by Medvidovic *et al* [30]. UML profiles have mainly been used for introducing domain specific concepts into UML (e.g. EJB profile by OMG [40]) and for introducing the concepts of architecture description languages into UML, as presented by Zarras *et al* [62], and Hudaib and Montangero [19]. An example of an alternative approach that modifies the UML metamodel is illustrated by Kandé and Strohmeier [23]. More in line with the work presented in this thesis, Egyed and Medvidovic [14] discuss mapping ADL specifications to UML models using a view integration process. They also define conformance and consistency relationships, but between UML models, not between profiles and views. In contrast, the approach defined in this thesis uses a profile hierarchy and a set of validation rules to establish a language for defining

[1] http://www.fujaba.de
[2] http://mde.abo.fi/tools/SMW/
[3] http://mde.abo.fi/tools/Coral/

domain specific architecture conventions. The approach focuses on giving support for configurable validation processes that can be customized to support the constraints and conventions of a given product line or domain. In addition, it relies solely on UML.

Transformation operations. Individual transformation operations have been studied by various researchers. Synthesis of statechart diagrams from collaboration diagrams are discussed by Schönberger *et al* [49] and Khriss *et al* [24]. A method for automatic generation of UML statecharts from sequence diagrams is introduced by Whittle and Schumann [59]. Moreover, the synthesis of statechart diagrams from trace diagrams is presented by Koskimies *et al* [27] and Systä [55] The synthesis of class diagrams from object diagrams is given by Engels *et al* [15]. Transforming scenarios to static model information is discussed by Egyed [11] and Nørmark [32]. Biermann and Krisnashwamy present the synthesis of programs from their traces [4], suited for pseudocode generation. The generation of code from state machines is a well-known technique, and exploited in many tools, e.g. Rhapsody[4]. Of existing commercial UML CASE tools, IBM Rational Rose is also able to perform transformations between sequence and collaboration diagrams. All these sources go into detail on specifying individual transformation operations, whereas the work presented in this thesis aims at identifying, categorizing, and bringing together the operations to form a comprehensive framework of transformations. In addition to the abovementioned operations, this work aims at completing the set of operations by also addressing the supported and weak tranformations.

Set operations. Techniques similar to the UML set operations presented in this thesis have been described by Ohst *et al* [33], and by Porres and Alanen [44]. The former discuss visualizing the differences between two UML diagram versions, while the latter formalize how to calculate the union and difference of UML models. However, both assume that there exists unique repository identifiers and that the models result from a common ancestor. The set operations can also be seen as a composition and decomposition mechanism for model fragments, each representing a single aspect or concern. Such mechanism is assumed in e.g. Subject-Oriented Design, as pointed out by Clarke *et al* [10], and described in the Theme/UML approach [5]. IBM Rational XDE[6] provides most support for merging UML models,

[4] http://www.ilogix.com/rhapsody/rhapsody.cfm/

[5] http://www.dsg.cs.tcd.ie/index.php?category_id=355

[6] http://www-306.ibm.com/software/awdtools/developer/plus/

but its functionality for deriving the correspondence relationship between model elements is proprietary, preventing the user from affecting it. Similarly, IBM Rational Rose has a model integrator tool for model differencing and merging. The set operations have been successfully used for comparing the reverse engineering capabilities of different UML CASE tools (Kollman *et al* [26]) and for comparing Web service descriptions (Jiang *et al* [22]).

Projection operations. Examples of compression operations are presented e.g. by Dósa Rácz and Koskimies [45], and by Egyed and Kruchten [13]. Refactoring operations are natural candidates for projection operations. A good introduction to refactoring and refactoring patterns books is presented by Fowler [17]. Design and architecture patterns (e.g. Gamma *et al* [18], Buschmann *et al* [8]) are a well-established mechanism for collecting the industry best practices and common know how.

General model processing tools. In general, several CASE tools support model synthesis, checking, and merging to some degree. From a model checking perspective, a transformation operation defines the information that must be shared by consistent diagrams. Bodeveix *et al* [5] introduce a tool called NEPTUNE for checking the consistency of different UML models, where the consistency criterion is defined in terms of (extended) OCL ([38] Sect. 6). For example, to check that a class diagram is consistent with a sequence diagram, an OCL expression, which applies the same principles as our transformation operation to infer possible conflicts, can be given. Hence NEPTUNE assumes the additional specification of consistency rules whereas we rely only on the information included in the UML model itself. Prosa[7] supports interactive checking between UML diagrams, and contains a model analyzer, and simulation tools. ILogix Rhapsody provides the designer with the possibility of observing the execution of an instrumented system via animation of UML models.

Many current UML CASE tools, both commercial (e.g. Rational Rose, Together[8], Poseidon[9]) and non-commercial (e.g. ArgoUML[10]) offer extensibility and interoperability capabilities, for example, by providing a proprietary API for model repository access, by introducing a scripting language, or by providing libraries for tool developers. However, these CASE tool dependent solutions are not generally well-suited for performing a chain of transactions

[7] http://www.prosa.fi
[8] http://www.borland.com/together/designer/
[9] http://www.gentleware.com
[10] http://argouml.tigris.org

or queries on the models. One of the main goals of xUMLi is to support the combining of small model operations to achieve high-level functionality, customizable for a given process, domain, or a platform.

Of the existing UML model processing platforms, the UML all purpose transformer (UMLAUT)[11], the OCL4Java library of the Kent Modeling Framework (KMF)[12], and the FUJABA, SMW, and Coral tools come close to our approach. In comparison, xUMLi supports COM automation and is thus not restricted to any particular programming language, allowing the development of practically any kind of components (e.g. user interaction, connection to an external tool). In addition, it involves a dedicated OCL interpreter for querying the models. The platform does not involve a CASE tool itself, but is intended to be integrated with arbitrary CASE tools.

[11] http://www.irisa.fr/UMLAUT/
[12] http://www.cs.kent.ac.uk/projects/kmf/

Chapter 7

Introduction to the Included Publications

> Let me explain it to you. Our work is like this donut. Sure, it's all fluff, has no nutritional value and there's a big gaping hole in the middle... **But**, if you sugar coat it, and add colorful little sprinkles to it, people will eat it up.
>
> – Mike Slackenerny in "Piled Higher and Deeper"

This thesis comprises the introduction part and the eight included publications. In the following, the included publications are described in more detail with the emphasis on the author's contributions on them.

In paper [I], the author jointly contributed to the discussion on how transformation operations can be defined between UML diagrams of different types, what kinds of general approaches for defining the transformations exist, and how the different transformation operations can be categorized. Further, the author's contributions include defining the most interesting transformation operations in detail and providing the discussion on the relationship among the UML metamodel and diagrams while defining the operations.

Paper [II] discusses the how the conformance operations and the overall model processing operations framework has been applied when building a general architecture model maintenance process, targeting a real-life product platform architecture model for an embedded terminal system product. The paper discusses the results obtained using the operations and how they are interpreted in the context of the case study itself. It acts as a basis for the main evaluation of the approach described in this thesis. The author's contributions include deriving the general maintenance process, participat-

ing in defining the architectural profiles for the particular case study, and constructing the model analysis process. Further, the author designed and conducted the architecture validation and comparison tasks.

Paper [III] describes the integrated enviroment facilitating the architecture design, reconstruction, and maintenance for targeted software product lines. From the point of view of this thesis, it describes how the model processing operation framework and its implementation has been integrated into the larger context of a software architecting environment and places the approaches into a larger context. The author's contributions include describing the model analysis and processing toolset, and jointly describing the application of the framework.

Paper [IV] discusses techniques for extending the UML profile mechanism for expressing domain-specific architectural constraints and conventions. The paper presents a general schema for arranging architectural profiles and introduces a set of conformance operations that define how the profiles are interpreted, constituting a profile definition language for validating architectural views against profiles. Furthermore, the paper presents a concrete tool implementation together with an initial case study for assessing the feasibility of the approach. The author's contributions include defining the architectural conformance rules, designing the architecture validation toolkit and evaluating the approach on an example software system.

Paper [V] discusses how VISIOME can be used for combining the UML model operations to construct new operations with high-level functionality, and how to describe often-repeated software engineering tasks and to automate them using the approach. The paper further presents various usage scenarios and software engineering tasks. The author's contributons include the joint work of establishing the model processing approach and the presented usage scenarios, as well as presenting the model operations part of the paper.

Paper [VI] shows how a use case description, addressing a particular system functionality with a set of UML sequence diagrams, can be transformed into an intermediate statechart diagram, and further into an implementation scheme, presented in structured pseudocode. The paper further presents a prototype implementation of the transformation operations. The author's contributions include specifying and implementing the sequence diagram to class diagram transformation and pseudocode generation functionality, and establishing the overall synthesis process.

Paper [VII] describes the overall model processing approach, the UML processing platform, and the implementation of artDECO on top of this platform. It is a sequel to the paper by Airaksinen *et al* [2]. The author's con-

tribution includes presenting a synthesis and summarizing the overall model processing approach.

Paper [VIII] describes the concept of UML set operations for composing and decomposing models, and for creating transient views of the system for increased model comprehension. The paper defines the operations, gives an example, and also places the technique in context with analogous techniques, namely the composition relationship of Subject Oriented Design. The paper is exclusively contributed by the author.

Chapter 8

Conclusions

> Those are my operations, and if you don't like them... well, I have others.
>
> – Adaptation of Groucho Marx

This thesis proposed a set of model processing operations for manipulating architecture and design level software engineering models. It was shown how dependencies between models can be exploited when building the operations, and how they can be combined to form model processing tasks with higher level of functionality. It was further shown that the operations can be meaningfully used in real-life software engineering. The introduced techniques can be defined and implemented using UML as the target modeling language and used as a basis for tool support in computer aided software engineering environments.

8.1 Thesis Questions Revisited

The questions presented in Chapter 1 are returned to and addressed.

Synthesis of Models. The information implied by existing models can be expressed with another modeling paradigm using the transformation operations, if the semantic relationship between the paradigms is strong enough. With suitable projection operations, the model can also be restructured according to given criterion to emphasize an alternative viewpoint.

- *How to describe the information implied by an existing model using another modeling paradigm to express a different point of view (e.g. the*

structure model implied by a behavior model)? Transformation operations can be used for the synthesis of new models expressed using a different modeling paradigm. For example, the dynamic execution traces gathered during the ISA case study were transformed to a structure model.

- *How to compose a model according to a given composition criteria in order to emphasize an alternative point of view (e.g. architectural concern)?* Projection operations can be used to produce a new transient model of a system model by restructuring it according to different composition criteria. For example, the ISA architecture model was recomposed along layers during the case study.

Merging and Slicing of Models. The set operations provide a composition mechanism for merging together model increments. When accompanied by suitable transformation operations, the set operations can be further used for merging together diagrams and model fragments describing different concerns and views to the system, expressed using different modeling paradigms.

- *How to allow different stakeholders to introduce model increments to the system model?* Designers can introduce new model increments to the system model by using the union set operation, and when necessary, tranformation operations. While the ISA case study worked with complete reverse engineered models, it exploited similar mechanisms for merging together model increments.

- *How to support model comprehension by comparing models describing the system from different perspectives, possibly using different modeling paradigms (e.g. static and dynamic views)?* Models can be transformed and compared against each other using set operations, supported by suitable highlighting and transformation operations. These mechanisms were used when slicing the ISA architecture model against execution traces.

Checking of Models. The conformance operations can be used to check that the concepts and their relationships in a model are allowed by profiles capturing domain or architecture specific constraints and conventions. Similarly, system composition can be checked against a given composition criteria. Using the set operations together with transformation and projection operations, it is possible to check different model fragments against each other.

- *How to ensure that the concepts and their relationships in the system model are in agreement with domain specific conventions, rules, and restrictions?* By expressing the conventions, rules, and restrictions using architectural profiles, the conformance operations can be used for validating system models. The profile-based validation of ISA platform architecture model was based on these techniques.

- *How to confirm that architecture or design level models at (e.g. different stages of evolution) are in agreement with each other?* With set operations, models can be sliced and compared against each other. These mechanisms were used when comparing different ISA platform releases against each other.

While the set of questions answered is by no means complete, it goes to show that the presented model processing approach can be used for addressing relevant software engineering problems. The presented questions are general enough to be applied to different problem domains.

8.2 Future Work

While the model processing approach has been implemented and deployed in practice, there remains several topics for future research. A new version of the model processing platform is being actively developed and a major release is expected to be released during the first half of 2005. After the initial positive experiences, work on further case studies on the ISA platform and another similar system will continue, both with forward engineering and reverse engineering architecture models.

The scope of this study—processing of architecture and design level software engineering models—is broad. Consequently, the thesis only covers some aspects of the topic in detail while leaving others open for future research. Some of the future work topics include the following:

- Develop the model processing platform and underlying tool support further. Support the VISIOME approach by building a library of useful scripts and associated model processing operations. As our experiences show that the model processing approach is significantly influenced by the context it is applied to, it is important to ensure tool reconfigurability. This also includes providing additional tools for building the model processing tasks, related to e.g. user interaction and visualization techniques.

- Examine how to build stronger tool support for different software processes, organizations, and domains, and evaluate the usability of the operations especially during forward engineering.
- Continue developing the architecture maintenance process. One future direction for extending the approach is to provide tools and techniques for combining static and dynamic analysis to maintain the consistency between dynamic and static views of the architecture. Another interesting continuing research topic is the monitoring of architecture evolution.
- Examine the possibility of developing a reference profile engine for interpreting standard UML profiles and integrating it with the xUMLi platform for further increasing the generality and configurability of the profile based architecture validation approach.
- Examine the possibilities of extending the profile approach to new domains, like aligning the model composition mechanisms provided by the set and projection operations with the aspect oriented design (AOD) paradigm. Another example is to place the model processing approach in the context of OMG Model Driven Development and Engineering (MDD/MDE) movement (e.g. Miller and Mukerji [31]). Evaluate the applicability of the model processing approach and different model processing operations on additional domains. One example is the ongoing work on generating high-level architecture views based on different user-provided criteria.

Some work has already been completed on providing additional tools for building the tasks. Initial steps on integrating the approach with a dedicated software specification methodology have been addressed by Pitkänen and Selonen [42] in the context of the DisCo method.[1] The relationship between profile mechanims and architectural and design pattern concepts has been explored by Selonen *et al* [51]. The set operations have been successfully used for comparing the reverse engineering capabilities of different UML CASE tools (Kollman *et al* [26]) and for comparing Web service descriptions (Jiang *et al* [22]).

8.3 Concluding Remarks

The goal of the research partly presented in this thesis has been to solve practical software engineering problems. The presented research aims not solely

[1] http://disco.cs.tut.fi

to explore the theoretical foundations of the model processing approach, but also to implement and provide concrete tools to be deployed and used by software developers.

Commercial software development projects do not usually conform to the clean textbook examples. Time-to-market pressure, large organizations with hundreds of developers with varying skills, the legacy software systems, the product-lines and platforms, proprietary middleware and hardware, and the requirement to produce software artefacts of the right quality in financial terms all make it difficult to apply rigorous methods. Therefore, when not dealing with safety critical applications, there is a need for scalable architecture and design level tool support that can be configured by the designers to meet the requirements of the process, project, domain and organization in question. The presented approach aims at providing such support. The thesis itself presents primitive model processing operations that are expected to be among the core components when building the higher level tool support.

During the course of the research, it became evident that it is not feasible to give rigid definitions for individual model processing operations. This is not only a technological issue. Experience with applying the operations has shown that the operations have to be parametrized at the UML level for each usage context individually: it is therefore necessary to keep the operations and their implementation customizable. A one-to-one translation schema forces the operations to be tied to a particular context, forces them to lose their generality, and clutters their definitions with modeling language specific details. The model processing approach provides the necessary flexibility to address this issue. Tool support and customizability are key goals of the approach and they have been achieved to a certain degree.

To summarize, there is a need for building tool support for architecture and design level modeling to meet the requirements of a particular process, domain, product line, or product platform. Model processing operations provide building blocks for constructing such tools. By combining the operations, it is possible to derive with numerous usage scenarios for dealing with different problem areas in software engineering. As the development of the approach has been driven by the practical needs in industry, strong cooperation with both the industry and academia has been a key driver in the research and development process. The experiences gained while applying the approach to large-scale industrial software systems have been promising.

Bibliography

[1] J. Airaksinen. artDECO — UML-pohjaisten profiililähtöisten ohjelmistoarkkitehtuurien tarkastustyökalu. Master's thesis, Tampere University of Technology, September 2004. In Finnish.

[2] J. Airaksinen, K. Koskimies, J. Koskinen, J. Peltonen, P. Selonen, M. Siikarla, and T. Systä. xUMLi: Torwards a Tool-independent UML Processing Platform. In K. Østerbye, editor, *Proceedings of the 10th Nordic Workshop on Programming and Software Development Tools and Techniques, NWPER'2002*, pages 1–15. Copenhagen, Denmark, IT University of Copenhagen, August 2002.

[3] M. Alanen and I. Porres. The Coral Modelling Framework. In K. Koskimies, L. Kuzniarz, J. Lilius, and I. Porres, editors, *Proceedings of the 2nd Nordic Workshop on the Unified Modeling Language NWUML'2004*, TUCS General Publications, pages 93–98. TUCS Turku Centre for Computer Science, July 2004.

[4] A. Biermann and R. Krishnaswamy. Constructing Programs From Example Computations. *IEEE Transactions on Software Engineering*, 3(2):141–153, 1976.

[5] J.-P. Bodeveix, T. Millan, C. Percebois, C. L. Camus, P. Bazex, and L. Feraud. Extending OCL for verifying UML Models Consistency. In L. Kuzniarz, G. Reggio, J. Sorrouille, and Z. Huzar, editors, *Proceedings of UML'2002 Workshop on Consistency Problems in UML-based Software Development*, pages 75–90. Ronneby, Sweden, Blekinge Institute of Technology Research Report 2002:06, 2002.

[6] G. Booch. *Object-Oriented Design with Applications.* Benjamin/Cummings Publishing Company, 1991.

[7] S. Burmester, H. Giese, J. Niere, M. Tichy, J. Wadsack, R. Wagner, L. Wendehals, and A. Zündorf. Tool Integration at the Meta-Model

Level within the FUJABA Tool Suite. In *Proceedings of the Workshop on Tool-Integration in System Development (TIS), Helsinki, Finland, (ESEC / FSE 2003 Workshop 3)*, pages 51–56, September 2003.

[8] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: System of Patterns.* John Wiley and Sons, 1996.

[9] S. Clarke. *Composition of Object-Oriented Software Design Models.* PhD thesis, Dublin City University, January 2001.

[10] S. Clarke, W. H. Harrison, H. Ossher, and P. L. Tarr. Subject-Oriented Design: Towards Improved Alignment of Requirements, Design, and Code. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'99)*, pages 325–339, 1999.

[11] A. Egyed. Integrating Architectural Views in UML. Technical Report USCCSE-99-514, University of Southern California, 1999.

[12] A. Egyed. Automated Abstraction of Class Diagrams. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(4):449–491, 2002.

[13] A. Egyed and P. B. Kruchten. Rose/Architect: A Tool to Visualize Architecture. In *Proceedings of the 32nd Annual Hawaii Conference on Systems Sciences (HICSS-32)*, 1999.

[14] A. Egyed and N. Medvidovic. Extending Architectural Representation in UML with View Integration. In R. France and B. Rumpe, editors, *Proceedings of the Second International Conference on the Unified Modeling Language, UML'99*, pages 2–16. Fort Collins, CO, USA, Springer, October 1999.

[15] G. Engels, R. Heckel, G. Taentzer, and H. Ehrig. A Combined Reference Model and View Based Approach to System Specification. *International Journal of Engineering and Knowledge Engineering*, 4(7):457–477, 1997.

[16] A. Evans and S. Kent. Core Meta-Modelling Semantics of UML: The pUML Approach. In R. France and B. Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *LNCS*, pages 140–155. Springer, 1999.

[17] M. Fowler. *Refactoring – Improving the Design of Existing Code.* Addison-Wesley, 1999.

[18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Orented Software.* Addison-Wesley, 1995.

[19] A. Hudaib and C. Montagero. A UML Profile to Support the Formal Presentation of Software Architecture. In *Proceedings of 26th International Computer Software and Applications Conference (COMPSAC 2002), Prolonging Software Life: Development and Redevelopment*, pages 217–223. Oxford, England, IEEE CS Press, August 2002.

[20] J. Ivers, P. Clements, D. Garlan, R. Nord, B. Schmerl, and J. Silva. Documeting Component and Connector Views with UML 2.0. Technical Report CMU/SEI-2004-TR-008, Software Engineering Institute (SEI), April 2004.

[21] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard. *Object-Oriented Software Engineering. A Use Case Driven Approach.* Addison-Wesley, 1992.

[22] J. Jiang, J. Lipponen, P. Selonen, and T. Systä. UML-level support for analyzing and comparing Web service descriptions. In *Conference on Software Maintenance and Re-engineering (CSMR'05)*, 2005. To appear as a short paper.

[23] M. M. Kandé and A. Strohmeier. Towards a UML Profile for Software Architecture Descriptions. In A. Evans, S. Kent, and B. Selic, editors, *Proceedings of UML 2000 - The Unified Modeling Language, Advancing the Standard, Third International Conference*, volume 1939 of *Lecture Notes in Computer Science*, pages 513–527. York, UK, Springer, 2000.

[24] I. Khriss, M. Elkoutbi, and R. K. Keller. Automating the synthesis of UML StateChart diagrams from multiple collaboration diagrams. In J. Bézivin and P.-A. Muller, editors, *The Unified Modeling Language, UML'98 - Beyond the Notation. First International Workshop, Mulhouse, France, June 1998, Selected Papers*, volume 1618 of *LNCS*, pages 132–147. Springer, 1999.

[25] H. Köhler, U. Nickel, J. Niere, and A. Zündorf. Integrating UML Diagrams for Production Control Systems. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE'00)*, pages 241–251. Limerick, Ireland, ACM Press, June 2000.

[26] R. Kollman, P. Selonen, E. Stroulia, T. Systä, and A. Zündorf. A Study on Current State of the Art in Tool-Supported UML-Based Static Reverse Engineering. In *Proceedings of the 9th Working Conference of Reverse Engineering (WCRE'2002)*, pages 22–34. IEEE CS Press, October–November 2002.

[27] K. Koskimies, T. Männisto, T. Systä, and J. Tuomi. Automated Support for Modeling of OO Software. *IEEE Software*, pages 87–94, January/February 1998.

[28] J. Koskinen, J. Peltonen, P. Selonen, T. Systä, and K. Koskimies. Model Processing Tools in UML. In *Proceedings of ICSE'01*, pages 819–820. IEEE CS Press, May 2001. Formal Research Tool Demo.

[29] J. Koskinen, J. Peltonen, P. Selonen, T. Systä, and K. Koskimies. Towards Tool Assisted UML Development Environments. In T. Gyimóthy, editor, *Proceedings of the 7th Symposium on Programming Languages and Tools (SPLST'01)*, pages 1–15. University of Szeged, June 2001.

[30] N. Medvidovic, D. Rosenblum, D. Redmiles, and J. Robbins. Modeling Software Architecture in the Unified Modeling Language. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 1(11):2–57, January 2002.

[31] J. Miller and J. Mukerji. MDA Guide Version 1.0.1. Technical report, The Object Management Group, June 2003. On-line at http://www.omg.org/mda.

[32] K. Nørmark. Synthesis of Program Outlines from Scenarios in DYNAMO, 1998. On-line at http://www.cs.auc.dk/∼normark/dynamo.html.

[33] D. Ohst, M. Welle, and U. Kelter. Differences Between Versions of UML Diagrams. In *Proceedings of the 9th European software engineering conference held jointly with 10th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 227–236. ACM Press, 2003.

[34] OMG. Request for Proposal: UML 2.0 Superstructure RFP, August 2001. ad/00-09-02.

[35] OMG. Meta Object Facility (MOF) Specification, Version 1.4, April 2002. On-line at http://www.omg.org/mof.

[36] OMG. Request for Proposal: MOF 2.0 Query / Views / Transformations RFP, April 2002. ad/2002-04-10.

[37] OMG. *Meta Object Facility (MOF) 2.0 Core Specification.* Object Management Group, October 2003. On-line at http://www.omg.org/-technology/documents/modeling_spec_catalog.htm.

[38] OMG. OMG Unified Modeling Language Specification, Version 1.5 (formal/03-03-01), March 2003. On-line at http://www.omg.org/uml.

[39] OMG. Unified Modeling Language: Superstructure version 2.0 Final Adopted Specification ptc/03-08-02, August 2003. On-line at http://www.omg.org/uml.

[40] OMG. UML Profile for Enterprise Distributed Object Computing (EDOC), version 1.0, February 2004. On-line at http://www.omg.org/uml.

[41] J. Peltonen. Visual Scripting for UML-Based Tools. In *Proceedings of ICSSEA 2000*, December 2000.

[42] R. Pitkänen and P. Selonen. A UML Profile for Executable and Incremental Specification-Level Modeling. In T. Baar, A. Srohmeier, A. Moreira, and S. J. Mellor, editors, *≪UML≫ 2004 – The Unified Modeling Language: Modeling Languages and Applications*, LNCS 3273, pages 158–172. Springer, October 2004.

[43] I. Porres. A Toolkit for Manipulating UML Models. Technical report, TUCS Turku Centre for Computer Science, January 2002.

[44] I. Porres and M. Alanen. Difference and Union of Models. In P. Stevens, J. Whittle, and G. Booch, editors, *UML 2003 - The Unified Modeling Language, Modeling Languages and Applications, 6th International Conference, San Francisco, CA, USA, October 20-24, 2003, Proceedings*, volume 2863 of *Lecture Notes in Computer Science*, pages 2–17. Springer, 2003.

[45] F. D. Rácz and K. Koskimies. Tool-Supported Compression of UML Class Diagrams. In R. France and B. Rumpe, editors, *UML'99 - The Unified Modeling Language, Fort Collins, CO, USA, October, 1999, Proceedings*, Lecture Notes in Computer Science, pages 172–187. Springer, 1999.

[46] C. Riva. Reverse Architecting: an Industrial Experience Report. In *Proceedings of the 7th Working Conference of Reverse Engineering (WCRE'2000)*, pages 42–51. IEEE CS Press, 2000.

[47] C. Riva, J. Xu, and A. Maccari. Architecting and Reverse Architecting in UML. In A. Brown, W. Kozaczynski, P. Kruchten, and G. Larsen, editors, *Proceedings of ICSE 2001 Workshop for Describing Software Architecture with UML*, pages 88–93. Toronto, Ontario, Canada, IEEE CS Press, May 2001.

[48] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

[49] S. Schönberger, R. Keller, and I. Khriss. Algorithmic Support for Model Transformation in Object-Oriented Software Development. *Concurrency and Computation: Practise and Experience*, 13(5):351–383, 2001.

[50] P. Selonen, K. Koskimies, and M. Sakkinen. How to Make Apples from Oranges in UML. In *Proceedings of the 34th Hawaii International Conference on System Sciences (HICSS-34)*. IEEE CS Press, January 2001. CD-ROM.

[51] P. Selonen, M. Siikarla, K. Koskimies, and T. Mikkonen. Towards the Unification of Patterns and Profiles in UML. *Nordic Journal of Computing*, 11(3):235–253, 2004.

[52] M. Siikarla. Implementation of a Component-Based Visual Scripting Language. Master's thesis, Tampere University of Technology, January 2003.

[53] M. Siikarla, J. Peltonen, and P. Selonen. Combining OCL and Programming Languages for UML Model Processing. In P. H. Schmitt, editor, *Proceedings of the Workshop, OCL 2.0 – Industry Standard or Scientific Playground*, volume 102 of *Electric Notes in Theoretical Computer Science (ENTCS)*, pages 175–194. Elsevier, 2004.

[54] D. Soni, R. L. Nord, and C. Hofmeister. Software Architecture in Industrial Applications. In *Proceedings of International Conference on Software Engineering ICSE 1995*, pages 196–207. Seattle, Washington, USA, April 1995.

[55] T. Systä. *Static and Dynamic Reverse Engineering Techniques for Java Software Systems*. PhD thesis, University of Tampere, 2000.

[56] P. Tarr, H. Ossher, W. Harrison, and S. M. S. Jr. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proceedings of the International Conference on Software Engineering (ICSE 21)*, pages 107–119. Los Angeles, CA, USA, ACM Press, 1999.

[57] S. Tolvanen. Development Environment for Visual Programing Languages. Master's thesis, Tampere University of Technology, Institute of Software Systems, 2004.

[58] J. van der Ven. An Implementation of Set Operations on UML Diagrams. Master's thesis, Rijksuniversiteit Groningen, Instituut voor Wiskunde en Informatica, 2004.

[59] J. Whittle and J. Schumann. Generating Statechart Designs from Scenarios. In *Proceedings of ICSE'00*, pages 314–323. Limerick, Ireland, IEEE CS Press, June 2000.

[60] J. Wikman. Evolution of a Distributed Repository-Based Architecture. In *Proceedings of the first Nordic Workshop on Software Architecture (NOSA'98)*, 1998.

[61] E. Yourdon. *Modern Structured Analysis*. Yourdon Press, 1988.

[62] A. Zarras, V. Issarny, C. Kloukinas, and V. K. Kguyen. Towards a Base UML Profile for Architecture Description. In A. Brown, W. Kozaczynski, P. Kruchten, and G. Larsen, editors, *Proceedings of ICSE 2001 Workshop for Describing Software Architecture with UML*, pages 22–26. Toronto, Ontario, Canada, IEEE CS Press, May 2001.

[I] P. Selonen, K. Koskimies, and M. Sakkinen. Transformations Between UML Diagrams. *Journal of Database Management*, 3(14):37–55, 2003.

[II] C. Riva, P. Selonen, T. Systä, and J. Xu. UML-based Reverse Engineering and Model Analysis Approaches for Software Architecture Maintenance. In *Proceedings of the International Conference on Software Maintenance (ICSM'04)*, pages 50–59. IEEE Computer Society, September 2004.

[III] C. Riva, P. Selonen, T. Systä, A.-P. Tuovinen, J. Xu, and Y. Yang. Establishing a Software Architecting Environment. In *Proceedings of the 4th Working IEEE/IFIP Conference on Software Architecture (WICSA'04)*, pages 188–200. IEEE Computer Society, June 2004.

[IV] P. Selonen and J. Xu. Validating UML Models Against Architectural Profiles. In *Proceedings of the 9th European Software Engineering Conference held jointly with 10th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE 2003)*, pages 58–67. ACM Press, 2003.

[V] J. Peltonen and P. Selonen. Processing UML Models with Visual Scripts. In *Proceedings of the 2001 Human-Centric Computing Languages and Environments (HCC'01)*, pages 264–271. IEEE Computer Society, September 2001.

[VI] P. Selonen, T. Systä, and K. Koskimies. Generating Structured Implementation Schemes from UML Sequence Diagrams. In L. QiaYun, R. Riehle, G. Pour, and B. Meyer, editors, *Proceedings of the 39th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS USA 2001)*, pages 317–328. IEEE Computer Society, July-August 2001.

[VII] J. Peltonen and P. Selonen An Approach and a Platform for Building UML Model Processing Tools. In *Proceedings of the ICSE'04 Workshop on Directions of Software Engineering Environments (WoDiSEE'04)*, pages 51–57, May 2004.

[VIII] P. Selonen. Set Operations for the Unified Modeling Language. In P. Kilpeläinen and N. Päivinen, editors, *Proceedings of the 8th Symposium on Programming Languages and Tools (SPLST'03)*, pages 70–81. University of Kuopio, June 2003.