DISSERTATION

AN APPROACH TO COMPOSING ASPECT-ORIENTED DESIGN MODELS

Submitted by

Y. Raghu Reddy

Department of Computer Science

In partial fulfillment of the requirements

for the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Summer 2006

COLORADO STATE UNIVERSITY

12 April, 2006

WE HEREBY RECOMMEND THAT THE DISSERTATION PREPARED UNDER OUR SUPERVISION BY Y. RAGHU REDDY ENTITLED AN APPROACH TO COMPOSING ASPECT-ORIENTED DESIGN MODELS BE ACCEPTED AS FULFILLING IN PART REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY.

Committee on Graduate Work

Sudipto Ghosh

Sudipto Ghosh, Ph.D., Committee Member

James Bieman, Ph.D., Committee Member

Daniel E. Turk, Ph.D., Committee Member

Robert B. France, Ph.D., Advisor

L. Darrell Whitley, Ph.D., Department Head

ABSTRACT OF DISSERTATION

AN APPROACH TO COMPOSING ASPECT-ORIENTED DESIGN MODELS

Developers of complex software systems are often required to address security, fault-tolerance and other extra-functional goals. The features that realize these goals are often spread across and tangled with other features in a design. It can be difficult to modify and evolve these crosscutting features. Localizing crosscutting features in a design can help developers manage the evolution of the features.

The aspect-oriented modeling (AOM) approach supports separation of crosscutting features from other features during design. An aspect-oriented design consists of a primary (base) model and a set of aspect models. A primary model describes the dominant structure of a design and an aspect model describes a feature that crosscuts the dominant structure. The aspect models and the primary model must be composed to obtain a system view that integrates features described by the models.

Composing aspect and primary models using a general-purpose model composition procedure may produce a composed model with undesirable properties. Composition directives can be used to alter how a general-purpose composition procedure composes models in the case where it is known or expected that the procedure will produce incorrect results.

In this thesis, we provide a technique for composing aspect and primary design models consisting of class and sequence models. Model composition involves merging aspect class models with primary class models and incorporating sequences of

interactions specified in aspect models into primary sequence models.

The aspect and primary class models are merged using signatures consisting of model element properties. The signatures provide a more flexible way to merge class models than the name based approach used by other researchers. We describe a general-purpose composition procedure that can be varied using composition directives. The research provides composition metamodels that can be used as the basis for developing model composition tools.

The class model composition metamodel developed in this research is very detailed than the metamodel provided by other researchers. The composition metamodel described in this thesis contains specifications of composition behavior that can be used to implement the composition. We have developed a prototype tool that implements the composition metamodel.

The interaction model composition technique developed in this research for composing aspect and primary sequence models is novel. Composition of sequence models has not been addressed by any other researcher previously.

The class model and sequence model composition techniques are applied in pilot studies to demonstrate the usage of the composition techniques.

Y. Raghu Reddy
Department of Computer Science
Colorado State University
Fort Collins, Colorado 80523
Summer 2006

ACKNOWLEDGEMENTS

I thank my advisor, Dr. Robert France, and other committee members Dr. Sudipto Ghosh, Dr. James Bieman, and Dr. Daniel Turk for their support during my research. I am grateful to Dr.France for his support and guidance in completing this dissertation. He has always been a positive influence through out this research and has motivated me to publish my research in the best conferences and journals. I thank Dr. Ghosh for his input in my research and in writing this report. I thank Dr. Bieman and Dr. Turk for their support during my years at CSU.

I am thankful to Devon Simmonds, Arnor Solberg, Franck Fleurey, Eunjee Song, and Ding Trung for their valuable inputs towards the completion of my disseration. It is my privilege to express my gratitude to these people for their efforts. Thanks to Nilesh Kawane, Dae Kyoo Kim and other friends who have made my stay at CSU a pleasurable experience.

My deepest love and respect go to my parents and brother who have endured a lot of pain to make me who I am today. My heart goes to my wife, Krishna Vandana who has always been supportive of me and has kept me in good spirits during the hard times.

TABLE OF CONTENTS

1 Introduction **1**

1.1 Problem Statement 1

1.2 Research Overview 3

1.3 Scope of Research 6

1.4 Dissertation Structure Overview 8

2 Background **9**

2.1 Unified Modeling Language 9

2.1.1 UML Models 10

2.1.2 The UML Metamodel 12

2.2 Role Based Meta-modeling Language 13

2.2.1 Class Diagram Template 14

2.2.2 Sequence Diagram Template 18

2.3 The Aspect Oriented Model-driven Development Framework (AOMDF) 19

3 Related Work **23**

3.1 Asymmetric composition approaches 24

3.2 Symmetric composition approaches 26

3.3 Hybrid composition approaches 29

3.4 Other AOSD approaches 31

3.5 Summary 34

3.5.1 Composition symmetry 35

3.5.2 Composition procedure 36

3.5.3 Level of Abstraction 37

4 Class Model Composition **39**

4.1 Matching Model Elements 41

4.1.1 Name-based Matching 42

4.1.2 Signature-based Matching 44

4.2 Composition metamodel for class diagrams 48

4.2.1 KerMeta 51

4.2.2 Essential Meta-Object Facilities 53

4.2.3 Implementation of class model composition 55

5 Composition Directives **58**

5.1 Using directives to resolve composition problems 61

5.2 Classification of directives 68

5.3 The Directives 69

5.3.1 Element Directives List 70

5.3.1.1 Creating new model elements 71

5.3.1.2 Adding model elements to a namespace 73

5.3.1.3 Removing model elements from a namespace 74

5.3.1.4 Changing properties of model elements in a namespace 75

5.3.1.5 Replace references to a model element in a namespace 76

5.3.1.6 Overriding a model element 77

5.3.1.7 Overriding default composition rules 78

5.3.2 Element Directives Examples 80

5.3.2.1 Example 1: Buffering Application 80

5.3.2.2 Example 2: User Management Application 83

5.3.3 Combining Element Directives 87

5.3.4 Model Directives 87

5.3.4.1 Precedes 88

5.3.4.2 Follows 88

5.3.5 Weave Ordering Example 89

6 Interaction Model Composition **91**

6.1 Composing message sequences 91

6.1.1 Overview of simpleAspect and compositeAspect Tags 92

6.1.1.1 Binding semantics for simpleAspect Tags 94

6.1.1.2 Binding semantics for compositeAspect Tags 95

6.1.2 Fragments in compositeAspect Sequence Model 97

6.2 Composition metamodel for merging sequence models 101

7 Pilot studies **104**

7.1 Transaction Aspect and Net Banking Application - Pilot Study . . . 104

7.1.1 A Money Transfer Service Primary Model 104

7.1.1.1 Money transfer service class model 105

7.1.1.2 Money transfer service sequence model 105

7.1.2 A Transaction Aspect Model 106

7.1.2.1 A Transaction aspect class diagram template 107

7.1.2.2 A Transaction aspect sequence diagram template . . 109

7.1.3 Applying class model composition 112

7.1.4 Applying interaction model composition 116

7.2 Discussion 120

7.2.1 Consistency of composition techniques 120

7.2.2 Tags in the primary model 123

7.2.3 Effect of signatures on composition directives 124

7.3 Authorization Aspect, Replication Aspect and Banking Application - Pilot Study 126

7.3.1 Banking Application Primary Model 127

7.3.1.1 Banking aplication primary class model 127

7.3.1.2 Banking application primary sequence model 127
7.3.2 An Authorization aspect model 127
7.3.2.1 An Authorization aspect class diagram template . . . 128
7.3.2.2 Authorization aspect sequence diagram template . . 129
7.3.3 A Replicated Repository aspect model 130
7.3.3.1 A Replicated repository class diagram template . . . 131
7.3.3.2 A Replicated repository sequence diagram template . 132
7.3.4 Applying class model composition 133
7.3.5 Applying Interaction model Composition 137

8 Conclusion and Future work **142**
8.1 Contribution 144
8.2 Composing hierarchical structures 145
8.3 Selection of signatures 147
8.4 Validation of the composition techniques 147
8.5 Tool Support 148
8.6 Consistency of Framework 149

A Money Transfer Service with Transaction management **152**

B Merge algorithm **154**

References **157**

LIST OF TABLES

2.1 Example bindings for the template class diagram 17

3.1 Summary of composition procedures 36

3.2 Summary of aspect oriented design approaches 37

3.3 Summary of level of abstraction 38

4.1 Code snippet for getMatchingElements implemented in KerMeta 56

4.2 Code snippets from the merge part involving conflict detection 57

5.1 Bindings for buffering examaple 64

7.1 Bindings for money transfer specific transaction aspect 113

7.2 Bindings for authorization aspect and banking application 134

7.3 Bindings for the authorized banking application and replicated repository aspect 136

LIST OF FIGURES

1.1 An overview of aspect oriented modeling approach 4

2.1 UML four layer metamodel architecture 10
2.2 Example of UML class and sequence diagrams 11
2.3 Example of process described using an UML activity model 12
2.4 A simple metamodel and its instance 13
2.5 Example of a class diagram template 15
2.6 Example of an instantiated class model 18
2.7 Example of a sequence diagram template 19
2.8 Example of an instantated sequence model 20
2.9 Aspect-Oriented Model-driven Development Framework. 21

4.1 An overview of the class model composition in AOM 40
4.2 An example of name-based composition 42
4.3 Composed model obtained using name-based composition 43
4.4 An example of a problem that can arise using name-based composition . 44
4.5 An Example of Model Element Matching and Merging 46
4.6 Composition metamodel for merging design models 49
4.7 The composer part (main) of sequence diagram for composition metamodel 50
4.8 Merge part of sequence diagram for composition metamodel 51
4.9 Kermeta (as shown in [80]) 52
4.10 EMOF classes used in the composition technique 54

5.1 Using composition directives to resolve composition problems 59
5.2 An example of faulty composition 63
5.3 Example of a property conflict 65
5.4 An example of cyclic ordering conflict 67
5.5 Classification of composition directives 68
5.6 Example 1. Before Application of directives. 81
5.7 Example 1. After Application of remove directives. 81
5.8 Example 1. After Application of remove and override directives. 82
5.9 Example 1. Resultant composed model after application of directives. . . 82
5.10 Example 2. Before application of directives. 84
5.11 Example 2. After application of directives. 85
5.12 Example 2. Composed model after application of directives. 86
5.13 Example 4. Specifying Weave Order 89

6.1 Tags on the Primary model 93
6.2 Primary sequence model with simpleAspect tags 95
6.3 An example simpleAspect sequence model 95
6.4 Composed model obtained from simple tagged primary model 96
6.5 Fragments in the compositeAspect sequence model 97
6.6 Primary model with compositeAspect tag 99
6.7 An example compositeAspect sequence model 100
6.8 Composed model obtained from compositeAspect tagged primary model 100
6.9 Interaction model composition metamodel. 102

7.1 A money transfer service primary class model 105
7.2 Primary sequence model describing money transfer scenario 106
7.3 Transaction aspect class diagram template 108
7.4 A transaction aspect sequence model 110

7.5 Sequence diagram for two phase commit protocol 111
7.6 Context-specific transaction aspect class model 114
7.7 Composed model for transaction aspect with money transfer service . . . 115
7.8 Tagged money transfer sequence model 117
7.9 Money transfer sequence model composed with transaction aspect sequence model 118
7.10 Tagging a message in the primary model with multiple aspect sequence models 121
7.11 Tagged money transfer sequence model consistent with class model . . . 122
7.12 Money transfer service primary Model with simpleAspect Lookup Tag . . 123
7.13 Money transfer service primary model with compositeAspect Lookup Tag 124
7.14 Composed model for transaction aspect with money transfer service not well-formed 126
7.15 Banking application primary model 127
7.16 The AddAccount sequence model 128
7.17 A class diagram template for the Authorization aspect model 128
7.18 A sequence diagram template for the authorization aspect model 130
7.19 A class diagram template for the Replicated Repository aspect model . . 131
7.20 A sequence diagram template for the Replicated Repository aspect model 132
7.21 Context-specific authorization aspect class model 135
7.22 Primary class model composed with Authorization aspect class model . . 135
7.23 Context-specific replicated repository aspect class model 136
7.24 Primary class model composed with Authorization and Replicated Repository aspect class models 138
7.25 The tagged AddAccount sequence model 139
7.26 AddAccount sequence model composed with Authorization and Replicated Repository aspect sequence models 140

8.1 Composition of classes with hierarchical structures 146

8.2 Two paths to obtain composed model 150

A.1 Expanded Money transfer sequence model composed with transaction aspect sequence model 153

Chapter 1

Introduction

1.1 Problem Statement

Separation of Concerns (SoC) is a fundamental software engineering principle that can be used to manage complexity, improve understandability, and facilitate evolution of software [20, 23, 52]. It allows a developer to handle concerns individually. A concern can be anything of interest in a product or process [28]. Examples of product based concerns are reliability, efficiency, and usability.

Abstraction is an example of SoC in which only the important details are considered [20]. For example, at the requirements level of abstraction one focuses only on details needed to understand and document software requirements while at the architectural design level of abstraction one focuses on details needed to understand and document the system architecture. Another form of abstraction involves separating technology specific concerns from technology independent concerns. For example, the Model Driven Architecture (MDA) [76] initiative emphasizes modeling techniques that can be used to separate technology specific concerns from technology independent concerns.

Modularization is another SoC mechanism in which the concerns are parts (modules) of a system [20]. Modularization is the process of breaking a software design into distinct parts that overlap in functionality as little as possible [51].

A design can be viewed as an integrated set of modules (e.g., subsystems, components, classes) that are developed in a relatively independent manner and work in concert to accomplish system goals. A decision to modularize the design around a set of concerns may make it difficult to modularize other concerns.

Current modeling approaches provide good support for modularizing design models but poor support for isolating *crosscutting features*, that is, functionality that is spread across the modules of the software and tangled with other functionality. For example, dependability concerns like security, fault-tolerance, and error recovery are often realized in software as crosscutting features. In this thesis, a *feature* is a logical unit of behavior [8], i.e., it is a piece of functionality that addresses a requirement or a set of requirements. Henceforth the term feature will be used to refer to functionality that addresses a concern.

Crosscutting features that address dependability concerns can be problematic for the following reasons:

- Understanding a crosscutting feature can be difficult when the description of the feature is distributed across the modules of the design.
- Changing the features requires making changes in a number of places in the software.
- The above two problems contribute to the difficulty of evaluating alternative forms of crosscutting features that address dependability concerns.

The above problems can be addressed by providing support for separating crosscutting dependability features from other features. At the code level, aspect-oriented programming (AOP) [39] and subject-oriented programming [26] support separation of crosscutting features from other features. Isolation of crosscutting features at design level can help developers better manage complexity during the design phase.

A crosscutting feature that has been isolated needs to be composed with other features to provide an integrated view of the system design. An integrated view of the system design enables one to understand the effects of composing the crosscutting features with other features. For example, a modeler may use the integrated view to implement the system design using a non-AOP language or may choose to implement the crosscutting and others feature individually and then compose them at the code level.

Composing a crosscutting feature with other features requires one to define a composition algorithm that uses some matching criteria to determine the model elements that need to composed. The model elements are said to match if the syntactic properties of the element are the same. For example, elements with the same name are considered matching model elements. Automation of the composition based on syntactic matching may not always produce the desired results, since two elements may match with respect to the syntactic properties but may represent different concepts.

The algorithm used in the composition procedure may be general-purpose or application-specific. A general-purpose algorithm can be used to compose any model element independently, however the composition may produce a model with errors, because the general-purpose algorithm does not take into consideration the application-specific properties. Application-specific composition algorithms can guarantee that the composition will yeild correct results by using application-specific properties to drive the composition but one may need to write different composition algorithms for different applications.

1.2 Research Overview

The aspect oriented modeling (AOM) approach [15, 17, 18] supports separation of crosscutting features from other features during design modeling. The approach can be used to produce designs that can be implemented using non-AOP languages.

In the AOM approach, crosscutting features are treated as patterns [15] described by *(template) aspect models*, and other features are described by a *primary model*. The result of composing aspect and primary models is an integrated design model called the *composed model*. *Bindings* determine where in the primary model the aspect models are to be composed. *Composition directives* determine how the aspect and primary class models are composed [60]. An overview of the AOM approach is shown in figure 1.1.

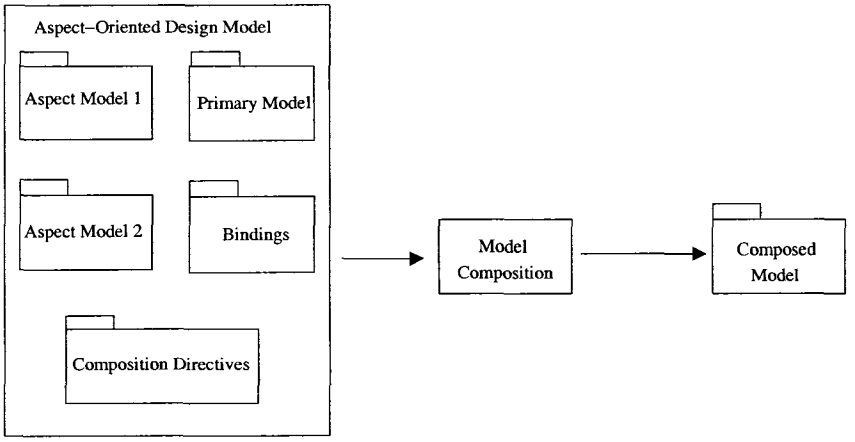


Figure 1.1: An overview of aspect oriented modeling approach

A primary model is described using Unified Modeling Language (UML) [77, 78]. In this research, class models are used to describe structural views and interaction models are used to describe behavioral views. Aspect models are specified using a template version of the Role-Based Metamodeling Language(RBML) [16]. The RBML is used to describe patterns and it's syntax is based on the UML.

An aspect model consists of a class diagram template and sequence diagram templates. The template elements in the aspect models need to be bound using application-specific values before they can be composed with a primary model. The application-specific value can be the name of a primary model element or the name of an application-specific model element that is to be added to the composed model

during composition. In our work, the bindings are explicitly specified for class models, and can be implicitly or explicitly specified for interaction models. For aspect class models, bindings produce a *context-specific aspect class model.* The context-specific aspect class model is composed with the primary class model to obtain an integrated structural view of the system design. The aspect interaction models and the primary interaction model are integrated to obtain an integrated behavioral view of the system design.

Model composition technologies that automate significant parts of the composition activity are needed if the composition procedure is to scale-up to models of complex software systems. There are many ways to support model composition. At one extreme one can develop a general-purpose composition algorithm that composes aspect and primary models without developer intervention. This approach provides very little flexibility in how the crosscutting features are incorporated into the dominant structure. The domain knowledge is not useful while using a general-purposed composition algorithm and this can lead to errors. At the other extreme, developers define application-specific composition procedures that are applied to the aspect and primary models they develop. This approach is very flexible, but requires more effort from developers since different application-specific algorithms need to be developed for different applications. More practical solutions are likely to lie between these two approaches. For example, a tool can codify a default general-purpose composition procedure and allow developers to vary some aspects of the composition procedure. In our work, we provide an approach that utilizes a default composition procedure that can be altered using composition directives.

Model element *signatures* determine how class model elements are merged during composition. A signature is a set of model element properties (e.g., name, attribute. If signatures of the model elements match then they are merged. We have described and implemented a composition algorithm that uses signatures and can be varied

using composition directives. The composition algorithm has been implemented in the KerMeta language [80].

The work described in this thesis builds upon the work by Clarke et al. [12] by providing a detailed metamodel that contains specifications of composition behavior. The composition directives developed in this research are more extensive and well-defined. The use of composition directives and signatures allow modelers to define and apply their own integration and reconciliation strategies.

The composition of interaction models is done by specifying *tags* on the primary sequence model. The tags indicate where and how the aspect sequence models are composed with the primary sequence model.

The AOM is the basis for an Aspect Oriented Model-driven Development Framework (AOMDF) [62, 63, 64] currently being developed at CSU. The AOMDF is an MDD framework that supports modeling of aspect and primary models at different levels of abstraction, transformation from one abstraction level to another, and composition of aspect and primary models. This work contributes to the AOMDF by providing techniques and limited tool support to compose the aspect and primary models.

We demonstrate the utility of the AOM composition techniques by using it to compose aspect models and primary model in pilot studies.

1.3 Scope of Research

In this research we use UML to describe the models because of the widespread usage of UML by practitioners and the availability of a number of tools for UML. The UML version 2.0 models used in this work are class, sequence and activity models. Class models are used to describe the structure and sequence models are used to describe the interactions of design artifacts. Refinement of these models to other levels (e.g., design to code) of abstraction is beyond the scope of this research. Activity models

are used to describe the AOM composition process. They are not used to describe design artifacts.

The research provides techniques for composing class models and composing sequence models. It is assumed that the structural view described by class models and behavioral view described by sequence models are consistent with each other. This research does not provide techniques for establishing the consistency of design views.

In AOM, the design models can be composed to analyze the interactions between the aspect and primary model elements. However, this research does not address model analysis. Existing model checking techniques can be used to analyze the design models. For example, Ivan Porres [53], Jan Jürgens [34, 35] have proposed techniques that can be used to perform model analysis on design models.

The class model composition technique developed in this research uses signatures and composition directives to compose design class models. The composition directives that need to applied depend on the signature chosen for matching the model elements. A systematic process for selecting a particular signature that can optimize the usage of composition directives is beyond the scope of this research.

The composition directives developed in this research may not be comprehensive and are applicable only to class model composition. The application of composition directives to interaction model composition is not a part of this research.

The research does not validate the composition technique using formal experimentation techniques. Instead, the composition technique developed in the research is implemented in the KerMeta language as proof of concept. The composition metamodel developed in this thesis and the implemented metamodel in KerMeta has a one-one correspondence and hence composition is correct with respect to implementation. We present pilot studies that demonstrate how the composition techniques develped in this thesis can be applied. Formal experimentation would require the existence of a more robust tool set which is beyond the scope of this research.

1.4 Dissertation Structure Overview

Chapter 2 presents background information on UML, the RBML, and the AOMDF. Chapter 3 describes related work on composition techniques. Chapter 4 presents the class model composition tehnique and chapter 5 presents the composition directives that can be used to compose class models. Chapter 6 describes the interaction model composition technique. Chapter 7 presents pilot studies in which crosscutting security, fault-tolerance and transaction features are modeled as aspects. Chapter 8 discusses conclusions and outlines directions for future work.

Chapter 2

Background

The purpose of this chapter is to provide background information needed to understand the concepts used in this research. Section 2.1 gives an overview of the UML and a brief description of the model types used in this research. Section 2.2 presents the language used to describe aspect models and section 2.3 gives an overview of the AOMDF that provides the context for this research.

2.1 Unified Modeling Language

The Unified Modeling Language (UML) [75, 77, 78] is a programming language independent notation for specifying, visualizing, constructing, and documenting systems. It is an Object Management Group (OMG) standard language for object-oriented modeling. The UML started out as a modeling language developed collaboratively by Grady Booch, James Rumbaugh, and Ivar Jacobson. Several revisions have been produced since the first release, and the most recent revision, version 2.0 [77, 78], was recently approved by the OMG.

The UML infrastructure is defined as a four-layer architecture (see Figure 2.1).

- Level M3 (meta-metamodel layer) defines a language for specifying metamodels. The Meta Object Facility (MOF) ?? is an example of meta-metamodel.
- Level M2 (metamodel layer) contains models that specify modeling languages.

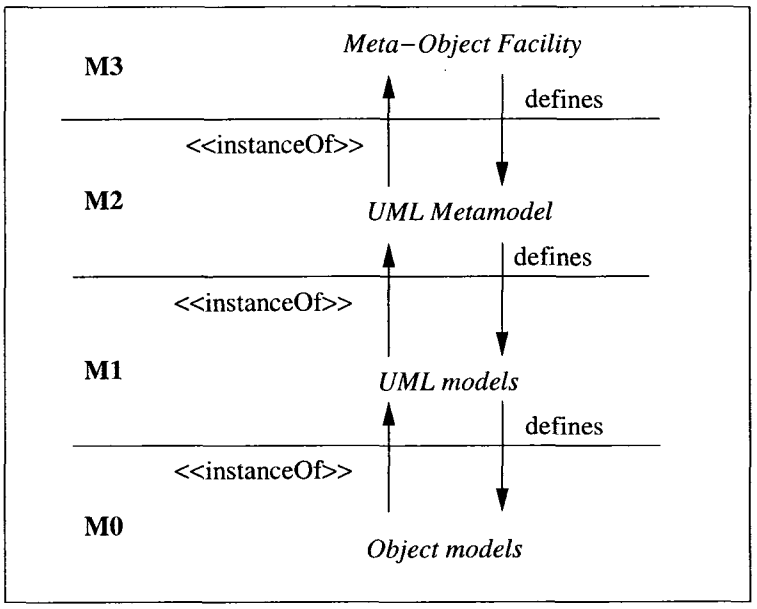


Figure 2.1: UML four layer metamodel architecture

The UML metamodel and the Common Warehouse Metamodel (CWM) [74] are examples of metamodels.

- Level M1 (model layer) contains models that describe semantic domains. The model layer consists of models expressed in languages specified by M2 metamodels.

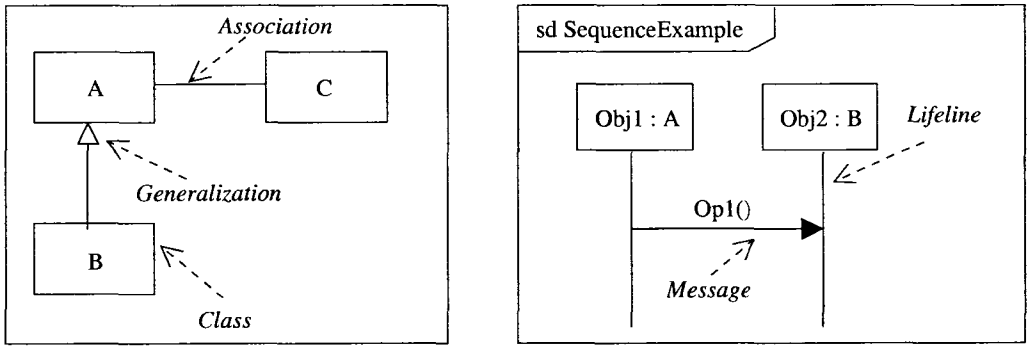
- Level M0 (user data) consists of object configurations specified by the models at level M1.

In this research template aspect models are defined at the M2 level since aspect models are generic descriptions of model families. The context-specific aspect models and primary model are defined at the M1 level.

2.1.1 UML Models

The UML provides several modeling views. Use case models, class models, interaction models, statechart models, and activity models are some of the views provided by the

UML. In this thesis UML models are used in two different ways. Class models and sequence models are used to describe software designs. Activity models are used to describe composition processes.



Class Diagram Sequence Diagram

Figure 2.2: Example of UML class and sequence diagrams

A UML class model consists of a set of classifiers (for example, classes, interfaces) and relationships (for example, association, generalization) between classifiers. Associations between classifiers specify the kinds of links that can exist between class objects. Association multiplicities restrict the number of links possible at the object level. Classifiers may have structural features (for example, attributes) and behavioral features (for example, operations). Operation specifications and constraints on attributes can be expressed using the Object Constraint Language (OCL) [48].

UML interaction models describe how objects in a system interact with each other to accomplish specific goals. Interactions can be described using sequence, interaction overview and communication models. In this work, UML 2.0 sequence models are used for describing interactions. In a sequence model, a collaboration among objects is described in terms of lifelines and messages. A lifeline represents an individual participant in the interaction and the message defines a particular communication between participants of an interaction [78]. Figure 2.2 shows simple examples of the UML class and sequence diagrams.

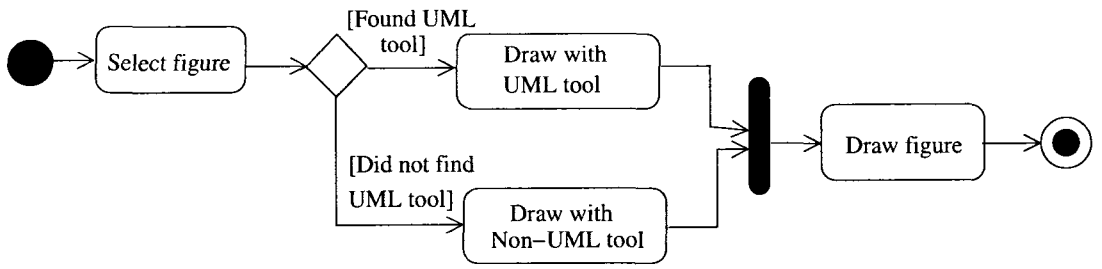


Figure 2.3: Example of process described using an UML activity model

UML Activity models are used to describe the flow of control and data through steps of a computation. They are typically used for modeling processes, for modeling behaviors described by use cases and for describing the behavior of operations. An activity is a structure of actions, where an action is a fundamental unit of behavior specification that represents some transformation or processing in the modeled system [78]. The action may be initated because of completion of other actions, or because objects and data become available. Figure 2.3 shows a simple example of an UML activity model that describes the process of drawing a figure using a UML tool or a non-UML tool.

2.1.2 The UML Metamodel

The UML metamodel characterizes syntactically valid UML models. It consists of a class model and a set of well-formedness rules. The metamodel class model consists of classes whose instances are UML model elements. For example, instances of the metamodel class *Association* are UML associations. Figure 2.4 shows a part of the metamodel and an example instance (i.e. a UML model). Meta-classes[1] may have attributes referred to as meta-attributes. For example, the meta-class *Class* has an *isAbstract* meta-attribute that determines whether a class is abstract or not. Note

[1]Classes in the UML metamodel are referred to as meta-classes

that meta-attributes are different from the attributes of a class: attributes are defined as instances of the meta-class *Property* as shown in the Figure. The multiplicty "2..*" shown on the association between *Association* and *Property* specifies that an association must have at least two association ends.

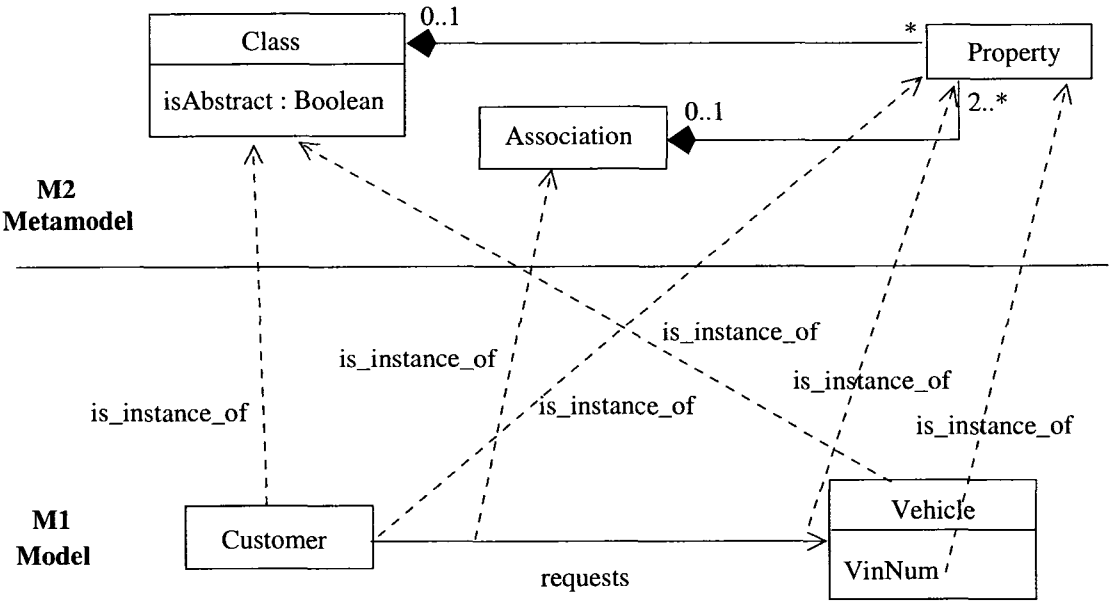


Figure 2.4: A simple metamodel and its instance

2.2 Role Based Meta-modeling Language

In the AOM approach aspect models describe patterns that characterize families of design models. The UML was designed primarily as a notation for modeling single applications, and its use to model application families is problematic. The Role Based Meta-modeling Language (RBML) [16, 41, 42] is a UML based language that supports rigorous specification of patterns that characterize a family of design models. Since RBML uses the UML syntax, UML tools can be used to create RBML specifications.

An RBML specfication consists of a set of role models that describe pattern properties from different perspectives [42]. A role model consists of roles where a role

specifies properties of pattern participants. A UML model can be obtained from a role model by binding the roles to model elements that have the properties specified by the role. However, obtaining a UML model and establishing that a UML model conforms to a role model are currently manual tasks. Automated support for checking conformance of UML models to role models is not available currently.

A variant of the RBML that facilitates generation of compliant models from pattern descriptions is used in this work to specify aspect models. Template diagrams rather than role models are used to specify families of models. UML models are obtained from template diagrams by binding the template parameters to actual values. The result of binding aspect models using actual values that satisfy constraints associated with the template parameters is a UML model that conforms to the aspect model.

Aspect models consist of class diagram templates and sequence diagram templates. Instantiating these templates produces UML design models consisting of class and sequence models.

2.2.1 Class Diagram Template

A class diagram template consists of parameterized class model elements, for example, class templates and association templates. It defines a family of class models where each class model is obtained by binding the parameters to actual values. An example of a class diagram template is shown in Figure 2.5 and the instantiated class model is shown in Figure 2.6. Template parameters are marked using the "|" symbol.

A class template consists of two parts: an attribute template section and an operation template section. Attribute templates produce attributes when instantiated, and operation templates produce operations when instantiated.

The class diagram template shown in figure 2.5 consists of the following class templates: |*Requestor*, |*Authorizer*, and |*AuthRepository*. |*Requestor* has an at-

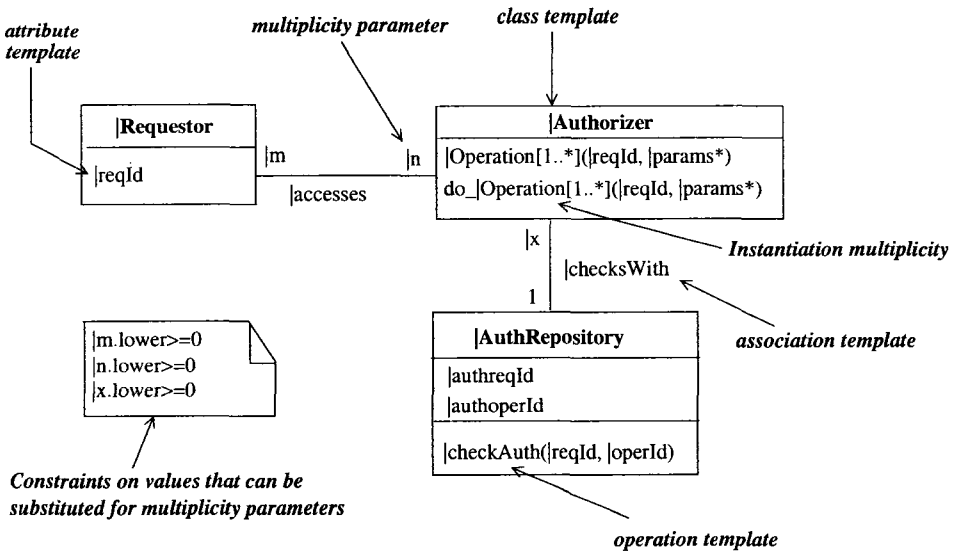


Figure 2.5: Example of a class diagram template

tribute template named |*reqId*. |*AuthRepository* has attribute templates named |*authreqId* and |*authoperId*. |*Authorizer* class template has operation templates named |*Operation* and *do_*|*Operation*. |*AuthRepository* has an operation template |*checkAuth*.

Association templates, for example, |*accesses* and |*checksWith*, produce associations between instantiations of the class templates they connect. An association template consists of multiplicity parameters (one at each end) that yield association multiplicities when instantiated. The multiplicity "1" on the |*AuthRepository* end of the |*checksWith* template is strict, i.e., an instantiation of the association end template must have this multiplicity. One can also specify constraints on multiplicities in class diagram templates (as done in figure 2.5 using a note).

A parameter can be associated with an instantiation multiplicity. The instantiation multiplicity associated with a feature indicates the number of times a feature can be instantiated. For example, a template of the form |*Operation 1..1* indicates that |*Operation* can instantiated exactly once in a class. If a template does not have an instantiation multiplicity, then the number of instantiations is not restricted. One can

specify a list of parameters using the collection parameter *params**. The collection parameter *params** in |*Operation* is bound to a collection of values during instantiation. Also, the presence of *params** in *do_|Operation* indicates that the collection of values bound to *params** in *do_|Operation* must be same as the values obtained from the instantiation of *params** in *Operation*.

An operation template can be associated with an operation specification template. An operation specification template is a parameterized OCL expression. For example, the operation specification template associated with the |*Operation* is shown below:

```
Context |Authorizer::|operation(|reqId, |params*) :
Pre:
  true
Post:
  let authmessage : OclMessage =
  |AuthRepository^|checkAuth(|reqId,|operId) in
  (authmessage.hasReturned() and authmessage.result() = true
   implies |Authorizer::do_|operation(|reqId, |params*))
```

The constraint template states that the operation can be performed only after it has been authorized. Operation specification templates are instantiated to obtain operation specifications (expressed in OCL).

The template class diagram can be instantiated using application-specific values to obtain a UML class model. Figure 2.6 shows a UML class model obtained by binding the class diagram template shown in Figure 2.5 to application specific values. The bindings used for instantiating the class diagram template are shown in table 2.1. For example, the class template |*Authorizer* is instantiated to form the class *Controller*. The attribute template |*reqId* in the class template |*Requestor* is instantiated to form the attribute *userId* in *Client*. The operation template |*Operation* in

the class template |*Authorizer* is instantiated to form the operation *addAccount* in *Controller*.

Table 2.1: Example bindings for the template class diagram

Aspect model parameter	**Application specific element**
|Requestor	Client
|Authorizer	Controller
|AuthRepository	AuthorizationRepository
|Requestor::|reqId	Client::userId
|AuthRepository::|authreqId	AuthorizationRepository::authuserId
|AuthRepository::|authoperId	AuthorizationRepository::authoperId
|Authorizer::|Operation	Controller::addAccount
|Authorizer::do_|Operation	Controller::do_addAccount
|AuthRepository::|checkAuth	AuthorizationRepository::checkAuth
|Operation::|reqId	addAccount::userId
|Operation::|params*	addAccount::accountId
do_|Operation::|reqId	do_addAccount::userId
do_|Operation::|userId	do_addAccount::accountId
|accesses	accesses
|checksWith	checksWith
|m	*
|n	1
|x	*

The operation specification obtained by binding the operation specification template of the operation template |*Operation* with application specific values is given below:

```
Context Controller::addAccount(userId,accountId) :
Pre:
  true
Post:
  let authmessage : OclMessage =
  AuthorizationRepository^checkAuth(userId,operId) in
  (authmessage.hasReturned() and authmessage.result() = true
```

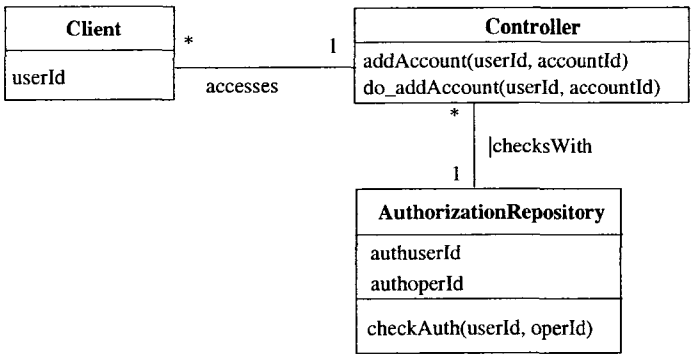


Figure 2.6: Example of an instantiated class model

```
implies Controller::do_addAccount(userId,accountId))
```

2.2.2 Sequence Diagram Template

Sequence diagram templates are used for specifying interaction patterns. A sequence diagram template specifies a family of scenarios. The |*exampleTemplate* (see figure 2.7) sequence template diagram consists of lifeline templates |*Requestor*, |*Authorizer* and *AuthRepository* and the following operation call message templates: |*Operation*, |*checkAuth*, and |*do_Operation*. The Lifeline templates produce lifelines, and operation call message templates produce operation call messages when bound to application specific values.

The sequence diagram template describes the following pattern of behavior:

- A requestor sends an operation call message to an authorizer object
- An authorizer object checks whether the requestor is authorized to execute the requested operation by calling a *checkAuth* operation.
- If the access is authorized, then the authorizer object invokes an operation *do_Operation* to perform the operation.

Figure 2.8 is obtained by binding the lifeline templates and message templates to application specific values. The message template *Operation* is bound to *addAccount*

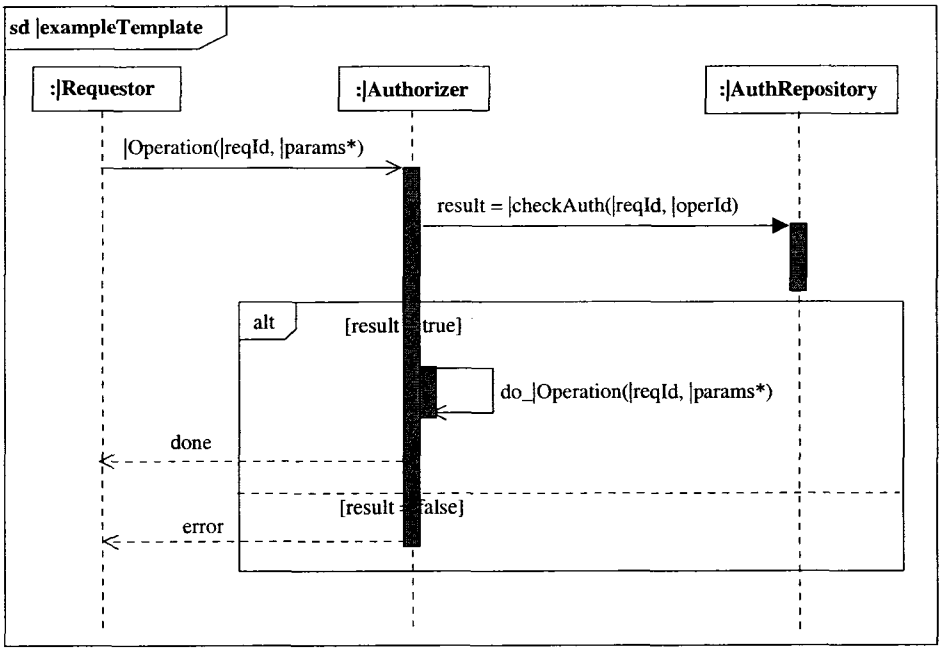


Figure 2.7: Example of a sequence diagram template

message. The figures shows that `addAccount` message needs to be authorized before it invokes `do_addAccount` message.

2.3 The Aspect Oriented Model-driven Development Framework (AOMDF)

The aspect oriented model-driven development framework (AOMDF) [62, 63, 64] that provides the context for this research is a model driven development (MDD) framework that supports vertical and horizontal separation of concerns. Horizontal separation of concerns is achieved by separation of crosscutting features from other features. Crosscutting features that have been separated are composed with other features to provide an integrated view of the system functionality. Vertical separation of concerns is achieved by providing support for transforming models across different levels of abstraction.

A process view of the AOMDF is shown in Figure 2.9. The process framework

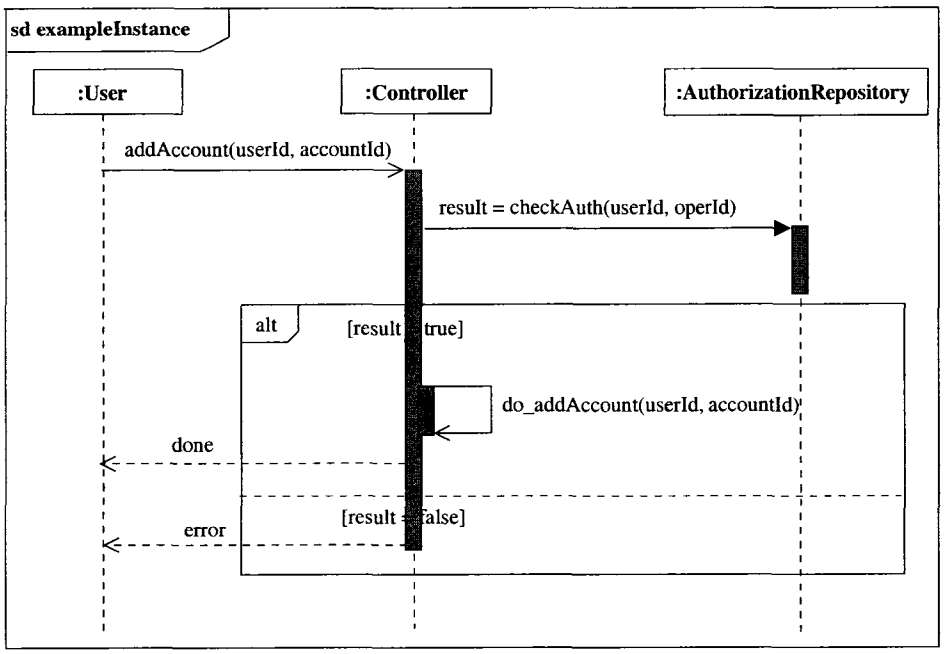


Figure 2.8: Example of an instantated sequence model

consists of two parts: Model Transformation and Model Composition. The model transformation part consists of *source level*, *mapping specification*, and *target level* activities.

The source level includes activities for acquiring or developing aspect and primary models. The system architect decides which services will be included in the primary model and which will be treated as aspects. The aspect models can be acquired from an aspect repository if one is available or they are developed by the system architect. The primary model is developed by the system architect.

The mappings specification part of the process framework includes activities for developing or acquiring the specifications for mapping source level models to target level models. Separate mappings are defined for each aspect model and the primary model. In AOMDF, mappings are specified using MOF 2.0 Query View Transformation (QVT) [54].

The target level includes activities for transforming the source level primary and

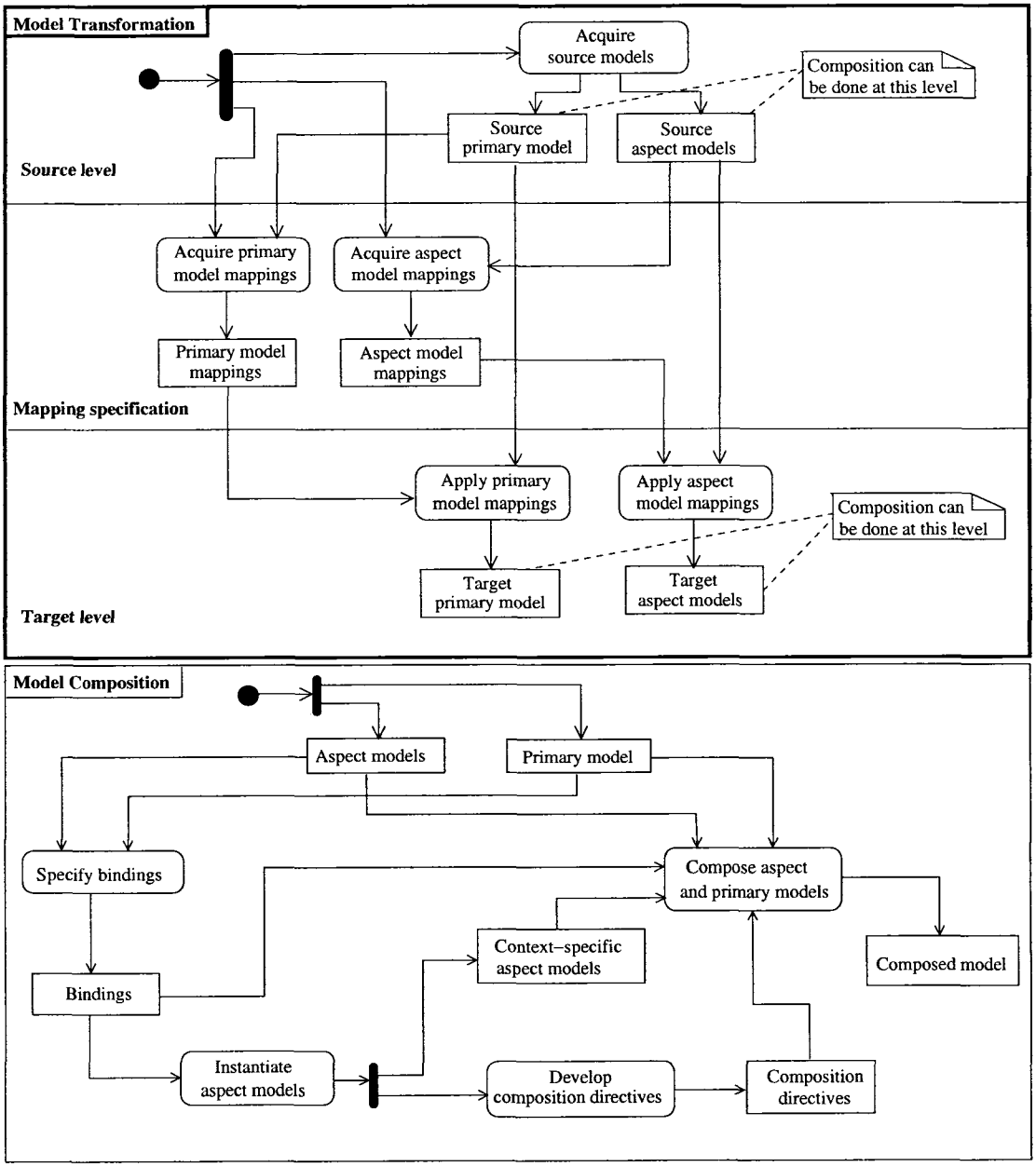


Figure 2.9: Aspect-Oriented Model-driven Development Framework.

aspect models to target level models. The transformations used at this level are implementations of the mappings specified by the mapping specification activities.

The *model composition* part of the AOMDF includes activities for binding the aspect models with application-specific values and composing the aspect and primary

models. A binding is specified as a list of *(aspect model parameter, application specific element)* pairs. The context-specific aspect class model is composed with the primary class model using composition directives to produce an integrated view of the system structure. The aspect sequence model and primary sequence model are composed to obtain an integrated view of the system behavior. Composition directives and tags determine how the aspect and primary models are to be composed.

Chapter 3

Related Work

A number of researchers have developed aspect oriented software development (AOSD) approaches (for example, see [2, 4, 12, 13, 22, 25, 31, 39, 56]). A significant body of work in AOSD focuses on the development of suitable programming abstractions and composition mechanisms. At the design level most of the aspect oriented design (AOD) approaches use UML as is or provide extenstions to UML to support modeling of aspects. A survey of AOSD approaches above the code level can be found in Chitchyan et al. [11].

AOSD approaches can be broadly categorized as *asymmetric* and *symmetric* [27] based on the decomposition and composition procedures used. In *asymmetric* composition, aspects and base artifacts (e.g. base code in AOP) play clearly distinguished roles during composition. Since the artifacts have clearly distinguished roles asymmetric composition approaches do not support composition of an aspect artifact with other aspect artifacts, and composition of a base artifact with other base artifacts. In *symmetric* composition both aspect and base artifacts are treated the same. Since aspect and base artifacts do not play any clearly distinguished roles, artifacts of the same type can be composed.

This chapter presents related asymmetric and symmetric composition approaches and the problems associated with each approach. We also discuss the extent that our research addresses the identified problems.

3.1 Asymmetric composition approaches

Several aspect languages have come up in relatively short period of time (e.g., AspectJ, AspectC++, JAC, JBOSS AOP, JAsCo), most popular of them being AspectJ [38]. An extensive survey of prominant aspect-oriented languages, and other prototypical and research oriented aspect-oriented languages can be found in [9].

AspectJ uses an asymmetric composition procedure. AspectJ extends the Java language. A Java program is also a valid AspectJ program. An AspectJ aspect is the unit of modularity for crosscutting features. In an aspect language like AspectJ, aspects are defined in strictly programming language specific specific terms. The aspect weaving (composition) depends on the language in which the aspects are specified, since different aspect languages use different compilers.

The aspects in AspectJ are Java classes with additional aspect-specific constructs (for example, pointcuts, joinpoints). The constructs are used when the aspect code is woven (composed) with the base code. Hence, aspects cannot be woven with other aspects in AspectJ. The composition is limited to a particular set of joinpoints specified in AspectJ. The type of joinpoints and pointcuts specified can vary from one aspect language to another.

Jacobson [31, 32] describes the development of design aspects based on use cases, which are then composed to create different views of the system. The work maps directly to program level aspects, using the composition techniques originally developed for AspectJ. The result is that their aspects contain the same composition constructs as AspectJ. The composition techniques developed our work can be used to compose any model elements and does not impose the limitations of the AspectJ composition and therefore can be used to perform a wider range of compositions.

Tkatchenko et al. [79] extend the AOP notion of join points and propose a simple join point model that extends the UML metamodel. They add a *joinPoint* class to the UML metamodel and the meta-classses *Operation* and *Property* extend the

joinPoint class. The join point model is an abstraction of AOP at the design level and hence the composition is asymmetric. In their approach, the join points where a crosscutting feature can be introduced to are limited to just operations, attributes and association end properties. However to compose design models other join points may be needed. For example, sequence model properties like lifeline and messages may need to be composed. In the AOM approach any UML model element is a join point and hence does not require an explict modeling construct to be added to the metamodel to represent join points.

Suzuki and Yamamoto [70] propose an extension to UML to support aspects. They extend UML by introducing the notion of *aspect* and *woven class*. An aspect is a classifier in the UML meta-model. They also propose a XML-based aspect description language called UXF/a (UML eXchange Format, aspect extension) for reusing aspect information across different tools. The approach lacks design composition rules and is restricted to design aspects that can be represented as aspects in AspectJ.

Stein et al. [66, 67] propose a similar UML extension in their Aspect-Oriented Design Modeling (AODM) approach. They represent aspects as classes with the stereotype <<aspect>>. Since design aspects in AODM are mapped to AspectJ the composition is asymmetric. The AOM approach described in our work is independent of any programming language and the aspects modeled can be implemented using a non-AOP language.

Kande et al. [43] support the intergration of AOP and Model-driven development. They propose a code driven modelling approach called AOP-to-UML. In their approach UML extension mechanisms are used to create models from AspectJ code. The composition is done using AspectJ. They argue that when an aspect design model is composed with other design models it becomes difficult to make changes to the aspect models. They further state that the elements that are modularized in the design model are not as modularized as they are in code. Hence, they propose a bottom-

up approach in which aspect oriented programming constructs are modeled at the design level using UML extension mechanisms. In our AOM approach the aspect class diagram and sequence diagram templates are pattern descriptions and hence changes can be made at different levels: (1) The pattern specification can be changed (2) the bindings can be changed to obtain different application specific models, and (3) composition directives or tags can be used to change how the design models are composed.

3.2 Symmetric composition approaches

The composition techniques used in the work on viewpoints [46], subject-oriented programming [26, 49], and multi-dimensional separation of concerns (MDSOC) [71] are symmetric.

At the code level, the subject-oriented programming (SOP) approach is closest to the composition techniques described in our work. SOP supports building object-oriented systems as compositions of *subjects*. A subject is a collection of classes or class fragments. In SOP, program elements such as classes and methods are composed by merging corresponding elements. A subject may be just a fragment or a complete application. If the subject is not a complete application, it must be composed with other subjects to obtain a complete application. The composition procedure does not distinguish one subject from another and hence it is symmetric.

During composition of subjects, composition designers consider issues of both correspondence and combination. The correspondence is established based on specified composition rules. The default correspondence is name-based and this can be altered by writing additional composition rules. Combination can be performed in different ways. For example, one subject's elements may be replaced by another subject's element. A join combination is used for aggregation rather than replacement. The composition rules used to control SOP can be classified under three categories: rules

that establish correspondence, rules that control combination, rules that control both correspondence and combination.

Hyperspaces is an extension of the SOP approach used to achieve multi-dimensional separation of concerns (MDSOC) [50, 71]. In their approach, a set of modules that address a single concern is called a *hyperslice*. The hyperslice encapsulates concerns in dimensions other than the one used for the dominant formalism. The hyperslices are composed to obtain the overall system. They have a tool, called HyperJ, which provides support for hyperspaces in Java. The composition is done using subject oriented programming composition rules.

The composition rules in subject-oriented programming and hyperspaces are analogous to the use of signatures to determine matches and the use of directives to alter model elements and override default composition rules. Our class model composition procedure depends on the model element properties specified in the signature rather than just names of model elements, primarily because developers should have the option of composing aspect and primary models using any model element properties. The composition procedure in our approach allows for finer tuning of matching criteria by allowing the user to define the signatures. The composition directives defined in our work are more extensive and do not just restrict themselves to replacing or aggregating model elements.

Brito and Moreira describe an aspect composition process that identifies match points in a design element and defines composition rules [10]. Rules use identified match points, a binary contribution value (either positive or negative) that quantifies the affects on other aspects, and a priority for a given aspect. In the context of AOP, Kienzle et al. describe composition rules based on dependencies between aspects [40].

Katara and Katz [36] propose using architectural views to reason about the influence of one aspect over another in the system. They introduce a conceptual model called the *aspect architecture*, where aspects are the building blocks. They describe

aspect as an augmentation of existing design modules that encapsulate a cross-cutting concern. Sub-aspects are aspects that support overlapping functionality. A composite aspect is an integration of sub-aspects, and corresponds to a single concern. Their notion of an aspect is split into two disjoint parts: a *uses part* and a *defines part.* The uses part describes the join points, to which the aspect is to be applied. The defines part is for specification of crosscutting concern. The overall system is obtained by composing all the aspects. Their notion of a composite aspect is similar to the template aspect models described in our work.

The approaches [10, 36, 40] described above focus primarily on relationships that can exist between aspects. They use a symmetric composition procedure to analyze the influence of an aspect over another. We describe the possible relationships between aspects as weave-order relationships and override relationships, but it may also be possible to use priorities and dependencies as done by Kienzle, Brito et al., and Katara et al. in our techniques.

Nuseibeh [45, 46] defines viewpoints as cross-cutting, partial knowledge of the system and its environments from the perspective of different stakeholders. The viewpoints should be integrated to provide a complete representation of the system. Their approach does not directly support composition and is limited to consitency checks and viewpoint consistency rules. Since one viewpoint is not distinguished from another, their support for composition is symmetric. Our work is complimentary in that we can compose the aspect models and our composition procedure can be used to detect potential problems that can arise as a result of interactions between different aspect models.

Aldawud et al. [3] propose a mechanism for composing state charts where a crosscutting behavior is an event that triggers a state transition. The aspects and base artifacts are specified using state charts and the composition is specified implicitly by linking events across state charts. Since, there is no distinguishing role between a

state chart in the aspect model and a state chart in the base model, the composition is symmetric. They use unique event names each time an object needs to invoke an event. A problem with their approach is that some transitions between states may occur without the occurance of any event. In such cases it becomes difficult to specify the composition using event names. Our AOM approach deals with composition of class models and sequence models and is not just name-based. We have not considered composition of state charts in our work.

Giese and Vilbig [21] propose a formal model of component behavior using UML description techniques for separation of non-orthogonal concerns in software architecture and design. They define a concern as a logically coherent subset of the overall system functionality. Their approach is to develop solutions (architectural views) for individual concerns and combine the solutions to obtain an overall solution for the complete system. The approach is based on formal contract specification of component behavior. The contract consists of a set of available operations, and a protocol state diagram of the supported interactions. Once the contracts are specified formally, they are combined to obtain the overall behavior. The approach mainly describes composition and support for evolution is made possible by changing the behavior associated with the contracts. The approach is formal, but does not address solutions that may crosscut the different architectural views. Our work provides composition techniques for solutions that crosscut modules of a design, and the technique can be used to detect syntactic conflicts that occur during composition.

3.3 Hybrid composition approaches

Some other approaches [1, 6] use a hybrid composition procedure consisting of both asymmetric and symmetric composition procedures.

Composition-Filters is an AOP technique where different aspects are expressed in *Filters* as declarative and orthogonal message transformation specifications [1, 2].

The filters can express crosscutting features with well-defined interfaces and orthogonal enhancements to objects. All messages to and from the attached objects pass through the filters in a sequence. Each message that passes through the filter can be evaluated or manipulated by the filter. The filter specifications are orthogonal and hence they can be composed together and this makes the composition procedure symmetric. Composition filters are tailor-made abstractions for composing the behavior of objects and filters can implement new methods not directly supported by the objects. This leads to asymmetry in the composition since filters and objects are different entities. The Composition-Filters are expressed in languages such as C++ and smalltalk. The composition procedure in our work complements the composition filter approach at the model level. The composition directives are similar to filters, since they can be used to modify the model elements.

At the model level, a related AOM approach is the Theme approach proposed by Baniassad and Clarke [6, 12, 13]. In the Theme approach, a design, called a *theme*, is created for each system requirement. These themes, like context-specific aspect and primary class models, are design views. A comprehensive design is obtained by composing themes. A crosscutting theme is triggered by the base theme and a non-crosscutting theme is not. Composition in the Theme approach is both symmetric and asymmetric. The themes are composed based on the symmetric approach used in subject oriented programming.

Composition relationships specify how models are to be composed by identifying overlapping concepts and specifying how models are integrated. Two types of integration strategies are used: *Override* and *merge*. Override integration is used when existing behavior in a subject needs to be updated to reflect new requirements. Merge integration is used when subjects for different requirements are to be integrated. Operations in related subjects may need to be merged into a unified operation. Reconciliation strategies resolve conflicts between property values of corresponding subject

elements. Precedence relationships, transformation functions applied to conflicting elements, explicit specification of reconciled elements, and default values may be used for reconciliation.

Clarke [12] also extends the UML metamodel with the notion of *composableElements* that can be composed using a composition relationship. They have a *Match* metaclass that supports specification of matching criteria. Their matching criteria includes *matchByName* and *dontMatch*. They leave the details of implementing the *matchByName* and *dontMatch* to the user of the metamodel. In this sense the metamodel describes a framework for composing UML models. In our work we have developed a more specialized metamodel that contains specifications of composition behaviors. The composition directives that we have developed are not limited to merge and override integration strategies. The use of composition directives and signatures, as described in our work, allow modelers to define and apply their own integration and reconciliation strategies, and thus gain finer control over how models are composed. In addition, we have developed a technique to compose sequence models.

3.4 Other AOSD approaches

Grundy [24, 25] proposes an Aspect Oriented Component Engineering (AOCE) approach that focuses on capturing concerns that crosscut many components. AOCE supports the identification, description and reasoning about components and aspects. Aspects are specified as components in AOCE but an aspect is explicitly differentiated from a component. Components are differentiated based on the service provided and required. In AOCE, an aspect extends a component. AOCE is aimed at design and deployment level and uses a set of UML meta-model extensions to specify the aspect information. The crosscutting concerns and other components are not merged at the design level, they merely refer to each other. The composition in AOCE happens

at run-time and depends on the underlying implementation platform. As a result, it is difficult to comprehend the overall system at the design level. In our work, the primary model and aspects are composed and composition directives or tags can be used to specify how the composition is done. The integrated view makes it possible to comprehend/evaluate the entire system at the design level.

Araujo et al. [4, 5] use a scenario-based approach to distinguish aspectual and non-aspectual scenarios. The behaviors that are repeated across system requirements are generalized into aspectual scenarios and areinstantiated when required. The aspectual scenarios are represented as interaction pattern specifications (IPS) where the IPS describes a pattern of interaction between its participants in terms of roles that the participants must fill in. The IPS is specified using RBML. Non-aspectual scenarios are represented as sequence models. The aspectual scenarios are translated from IPS to State Machine Pattern Specifications (SMPS) and the non-aspectual scenarios are translated from sequence models to Finite state machines (FSM) before they are composed. Their approach compliments our work by providing a technique to compose state models and uses the RBML notation developed at CSU. Further, their approach does not provide any insight into how one can resolve problems that may arise due to the composition of aspect and primary models.

Awais *et al.* [55, 56] propose an Aspect Oriented Requirements Engineering approach called "AORE with Arcade" for modularizing and composing requirements level concerns that cut across other requirements. The approach involves identifying requirements using stakeholders' viewpoints, use cases/scenarios, goals or problem frames. The approach basically uses a set of matrices consisting of the viewpoints and concerns represented in XML. The composition rules are defined using XML. The approach is supported by the Aspectual Requirements Composition and Decision (ARCaDe) support tool. The composition rules, conflict resolution mechanisms are defined at the requirements level. Our work complements their work by support-

ing UML based aspect representation and composition at the design level.

Tekinerdogan *et al.* [72, 73] supports an aspect-oriented software architecture design that integrates the ideas in software architecture design and aspect-oriented software development. Their research looks at advanced separation of concerns at the architecture design level. Functional components are addressed as architecture design components, and crosscutting architectural concerns are called architectural aspects. They propose Aspectual Software Architecture Analysis Method (ASAAM) to explicitly identify and specify architectural aspects. The approach is an extension of SAAM [37] and uses scenarios to identify aspects. They classify scenarios into direct scenarios, indirect scenarios, aspectual scenarios and architectural aspects. Scenarios that are directly supported by the architecture are called direct scenarios. Scenarios that require modification to the architecture are called indirect scenarios. Aspectual scenarios are derived from direct or indirect scenarios and represent potential aspects. The approach is highly intuitive. As the authors themselves state, the scenario may be direct or indirect depending on the components and the type of architecture chosen. The main goal of their work is identification of aspects at the architectural level. They do not compose or analyze the scenarios to provide an overall view of the architecture. Our work does not address identification of aspects, rather the aspect models are assumed to be available in a repository. If the aspect models are not available they need to be specified. Aspect models are specified as pattern descriptions and we describe composition techniques for incorporating the aspect models in software designs.

Jeźequel *et al.* [29, 30, 33] have proposed an approach where they introduce specialized stereotypes for each crosscutting concern. They have developed an UMLAUT model transformation framework that can be used for building application specific weavers to weave multi-dimensional high level UML design models into detailed design models suitable for either implementation, simulation or validation. UMLAUT

framework allows complex model transformations. The transformation operations are written in the MTL transformation language. The meta models are defined using the Meta Object Facility. UMLAUT uses a form of roles that serves the same purpose as template parameters defined in the RBML, but the treatment of properties is not as extensive. In our work, the properties are explicitly specified using the attribute templates, operation templates and the constraint templates. The AOMDF framework is complimentarty to UMLAUT and uses QVT [54] to specify transformations.

Sutton *et al.* [68, 69] propose a Concern-Space Modeling Schema (COSMOS) that can be used to model early-stage concerns and relationships. Cosmos provides a framework in which concerns and their relationships can be modeled independent of the life cycle stages, methods, and technology. Cosmos is used with HyperJ to support concern-driven composition. As pointed out by the authors themselves, Cosmos is not intended to replace other modeling approaches rather it is to be used in conjunction with other approaches. The class model composition technique can be used along with COSMOS to reason about the relationships between crosscutting features.

Gray *et al.* [22] use aspects in domain-specific models that target embedded systems. Requirements, architecture and the environment of a system are captured in the form of formal high-level models that allow representation of concerns. Their research is part of the Model-Integrated Computing (MIC) and extends the scope and usage of models such that they form the backbone of a development process for building embedded software systems. Our work can complement their research by providing an UML based approach for describing aspects.

3.5 Summary

All the AOSD approaches surveyed in this chapter specify crosscutting features and relationships between crosscutting features and other features. A lot of AOSD approaches use AspectJ to describe aspect-specific constructs. UML is used in most of

the aspect oriented (AO) design approaches.

The main contribution of AO design has been to provide designers with the means to model aspect-oriented systems. An AO design provides developer ways to reason about design. The composition techniques can be used to compose crosscutting or non-crosscutting features. This includes techniques for composition at the code level or at the design level. Hence a designer can choose to compose at the design level and refine the design artifacts using an object-oriented or an aspect-oriented programming language in the later stages of development lifecycle.

The following subsections summarize the surveyed AOSD approaches based on the composition symmetry, composition procedure, level of abstraction and the type of aspects specified.

3.5.1 Composition symmetry

An aspect oriented approach may use symmetric or asymmetric or hybrid composition. Symmetric approaches provide support for modularization of crosscutting and non-crosscutting features. Most of the asymmetric approaches use design constructs that can be directly mapped to some aspect oriented programming language like AspectJ.

The composition described in our work is a hybrid composition procedure: The template aspect models are patterns that cannot be directly composed with base models, but the instantiated forms of the aspect models (i.e., context-specific aspect models) are not distinguished from the primary model by the composition procedure. The class model composition procedure we developed can be used to compose aspect models to obtain new aspect models or can be used to compose context-specific aspect model with the primary model to obtain the composed model. The composition of class models is symmetric and is not applicable to sequence models.

In the interaction model composition technique the aspect sequence model and

Table 3.1: Summary of composition procedures

Approach	**Composition Procedure**
Kande *et al.* - AOP to UML	Asymmetric
Suzuki *et al.* - UXF/a	Asymmetric
Stein *et al.* - AODM	Asymmetric
Tkatchenko *et al.* - Join point model	Asymmetric
Jacobson - AOSD with use cases	Asymmetric
Ossher *et al.* - Hyperspaces	Symmetric
Aldawud *et al.* - State chart weaving	Symmetric
Katz *et al.* - Architectural views	Symmetric
Giese *et al.* - Contracts	Symmetric
Nuseibeh *et al.* - Viewpoints	Symmetric
Clarke *et al.* - Theme	Hybrid
Aksit *et al.* - Composition Filters	Hybrid
Our AOM approach	Hybrid

primary sequence model play clearly distinguished roles. We use tags on the primary sequence model to specify the ordering of messages. Hence composition of interaction models is asymmetric. A summary of the composition procedures used in the aspect oriented approaches is provided in table 3.1.

3.5.2 Composition procedure

Many researchers use UML extension mechanisms like stereotypes, tag values, to provide aspect orientation to UML. Some have found that extension mechanisms are not enough to support the composition procedure and have extended the UML metamodel to support composition. While some procedures support only composition of crosscutting features with other features, others support composition of crosscutting features and other features.

The composition procedure described in our research makes use of signatures and tags. The class models are composed using signatures and composition directives. The composition directives can be used to alter the class model composition in cases

where the composition is know or expected to yield incorrect results. The composition procedure for sequence models is specified using pre-defined schemas. A summary of the composition procedures provided in the various aspect oriented approaches is provided in table 3.2.

Table 3.2: Summary of aspect oriented design approaches

Approach	**Composition procedure**
Kande *et al.* - AOP to UML	Based on AspectJ
Suzuki *et al.* - UXF/a	Do not compose at design level - use AspectJ at code level
Stein *et al.* - AODM	Similar to AspectJ composition
Tkatchenko *et al.* - Join point model	Based on AspectJ
Jacobson - AOSD with use cases	AspectJ composition procedure
Ossher *et al.* - Hyperspaces	Name based composition using a set of matching rules
Aldawud *et al.* - State chart weaving	Implicit event name based
Katz *et al.* - Architectural views	Defined at package and class level
Giese *et al.* - Contracts	Formal composition semantics using state charts
Nuseibeh *et al.* - Viewpoints	Using inter and intra viewpoint check rules
Clarke *et al.* - Theme	Using merge and override composition specification
Aksit *et al.* - Composition Filters	Depends on the language
Our AOM approach	Based on signatures and pre-defined schemas

3.5.3 Level of Abstraction

A lot of AOSD approaches offer separation of crosscutting features from other features in a system at one abstraction level, mostly at lowest level of abstraction (implementation). Some researchers have abstracted to the design level but still model code aspects at design level rather than specify design level aspects. An advantage of specifying aspects at the middle-high (design) level is that the design aspects may be mapped to code using a non-AOP language. Table 3.3 summarizes the type of

aspects specified and the level of abstraction at which aspect in some of the aspect oriented approaches.

Table 3.3: Summary of level of abstraction

Approach	**Level of Abstraction**	**Type of Aspects**
Kande *et al.* - AOP to UML	low	Models Code aspects
Suzuki *et al.* - UXF/a	low-middle	Design aspects independent of implementation
Stein *et al.* - AODM	low-middle	Design aspects abstracted from AOP lanaguges
Tkatchenko *et al.* - Join point model	low-middle	Design aspects that can be mapped to Aspect languages
Jacobson - AOSD with use cases	low-high	High level Design aspects mapped to AspectJ
Ossher *et al.* - Hyperspaces	low-middle	Design aspects
Aldawud *et al.* - State chart weaving	middle	Design aspects
Katz *et al.* - Architectural views hline Giese *et al.* - Contracts	middle-high middle-high	Design aspects design aspects
Nuseibeh *et al.* - Viewpoints	low-high	programming language independent aspects
Clarke *et al.* - Theme	low-middle	programming language independent design aspects
Aksit *et al.* - Composition Filters	low-middle	design aspects depend on the implementation platform
Our AOM approach	middle-high	programming language independent design aspects

Chapter 4

Class Model Composition

In this chapter, we present a technique for composing class models. An overview of the class model composition technique is shown in Figure 4.1. The figure shows the application of bindings to obtain context-specific aspect class models. The bindings are specified as a list of (aspect model parameter, application specific model element) pairs, where for each pair, the application specific model element is obtained by instantiating the aspect model parameter with application-specific values. The namespace from which the bindings are obtained is called as the *application domain namespace.* The context-specific aspect class models are then merged with the primary class model using the prototype tool implemented in KerMeta to obtain the composed class model. Composition directives can be used to specify how the context-specific aspect class models are composed with the primary class model.

Sometimes, a single aspect class model may have to be instantiated multiple times for a given application. For example, consider the case where a decision has been made to make an application design fault tolerant and highly available by replicating critical resources such as data repositories and service providers. Incorporating the crosscutting replication feature into the (primary) design class model proceeds as follows:

1. An aspect class model describing the replication feature for a generic resource is developed or acquired.

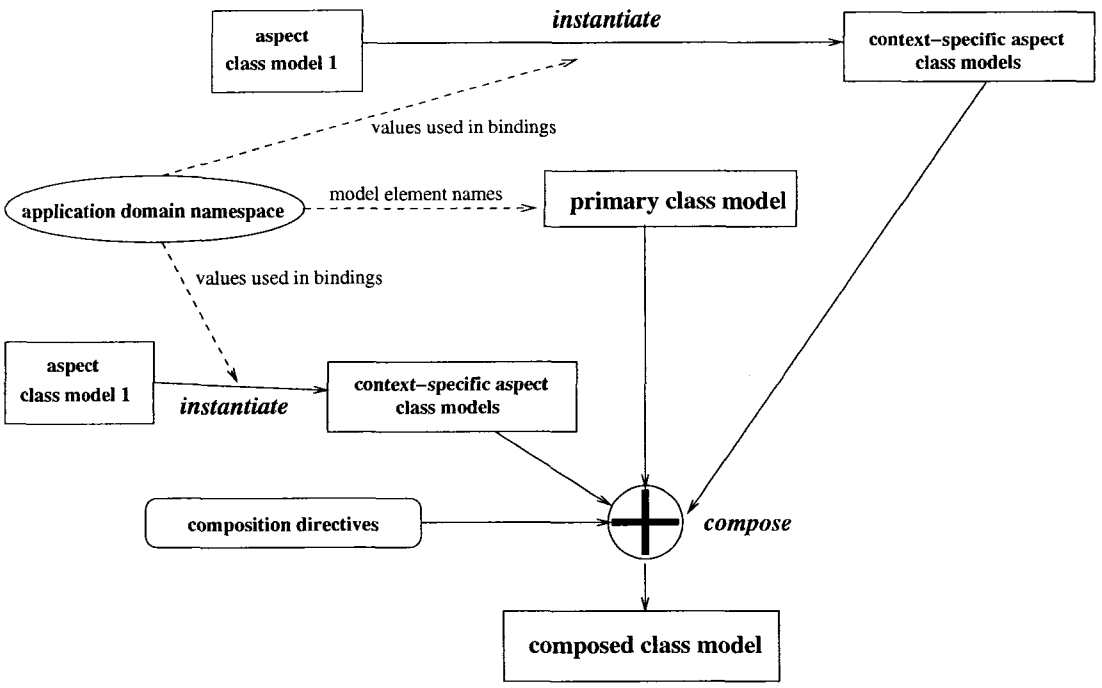


Figure 4.1: An overview of the class model composition in AOM

2. The replication aspect class model is instantiated multiple times. Each instantiation is a context-specific aspect class model that describes the replication feature for a specific application resource.

3. The context-specific aspect class models are composed with the primary application class model to produce a design in which specified resources are replicated.

In cases where multiple aspect class models are to be composed with a primary class model, composition directives can be used to specify the order in which the aspect class models are to be composed with the primary class model. For example, an inventory management system may need to incorporate authorization and auditing features. An ordering relationship between the authorization and auditing features needs to be established before the features are incorporated into the inventory management system primary model.

4.1 Matching Model Elements

The composition of class models is performed by matching context-specific aspect class model elements with the primary class model elements. Only model elements of the same syntactice type[1] can be merged. For example, an operation can be merged only with other operations. An attribute cannot be merged with operations, classes, etc. Model elements that are of the syntactic same can be merged if they have matching properties (e.g., class names, class attributes). The matching may be done using a name-based or signature-based composition procedure. For example, two classes are said to match if they have the same name in a name-based composition procedure. Some of the rules that determine how the model elements are composed are given below [15]:

- If model element properties match, they are merged to form a single model element in the composed model.
- If the *matching model elements*[2] are operations with operation specifications, the operation specification (pre and post conditions) should be merged in the composed model. The precondition of the merged operation in the composed model is a disjunction of the preconditions associated with the matching operations, and the postcondition of the merged operation is the conjunction of their postconditions. A composition directive can be used to vary how the pre and post conditions are merged.
- If the matching model elements have constraints (class invariants), the constraint associated with the element in the composed model is the conjunction of

---

[1]The syntactic type of a model element is the class of the model element in the UML metamodel

[2]Model elements with the matching properties

the constraints. A composition directive can be used to vary how the constraints are merged.

- If the matching elements are associations, then the stronger (more restrictive) multiplicity at an association end is used in the composed model. A composition directive can be used to override this rule.

- If a model element property in one matching element is not in the other, then it appears in the composed model.

4.1.1 Name-based Matching

The name-based composition procedure uses model element names to identify the elements that need to be merged. Model elements of same syntactic type with same name are merged to form a single model element [15].

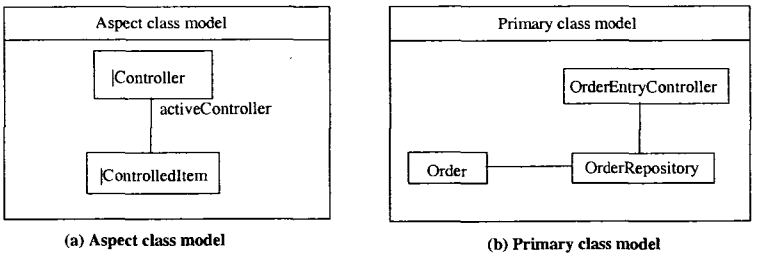


Figure 4.2: An example of name-based composition

Consider an example in Figure 4.2 which shows a partial aspect class model and a partial primary class model. The aspect class model consists of two class templates: `Controller` and `ControlledItem`. The aspect class model is meant to increase the fault tolerance of the system it's woven with by using redundant controllers. In the aspect model specified, at any given time only one controller (`activeController`) is active. If the `activeController` fails, another controller in the redundant set becomes the `activeController`. The primary class model is an order entry system consisting of `OrderEntryController`, `OrderRepository` and `Order` classes.

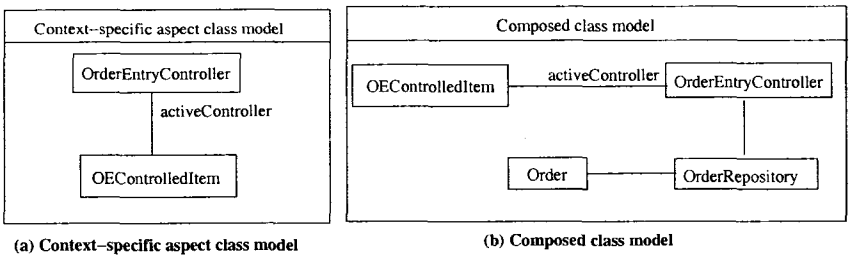


Figure 4.3: Composed model obtained using name-based composition

The context-specific aspect class model is obtained by instantiating the template aspect class model (see Figure 4.2(a)) using the following list of bindings: (|`Controller`, `OrderEntryController`), (|`ControlledItem`, `OEControlled-Item`). The bindings are applied on the template aspect class model to obtain a context-specific aspect model shown in Figure 4.3(a). The name-based composition procedure merges matching models elements. In this case the context-specific aspect model element `OrderEntryController` and primary model element `OrderEntry-Controller` are merged. Other model elements are included in the composed model (see Figure 4.3(b)).

Ideally, the values used to bind the aspect model elements and the values used to name the primary model elments are obtained from a repository of namespaces. If such a repository exists, naming conflicts can be avoided. Unfortunately, a repository of namespaces is often not available in design development environments, and thus naming conflicts may occur.

The name-based composition procedure is relatively easy to implement but as a matching criterion, it can be too permissive in some cases. For example, matching operations using only their names could lead to merging problems when the operations have incompatible return types or when the argument lists differ (see Figure 4.4). The Figure shows an operation `updateAcct` in `Model 1` with the return type `boolean` and an operation `updateAcct` in `Model 2` with the return type `int`. If the two operations

are merged using a name-based composition procedure, it results in a conflict due to the incompatible return types. Similarly, matching attributes using only their names can lead to merging problems when the types associated with the attributes are incompatible.

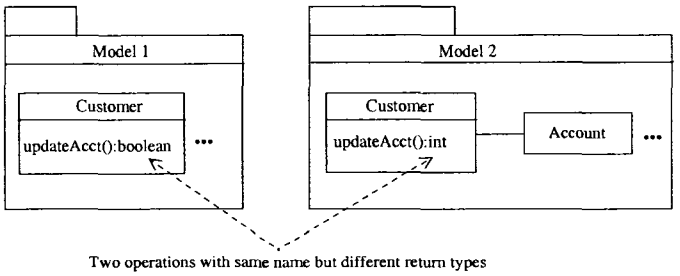


Figure 4.4: An example of a problem that can arise using name-based composition

One would like to have matching criteria that takes into consideration additional properties of the elements being matched. For example, one should be able to express a matching criterion for attributes that requires matching attributes to have the same name and return type. This led to the development of the signature-based composition procedure.

4.1.2 Signature-based Matching

The *signature-based composition* procedure uses a finer-grained matching criteria in which model elements with matching signatures are merged to form a single model element in the composed model. A model element signature is defined in terms of its syntactic properties, where a syntactic property of a model element is either an attribute or an association end defined in the element's UML metamodel class. For example, *isAbstract* is a syntactic property defined in the metamodel class called `Class`. If $isAbstract = true$, then an instance of the `Class` is an abstract class, otherwise the instance is a concrete class (i.e., $isAbstract = false$).

The set of syntactic properties used to determine a model element's signature

is called a *signature type*. For example, the signature type for an operation can be defined as a set consisting of the following properties defined in the `Operation` class: `name` (value is the operation's name) and `ownedParameter` (value is the collection of parameters associated with the operation). Using this signature type, the signature of an operation $update(x : int, y : int)$ is the set $\{update, (x : int, y : int)\}$. If this signature is used to match operations, two operations match if and only if they have the same name and parameter list. If the signature type of an operation consists only of the operation name, then the signature of the operation is $\{update\}$. Use of this name-only signature type results in a weaker matching criterion for operations: two operations match if and only if they have the same name. As stated earlier using a name-only signature can sometimes result in merging problems.

A signature type that consists of all syntactic properties associated with a model element is called a *complete* signature type. Complete signature types require that matching model elements have equivalent values for all syntactic properties (i.e., the matching elements must be syntactically identical). Complete signature types are typically used for matching contained model elements such as class attributes and operation parameters. Composite model elements that contain a variety of model elements (e.g., classes) tend to have signature types that are not complete.

A signature type that consists of a default set of syntactic properties associated with a model element is called a *Default* signature type. Default signature types (for example, the `name` property of a model element) are used when the signature type is not specified explicitly. Default signature types are typically used for matching model elements such as packages.

A proper signature type should be defined to obtain the desired result. It may be necessary to use the default signature type, complete signature type or a subset of properties of the model elements depending on the desired composed model. As an example, consider a model, *Model 1*, containing a concrete class named *Customer*

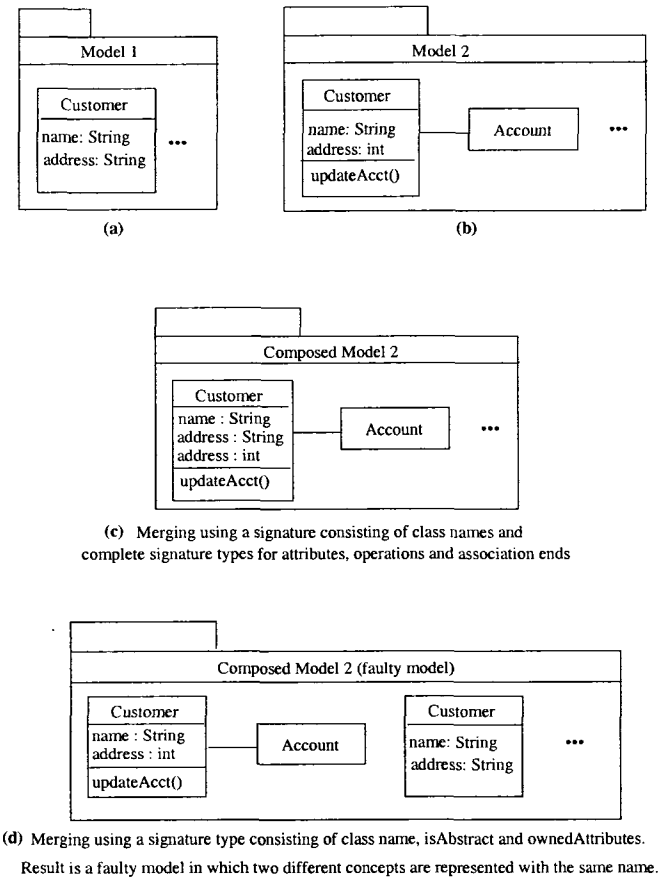


Figure 4.5: An Example of Model Element Matching and Merging

with attributes *name* and, *address*, (see Figure 4.5(a)) and another model, *Model 2*, which contains a concrete class named *Customer* with attributes *name*, *address* and a reference to an *Account* object (see Figure 4.5(b)). If the signature type used to compose the classes in Figure 4.5(a) and Figure 4.5(b) consists of the class name property and the *isAbstract* property then the two classes match (they are both concrete) and their contents are merged to form a single class. The issue of merging syntactic properties that are not part of a model element's signature type arises in this case. The matching classes in this example have different attribute, operation and association end sets. Merging the constituent model elements involves matching them using signature types defined for the elements. The constituent elements that

are matched are merged in the composed model. Those elements that are not matched are included in the composed model.

The composed model shown in Figure 4.5(c) is obtained by using complete signature types for attributes, operations and association ends:

- The attribute *name* : *String* in *Model 1* and *Model 2* match and is included once in the composed model.
- The attribute *address* : *String* in *Model 1* does not appear in *Model 2* and thus is not matched. It appears in the composed model.
- The attribute *address* : *int* in *Model 2* does not appear in *Model 1* and thus is not matched. It appears in the composed model.
- The operation *updateAcct*() in *Model 2* does not appear in *Model 1* and thus is not matched. It appears in the composed model.
- The class *Account* in *Model 2* does not appear in *Model 1* and thus is not matched. It is included in the composed model.
- The assocation between the class *Account* and class *customer* in *Model 2* does not appear in *Model 1* and thus is not matched. It is included in the composed model.

The usage of just the *name* property for attribute in the example would have led to problems because the attribute *address* is of type *String* in *Model*1 and *int* in *Model*2. The use of particular signature types can lead to models that are not syntactically well-formed in some cases. For example, in Figure 4.5(c) the attribute *address* in class *Customer* is of type *int* and type *String* which is a violation of the namespace. As another example, consider the case in which the signature type for class is defined as consisting of the following properties: `Name`, `isAbstract`, and

`ownedAttribute`. Two classes match using this signature type if and only if they have the same name, are both abstract or are both concrete, and they have the same set of attributes (instances of class `Property`) and association ends (instances of class `Property`). If this signature type is used to compose the class models shown in 4.5(a) and 4.5(b), then the result is shown in 4.5(d). The *Customer* classes in *Model 1* and *Model 2* do not match because they do not have the same set of attributes or association ends and thus they are not merged. The model is not well-formed because there are two classes with the same name in the same namespace.

To resolve the above problem one must understand the intent behind the signature type. If it is determined by the modeler that the signature type correctly reflects the syntactic form of classes that represent the same concept, then the problem is resolved by renaming either the *Customer* class in *Model 1* or the *Customer* class in *Model 2*. As will be described later in this thesis, this can be accomplished by using a composition directive. On the other hand, if the modeler determines that the classes actually represent similar classes then the signature type can be changed so that the classes match.

The signature-based composition procedure has been implemented in the KerMeta language. The composition metamodel describing the signature-based composition and a prototype implementation that uses the signature type to merge two class models is described in the next section.

4.2 Composition metamodel for class diagrams

The composition metamodel describes how signature-based composition can be accomplished. The composition metamodel describes the static and behavioral properties needed to support signature-based model composition (see Figure 4.6). In this thesis, we describe the behavioral properties in terms of class operations and narrative descriptions of the operations. A sequence model that shows the interactions between

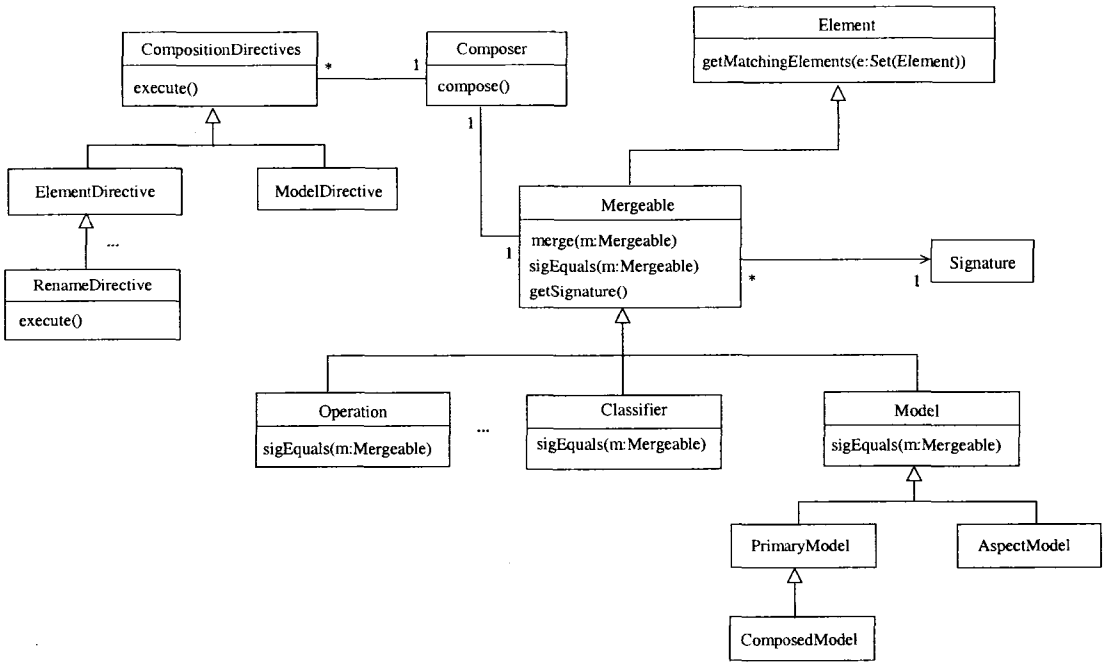


Figure 4.6: Composition metamodel for merging design models

the model elements in the composition metamodel is shown. Alternatively, activity models can be used to describe the behavior that takes place during composition.

The composition metamodel can be used to compose any two class models. The composition assumes the primary model as the initial model. The aspect model is merged with the primary model to obtain the composed model. At any given time, only one aspect model is composed with the primary model and the order in which multiple aspect models are composed with the primary model can be specified using model directive. The composition metamodel shown in Figure 4.6 has been implemented using the KerMeta language [58, 80].

The sequence models for the composition metamodel are shown in Figures 4.7 and 4.8. Figure 4.7 shows an invocation of the `compose` method with the mergeable models as parameters. Depending on the signature the constituent model elements are merged. The constituent model elements are recursively merged using the mergeElements sequence (see Figure 4.8. This is shown as a reference, since sequence models

do not directly support recursion. An explanation of the metamodel elements is explained later in the implementation subsection.

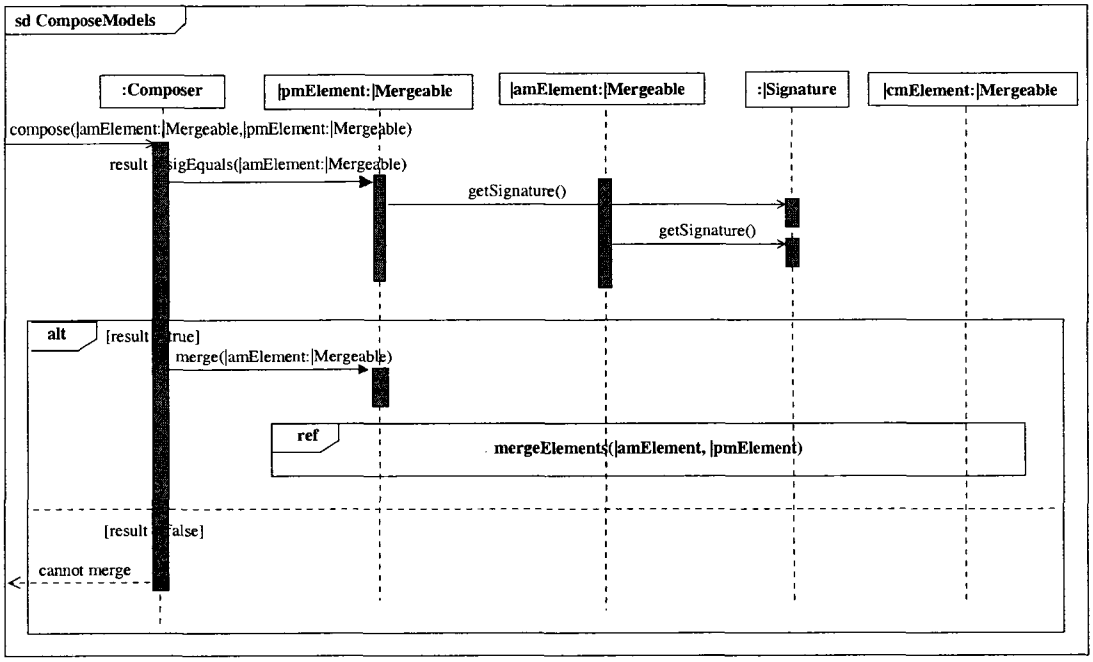


Figure 4.7: The composer part (main) of sequence diagram for composition metamodel

The composition metamodel is independent of the model types and hence can be used to compose class models, activity models, state models, etc. Depending on the model type the corresponding signature type consisting of properties of that model type needs to be specified. For example, composing two activity models requires a signature with the set of properties associated with activity models. To compose sequence models, properties like lifeline, message can be used. However, UML does not specify properties that can be used to determine the predecessor and successor of messages. This is a major limitation in the composition of sequence models since a modeler needs to understand the flow of messages to determine where in the primary model the aspect model sequences need to be introduced. Hence the composition metamodel described in Figure 4.6 cannot be used to compose sequence models.

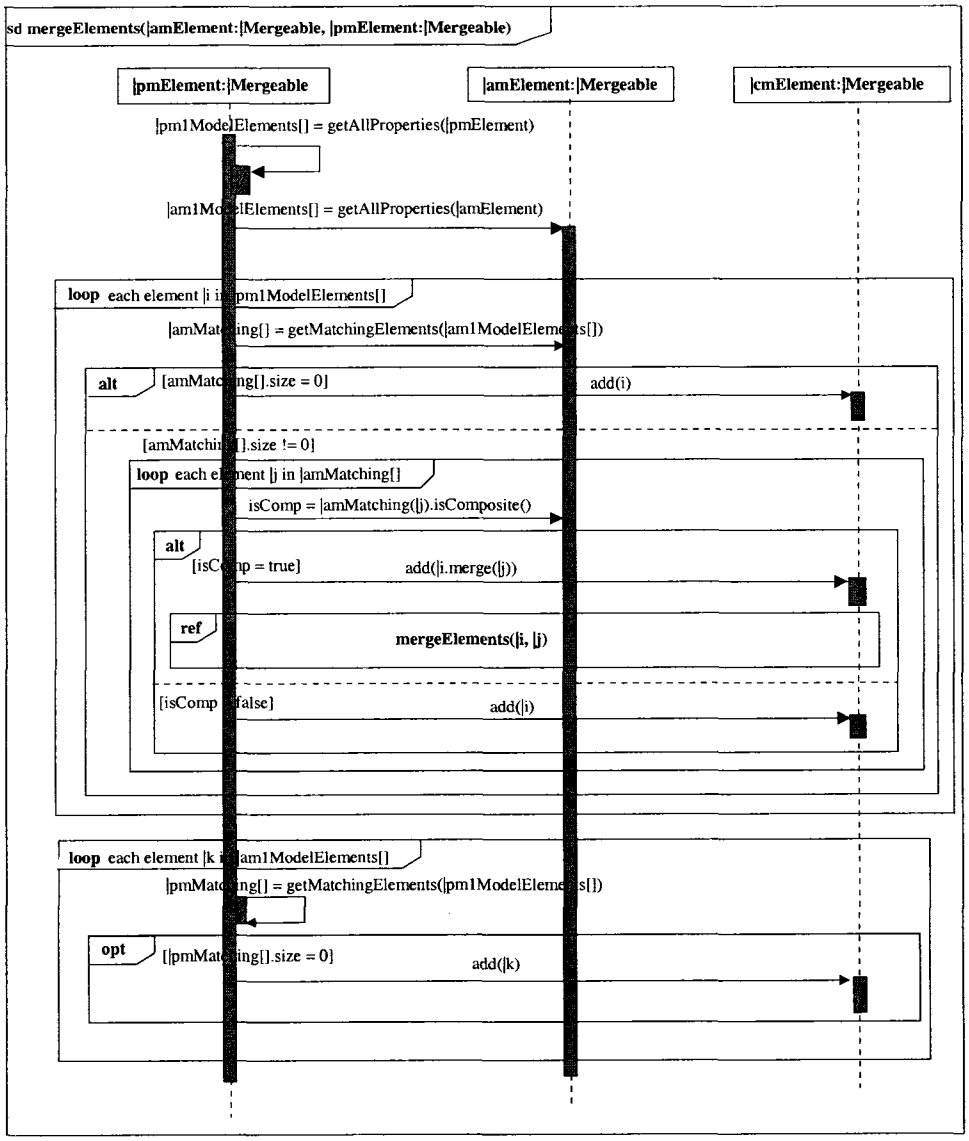


Figure 4.8: Merge part of sequence diagram for composition metamodel

4.2.1 KerMeta

KerMeta [44, 80] is an open-source metamodelling language developed by the Triskell team at IRISA. It has been designed to be a common basis for implementing Metadata languages, action languages, constraint languages or transformation language (see figure 4.9) [80]. It can be used to define the structure and behavior of a user-designed metamodel.

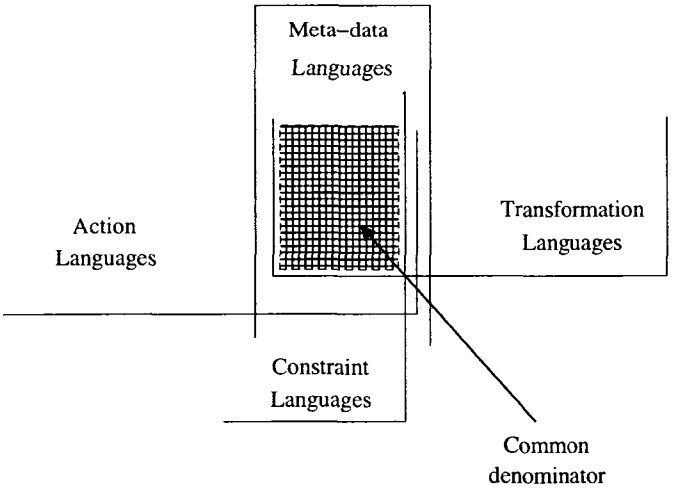


Figure 4.9: Kermeta (as shown in [80])

The KerMeta metamodel is divided into two packages: *structure* and *behavior*. The structure corresponds to the OMG metamodeling language Essential Meta-Object Facilities (EMOF)[47]. The behavior corresponds to actions.

The KerMeta action language includes object-oriented features and model-specific features. Some of these model-specific features are used in our research for the implementation of the model composition. For example, association end references have a property called *opposite* which makes it possible to model the opposite association end reference to which its associated. Some other features like object containment have been included. In addition to this, KerMeta implements OCL closures such as *each*, *collect*, and *select*. Inclusion of the above given features makes it easier to implement operations defined in the metamodels.

The KerMeta language was chosen to implement the model composition for the following two reasons:

- KerMeta allows implementing the model composition by adding the composition algorithm in the body of the operations defined in the composition metamodel.
- KerMeta tools are compatible with the Eclipse Modeling Framework (EMF) and

thus Eclipse tools can be used to edit, store, and visualize models manipulated in the aspect oriented modeling and development framwork (AOMDF).

The EMF uses *ecore*, a meta language that describes models based upon a subset of the OMG Meta Object Facility 2.0 (MOF) called Essential MOF (EMOF). Since KerMeta is compatible with EMF, the ecore model is imported as a KerMeta model and the composition metamodel is added to the ecore model. Once the composition metamodel is added to the ecore model, the algorithm implementing the composition is added in the body of the operations defined in the composition metamodel.

4.2.2 Essential Meta-Object Facilities

EMOF 2.0 [47] can be used to describe metamodels using object-oriented concepts. It utilizes concepts from UML 2.0, and thus allows one to use UML tools to build metamodels. The EMOF classes used in the implemenation of the composition technique are shown in figure 4.10.

All EMOF classes inherit properties from the `Object` class. This class contains a list of operations that are used in the class model composition:

- The *getMetaClass*() operation returns the *Class* of an object. For example, if the *getMetaClass()* is used on an operation, it will return the metaclass `Operation`.
- The *container*() operation returns the containing parent object.
- The *equals*(*element*) determines if the element (an instance of `Element` class) is equal to this Element instance.
- The *set*(*property*, *element*) operation sets the value of the property to the element.

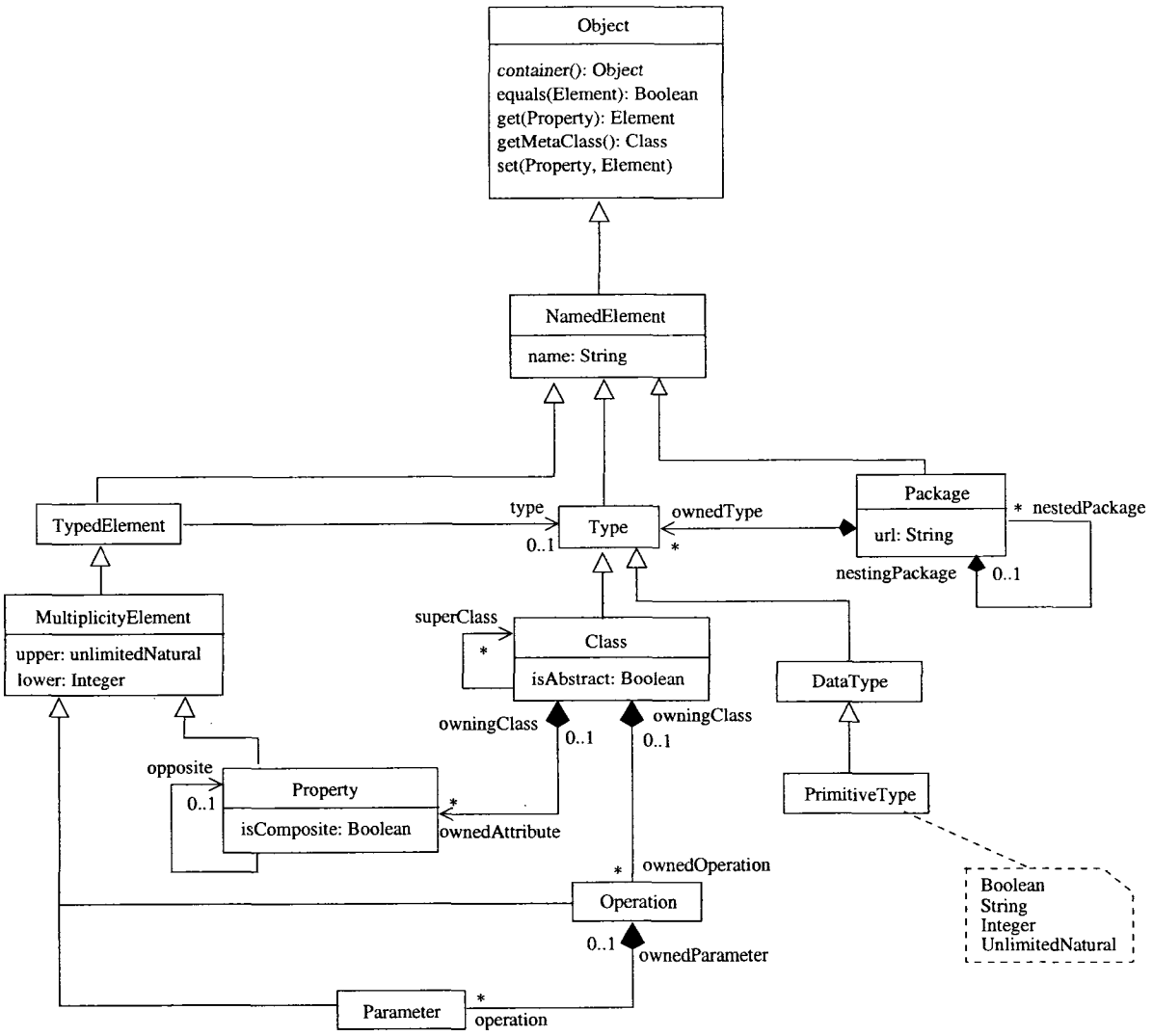


Figure 4.10: EMOF classes used in the composition technique

- The *get(property)* operation returns a List or a single value depending on the multiplicity.

The *isComposite* attribute defined in the class *Property* returns true if the object is contained in the parent object. Cyclic containment is not possible, i.e. an object can be contained in only one other object. The attributes, upper and lower, of class *MultiplicityElement*, represent the multiplicities of the associations at the metamodel level. For example, "0..1" represents a lower bound "0" and an upper

bound "1".

Additionally, the *getAllProperties*() operation is added to the `Object` class. It returns all the properties (including inherited properties) associated with the object instance. This will return elements that are composite as well as primitive. The primitive elements shown in the figure are String, Boolean, Integer datatypes.

4.2.3 Implementation of class model composition

The Kermeta implementation of the core parts of the composition metamodel (i.e., the metamodel obtained by excluding the CompositionDirective hierarchy) treats the model elements and instances of the other classes in the metamodel as objects (i.e., instances of the `Object` class shown in Figure 4.10). Hence, the model elements instances can use all the operations present in the `Object` class. The implementation is written independently of model element types and it uses reflection to obtain type information.

The composition metamodel extends the UML metamodel by adding behavior to the classes in the metamodel. The core concepts shown in figure 4.6 are described below:

- **Element**: Element is an extension of the UML metaclass, Element. It is extended by the operation *getMatchingElements(e[]: Element)*. Other operations *container(), get(property), set(property,element), getMetaClass()* of the EMOF Object class are used by the Element.

  - **getMatchingElements**: This operation takes in a set of elements and returns an element or set of elements that have the same syntactic type and signature as the element that invokes it (see table 4.1). The syntactic type check is performed by invoking the *getMetaClass()* and the *getAllPropeties()* operation defined in the EMOF Object class. The signature is obtained using *getSignature()*.

Table 4.1: Code snippet for getMatchingElements implemented in KerMeta

```
operation getMatchingElements(elements : Collection<EModelElement>)
                                    : seq EModelElement[0..*] is do
        var e1 : Mergeable e1 ?= self
        if e1 == void then
                result := Sequence<EModelElement>.new
        else
                result := elements.select{ e |
                                    var e2 : Mergeable e2 ?= e
                if e2 == void then
                        false
                else
                        e1.getSignature == e2.getSignature
                end
                }
        end
```

- **Mergeable**: This is an abstract class. Instances of Mergeable class are elements that are mergeable. Examples of mergeable elements shown in the figure are *Classifiers, Operations, and Models.*
  - **merge**: This operation merges the element with another mergeable element.
  - **sigEquals**: This operation checks if the element's signature is equal to the signature of another element.
  - **getSignature**: This operation gets the signature of the element based on the signature type.
- **Signature**: This class is used to obtain the signature of the mergeable elements. This class is linked to every mergeable element.
- **Composer**: This class composes the models with the compose method. The inputs to the compose method are the primary model and conext-specific aspect model.

Table 4.2: Code snippets from the merge part involving conflict detection

```
if me1.getSignature == me2.getSignature then
        if p.isComposite then
                stdio.writeln("Recursive merge")
                result.set(p, me1.merge(me2))
        else
                result.set(p, me1)
        end
else
        stdio.writeln("ERROR : Elements signature do not match")
end
```

```
if self.get(p) == other.get(p) then
        result.set(p, self.get(p))
else
        stdio.writeln("WARNING : conflicting value for property")
        result.set(p, self.get(p))
end
```

The signature type needs to be specified for composing the design class models if a default signature is not meant to be used. Problems may occur when the signatures don't match or the property values (for example, return type of an attribute) associated with the model elements don't match (see table 4.2). The developer can then resolve the conflict by modifying the model element(s) using composition directives or by changing the signature type. In the code snippet shown, the problem of conflicting values for a property is resolved by choosing the property value associated with the primary model element. Alternatively, a change property composition directive (described later in this thesis) can be used to change the value associated with the property. The result of the composition is a composed class model.

Chapter 5

Composition Directives

The signature-based composition can produce design class models with undesirable properties when the views of matching elements contain inconsistent information. For example, if two classes containing operations need to be composed and only a default signature type (*name* property) is used the resulting set of operations in the composed class may not exhibit the desired properties. In some cases, the problems can be resolved by modifying the context-specific aspect and primary models or by overriding some of the composition rules. Composition directives can be used for these purposes.

Figure 5.1 shows activities related to the application of composition directives. The composition activity, *Compose aspect and Primary models*, takes in three inputs: a primary model, a non-empty set of context-specific aspect models, and a (possibly empty) set of composition directives. In this activity, the aspect and primary models are composed using the signature-based approach and composition directives to produce a composed model.

After composition, the composed model can be analyzed against desired properties (referred to as *Properties to Verify* in Figure 5.1) to uncover design errors. For example, one can analyze the models against well-formedness rules to identify badly formed models or one can analyze the models against desired semantic properties (e.g., "only the owner of a file can delete the file") to uncover undesirable properties.

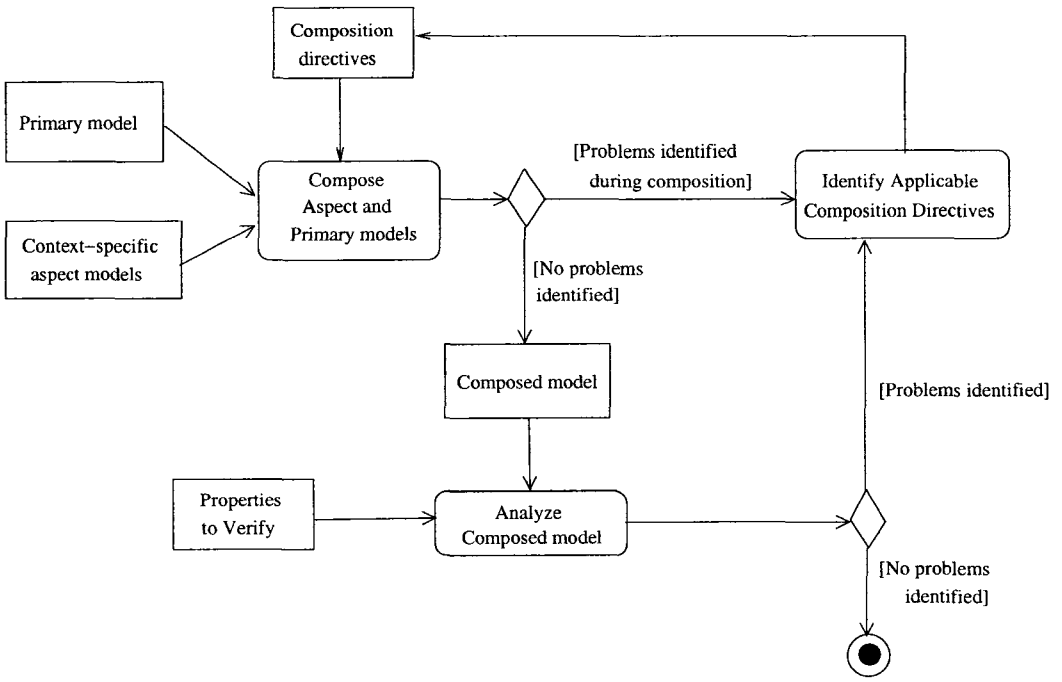


Figure 5.1: Using composition directives to resolve composition problems

In related work, we developed an approach to verify the correctness of the composition with respect to desired properties that the composed models must exhibit [65]. In the approach, the desired behavioral property to be verified is used in the composition procedure to generate proof obligations. Given (context-specific) aspect sequence models, a primary sequence model, and a behavioral property (specified in OCL), composition of the sequence models proceeds with the proof obligations being discharged as the behavioral views of the aspect and primary models are composed. Correctness of the composition with respect to the specified property is determined by discharging the proof obligations. The composition can be stopped as soon as it is determined that the proof obligation does not hold. This approach allows composers to determine at which point in the composition the property fails to hold. The information that is available when the composition is stopped can be used by a developer to determine what needs to be done to correct the situation.

In cases, where the uncovered problem is syntactic, an appropriate set of directives are identified and used to compose the context-specific aspect and primary models. For example, if two operations have the same name but are conceptually different (operation pre and post conditions are not the same), one of the operations can be renamed using a composition directive. In some cases, a single composition directive may not resolve the problem. For example, if a composed model does not exhibit a desired property due to the presence of an operation, the operation needs to be deleted from the composed model. It is not enough if the operation is deleted, all references to that operation also need to be deleted. In such cases, a set of composition directives need to be used.

In other cases where substantial changes to the aspect or primary model are required the application of composition directives may not be feasible. For example, it may be determined that the aspect model to be incorporated does not address the required feature, then another aspect model may be needed. In some cases where the primary model has to be significantly refactored usage of composition directives may not be feasible. Consider the case where a primary model needs to be composed with a replication aspect. The replication aspect used in the composition has a single active repository. The composition with a primary model yield a composed model with a single active repository. If the intent is to have multiple active repositories, one may need to change the replication aspect being used rather than instantiating the replication aspect multiple times. The composition directives described in this thesis do not address such problems.

The list of problems that are identified in this thesis are described below:

- If two model elements are of the same type and represent the same concept but have different names, it can lead a problem where the the resultant model is inconsistent.

- If the composed model does not exhibit the desired behavior, model elements

may need to be added or removed. This can lead to references that may point to elements that have been removed or references that do not exist for elements that have been added.

- If two model elements match with respect to the signatures but have conflicting values, the model element values need to be changed. For example, if a model element has $isAbstract = true$ and the matching model element has the property value $isAbstract = false$ then the values are considered inconsistent. In such cases, one value may need to be overriden by another.

- In a system with multiple aspects, the order in which aspect models are composed with a primary model may be important in the cases where different orderings produce different composed models. A cyclic-ordering conflict occurs when there is a cycle among ordering relationships defined over multiple aspects.

5.1 Using directives to resolve composition problems

In this section we provide examples of composition problems that can be resolved using composition directives. It is important to note that the composition approach discussed in the previous chapter does not provide systematic techniques for analyzing composed models nor for identifying appropriate composition directives once problems are uncovered. As stated earlier, the composition algorithm will flag cases where conflicting syntactic properties exist for model elements that are merged. It does not, however, detect conflicts that can arise as a result of inconsistent specifications of behavior or other semantic properties. For example, consider an example in which the primary class model has an operation specification associated with the *withdraw* operation. The post-condition states that amount *amt* is reduced from the *balance* after withdrawl from the original balance.

```
context Aspect::Account::withdraw(amt:Integer)
  pre  : balance > amt and amt <> 0
  post : balance = balance@pre - amt
```

Consider a primary called model that has an operation called *withdraw* with the following operation specification:

```
context Primary::Account::withdraw(amt:Integer)
  pre  : -- None
  post : balance = balance@pre
```

The post condition states that the *balance* after withdrawl should be the same as the original balance. The composition of two operations will result in conjunction of the two post-conditions, which means that the resultant post-condition should evaluate to true. However, sematically this is inconsistent because the value of balance after the execution of *withdraw* operation in *Aspect* is not the same as the *withdraw* operation in *Primary*. Uncovering such semantic properties requires semantic analysis of the composed model. However, the default composition rule that the composition of postconditions is a conjunction can be altered using a composition directive. Also, composition directives can be used to remove the pre and postconditions associated with an operation or to override the operation in the primary model with the operation in the aspect model.

Consider a simple example of a composition that leads to a faulty composed class model (see Figure 5.2). In the example, a modeler creates a primary model (see Figure 5.2(a)) in which an output producer (an instance of `Writer`) sends outputs directly to the output device it is linked to (instance of `FileStream`). The `writeLine` operation is associated with an operation specification. The modeler then decides to incorporate a buffering feature described by a buffering aspect model. Figure 5.2(b) shows the template class model for the buffering aspect. The aspect model describes

how entities that produce outputs (represented by instantiations of *BufferWriter*) are decoupled from output devices through the use of buffers. The operation templates `|bwrite()` in `|Buffer` and `|write()` in `|BufferWriter` are associated with template forms of operation specifications.

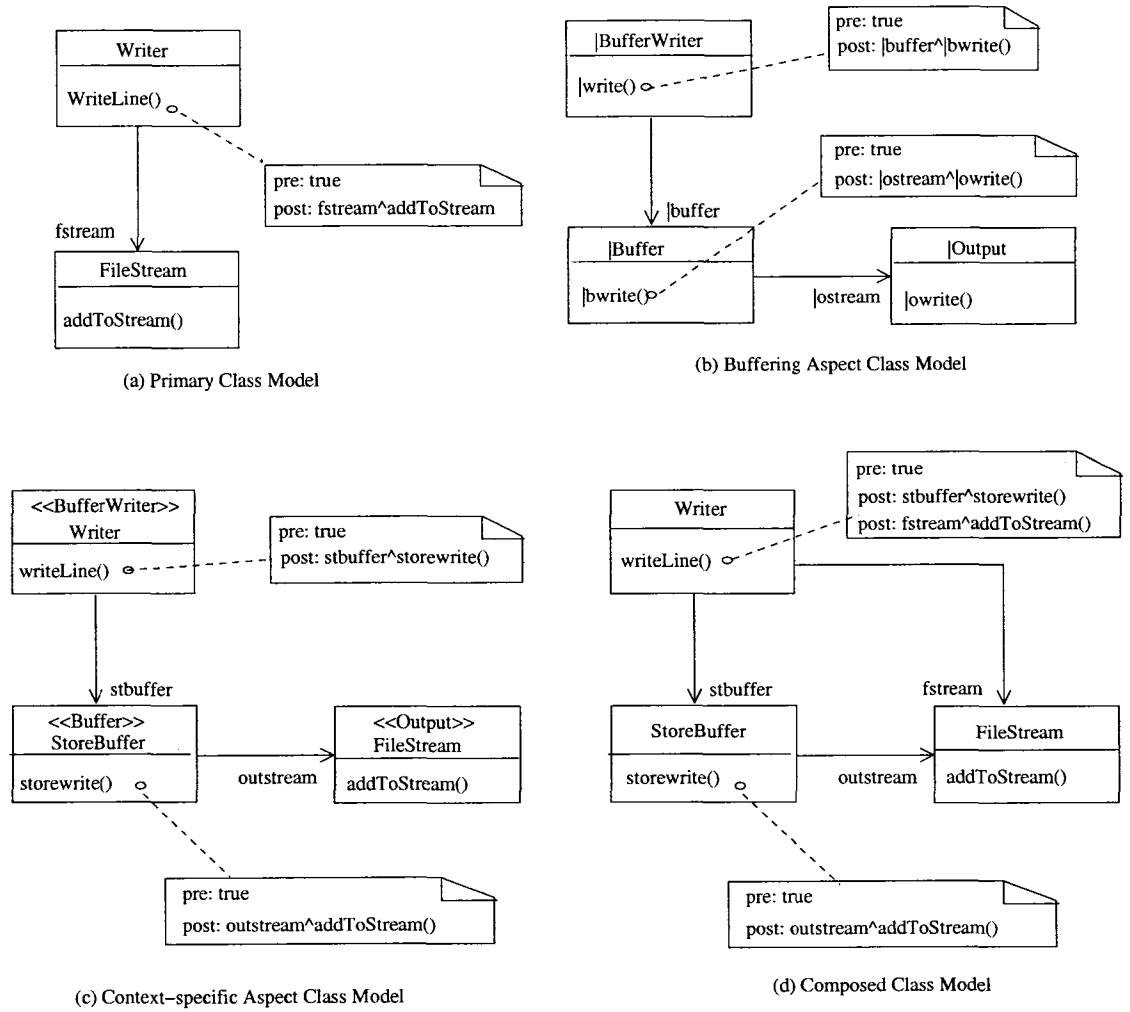


Figure 5.2: An example of faulty composition

Instantiating the buffering class diagram template produces a class diagram that describes how buffering is to be accomplished in the context of the primary model. The class diagram shown in Figure 5.2(c) is obtained from the buffering aspect class diagram using bindings shown in table 5.1.

Table 5.1: Bindings for buffering examaple

Aspect element name	**Application specific element name**
|Buffer	StoreBuffer
|Output	FileStream
|BufferWriter	Writer
|Buffer::|bwrite()	StoreBuffer::storewrite()
|BufferWriter::|write()	Writer::WriteLine()
|Output::|owrite()	FileStream::addToStreams()
|buffer	stbuffer
|ostream	outstream

The result of composing the class diagram shown in Figure 5.2(c) with the primary model class diagram shown in Figure 5.2(a) is presented in Figure 5.2(d). Composition is carried out by matching model elements using signatures consisting only of model element names. As per the composition rules desicribed in the previous chapter, the operation specifications associated with the operations are also merged.

The merging of the `writeLine()` operations in the primary and context-specific aspect models produces an operation that calls the buffer's operation `storeWrite()` and the filestream's operation `addToStream()`. This is not the desired result: The intent is to completely decouple `Writer` from `FileStream` using `StoreBuffer`. To resolve this problem, the following composition directives can be used:

- A composition directive that removes the association between `Writer` and `FileStream` in the primary model.
- A composition directive that removes the operation specification associated with the `writeLine()` operation in the primary model.

As another example, consider the partial context-specific and primary class models shown in 5.3. The `addUser()` operation in the primary model adds a user (instance of `User`) to the `Userlist` in `UserRepository` class. The `addUser()` operation

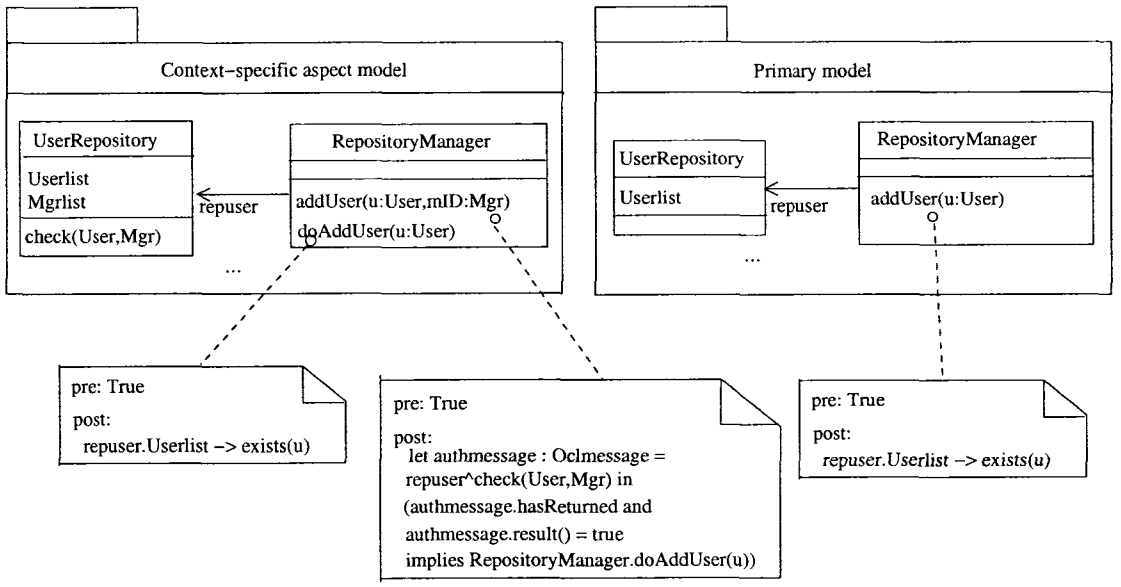


Figure 5.3: Example of a property conflict

in the context specific aspect model calls the `doAddUser` operation only when the manager (instance of `Mgr`) is authorized. The `doAddUser()` operation adds a user to the collection. Using signatures that consist only of model element names, the two `RepositoryManager` classes match and thus their properties are merged. During the merge of these two classes, the `addUser()` operations are matched and their specifications are merged. The resulting `addUser()` operation specification will have a property conflict: The specification from the primary model allows unconditional adding of users, but the specification from the context-specific model will allow adding of users only if the operation is authorized for the client. This is an example of a *property conflict*: A property conflict occurs when two matching elements (elements with the same signature) are associated with conflicting semantic properties. In this example, the intent is to merge the `doAddUser()` operation in the context specific aspect model with the `addUser()` operation in the primary model. To resolve this conflict and reflect the intent, a composition directive that renames the `addUser()` operation in the primary model to `doAddUser()` can be used. After this renam-

ing, signature-based composition will produce a composed model with the required properties.

In some cases, renaming elements may not be the appropriate way to resolve a conflict. Consider a context specific aspect class model that includes a class `FileStream` with an attribute `maxWriters : int` that is associated with the constraint {`maxWriters = 1`}, and a primary class model with a class named `FileStream` that contains an attribute `maxWriters : int`, with the constraint {`maxWriters = 2`}. If the matching attributes are merged, a property conflict will arise because the merged constraint ({`maxWriters = 1 and maxWriters = 2`}) is inconsistent. This conflict can be resolved by specifying, through a composition directive, that one element overrides the other, such that properties from the overriding element take precedence over those in the element being overridden.

In some cases, composition directives may be needed modify the composed model rather than primary model or context-specific aspect model to produce a model that satisfies the desired properties. For example, associations may be added between a class introduced by the primary model and another class introduced by a context-specific aspect model to provide required access to behaviors defined in the classes, or they may be removed to prevent access that is to be prohibited in the composed model.

The ability to rename, add, and remove elements can result in another type of conflict: *reference conflict.* A reference conflict arises when a reference is made to an element that no longer exists. To resolve this conflict, the affected references in a model must be identified and updated. Composition directives that identify and update specified references are needed.

In an aspect-oriented model that contains multiple aspect models, different composition orderings may produce different composed models [19, 60]. A particular ordering can lead to undesirable emergent behaviors. For example, consider an au-

diting feature and a password feature that are to be composed with a primary model. If the password feature is composed with the primary model before the auditing feature, then the end result could be a model in which the auditing feature captures and stores passwords. This may be an undesirable emergent behavior. Composition directives that can be used to specify the order used to compose multiple aspects with a primary model are needed.

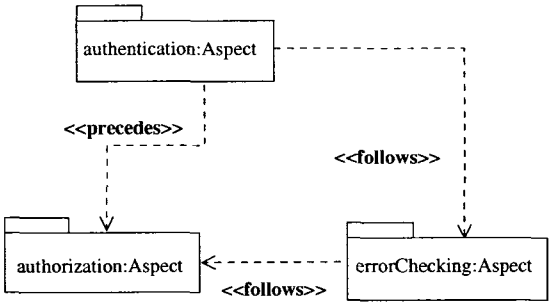


Figure 5.4: An example of cyclic ordering conflict

Defining composition ordering raises another type of conflict. A cyclic-ordering conflict occurs when there is a cycle among ordering relationships defined over multiple aspects (see Figure 5.4). Analysis can detect and correct ordering conflicts. For example, an authorization, replication and transaction aspect need to be composed with the primary model. If the order of composition states that replication precedes authorization, replication follows transaction, and transaction follows authorization. This can result cyclic conflict.

The above discussion indicates that the following list of actions should be captured by composition directives:

- Creating new elements.
- Adding elements to a Namespace.
- Deleting elements from a Namespace.

- Changing property values of elements.
- Finding and changing references to specified model elements.
- Specifying override relationships between matching elements.
- Changing default composition rules
- Specifying ordering relationships among multiple aspects.

The above list of actions reflects our current experience and may be incomplete.

5.2 Classification of directives

Composition directives can be classified as *Model Directives* and *Element Directives* (see Figure 5.5). Model directives are used to determine the order in which multiple aspect models are composed with a primary model. Element directives are used to determine how an aspect model elements are composed with the primary model elements. Element directives can be classified as *pre-merge*, *merge* and *post-merge* in terms of when they are applied in the composition process.

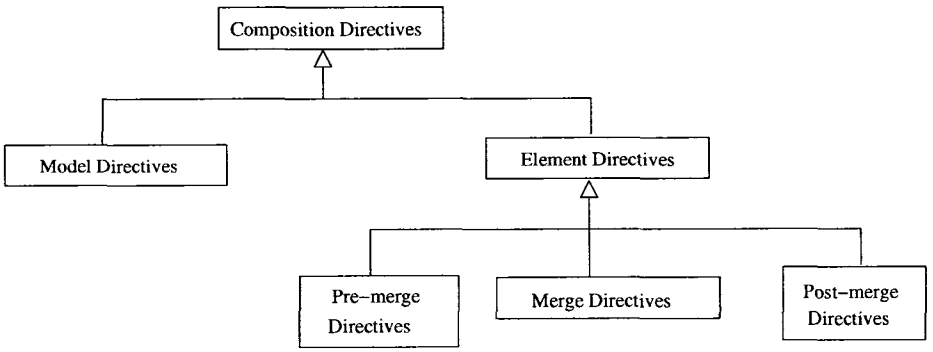


Figure 5.5: Classification of composition directives

Pre-merge Directives: These directives are used to modify the primary class model or the context-specific aspect class models before they are merged. For example, one

can rename model elements, delete model elements, or replace model elements (delete and add model elements) in the models. The application of pre-merge directives produces a changed aspect and primary model that are as inputs for the merge algorithm. In the previous chapter, the merge algorithm deviod of the composition directvies was described.

Merge Directives: These directives are used to override rules for merging model elements. For example, one can specify that a model element in one model completely replaces an element in another model. The inputs to the merge are primary, context-specific aspect models, and the merge directives and the output is a composed model that has default rules overridden.

Post-merge Directives: These directives are used to carry out simple modifications on the model produced after merging possibly modified primary and context-specific aspect models. The desired properties of the composed model can be verified against a set of properties to identify the applicable post-merge directives. The directives for renaming, adding, deleting, and replacing model elements also fall into this category.

5.3 The Directives

In this section we describe the composition directives that we have identified through application of the composition procedure on small case studies. Directives that affect only aspect and primary models are applied to the models before their elements are merged. Those that add elements to composed models and those that override composition rules are applied during merging. The directives that modify the model can be viewed as transformations on the models [7].

Each directive (except for the directives that override composition rules) is described using the following format:

- *Directive Name:* This section states the name of the directive or the form of names for a family of directives.
- *Application:* This section describes the purpose of the directives and describes the entities that the directives operate on.
- *Form:* This section describes the syntactic form of the directives.
- *Constraint:* This section gives the conditions that must hold if the directives are to have the intended effect. The constraint in this section is referred to as the directive precondition.
- *Effect:* This section describes the effect of the directives on their targets. The specification of effect is called the directive postcondition.

The following subsections describe the directives and gives examples of their application.

5.3.1 Element Directives List

We have identified the following element directives thus far:

- Creating new model elements (a family of directives)
- Adding model elements to a namespace
- Removing model elements from a namespace
- Changing properties (a family of directives)
- Replacing references to a model element in a namespace
- Overriding model elements
- Overriding composition rules (a family of directives)

When an element is created by a create directive, a handle that can be used to reference the element is provided. These handles are used in composition directives that are applied after the creation of the model elements. The names that appear on model elements in aspect and primary class models serve as references to the model elements in directives. For example, an association name or a role name can refer to an association in a directive.

5.3.1.1 Creating new model elements

The following describes the family of create directives.

Directive Name: create<metamodel class name>

The following are examples of names for create directives:

create`Association`, **create**`Class`, where `Association` and `Class` are the names of concrete classes in the UML metamodel.

Application: The create directives are used to create new model elements (i.e., model elements that are not in the primary or aspect class models being composed). In the composition metamodel, each concrete `Element` class is associated with a constructor. The create directives use these constructors to create model elements to ensure that the created elements are syntactically well-formed. The new element is not a member of any namespace when it is created. The new element can be added to the namespace using the **add** directive.

A create directive has set of operands that determines the arguments passed to the constructors of the model elements. The operands are a set of (*property name* = *property value*) pairs, where the *property name* is the name of a model element property in the UML metamodel and the *property value* is the value of that *property name*.

Form:

```
newHandle = create<Element> {operands}
```

The following is an example of a create directive that creates a concrete class with a name "NewClass". In the example, the property names are `name` and `isAbstract` and the corresponding property values are `''NewClass''` and `false`.

```
newClass = createClass {name = ''NewClass'', isAbstract = false}
```

As another example, consider the following create directives used to create a strong aggregation relation between two existing classes: `UserStore`, and `UserRep`.

```
userRepEnd = createProperty { isComposite = false,
    aggregation = none, type = aspect::UserRep,
    opposite = userStoreEnd, lower = 1, upper = 1 }

userStoreEnd = createProperty { isComposite = true,
    aggregation = composite, type = primary::UserStore,
    opposite = userRepEnd, lower = 1, upper = -1 }

userRep-userStore = createAssociation { name = "UserRep-UserStore",
    isDerived = false, memberEnd = [userRepEnd,userStoreEnd] }
```

The above directives indicate that two association ends (property) `userRepEnd` and `userStoreEnd` must be created before the association `userAuth-userMgmt` is created. We assign the value of "-1" to upper (representing the upper limit of a multiplicity) where "-1" represents the multiplicity "*". The "[...]" notation

is used to denote a collection of association ends in the **createAssociation** directive. Notice that the strong aggregation has still not been added to the namespace.

Constraint: There are no constraints for these directives.

Effect: A create directive provides a reference to a new model element that is valid. The new `Element` is not a member of any namespace.

5.3.1.2 Adding model elements to a namespace

Directive Name: add

Application: The **add** directive is used to add a model element to a namespace in a model. It can be used to add a newly created model element (i.e., one created by a create directive) to a namespace and to add an element from another namespace into a target namespace. The latter action is needed when a model element is migrated to a new namespace in order to ensure that the composed model exhibits the desired properties. Such a migration would involve removing the element from its original namespace (using the **remove** directive described later) and then adding it to the new namespace.

The **add** directive has one operand, the model element to be added.

Form: add `owner::elem`

In the above, the model element, `elem` is added to the namespace, `owner`.

Constraint: The target namespace must exist, the element to be added must have a unique name within the namespace, and the element must be an instance of a concrete UML metamodel class that can be owned by the namespace.

Effect: The element is in the target namespace.

5.3.1.3 Removing model elements from a namespace

Directive Name: remove

Application: The **remove** directive is used to remove a model element from a namespace. It is used when the presence of certain model elements compromises desired properties of the composed model. For example, consider a security aspect model that requires that certain associations not exist in the composed model because their presence can lead to leaks of sensitive information. The **remove** directive can remove these associations in the primary model.

Removing a composite model element involves removing all its contained parts. For example, removing an association involves removing its association end properties (but not the classes at the association ends).

Removing a model element can result in models with hanging references: References to the removed element may be present in the namespace and elsewhere (e.g., in OCL expressions) after removal. Use of the directive should be coupled with the use of other directives that take care of the hanging references. For example, one can use the **replaceOccurrences** directive (described later) to replace references to the deleted element with references to other elements.

The **remove** directive has one operand, the model element that is to be removed.

Form: remove `owner::elem`

In the above, the model element, `elem` is removed from the namespace, `owner`.

Constraint: The namespace must exist in a model. The element must be in the

namespace before the directive is applied.

Effect: The element is not in the namespace.

5.3.1.4 Changing properties of model elements in a namespace

The family of directives for changing model element properties are described below.

Directive Name: change<property name>

Examples of change directive names are **changeisAbstract**, and **changename**. The **changename** directive is written more concisely as **rename**.

Application: The **changeProperty** directive is used to change the value of a model element property. This directive can be used to force or prevent matching of model elements by changing the property values used to determine element matches. For example, if the signatures of two classes match but the value of `isAbstract` property is `true` for one class and the value of `isAbstract` property for the other class is `false`, this directive can be used to change the property value of one of the classes to match with the other class. In cases where matching is based only on the names of elements, this directive can be used to rename elements so that they match or do not match.

This directive has two operands. The first is the model element with the property, the second is the new value of the property.

In our examples we often use this directive to rename model elements and thus we use a more concise name for the directive: **rename**. The renaming directive is often applied to the primary class model, because renaming of elements in the context-specific aspect class models can also be accomplished by binding specification. The application-specific values that are bound to the aspect model templates

can be changed instead of using a renaming directive.

Form: change<property name> `owner::targetElement` **to** `propertyValue`

The following is an example of a `changeProperty` directive that changes the value of `isAbstract` property of a class to `true`.

change`isAbstract Primary::Auth` **to** `true`

In the above example, `Auth` is the class and `Primary` is the namespace.

In the cases where the property to be changed is a model element name one can use the form below:

rename `owner::targetElement` **to** `newName`

The following is an example of a `rename` directive that changes the value of the `name` property of an operation to `doAddUser`.

rename `Primary::AddUser` **to** `doAddUser`

In the above example, `AddUser` is the operation and `Primary` is the namespace. The operation name `AddUser` is changed to `doAddUser`.

Constraint: The element must exist in a primary, aspect or composed model. The changed property value must be valid. For example, an `isAbstract` property cannot be changed to anything other than `true` or `false`.

Effect: The specified property value in the target model element has the new value.

5.3.1.5 Replace references to a model element in a namespace

Directive Name: replaceOccurrences

Application: The **replaceOccurrences** directive is used to replace references to a model element with references to another model element in a namespace. It is

often used in conjunction with directives that add and remove model elements. For example if an association referenced in an OCL expression is removed then one can use this directive to change the reference in the OCL expression.

The **replaceOccurrences** directive has two operands: The first is a reference to a model element, and the second is a reference to another model element.

Form: replaceOccurrences `owner1::elem` **with** `owner2::replacementElem`

The above states that references to `elem` in the namespace `owner1` are to be replaced by references to `replacementElem` in the namespace `owner2`.

Constraint: The referenced elements must exist in the namespaces.

Effect: All existing references to the model element `owner1::elem` are changed to references to the element `owner2::replacementElem`.

5.3.1.6 Overriding a model element

This composition directive is similar to the override relationship proposed by Clarke et al. [12].

Directive Name: override

Application: The **override** directive defines an override relationship between two potentially conflicting model elements. It indicates that the properties of a model element takes precedence over properties of a matching model element during composition.

When an **override** relationship is defined for two model elements, the relationship propagates to the contained model elements. If a class in the aspect class model overrides a class in the primary model, the operations in the aspect class model override

the operations in the primary class model. However this may not be feasible for all cases. Since, the consequences of the implicit overrides are not immediately obvious, explicit **override** relationships should be defined for contained model elements whenever feasible and practical.

The **override** directive has two operands. The second operand is the model element that overrides the first operand.

Form: override `owner1::elem1` **with** `owner2::elem2`

The above states that `elem2` in namespace `owner2` overrides `elem1` in namespace `owner1`.

Constraint: `owner1::elem1` and `owner2::elem2` must exist in separate models, one in a primary class model, and the other in a context-specific aspect class model. The two elements must match.

Effect: During composition, the properties of `elem1` are replaced by properties of `elem2`.

5.3.1.7 Overriding default composition rules

When merging matching model elements with different property values, the composition mechanism can use default rules to determine the property values that will be used in the composed model. In the previous chapter we defined the default rules for composing model elments.

Sometimes one may want to change the default rules when composing models. For example, one may want to use the weaker multiplicity constraint at the ends of composed associations, or one may want to have a conjunction of preconditions

rather than disjunction. *Override composition rule* directives are used for this purpose. In our approach, each rule is associated with a set of possible variations and a directive for each variation is defined. For example, the association end multiplicity rule is associated with the following directive:

association end multiplicity rule `owner1::assocend1; owner2::assocend2` **weaker**

Use of this directive indicates that the weaker of the two multiplicities at the specified associations are to be used in the composed model. One can also override the rule globally using the following directive:

association end multiplicity rule weaker

For the operation specification rule we have the following directive:

operation specification rule `owner1::aclass1::`**PreSpec**`(anoperation1),`
`owner2::aclass2::`**PreSpec**`(anoperation2)` **conjunct**

The above states that the precondition of the operation formed by merging the matching operations `anoperation1` and `anoperation2` is the conjunction of their preconditions. A similar directive for postconditions can be defined:

operation specification rule `owner1::aclass1::`**PostSpec**`(anoperation1),`
`owner2::aclass2::`**PostSpec**`(anoperation2)` **disjunct**

This thesis has a limited number of composition rules. In the cases where we do not have such rules, composition results in a conflict when the property values differ.

5.3.2 Element Directives Examples

The following are examples of composition scenarios that require the use of element directives to produce desired results. In the examples we show the effect of element directives in terms of before and after diagrams. We also show the resultant composed model obtained after the application of composition directives.

5.3.2.1 Example 1: Buffering Application

The buffering application described in section 5.1 (see Figure 5.2(d)) results in a faulty composition since the intent is to completely decouple `Writer` from `FileStream` using `StoreBuffer`. The faulty composition can be avoided by using composition directives that do the following (the aspect and primary models are shown in Figure 5.6):

1. Remove the association between `Writer` and `FileStream` in the primary model: In the desired composed model, all writing to the `FileStream` is done via the `StoreBuffer`. The `Writer` should not have direct access to the `FileStream` in the composed model.

2. Remove the OCL specification for `writeLine()` in the primary model: A reference is made to the association in the OCL specification for `writeLine()` (the postcondition references the `fstream` association end). This constraint should not be in the composed model since all writing to the file stream is done via the buffer. For this reason, the OCL constraint associated with `writeLine()` in the primary model is removed.

The directives that accomplish the above are given below:

1. **remove** `primary::Writer::fstream`

2. **remove** `primary::Writer::Spec(writeLine)`

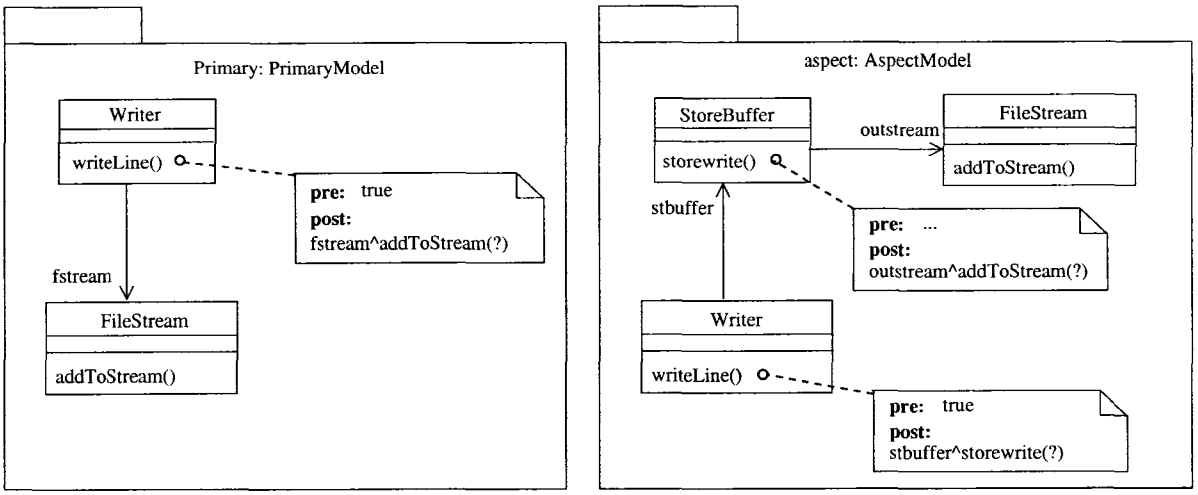


Figure 5.6: Example 1. Before Application of directives.

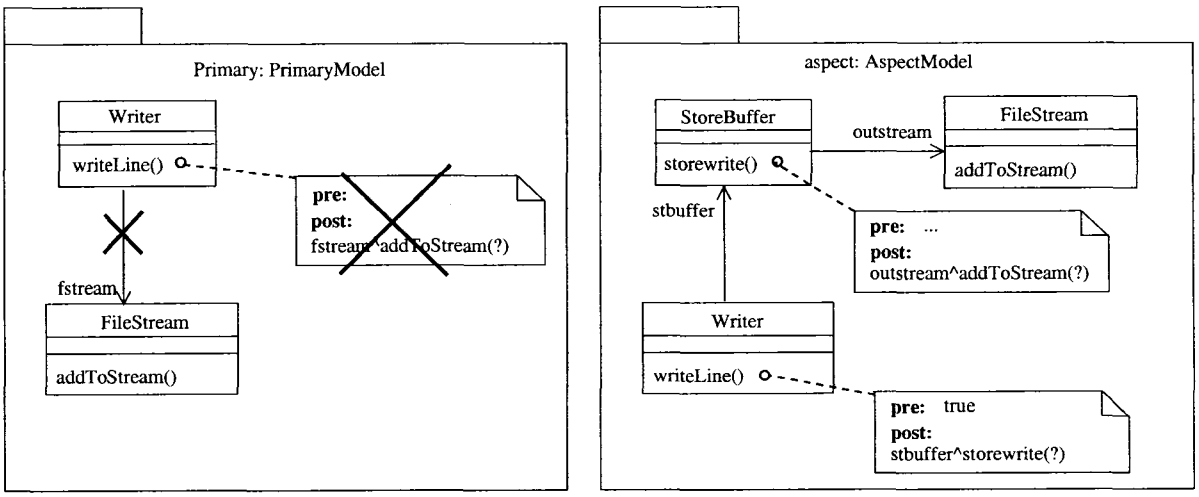


Figure 5.7: Example 1. After Application of remove directives.

In the above, **Spec**(writeLine) refers to the specification associated with the operation `writeLine()`. Figure 5.6 and Figure 5.7 illustrate the before and after effect of the directives on the primary and aspect class models. An "X" indicates the removal of an element.

An alternative way to accomplish the above would be to use the **override** directive instead of the second **remove** directive for overriding operations.

1. **remove** `primary::Writer::fstream`

2. **override** `primary::Writer` **with** `aspect::Writer`

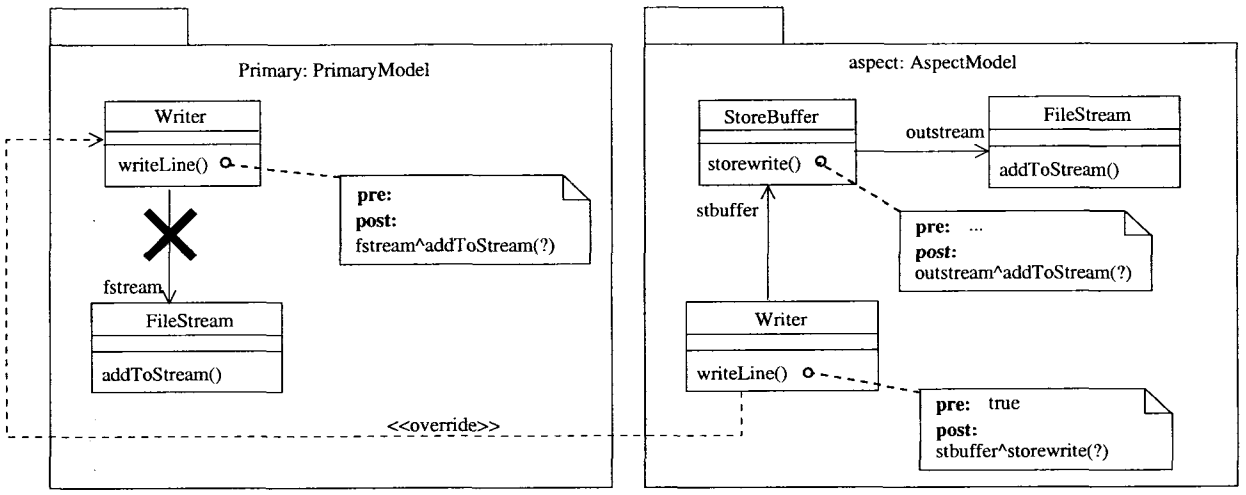


Figure 5.8: Example 1. After Application of remove and override directives.

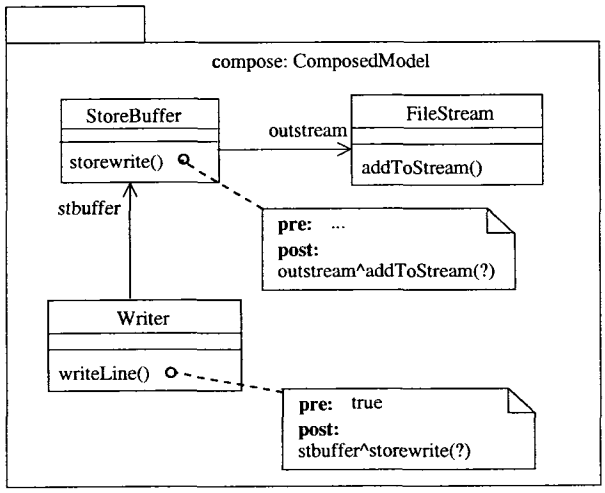


Figure 5.9: Example 1. Resultant composed model after application of directives.

Figure 5.8 illustrates the effect of the using the remove and override directives on the primary and aspect models. The override directive overrides the primary model

element with the aspect model element. Since the override is implicit, the operations and operation specification of the primary model elements are also overridden. The resultant composed model after the application of directives is shown in Figure 5.9.

5.3.2.2 Example 2: User Management Application

The following example, from France et al. [15], illustrates the use of the **create**, **add**, **remove** and **replaceOccurrences** directives. The aspect model shown in Figure 5.10 presents a design view in which add and delete user actions must be authorized before they are carried out. The primary model describes a design view in which authorization does not occur. The objective of the composition is to produce a composed model in which the authorization behavior in the aspect is incorporated into the primary model. In Figure 5.10, the `UserAuth` class in the aspect model performs authorization checks on clients requesting the addition or deletion of users from the system. In the composed model, `Manager` client must request the add and delete user operations by calling the corresponding operations in `UserAuth` and should have no direct access to the `UserMgmt` class. To accomplish this, a directive is used to remove the `accesses` association in the primary model:

(1) **remove** `primary::Manager::accesses`

There are references to the `accesses` association in `Manager` that must be replaced or removed. In this case, references to `accesses` in the primary model must be changed to `uaccesses` in the context specific aspect model, because all access to the operations is made via the `uaccesses` association in the composed model. The following directive is used to accomplish this:

(2) **replaceOccurrences** `primary::Manager::accesses`

with `aspect::Manager::uaccesses`

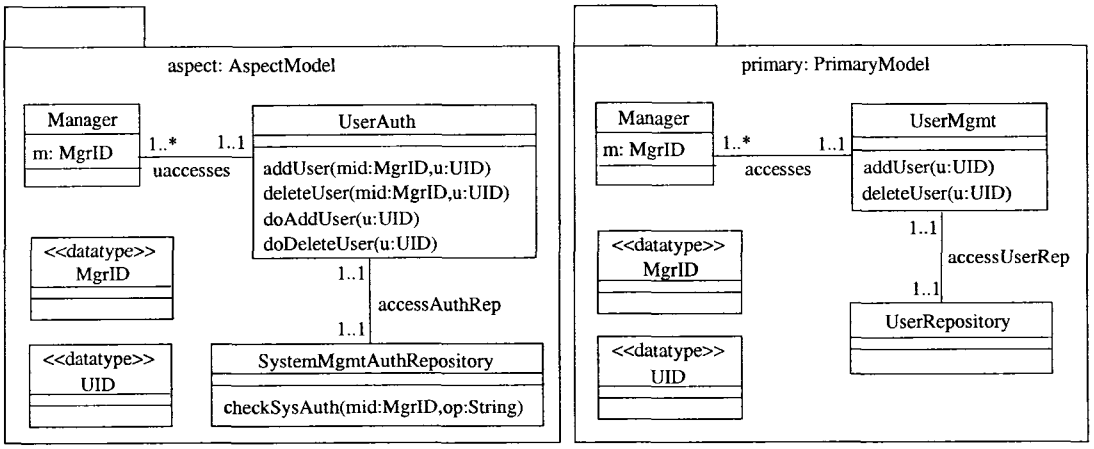


Figure 5.10: Example 2. Before application of directives.

The definitions of the `addUser` and `deleteUser` operations in `UserAuth` include an authorization check. In the aspect model, if a `Manager` client is authorized to carry out the add or delete action a call is made to the respective `doAddUser`, `doDeleteUser` operations. In the described composed model, the operations `addUser` and `deleteUser` in `UserMgmt` carry out the add and delete user actions, respectively. To make this possible a composition directive that adds an aggregation between the `UserMgmt` class and the `UserAuth` class is used:

```
(3) userAuthEnd = createProperty { isComposite = false,
      aggregation = none, type = aspect::UserAuth,
      opposite = userMgmtEnd, lower = 1, upper = 1 }

    userMgmtEnd = createProperty { isComposite = true,
      aggregation = composite, type = primary::UserMgmt,
      opposite = userAuthEnd, lower = 1, upper = -1 }
```

```
userAuth-userMgmt = createAssociation {
  name = "UserAuth-UserMgmt" , isDerived = false,
   memberEnd = [userAuthEnd,userMgmtEnd] }
```

Once the new `Association` is created, we need to add it to the composed model. The composition directive that accomplishes this is given below. We reference the composed model using the name *comp*:

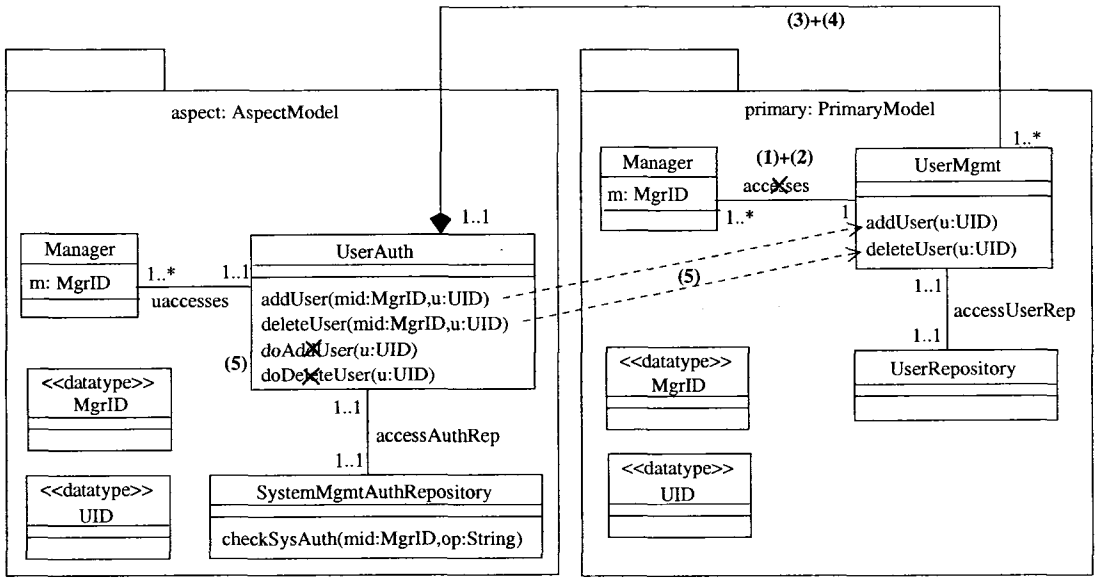


Figure 5.11: Example 2. After application of directives.

(4) **add** `comp::userAuth-userMgmt`,

add `comp::UserAuth::userAuthEnd`,

add `comp::UserMgmt::userMgmtEnd`

There are two options for creating a composed model in which authorized calls to `addUser` and `DeleteUser` are made: The first option is to replace the specifications

of doAddUser and doDeleteUser so that they delegate the actions to the respective operations in UserMgmt using the new association. The second option is to replace the calls to doAddUser and doDeleteUser by calls to the respective operations in UserMgmt. We give the directives that accomplish the latter option below:

(5) **replaceOccurrences** `aspect::UserAuth::doAddUser`
with `primary::UserMgmt::addUser()`,
remove `aspect::UserAuth::doAddUser`,
replaceOccurrences `aspect::UserAuth::doDeleteUser`
with `primary::UserMgmt::deleteUser()`,
remove `aspect::UserAuth::doDeleteUser`

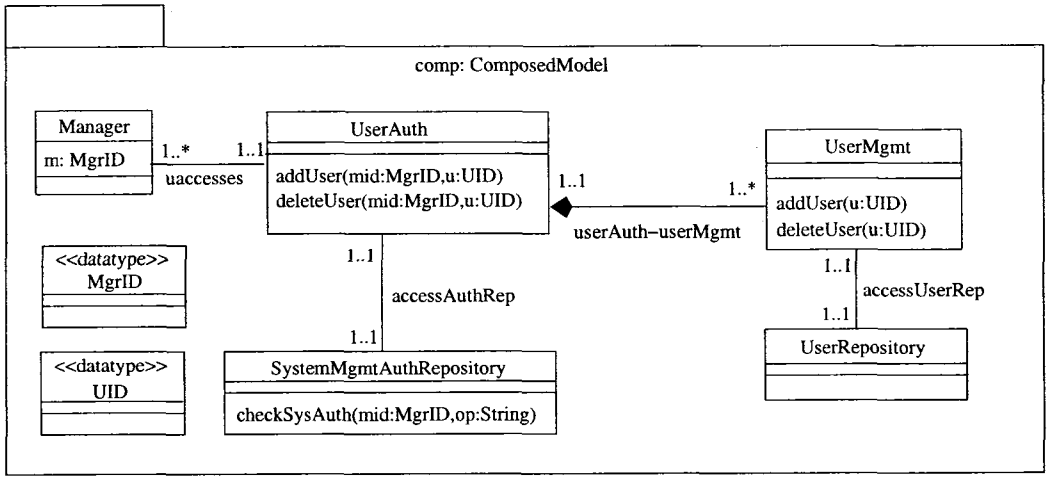


Figure 5.12: Example 2. Composed model after application of directives.

The effect of applying the directives on the aspect and primary models is shown in Figure 5.11. The association between UserMgmt and UserAuth is in the composed model but not in the aspect or primary models. The dependencies from the addUser and deleteUser operations in UserAuth indicate that they call the respective operations in UserMgmt. The resultant composed model after the application of the

composition directives is shown in Figure 5.12. The composed model has `addUser` and `deleteUser` operations in `UserMgmt` class that can be performed only when they are authorized.

5.3.3 Combining Element Directives

The examples and the descriptions of composition directives provide some indication that use of some element directives are often coupled with the use of others. For example, adding or removing a model element sometimes requires use of directives such as the **replaceOccurrences** directive to avoid hanging references. An overview of combined directives in the pre-merge, merge and post-merge categories are given below:

Pre-Merge Combined Directives: These directives are often combinations of directives for deleting model elements from a namespace or renaming model elements in a namespace. The directives are often combinations of **change-property** and **replaceOccurrences** directives.

Merge Combined Directives: Matching directives are combined directives that force the matching of elements or disallow the matching of elements during merge. Combinations of the **override** and **replaceOccurrences** directives are often used to override rules used to merge model elements.

Post-Merge Combined Directives: These directives are often combinations of directives for creating model elements, adding model elements to a namespace and deleting model elements from a namespace.

5.3.4 Model Directives

Model directives determine how a set of models are composed. The model directives we have identified constrain the order in which context-specific aspect models

are composed with a primary model. These directives can define weave-ordering relationships between aspect models. A weave-ordering relationship is a binary constraint that specifies an ordering between two aspect models. There are two cases: An aspect model must be composed before another, or an aspect model must be composed after another.

5.3.4.1 Precedes

Directive Name: precedes

Application: This directive specifies that one aspect model is to be composed with a primary model before another. This directive has two aspect models as operands. The first operand is the aspect model that is to be composed the second operand.

Form: `former` **precedes** `latter`

Constraint: Both aspect models must exist.

Effect: A weave-ordering relationship is created between the two aspect models, and added to the set of weave-ordering constraints maintained by the composer. This directive does not imply that `former` will be woven immediately before `latter`. It simply requires that `former` be woven some time before `latter`.

5.3.4.2 Follows

Directive Name: follows

Application: This directive specifies that one aspect model is to be composed with a primary model after another. This directive is provided only to increase the readability of composition directives. It may be interpreted as equivalent to the **precedes** directive with the operands switched. This directive has two aspect model

operands. The first operand is the aspect model to be composed after the second operand.

Form: `later` **follows** `earlier`

Constraint: Both aspect models must exist.

Effect: A weave-ordering relationship is created between the two aspect models, and added to the set of weave-ordering constraints maintained by the composer. This directive does not imply that `later` will be woven immediately after `former`. It simply requires that `later` be woven some time after `former`.

5.3.5 Weave Ordering Example

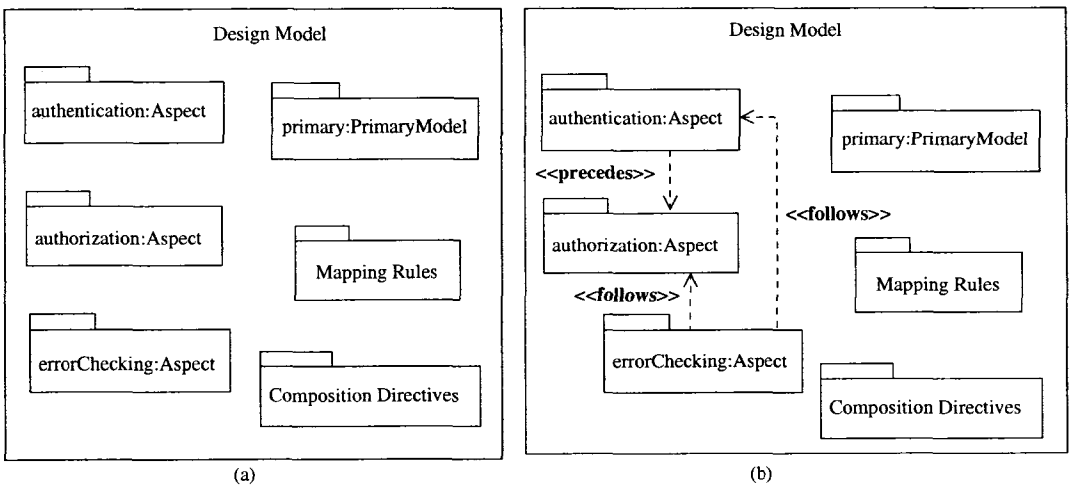


Figure 5.13: Example 4. Specifying Weave Order

Consider the aspect design model in Figure 5.13(a). There are three different aspect models and the primary model. In this example, the authentication aspect model needs to be composed before the authorization aspect model, because autho-

rization without authentication is meaningless. Therefore, we declare the following composition directive to make the order explicit.

1. `authentication` **precedes** `authorization`

We could have also defined a composition directive using the **follows** directive with the operands reversed to achieve the same result.

Suppose we also wish to weave the `errorChecking` aspect model last. The following composition directives accomplish this:

1. `errorChecking` **follows** `authorization`

2. `errorChecking` **follows** `authentication`

The application of the directives is shown in Figure 5.13(b). The dependency from authentication to authorization illustrates the weave-order relationship that specifies that authentication must be woven before authorization, and the dependencies from errorChecking to each of the other aspects illustrates the two binary weave-order relationships that specify errorChecking as the last aspect to be woven.

Chapter 6

Interaction Model Composition

In this chapter, we present a interaction model composition technique for composing aspect and primary sequence models. Composing sequence models requires one to merge the messages of the primary and aspect sequence models to obtain a corresponding merged sequence of messages in the composed model. Furthermore, the lifelines in the composed model need to correspond to the lifelines involved in the source sequence models. Thus, the aspect sequence diagram template should be bound to application-specific values before the models can be composed.

6.1 Composing message sequences

The composition of message templates in the aspect sequence diagram template with the messages in the primary sequence model requires the specification of how and where the application-specific values of the message templates are to be composed with the primary model messages. The interaction model composition technique we have developed involves tagging the primary sequence model. The *tags* identify the set of primary sequence model elements (e.g., lifelines and messages) to which the aspect sequence model elements need to be composed. The technique is analogous to weaving techniques used in aspect oriented programming. The model level tags specify the join points where the aspect sequence needs to be introduced, and also define the type of composition that needs to occur between the primary sequence

model and aspect sequence model. The interaction model composition occurs under the assumption that the default message flow is described in the primary sequence model. The aspect sequence models can be composed with the primary sequence model in two ways:

1. By inserting behavior represented by a message or a sequence of messages in the aspect model at a particular point in the primary model

2. By replacing a message or a sequence of messages in the primary model with a sequence of messages in the aspect model.

Two types of tags are applied on the primary sequence model to specify how the aspect sequence model needs to be composed: `simpleAspect` tags and `compositeAspect` tags. The `simpleAspect` tags are associated with a single lifeline or a single message in the primary model while `compositeAspect` tags are associated with multiple lifelines and multiple messages.

6.1.1 Overview of simpleAspect and compositeAspect Tags

A `simpleAspect` tag is a stereotype <<`simpleAspect`>> of a UML message to self. A `compositeAspect` tag is a stereotype <<`compositeAspect`>> of combined fragment. Stereotyping the interaction modeling elements enables one to use the current UML tools to tag the primary sequence model. The simpleAspect or compositeAspect tags are specified on the primary sequence model to imply the composition of a `simpleAspect` or a `compositeAspect`.

The tags specified in a primary sequence model have a syntactic structure as follows:

<<aspect>> <aspectname> (<paramlist>)[(<bindspecification>)]

The `aspect` refers to the type of aspect - `simpleAspect` or `compositeAspect`. The `aspectname` refers to the name of the aspect sequence model that needs to be

composed. For example, sdTransaction refers to transaction sequence model. The <paramlist> is a list of zero or more parameters used to configure the aspect. For example, one phase or two phase commit transaction protocols can be assigned as an input parameter in a transaction aspect sequence model to specify the transaction protocol. The `bindspecification` is used to bind the aspect sequence model elements with primary sequence model elements. Figure 6.1 shows examples of a primary sequence model tagged with `simpleAspect` and `compositeAspect` tags.

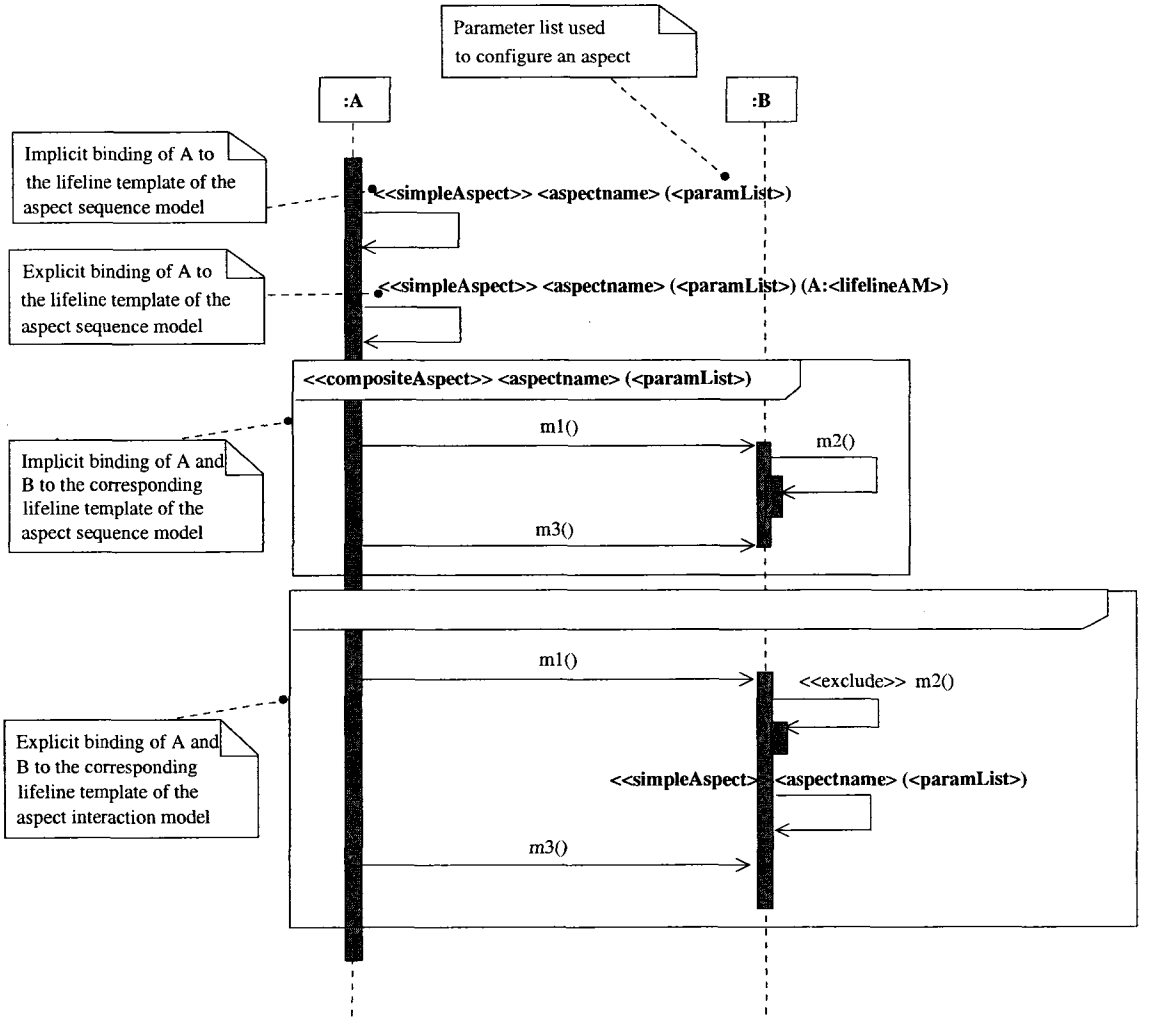


Figure 6.1: Tags on the Primary model

The binding specification binds the aspect sequence diagram template to

application-specific values. The binding can be done either explicitly or implicitly.

6.1.1.1 Binding semantics for simpleAspect Tags

An explicit bind specification has the following form <lifelinePM>:<lifelineAM>, where lifelinePM is the lifeline of the primary sequence model that should be bound to a particular lifeline template in the aspect sequence diagram template (lifelineAM). The bind specification is optional and is only required when the bindings need to be specified explicitly. If the bind specification is not shown explicitly implicit binding semantics are used.

In a primary sequence model with `simpleAspect` tag, the lifeline in the primary sequence model from which the tag originates will be bound to the corresponding lifeline template in the aspect sequence diagram template. This requires that there be only one lifeline template in the aspect sequence diagram template. A `simpleAspect` uses implicit binding semantics since only one lifeline template needs to be bound. The Figures 6.2, 6.3 and 6.4 illustrate `simpleAspect` tags that use implicit binding semantics. The primary model shown in figure 6.2 has two lifelines: A and B. The lifeline A is tagged twice with an aspect named `AspectModelTest`, first with parameter p, and second with parameter r. These parameters are used to configure the aspect model.

The aspect sequence model template is shown in Figure 6.3. The aspect has one input parameter (`x:String`), and one lifeline template (`|C`). The other two lifelines, `D` and `E`, are not template elements and are not bound to any element in the primary sequence model.

The composed sequence model is obtained using the implicit binding semantics of the `simpleAspect` tag. The `|C` is bound to `A` while `D` and `E`, and the corresponding messages in the sequence appear as specified in the aspect sequence model. The parameters `p` and `r` are passed to the aspect sequence model template. The result

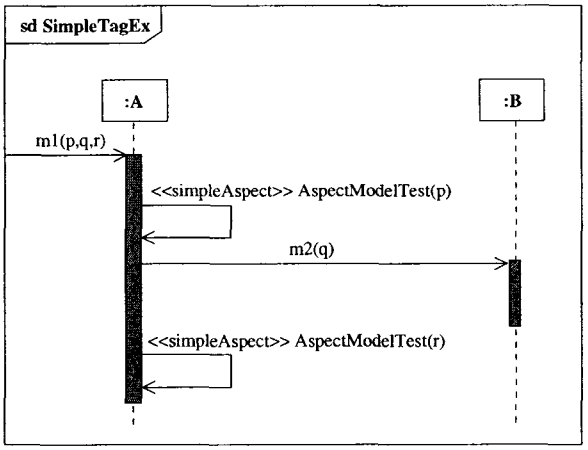


Figure 6.2: Primary sequence model with simpleAspect tags

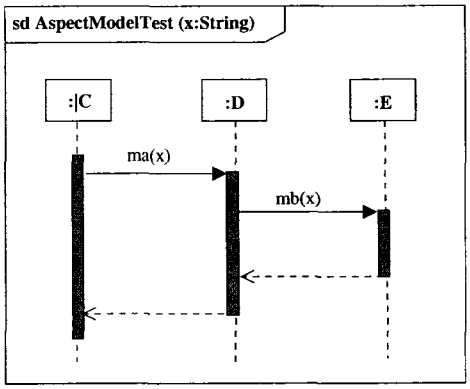


Figure 6.3: An example simpleAspect sequence model

of composing the application-specific values of the aspect sequence model template with the primary sequence model is shown in Figure 6.4.

6.1.1.2 Binding semantics for compositeAspect Tags

The implicit binding semantics of the `compositeAspect` tag specify that the lifeline from which a message within the `compositeAspect` tag originates will be bound to the corresponding lifeline template in the aspect model, and the lifeline to which the message is received will be bound to the corresponding template lifeline in the aspect model. The `compositeAspect` tag in the primary model has default semantics

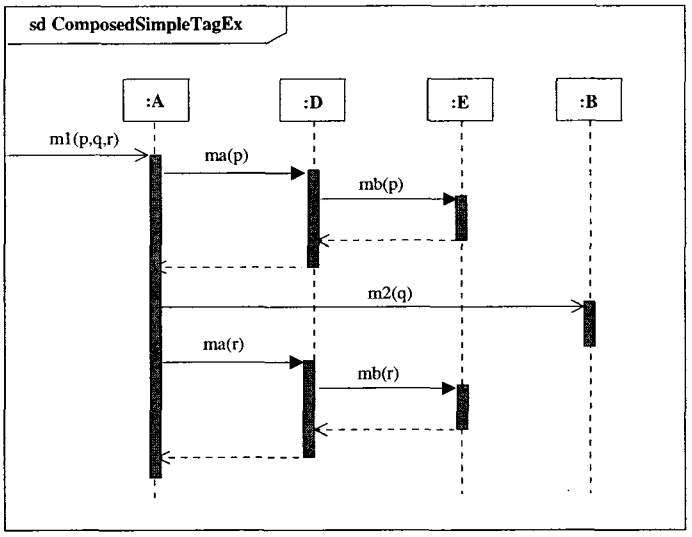


Figure 6.4: Composed model obtained from simple tagged primary model

where message templates in the aspect model are bound to application-specific values before they are composed with the corresponding messages in the primary model. The default semantics can be overridden using the an <<`exclude`>> tag. For example, the <<`exclude`>> tag in Figure 6.1 implies that the aspect sequence should not be applied on the message `m2()`. A primary model tagged with a `compositeAspect` can also have a `simpleAspect` tag within the `compositeAspect` tag as shown in figure 6.1.

During implicit binding care should be taken such that a lifeline in the primary sequence model is not bound to conflicting lifeline templates in the aspect sequence model templates. The bind specification has to be consistent with the set of bindings specified during class model composition. In cases where composition directives have been used to add or remove model elements during class model composition, the corresponding element instances should be added or removed from the sequence model.

6.1.2 Fragments in compositeAspect Sequence Model

Tagging the primary model using a `compositeAspect` tag implies that the messages in the primary model are composed based on a pre-defined schema specified as fragments in the composite aspect sequence model. Each fragment performs different roles and is treated differently during the composition of the aspect sequence model with the primary sequence model. The fragments are shown in figure 6.5. In our composition technique, the fragments are treated as stereotypes of interaction fragments. The different fragments in a composite aspect sequence model are `begin, before, body, after, end`. When a `compositeAspect` tag is used in the primary model, the messages present in the fragments of the corresponding composite aspect sequence model will be composed with the messages encompassed by the `compositeAspect` tag according to certain rules. The rules are as follows:

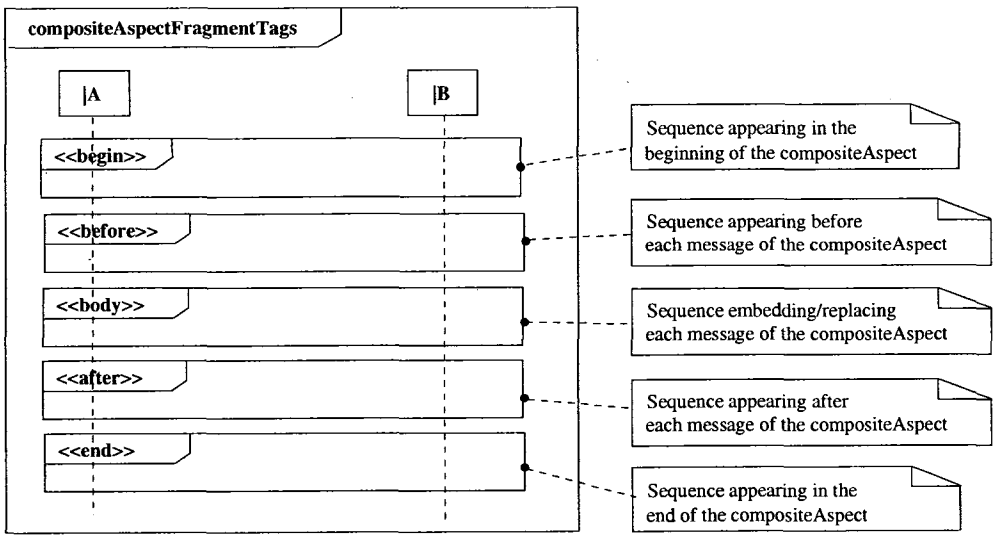


Figure 6.5: Fragments in the compositeAspect sequence model

- The <<`begin`>> fragment defines the message or sequence of messages in the aspect model that should appear in the beginning of the `compositeAspect` tagged primary model, thus, before any other sequence immediately preceding

the messages encompassed in the `compositeAspect` tagged primary model. In the composed sequence model the begin sequence of the aspect model appears exactly once for every `compositeAspect` tag in the primary model.

- The <<`before`>> fragment defines the message or sequence of messages that will appear before each message encompassed in the `compositeAspect` tagged primary model.

- The <<`body`>> fragment defines the message or sequence of messages that will refine each message encompassed within the `compositeAspect` tagged primary model. The semantics of the messages in the primary model should be maintained during the refinement.

- The <<`after`>> fragment defines the message or sequence of messages that will appear after each message encompassed in the `compositeAspect` tagged primary model.

- The <<`end`>> fragment defines the message or sequence of messages that will appear at the end of the `compositeAspect` tagged primary model. When the composite tagged in the primary model is composed with the aspect model, the end sequence appears exactly once for the composite tag.

The above listed fragments will appear in the aspect sequence diagram template and not in the primary sequence model. Furthermore, one or more of the above fragments are required to compose the instantiation of the aspect sequence model template with a `compositeAspect` tagged primary sequence model, but it is not necessary that all fragments exist in the aspect sequence model template. The messages in all the above fragments are composed as per the rules stated unless there is an explicit <<`replace`>> tag attached to the message templates in the aspect model sequence template. The `replace` tag identifies messages in the `compositeAspect`

tagged primary model that need to be replaced with application-specific values of message templates in the aspect sequence model template rather than being composed. The `replace` tag occurs mostly in the `body` fragment.

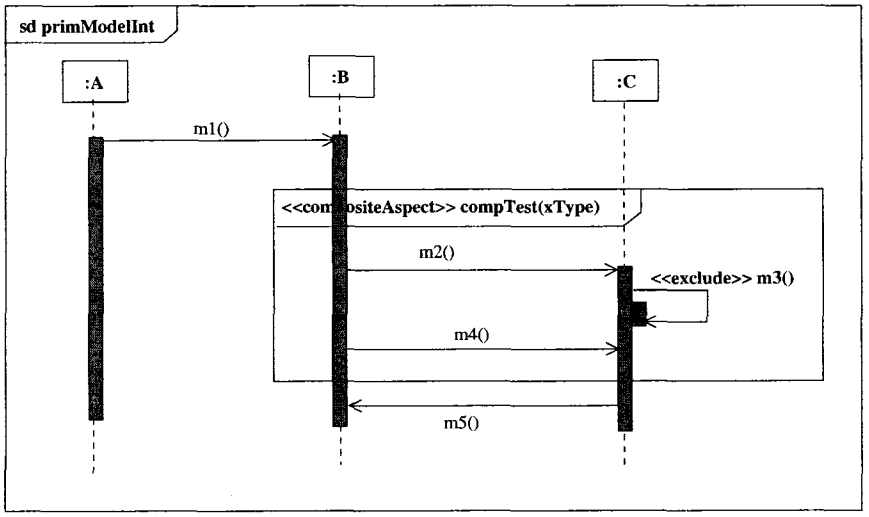


Figure 6.6: Primary model with compositeAspect tag

The Figures 6.6, 6.7 and 6.8 illustrate simple tags that use implicit binding semantics. The primary sequence model shown in figure 6.6 has three lifelines: A, B and C. The messages `m2()`, `m3()` are tagged with compositeAspect named `compTest` to specify that the messages need to be composed with `compTest` aspect. The `compositeAspect` has an input parameter `xType`. The message `m4()` is tagged with an <<`exclude`>> tag, so the message needs to remain unchanged when the messages encompassed in the `compositeAspect` tagged primary model are composed with the application-specific values of the aspect sequence model template.

The aspect model with the defined fragments is shown in Figure 7.4. The aspect has one input parameter `(t:Type)`, and two lifeline templates (`|X` and `|Y`) and one message template `|op3()`. Other messages in the different fragments are not templates and hence will appear as is in the composed sequence model.

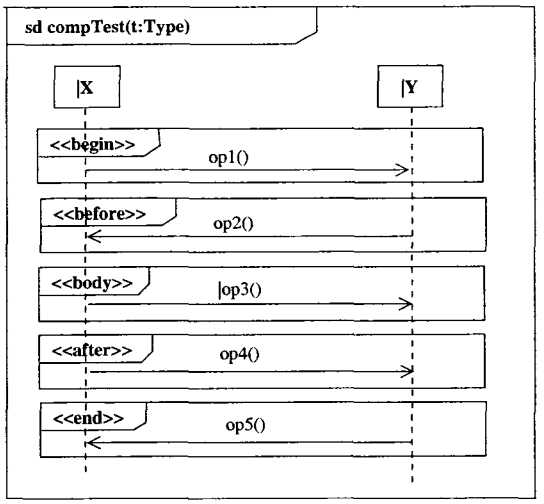


Figure 6.7: An example compositeAspect sequence model

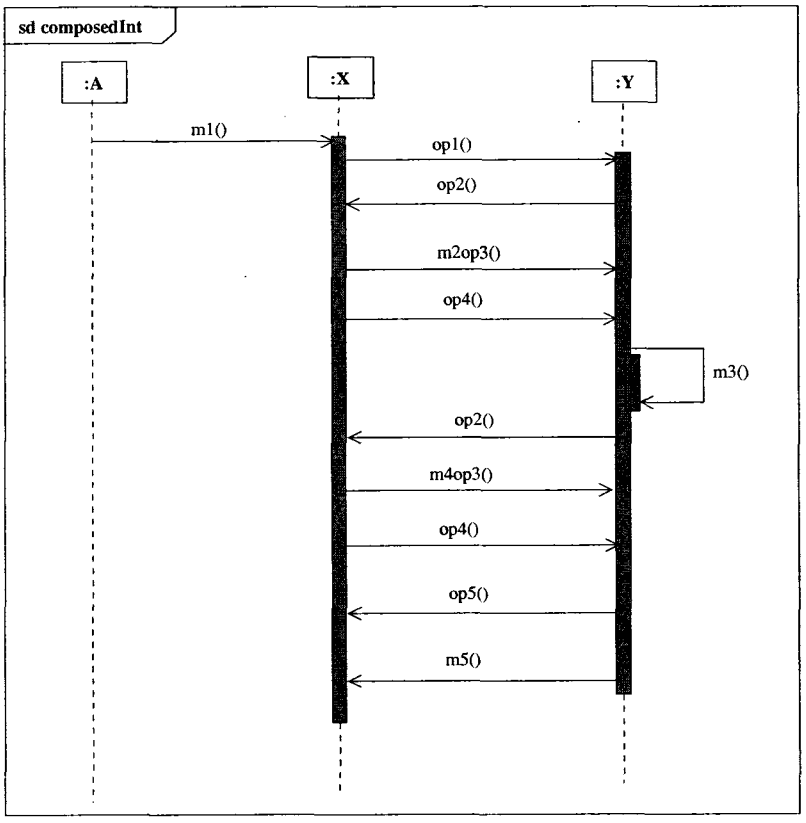


Figure 6.8: Composed model obtained from compositeAspect tagged primary model

The composed model is obtained using the implicit binding semantics of the `compositeAspect` tag. The lifeline template `|X` is bound to `B` and the `|Y` is bound to `C`, message template `|op3()` is bound to `m2()` and `m4()`. The message `m3()` is not bound due to the presence of the <<exclude>> tag. The messages are composed with the aspect sequence model template as per the rules specified for the fragments.

As per the rules, the message `op1()` is composed before any message in the compositeAspect tagged primary model. The messages `op2()` and `op4()` are composed before and after every primary model message that is within in the compositeAspect tagged primary sequence model. The message `op5()` is composed after all the messages in the compositeAspect tagged primary model. The message template `op3` specified in the <<body>> fragment refines the messages `m2()` and `m4()` into `m2op3()` and `m4op3()` respectively. The result of composing the aspect sequence model template with the primary sequence model is shown in Figure 6.8.

In case of a simple tagged primary model, the corresponding aspect model does have any defined fragments, since a `simpleAspect` is just an insertion of the entire aspect sequence in the primary model.

6.2 Composition metamodel for merging sequence models

The interaction composition metamodel describes static properties needed to support the composition of interaction models. Figure 6.9 shows the core part of the interaction metamodel, which is an extension of the UML metamodel. A `Primary Model Tag` and is a metamodel element that is specialized as `simpleAspect`, `compositeAspect` and `exclude` tags. The `simpleAspect` tags are associated with `Lifeline` while a `compositeAspect` tag is associated with `InteractionFragment`. The `Aspect Model Fragments` has five specializations: `begin`, `body`, `before`, `after`, and `end`. All aspect model fragments are associated with `InteractionFragment`.

The `replace` tag is associated with individual messages. The fragments are treated as stereotypes to enable the implementation of the fragments using existing UML tools.

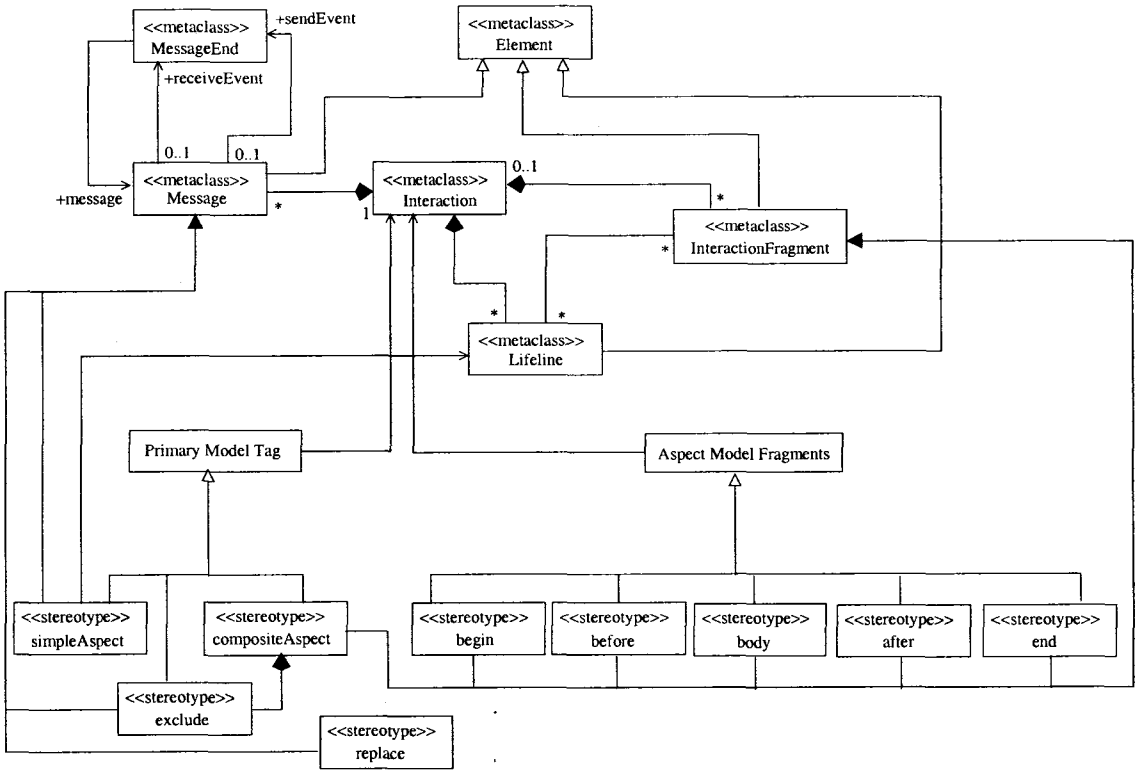


Figure 6.9: Interaction model composition metamodel.

Primary model tag instances can be used by the developer of a primary model to specify the lifelines and messages to which a specific aspect model should be applied. Similar to the class model composition implementation, the metamodel can be used as a basis for building a tool which will help in the automation of the sequence model composition process. To implement the sequence model composition, behavioral properties need to be added to the metamodel. Our approach is built upon the notion that the `simpleAspect`, `compositeAspect`, `exclude` and `replace` tags specify how the composition should occur and some atomic operations can implement these tags during composition. The atomic composition operations that implement the tags

in the sequence models are given below:

- **Merge**: A message in the primary model is merged with another message in the aspect model
- **Insert**: A single message or a set of messages are inserted at a particular point in the primary model
- **Replace**: A message in the primary model is replaced with a message or a sequence in the aspect model
- **Exclude**: A message in the primary model encompassed in a composite tag that needs to be excluded from the composition.

The atomic composition operations implement the tags specified in the primary model. The <<`replace`>> tag and <<`exclude`>> tags are associated with single messages and are implemented by the *replace* and *exclude* atomic operations respectively.

The <<`simpleAspect`>> tag is implemented by the *insert* operation, since the aspect message sequence is just inserted in the primary model sequence.

The `compositeAspect` tag can be implmented using a combination of the atomic operations, since the composition is based on the pre-defined fragments in the aspect model. The <<`begin`>>, <<`end`>>, <<`before`>, or <<`after`>> are implemented by the `insert` operation. For template messages that are to be bound, the *merge* operation composes the messages by default unless there is an explicit, <<`replace`>> or <<`exclude`>> specified on the message. If there is a <<`replace`>> or<<`exclude`>> tag specified then the corresponding atomic operations implement these tags.

The atomic operations described above are meant to be be used as guidelines for adding the behavior in the metamodels. We have not implemented the composition metamodel for sequence models in our work.

Chapter 7

Pilot studies

In this chapter we show the application of the class model and interaction model composition techniques using a money transer service application and a partial banking application. The money transfer service design models are composed with design models that address transaction management. The partial banking application design models are composed with design models that address security and replication.

7.1 Transaction Aspect and Net Banking Application - Pilot Study

The money transfer service application requires a transaction aspect to manage the money transaction. A money transfer service is used for payment of bills, transfering money between accounts, etc. A money transfer service application class model is composed with a transaction aspect class model to show class model composition. A money transfer sequence model is composed with a transaction aspect sequence model to show the interaction model composition.

7.1.1 A Money Transfer Service Primary Model

Money transfer services such as NetBank or PayPal, make requests for transferring money from one account to another. The account may exist in a single or may be from different banks.

7.1.1.1 Money transfer service class model

Figure 7.1 shows a part of the design class model for a money transfer service. The accounts are managed by `AccountManager` objects. An `AccountManager` object can manage multiple `Account` objects. The `MoneyTransferService` objects can make requests to the `Account` objects to withdraw or deposit a certain amount of money through the `AccountManager` objects.

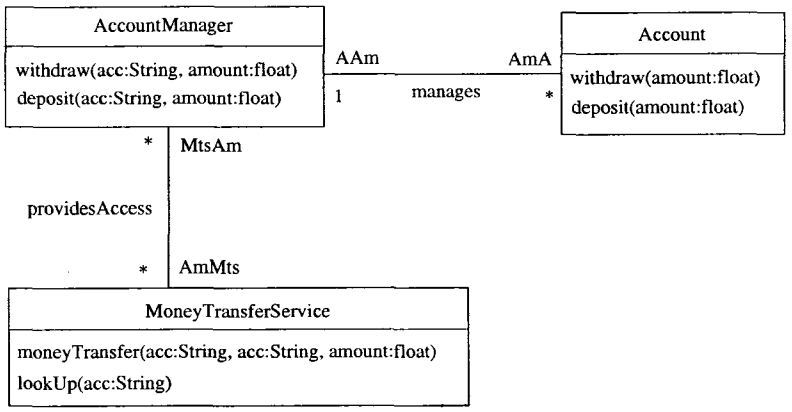


Figure 7.1: A money transfer service primary class model

7.1.1.2 Money transfer service sequence model

Figure 7.2 shows a sequence model that describes how money is transferred from one account to another. Invocation of the `moneyTransfer` operation in `MoneyTransferService` by some client (not shown in the figure) results in invocations of the `lookUp` operation to identify the account managers that manage the accounts involved in the money transfer. The sequence model shows that `acc1` is managed by `am1` and `acc2` is managed by `am2`, where `acc1` and `acc2` are account numbers and `am1` and `am2` are account manager instances. The amount that needs to be withdrawn from `acc1` and deposited into `acc2` is passed in using the string parameter `amount`. This is done using two operations, `withdraw` and `deposit`. The `withdraw` operation withdraws money from an account (`acc1`) and the `deposit` operation deposits money into an

account (acc2).

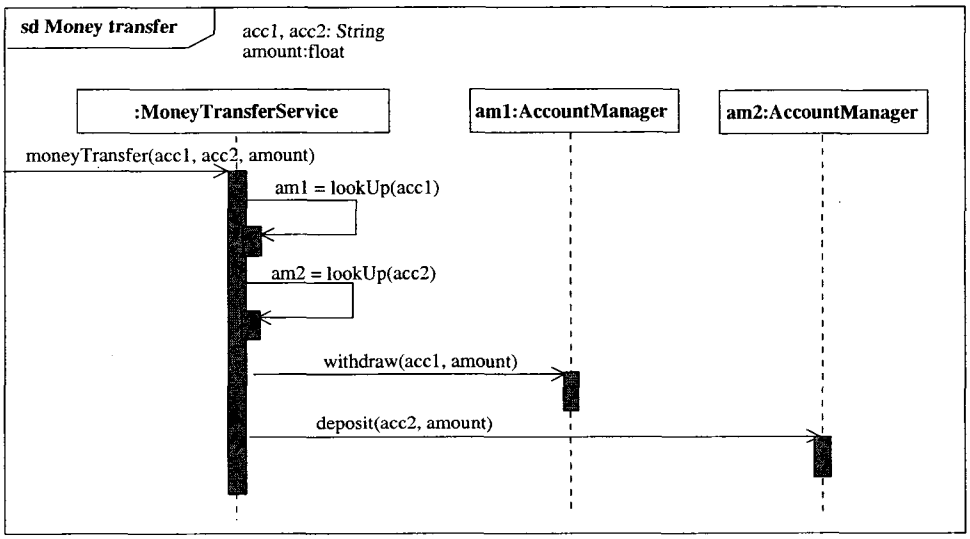


Figure 7.2: Primary sequence model describing money transfer scenario

The `moneyTransfer` operation needs to invoke both `withdraw` and `deposit` operations, otherwise the operation cannot be performed in the desired manner. For example, if the money is withdrawn from one account, it needs to be deposited in the other account, otherwise the `moneyTransfer` operation will be incomplete. Hence the `withdraw` and `deposit` operations should be made transactional in the context of `moneyTranser` operation. This can be accomplished by incorporating a transaction management feature.

7.1.2 A Transaction Aspect Model

A transaction is an indivisible collection of operations between servers and clients that remains atomic even if some clients and servers fail. An atomic operation is an operation that is free of interference from concurrent operations performed by other threads in a system. An atomic operation is guaranteed to have only one of the two known outcomes (i.e. complete success or complete failure). For example, when transferring an amount from one bank account to another if the machine loses

power, it is best that neither account's balance be changed. In other words the known outcome should be a complete failure.

The transaction aspect model consists of a class diagram template (see Figure 7.3) and an sequence diagram template (see Figure 7.4). The transaction aspect shown in Figures 7.3 and 7.4 describe a distributed transaction protocol that can use a one-phase commit or a two-phase commit sequence to commit the transaction depending on the success or failure of operations.

7.1.2.1 A Transaction aspect class diagram template

The class diagram template shown in Figure 7.3 has two class templates, `TransClient` and `Participant`, and a class `TransactionManager`.

- `TransClient` represents the set of classes that initiate the transaction and perform operations for the specific transaction.
- `Participant` represents the set of classes that provide some service required by the transaction clients.
- A `TransactionManager` is responsible for coordinating and managing transactions.

`TransClient` has an operation template named `Operation` representing operations that initiate transactional behavior. The `decision` operation represents functionality that records the outcome of a transaction. As stated earlier, the outcome of the transaction can only be a complete failure or a complete success and is recorded as a boolean value. A transaction may either be committed or aborted depending on the decision. The `processCommit` and `processAbort` operations are carried out by the `TransClient` when a transaction is committed or aborted respectively.

The `Participant` class template has two operation templates named `transOperation` and `do_transOperation`. The `transOperation` represents the `Partici-`

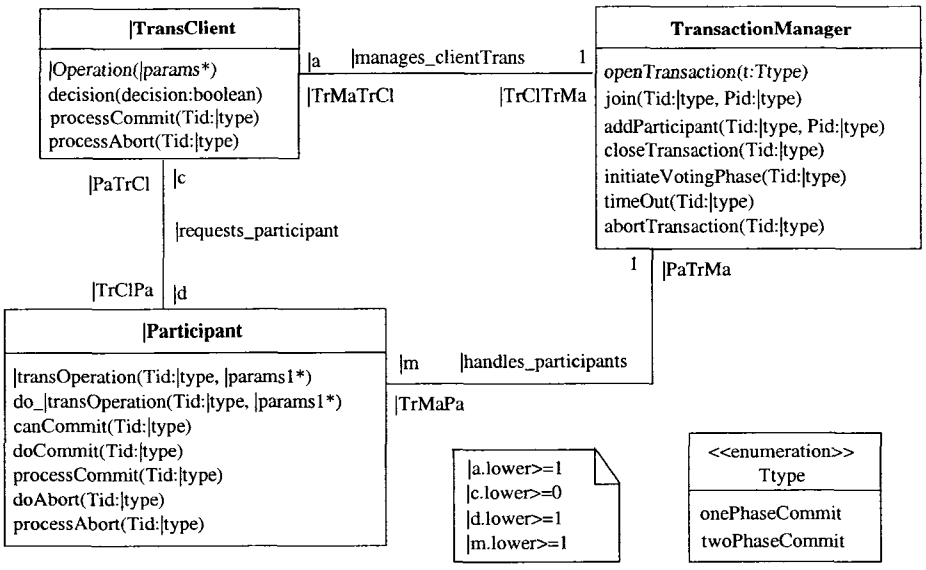


Figure 7.3: Transaction aspect class diagram template

pant operations that are transactional. The instantiation of the operation template (`Operation`) defined in the `TransClient` make the instantiations of `transOperation` transactional. For every instance of `transOperation` there should be a corresponding instantiation of `do_transOperation`. For example, if the `transOperation` template in `Participant` is instantiated with a name *op1*, then `do_transOperation` must be instantiated as *do_op1*. The `canCommit`, `doCommit`, `processCommit`, `doAbort` and `processAbort` are operations used by the `TransactionManager` object to communicate with a `Participant` object.

The `openTransaction`, `join`, `addParticipant`, `closeTransaction`, and `abortTransaction` operations in the `TransactionManager` class are operations used for interaction between `Participant` objects, `TransClient` objects and `TransactionManager` object. The `openTransaction`, `closeTransaction`, and `abortTransaction` are used to open, close and abort a transaction respectively. A `Participant` object can request to join a particular transaction using the `join` operation. A `Participant` object can be added to a particular transaction by the `TransactionManager` object

using the addParticipant operation. The timeOut operation is used to time out a particular transaction after a certain period of time. The initiateVotingPhase initiates the voting process that will lead to either completion or aborting of a particular transaction. Note that none of the operations specified in the TransactionManager class are template operations and hence need not be bound to application-specific values.

The multiplicity constraints state that a TransClient object should be linked to atleast one Participant object. Each Paricipant object is linked to exactly one TransactionManager object. The transaction protocol type (Ttype) is shown as enumeration consisting of onePhaseCommit andtwoPhaseCommit protocols.

7.1.2.2 A Transaction aspect sequence diagram template

The sequence diagram template describes a family of transaction scenarios (see Figure 7.4). A typical scenario in the family is as follows: A TransClient object (i.e., an object of a class obtained from binding TransClient to application specific values) initiates the transaction by sending the openTransaction operation call message to a TransactionManager object. When the TransactionManager object receives openTransaction message, it opens a transaction and returns a transaction id (Tid). This Tid is sent as a parameter in all subsequent operations to identify the transaction.

The TransClient then performs the collection of operations that exhibit transactional behavior for that transaction using the Tid. When a Participant object receives a message request, it checks whether it is already a member of the particular transaction. If not, the Participant object joins the transaction before it performs the requested operation. The Figure 7.4 shows a collection of Participant objects called theParticipants representing the set of Participants involved in the transaction. When the Participant object joins the transaction, the Participant object is added to the transaction by the TransactionManager object using the

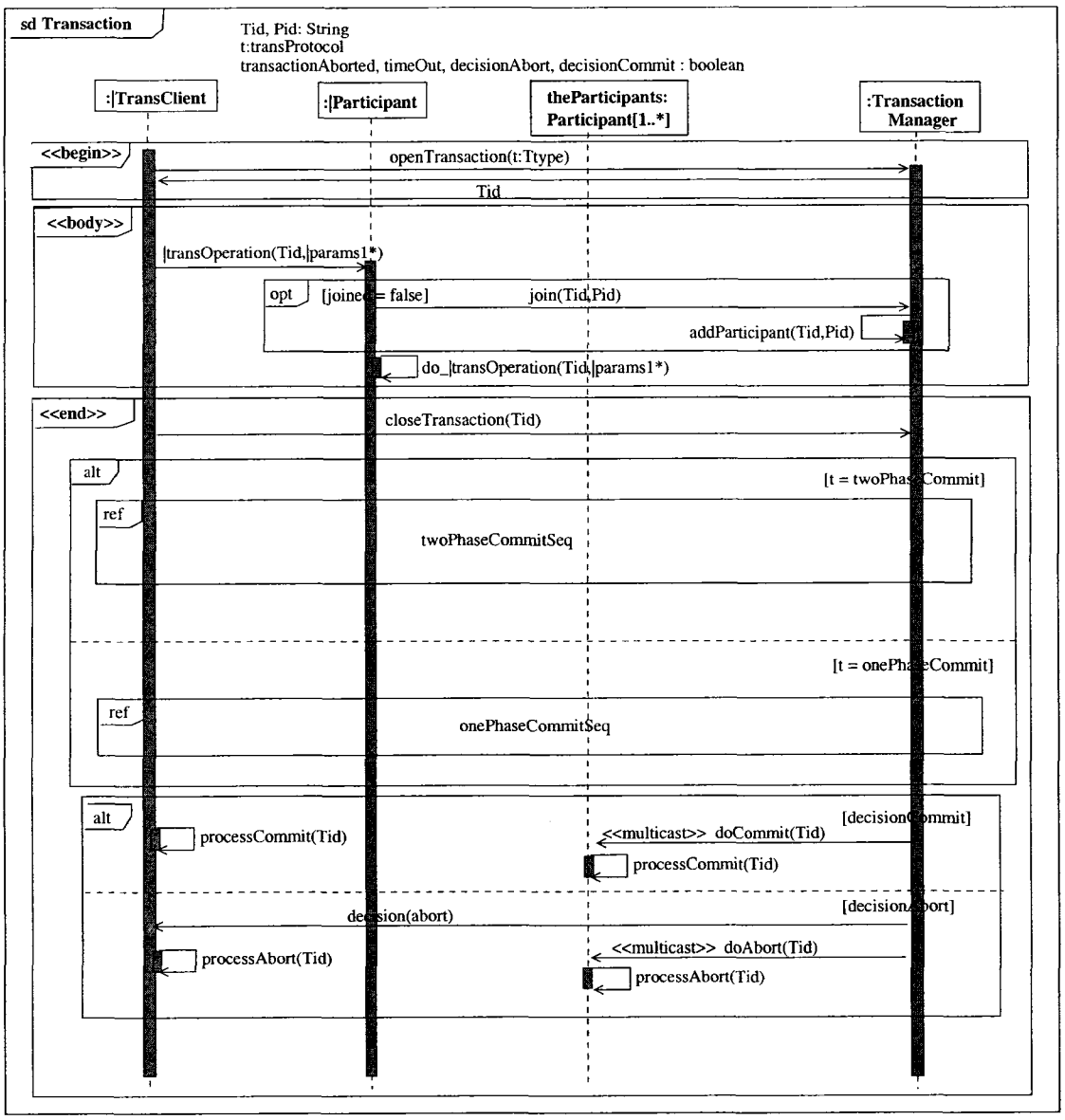


Figure 7.4: A transaction aspect sequence model

addParticipant message. The Participant object then delegates the transOperation responsibility by calling the corresponding instantiation of do_transOperation message template. The operations are then executed based on the type of protocol selected. In the Figure, the transaction one-phase and two-phase commit protocols are shown as alternate interaction occurences. Depending on the type of protocol

chosen the particular interaction occurence will occur.

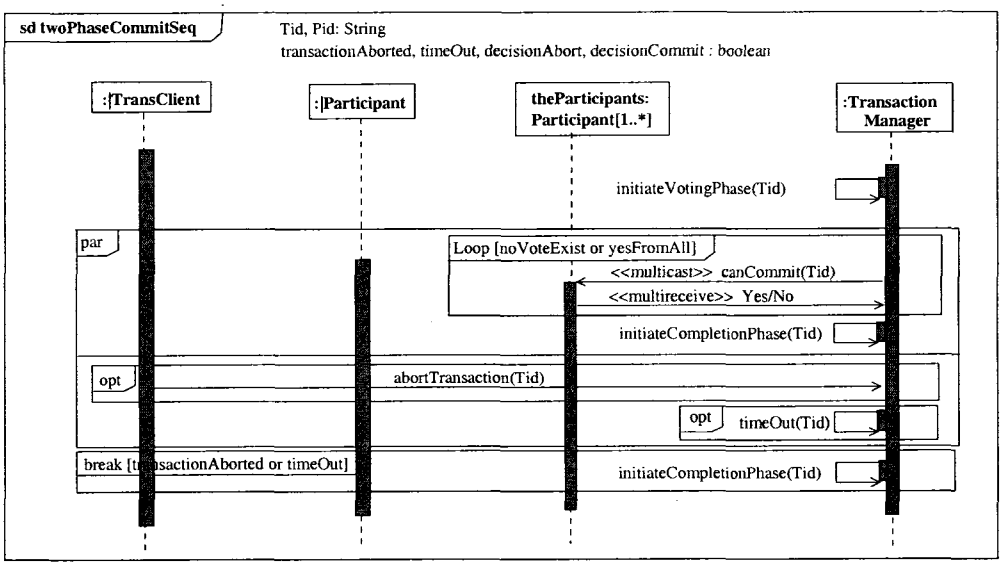


Figure 7.5: Sequence diagram for two phase commit protocol

Two-Phase Commit Protocol: When the `TransClient` object requests to close the transaction, the `TransactionManager` object starts the commit protocol depending on the chosen transaction protocol type. The sequence model in Figure 7.5 shows the details of the two-phase commit protocol. The one-phase commit protocol is shown as an interaction occurance in Figure 7.4 and the detials are not shown in this thesis. In the first phase (*voting phase*) of the two-phase commit protocol, the `TransactionManager` object polls all the `Participant` objects to check if they are ready to commit. In the second phase (*completion phase*), the `TransactionManager` object decides to abort or commit the transaction based on the result obtained from the voting phase. The decision is multicast to all `participant` object. At any time during the transaction, the `TransClient` objects can request to abort the transaction or the `TransactionManager` object may timeout. Both requests result in the initiation of the completion phase, even when the voting phase is incomplete. The `TransactionManager` object will then eventually decide to abort and all `Participant`

objects will be informed. The `Participant` objects will then roll back the transaction individually.

The transaction aspect model has multiple lifeline templates and message templates that are structured using defined fragments of a compositeAspect tag. The transaction model has a <<`begin`>> fragment, a <<`body`>> fragment and an <<`end`>> fragment. The transaction aspect does not have a <<`before`>> and <<`after`>> fragment.

7.1.3 Applying class model composition

The transaction aspect class diagram template needs to be instantiated before it can be composed with the primary model using the prototype implementation. The context-specific aspect model is obtained by instantiating the transaction aspect class diagram template shown in Figure 7.3 in the context of the money transfer service primary model shown in Figure 7.1. A system developer uses domain knowledge to obtain the bindings that are used during the instantiation. For example, the transaction aspect models has a template operation called `transOperation` which represent the operations that are transactional. In the primary model `withdraw` and `deposit` are transactional operations if they are invovled in the money transfer. So the `transOperation` is bound to `withdraw` and `deposit`. An instantiation of the `transOperation` produces a corresponding instantiation of `do_transOperation` to `do_withdraw` and `do_deposit`. The list of bindings are shown in table 7.1. The values used in the bindings shown in the table are names obtained from the primary model element or the name of an application-specific element that is to be added to the composed model during composition.

The |`type` is mapped to `String` for all the `Tid` and `Pid`. The |`type` is used in several operations but its binding is shown only once in the table. The application of bindings on the transaction aspect class diagram template results in a money transfer

Table 7.1: Bindings for money transfer specific transaction aspect

Aspect model parameter	Application specific element
|TransClient	MoneyTransferService
|TransClient::|Operation	MoneyTransferService::moneyTranfer
|Operation::|params*	moneyTranfer::acc, acc, amount
|Participant	AccountManager
|Participant::|transOperation	AccountManager::withdraw
|Participant::|transOperation	AccountManager::deposit
|params1*	acc:String, amount:float
|type	String
|a	1..*
|c	*
|d	1..*
|m	1..*
|requests_participant	providesAccess
|manages_clientTrans	manages_moneyTrans
|handles_participants	handles_accountManagers
|PaTrCl	AmMts
|TrClPa	MtsAm
|TrMaTrCl	TrMaMts
|TrClTrMa	MtsTrMa
|PaTrMa	AmTrMa
|TrMaPa	TrMaAm

context-specific transaction aspect class model shown in Figure 7.6.

Once the context-specific aspect class model is obtained, it is composed with the primary class model. The `withdraw` and `deposit` operation of the primary class model are persistant and used to withdraw and deposit money respectively. However, in the context-specific aspect class model the withdraw and deposit is not accomplished by the `withdraw` and `deposit` operations. The `withdraw` and `deposit` operations make a call to the `do_withdraw` and `do_deposit` operations. The `do_withdraw` and `do_deposit` operations are persistant.

The tool uses a default signature type to compose the `withdraw` and `deposit` operations in the primary class model with the `withdraw` and `deposit` operations of

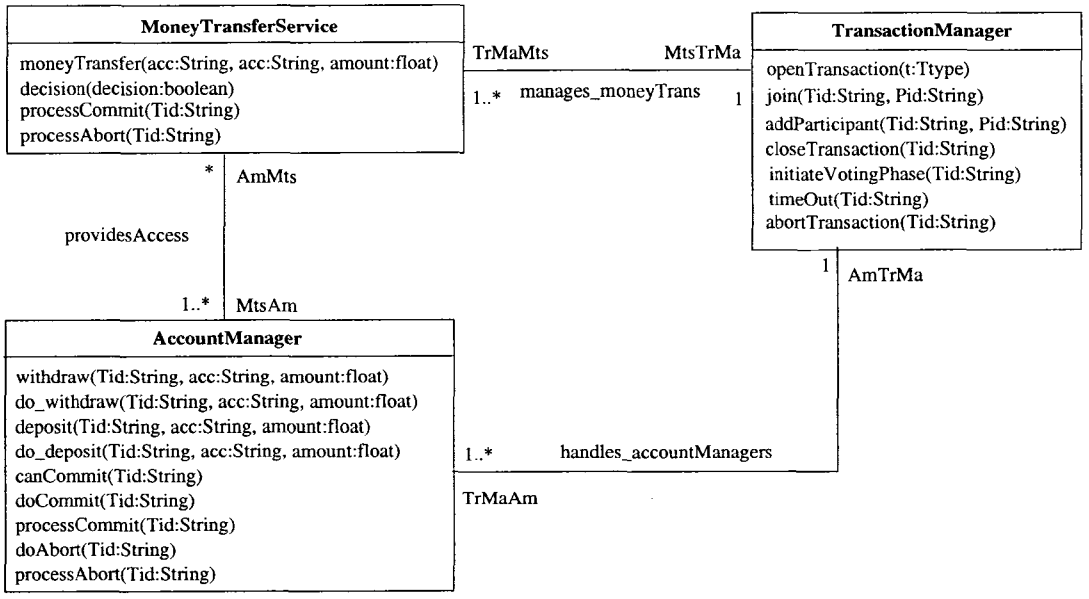


Figure 7.6: Context-specific transaction aspect class model

the context-specific aspect class model. If the models are merged using the tool the `withdraw` and `deposit` operations in the primary class model will be merged with `withdraw` and `deposit` operations of the context-specific aspect class model. This results in a faulty model since the operations do not have the same intent. This can be avoided by applying the following composition directives:

1. **remove** `primary::AccountManager::withdraw`

2. **replaceOccurrences** `primary::AccountManager::withdraw`
   with `aspect::AccountManager::do_withdraw()`

3. **remove** `primary::AccountManager::deposit`

4. **replaceOccurrences** `primary::AccountManager::deposit`
   with `aspect::AccountManager::do_deposit()`

The composition directives delete the `withdraw` and `deposit` operations in the primary model and replace them with `do_withdraw` and `do_deposit` operations of

the aspect model.

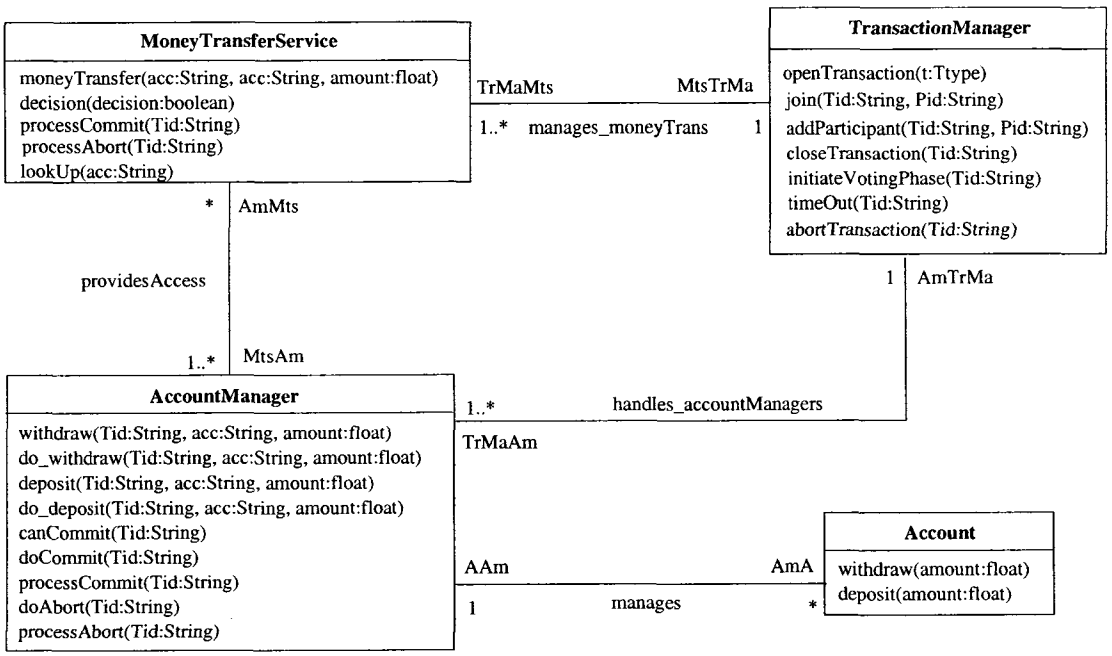


Figure 7.7: Composed model for transaction aspect with money transfer service

The context-specific transaction aspect model and the modified primary model are merged using a default signature type (*name*) for classes, operations and a complete signature types for parameters and association ends. The resultant composed model (see Figure 7.7) exhibits a functionality in which the the `withdraw` and `deposit` operations make a call to the `do_withdraw` and `do_deposit` operations. The withdrawl and deposit are performed by the `do_withdraw` and `do_deposit` and save to a persistant storage. Note that, in the primary model the multiplicity on the association end property `MtsAm` is "*" and in the context-specific aspect model the multiplicity on the association end property `MtsAm` is "1..*". The resultant composed model shows a stronger multiplicity of "1..*" as a result of the default composition. If the weaker multiplicity of "*" was needed the following merge directive can be used:

association end multiplicity `aspect::MoneyTransferService::MtsAm;`

`primary::MoneyTransferService::MtsAm` **weaker**

If an alternative signature type is used after renaming the operations, a modeler may need to apply more composition directives to obtain the desired results. As an example, consider a signature-based match with a default signature type (*name*) for classes, association ends and a signature type for operations consisting of *name* and *ownedParameters*.

The signature-based match will result in a composed model (see Figure ??) that does not exhibit the desired properties. The `withdraw(acc:String, amount:float)` in the primary class model and `withdraw(Tid:String, acc:String, amount:float)` in the aspect class model reflect the same intent but do not match with respect to the signatures and will both be included in the composed model. Similarly `deposit(acc:String, amount:float)` in the primary class model and `deposit(Tid:String, acc:String, amount:float)` in the aspect class model do not match and will both be included in the composed model. The following composition directives need to be applied to obtain the desired results :

1. **remove** `primary::AccountManager::withdraw()`
2. **replaceOccurrences** `primary::AccountManager::withdraw()`
   with `aspect::AccountManager::withdraw()`
3. **remove** `primary::AccountManager::deposit()`
4. **replaceOccurrences** `primary::AccountManager::deposit`
   with `aspect::AccountManager::deposit()`

7.1.4 Applying interaction model composition

The money transfer scenario shown in 7.2 talks about tranfer of money from one account to another. The `MoneyTransferService` object gains access to the accounts

using `AccountManager` objects by validating the input parameters. The parameters are account numbers (`acc1`, `acc2`) and amount (`amount`) of money that needs to be transferred from one account to another.

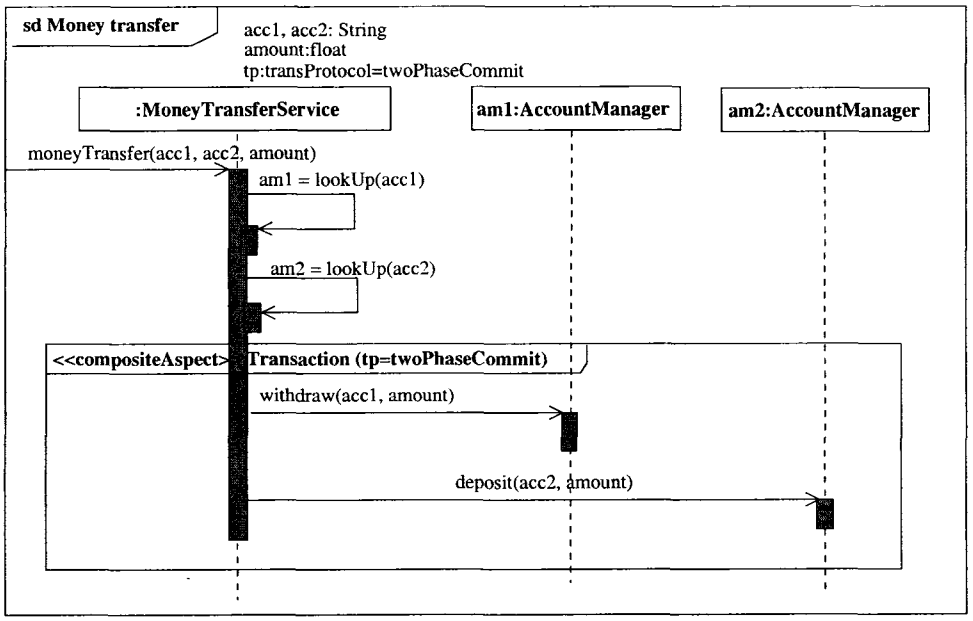


Figure 7.8: Tagged money transfer sequence model

The `withdraw` and `deposit` operations have to be composed with application-specific values of transaction aspect sequence diagram template to make them transactional. The `withdraw` and `deposit` operations span across more one lifeline and are specified using fragments of a `compositeAspect` tag (see Figure 7.8). The figure shows the name of the compositeAspect `Transaction` and the transaction protocol type `twoPhaseCommit` passed in as a parameter.

The composition of transaction aspect sequence model with the primary sequence model is done using the implicit binding semantics of the `compositeAspect` tag. The result of composing the transaction aspect sequence model shown in Figure 7.4 with the primary sequence model is shown in Figure 7.9.

The lookup sequences present in the primary sequence model are not affected

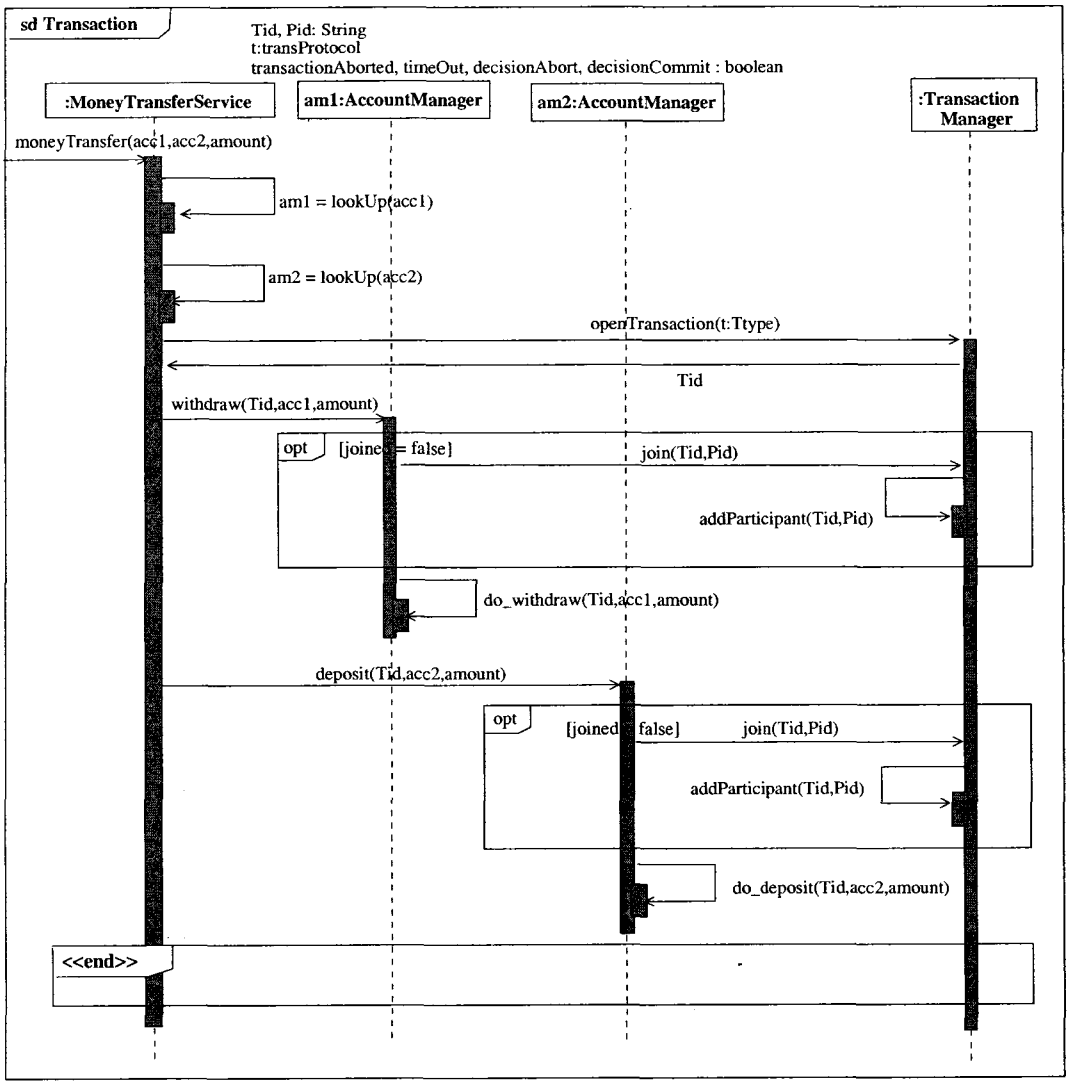


Figure 7.9: Money transfer sequence model composed with transaction aspect sequence model

by the transaction aspect since they are not composed with any aspect sequence model element. Using the implicit binding semantics, the `TransClient` is bound to a `MoneyTransferService` object and the `Participant` to `am1:AccountManager` and `am2:AccountManager`. Based on the rules defined for a `compositeAspect` tagged primary model, the message template `transOperation(Tid, params*)` sent by the `TransClient` lifeline template and received by the `Participant` template is bound to

messages that originate from a MoneyTransferService object (application specific value of TransClient lifeline template) and received by AccountManager objects (application specific value of Participant lifeline template).

For every instance of transOperation there is a corresponding instantiation of do_transOperation. In our case, the transOperation template in Participant is instantiated to withdraw and deposit. Hence do_transOperation is instantiated as do_withdraw and do_deposit.

The merge occurs as per the rules of the compositeAspect fragments.

The <<begin>> fragment is the first sequence to appear. The openTransaction message which a part of the <<begin>> fragment is sent to the TransactionManager object with the twoPhaseCommit as the parameter.

The <<body>> sequence is applied to each message within the composite fragment, hence, the <<body>> is applied to the messages: withdraw and deposit. The resultant behavior must exhibit the semantic properties of the withdraw and deposit message defined in the primary model. The body fragment refines the primary model messages by making them transactional. During the composition, the Tid parameter (representing the transaction Id) is inserted. The refinement results in a sequence in which withdraw and deposit are performed only when the participant objects that perform the services are part of that transaction.

The <<end>> sequence appears in the composed model at the end of the sequence of messages, similar to the <<begin>> sequence that appears at the start. The end sequence is a two phase commit protocol sequence, since the input parameter defined in the composite tagged primary model is a two phase commit protocol. The TransactionManager object multicasts canCommit to the AccountManager objects as per the two phase protocol. The transaction is committed only when both the AccoutManager objects commit to the transaction otherwise the transaction is aborted. The end sequence is shown as an interaction occurance for lack of space (see

figure 7.9). The full sequence model is shown in appendixA.

7.2 Discussion

The class and interaction model composition tecniques are described independently in this thesis. Class models and sequence models provide different views of a design. Hence, these views should be consistent with each other. For example, if a change is made in class model the corresponding sequence model should reflect that change. A modeler may choose to compose the structural design models before the behavioral models or vice versa. Hence the model views should be consistent with each other.

7.2.1 Consistency of composition techniques

The class model composition uses composition directives and the interaction model composition uses tags to support the composition of aspect and primary models. Tags may be considered as a form of composition directives, since they guide the composition process. For example, the <<`begin`>> fragment of compositeAspect tag orders the composition of specific elements by stipulating that the aspect model elements enclosed within the begin fragment must preceed other composed model elements that are obtained during composition. Hence, a modeler needs to understand the relationships between tags and composition directives to determine the limitations of using tags to specify composition directives. For example, determining the types of directives that are best specified as tags and directives that cannot or should not be specified as tags.

As another example, consider the `precedes` and `follows` model directives that specify the weave-order between aspect class models. In interaction model composition, the tagging of the primary model sequence models determine the order of composition. The weave-order can be easily enforced if the aspect model sequences are orthogonal to each other and if mulitple aspect sequence models are not com-

posed with the same message in the primary model. If the aspect sequence models are non-orthogonal and multiple aspect sequence models need to be composed with the same sequence of primary model messages, the weave-order can be specified as using an inner and outer interaction fragment. For example, if *connectivity* aspect needs to composed prior to the *transaction* aspect, we can specify that the connectivity aspect should be composed before the transaction aspect by encompassing the messages using the interaction fragments as shown in Figure 7.10. The inner fragment compositeAspect needs to be composed with the `addAccess` message before the outer fragment compositeAspect.

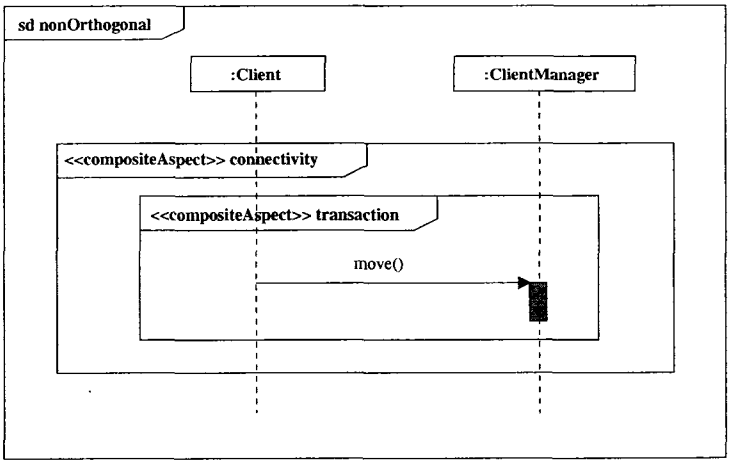


Figure 7.10: Tagging a message in the primary model with multiple aspect sequence models

Some of the composition directives may not have a corresponding tag fragment in the interaction model composition. For example, the `rename` composition directive does not have a corresponding tag fragment specified in the interaction model composition. In such cases, care should be taken such that the models are consistent. UML tools that automatically update the different views when a change is made to a particular view are useful in such cases.

In the transaction aspect and netbanking pilot study, the following composition

directives were applied:

1. **remove** `primary::AccountManager::withdraw`

2. **replaceOccurrences** `primary::AccountManager::withdraw`
   with `aspect::AccountManager::do_withdraw()`

3. **remove** `primary::AccountManager::deposit`

4. **replaceOccurrences** `primary::AccountManager::deposit`
   with `aspect::AccountManager::do_deposit()`

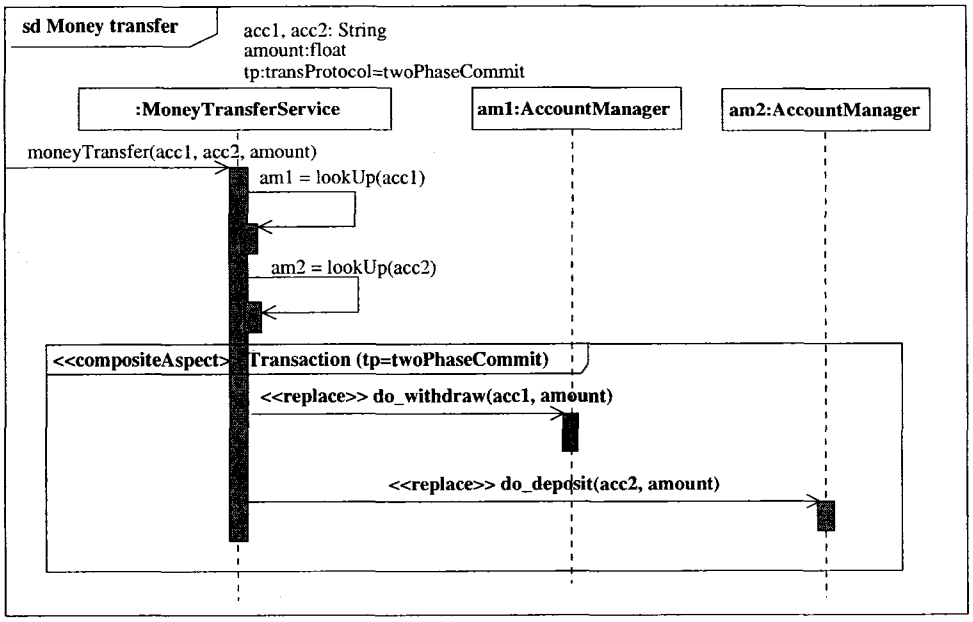


Figure 7.11: Tagged money transfer sequence model consistent with class model

The `remove` directive shown above is applied on `withdraw` and `deposit` operation defined in the `AccountManager` class in Figure 7.1. When the `remove` directive is applied during the class model composition, the `withdraw` and `deposit` operation call message from `MoneyTransferService` object to `AccountManager` objects in Figure 7.2 should reflect the change (see Figure 7.11).

The references to the primary model operations `withdraw` and `deposit` are replaced with an aspect model element using the `replaceOccurences` directive in the class model composition. The corresponding operation call messages `withdraw` and `deposit` in the primary sequence model are tagged with the <<replace>> tag to reflect the intent.

7.2.2 Tags in the primary model

The use of tags to support composition of some scenarios into the primary sequence models raises the issue of the using different ways to tag the primary model. The functionality provided by the primary model determines the type of tag applied on the primary model.

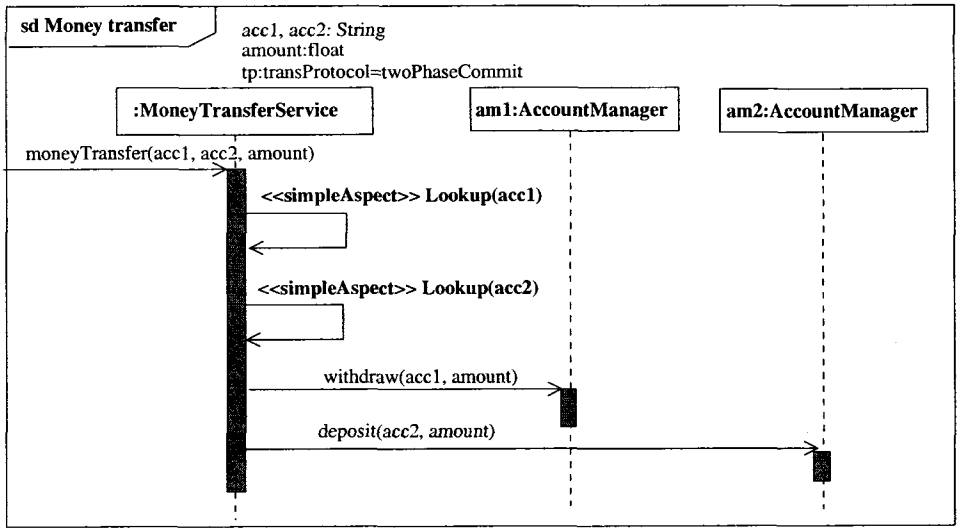


Figure 7.12: Money transfer service primary Model with simpleAspect Lookup Tag

Figure 7.2 shows a sequence primary model for the money transfer service. The `MoneyTransferService` object provides a *lookUp* message for getting a reference to each `AccountManager` object. The functionality provided by the *lookUp* message is required for both centralized and distributed systems. However for distributed systems, middleware normally provides this feature. Therefore, a developer may

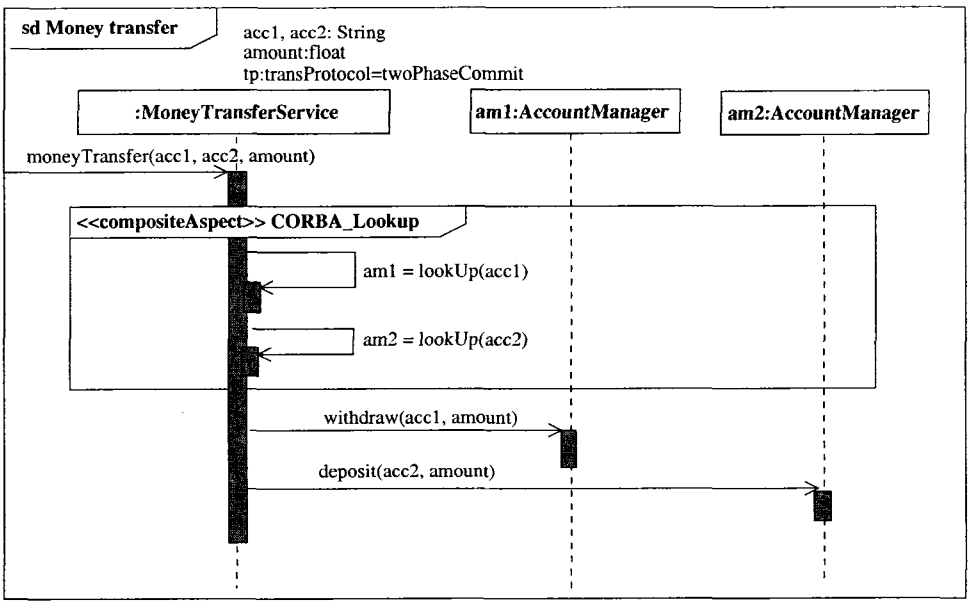


Figure 7.13: Money transfer service primary model with compositeAspect Lookup Tag

choose to omit the two *lookUp* messages in Figure 7.2 and specify a primary model without the *lookUp* methods.

The *lookUp* in the primary model enables the `MoneyTransferService` object to interact with the `AccountManager` objects that are managing the accounts involved in the money transfer. The developer may choose to tag the primary model using `simpleAspect` tags as shown in Figure 7.12. If the primary model has elements that are centralized and needs to be distributed, the primary model element must be overridden by the middleware aspect model using a `compositeAspect` tag (see Figure 7.13. In the Figure, the *lookUp* message is overridden with the `CORBA_Lookup` aspect.

7.2.3 Effect of signatures on composition directives

The composed model shown in Figure 7.7 is obtained using a default signature type (*name*) for classes, operations and a complete signature types for parameters and association ends. The signature type can be varied to obtain a different composed

model. However, improper usage of signature type can result in models that are not well-formed. In cases where the composition is expected to yield undesirable results, composition directives can be used to obtain the desired results.

Complete signature types are typically used for matching contained model elements such as class attributes and operation parameters. Default signature types are typically used for matching model elements such as packages. The usage of a particular signature type can affect the composition directives that may need to be applied. For example, consider two models with operations that have the same name but different return types. If the two operations are composed using a default signature type (only *name*) the operations match with respect to the signature but a conflict arises due to the different return types. The conflict may be resolved using a composition directive that overrides an operation in one model with the operation in the other model. If the two operations are composed using a signature type consisting of operation *name* and return *type*, then the two operations do not match and are both included in the composed model. This negates the usage of a composition directive if the intent is to obtain both the operations in the composed model.

Usage of particular signature types may sometimes lead to models that are not well-formed. As an example, consider matching the context-specific aspect class model 7.1 and primary class model 7.6 with signature type consisting of class `name` and `ownedOperations`. If the models are composed using such a signature type, it will result in a model that is not well-formed (see Figure 7.14). The composed model needs to be altered using multiple add, delete and changeOccurences composition directives to make it well-formed. This can be avoided by just changing the signature type to default signature type for classes and complete signature type for operations. A selection of a particular signature type may result in the application of more composition directives or vice versa.

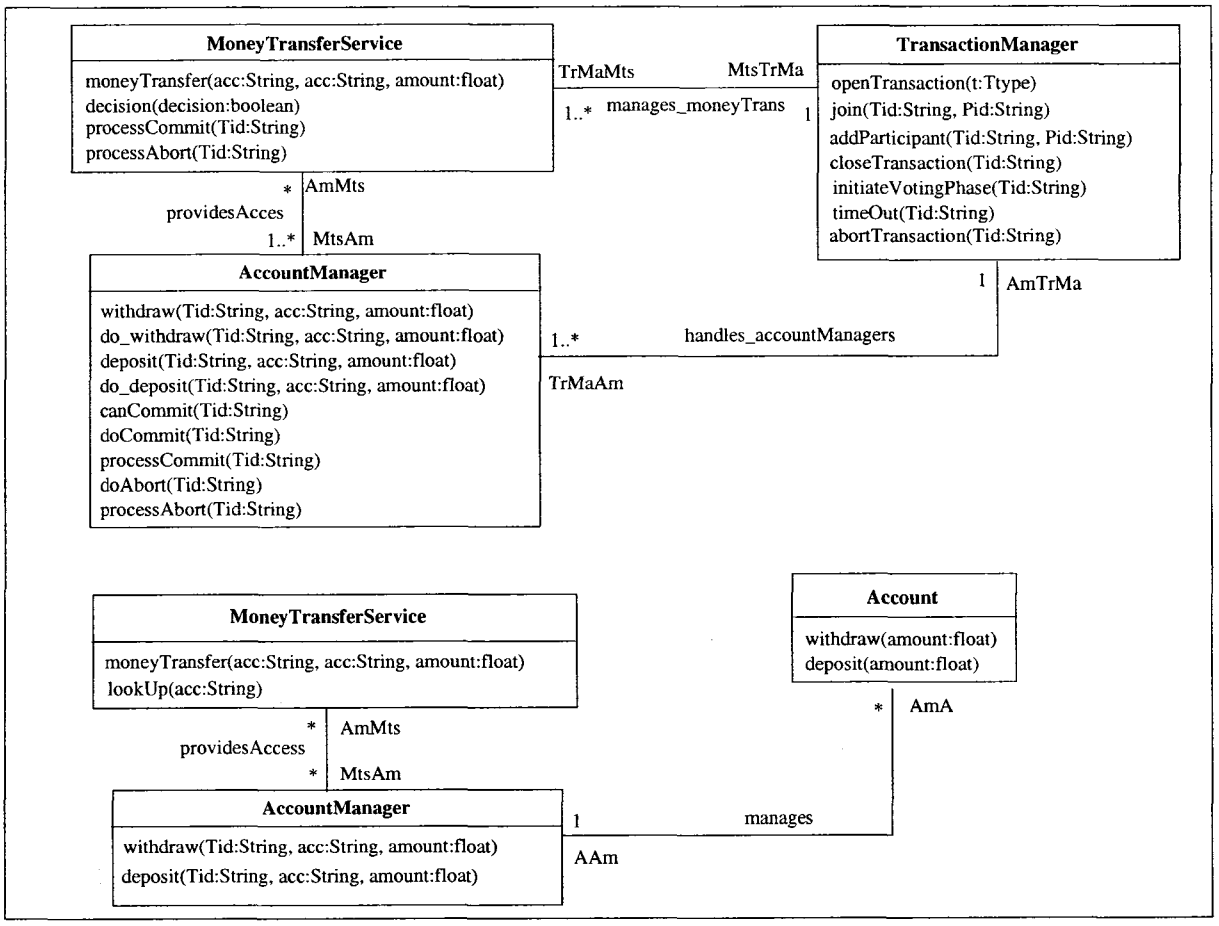


Figure 7.14: Composed model for transaction aspect with money transfer service not well-formed

7.3 Authorization Aspect, Replication Aspect and Banking Application - Pilot Study

In this pilot study we model two solutions that address dependability features: an authorization-based access control aspect and a replicated repository fault tolerance aspect [57, 59]. The aspect models are composed with a primary model describing a small banking system. We also show the application of composition directives on class models to obtain the desired results when the they do not exhibit the desired properties.

7.3.1 Banking Application Primary Model

7.3.1.1 Banking aplication primary class model

Figure 7.15 shows a class model describing a part of the banking application. `Client` objects, for example customers and tellers, make requests to the `ClientManager` objects to perform operations on accounts. Operations include adding or deleting accounts, and debiting or crediting accounts. Accounts are held in a repository that is an instance of `AccRepository`.

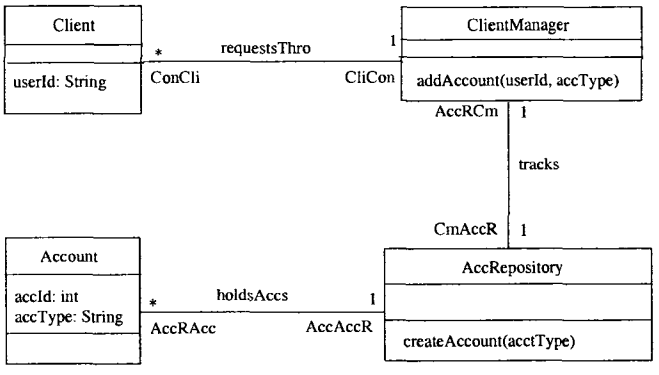


Figure 7.15: Banking application primary model

7.3.1.2 Banking application primary sequence model

Figure 7.16 shows a sequence model that describes how an account is added to the system. Invocation of the `addAccount` operation in `ClientManager` object with the type of account (`accType`) by a `Client` object results in the creation of a new account using the `createAccount` operation. Once the new account is created the account identity (`accId`) for the given account type.

7.3.2 An Authorization aspect model

Authorization is a security concern and is crosscutting in nature. A solution that addresses authorization, can be used to prevent unauthorized handling of resources. In our authorization aspect model, we describe a family of solutions in which accesses

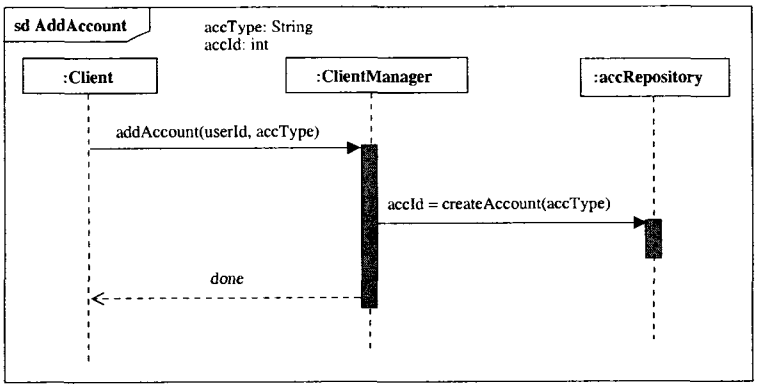


Figure 7.16: The AddAccount sequence model

to protected operations must first be authorized. The authorization aspect model consists of a class diagram and sequence diagram template.

7.3.2.1 An Authorization aspect class diagram template

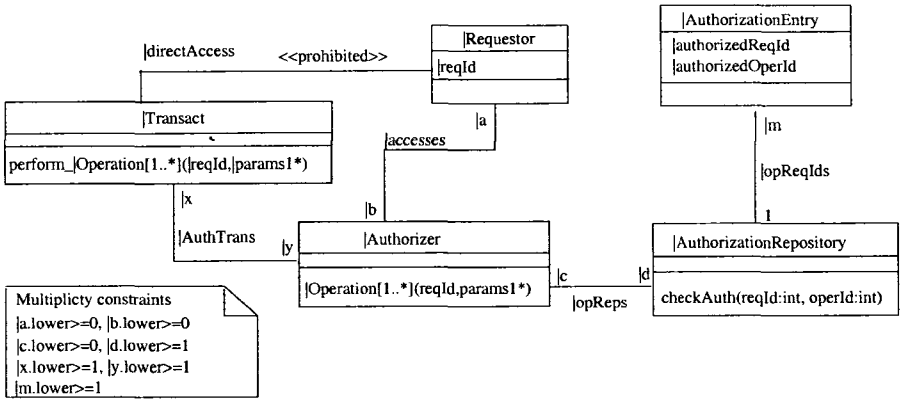


Figure 7.17: A class diagram template for the Authorization aspect model

The class diagram template shown in Figure 7.17 has five class templates.

- `Requestor` is a template for client classes. It has an attribute template named `reqId` representing a client's identifier.
- `Authorizer` is a template for classes that are responsible for authorizing clients. It has an operation template named `Operation` representing operations that

authorize clients. An instantiation of `Authorizer` can have one or more instantiations of operation associated with it. This is indicated by the instantiation multiplicity `[1..*]` associated with the operation template.

- `Transact` is a template for classes that contain protected operations (i.e. operations that can only be invoked by authorized clients). It has an operation template named `perform_|Operation` representing protected operations. If the `Operation` template in `Authorizer` is instantiated with a name *op1*, then `perform_|Operation` must be instantiated as *perform_op1*.

- `AuthRepository` is a template for classes representing repositories of client-operation authorizations.

- `AuthorizationEntry` is a template for classes representing authorization data.

The <<prohibited>> stereotype on the association template `directAccess` indicates that associations between application-specific values of `Transact` and `Requestor` class templates are prohibited. If such an association exists in a primary model then the association is deleted when an instantiation of the aspect class diagram template is composed with the primary class model.

7.3.2.2 Authorization aspect sequence diagram template

Figure 7.18 shows an sequence diagram template that describes a family of authorization scenarios. The following is a typical scenario in the family:

- A **Requestor** object (i.e., an object of a class instantiated from **Requestor**) sends an operation call message to an **Authorizer** object.

- The **Authorizer** object checks whether the client is authorized to invoke the requested operation by searching for an authorization in a **AuthRepository** object. This check is done using the `getAuth`.

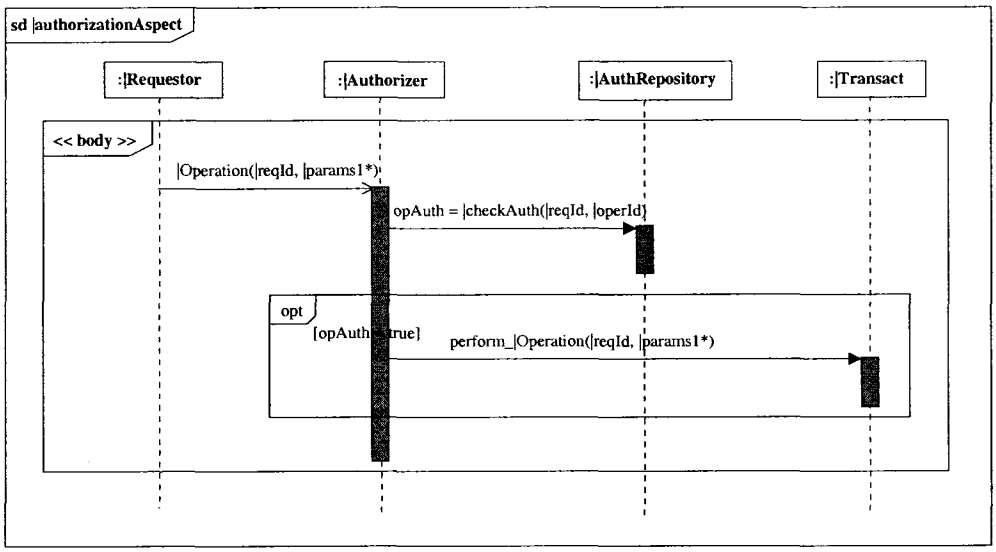


Figure 7.18: A sequence diagram template for the authorization aspect model

- If the authorization succeeds, then the **Authorizer** object delegates the responsibility of performing the requested operation to a **Transact** object by calling the corresponding instantiation of |**Operation**. This is represented using the *opt* interaction occurance in the sequence diagram template.

The authorization aspect model has multiple lifeline templates that need to bound with application-specific values and is modeled using the compositeAspect tag fragments. The sequence diagram template (see Figure 7.18) consists of only the <<body>> fragment.

7.3.3 A Replicated Repository aspect model

Resources can be replicated to improve fault tolerance [61]. We describe a simplified data redundancy solution based on the use of replicated data repositories. Data replication can be accomplished using either primary/backup techniques or active replication techniques. With primary/backup replication, if a failure occurs in the primary repository, another repository must take over. Active replication requires that some sort of group communication be available for messaging between the client

and replicas. In addition, messages must be totally ordered so that each replica performs the same updates in the same order. The simple replication aspect described in this study uses active replication, and therefore imposes an additional constraint on the system design that a totally ordered broadcast group communication mechanism be available. In the pilot study, the messages are broadcast to replicas that are in a *ready* state. A replica in a *not ready* state is not available for access.

7.3.3.1 A Replicated repository class diagram template

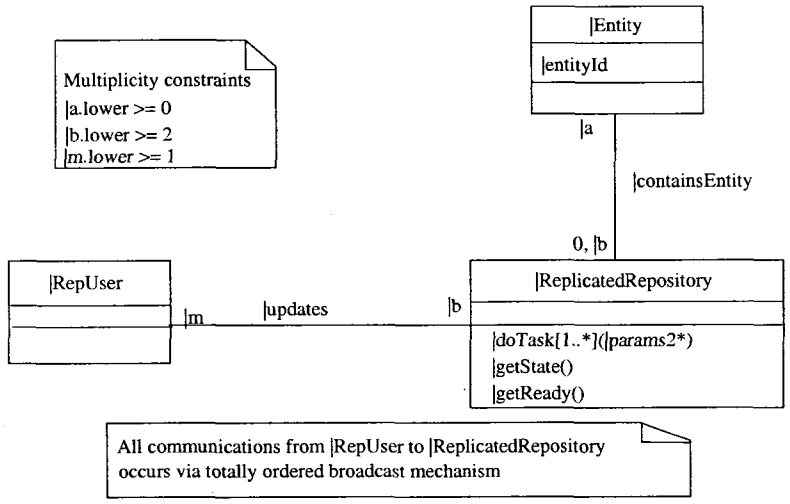


Figure 7.19: A class diagram template for the Replicated Repository aspect model

Figure 7.19 shows a class diagram template of a simple replicated repository aspect model. There are three class templates in the aspect model.

- `RepUser`: Instantiated classes represent repository users.
- `ReplicatedRepostiory`: Instantiated classes represent repositories.
- `Entity`:Instantiated classes represent items stored in repositories.

The `ReplicatedRepository` class template has an operation template `doTask` representing operations that access the repository. The multiplicity parameter con-

straint `b.lower >= 2` indicates that there must be at least two replicated repositories associated with the application-specific value of the `repUser` class template.

7.3.3.2 A Replicated repository sequence diagram template

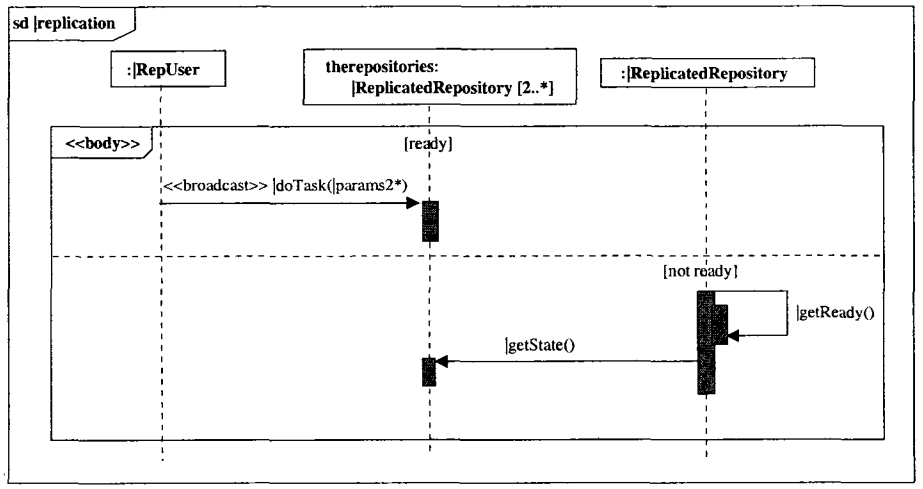


Figure 7.20: A sequence diagram template for the Replicated Repository aspect model

Figure 7.20 shows the sequence diagram template describing a family of scenarios with multiple replicated repositories. The replicated repository aspect has multiple lifeline templates and is modeled using `compositeAspect` tag fragments. The sequence diagram template consists of only the <<`body`>> fragment. An instantiation of the sequence diagram template should have atleast two instances of the `ReplicatedRepository` lifeline templates. A `|RepUser` object (i.e., an object of the application-specific value of `RepUser`) sends a `|doTask` message to each available repository via a totally ordered broadcast mechanism. If a repository is in a *ready* state (i.e., the repository and its contents are available for access), the task is performed by the repository. If the repository is a *not ready* state then the repository has failed. The repository may then update itself from other available repositories using the `|getReady` and `|getState` messages.

7.3.4 Applying class model composition

A banking system design in which the **addAccount** operation is protected (i.e., placed under access control) and fault tolerent (i.e, replicated) can be obtained by composing the banking system primary model shown in Figures 7.15 and 7.16 with application-specific values of the authorization aspect template diagram and replicated repository aspect template diagram.

The primary class model is composed with more than one aspect class model. Hence a weave-order needs to be established, before the aspect class models can be composed with the primary class model. This can be accomplished by using the model directive `precedes`:

```
authorization precedes replication
```

The directive states that the composition of authorization aspect class model with the primary class model precedes replicated repository aspect class model. The replicated repository aspect class model is composed with the resultant class model obtained from composing authorization aspect class model and the primary class model.

The authorization aspect class diagram template shown in Figure 7.17 must be bound to the banking application before it can composed using the tool. The bindings are shown in table 7.2. The context-specific aspect class model obtained by binding the template elements with application-specific values is shown in Figure 7.21.

The context-specific aspect class model shows that the association `requestsThro` is prohibited. The primary class model shown in 7.15 has an association between the `Client` object and the `ClientManager` object. Hence the assocation should be removed during the composition using the following composition directives:

1. **remove** `primary::requestsThro`

Table 7.2: Bindings for authorization aspect and banking application

Aspect model parameter	**Application specific element**
|Requestor	Client
|Requestor::|reqId	Client::userId
|Transact	ClientManager
|Authorizer	Session
|Authorizer::|Operation	Session::addAccount
|Operation::|reqId	addAccount::userId
|Operation::|*params1**	addAccount::accType
|AuthorizationEntry	AuthEntry
|AuthorizationEntry::|authorizedReqId	AuthEntry::authReqId
|AuthorizationEntry::|authorizedOperId	AuthEntry::authOperId
|AuthorizationRepository	AuthRepository
|AuthorizationRepository::|getAuth	AuthRepository::getAuth
|getAuth::|reqId	getAuth::userId
|getAuth::|operId	getAuth::operId
|directAccess	requestsThro
|accesses	accesses
|AuthTrans	AuthTrans
|opReps	opReps
|opReqIds	opReqIds
|a	*
|b	1..1
|c	*
|d	1..1
|m	1..*
|x	1..*
|y	1

2. **remove** `primary::ClientManager::CliCon`

3. **remove** `primary::Client::ConCli`

The removal of association requires the removal of association ends. Once the association is removed, the context-specific aspect and primary class model can be composed using the tool. The signature-based composition devoid of composition directives results in a faulty model that does not exhibit the desired properties.

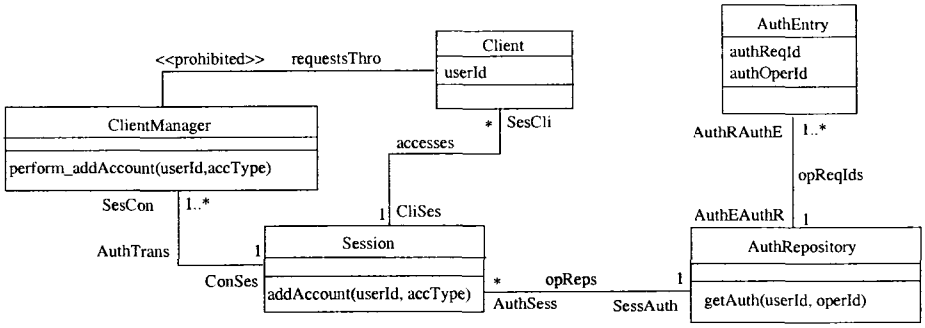


Figure 7.21: Context-specific authorization aspect class model

The resultant model has a `ClientManager` class with `addAccount` operation and `perform_addAccount` operation that accomplish the same goal. This can be avioded by applying the following composition directives:

1. **rename** `primary::ClientManager::addAccount`
   to `perform_addAccount()`

2. **replaceOccurrences** `primary::ClientManager::addAccount`
   with `primary::ClientManager::perform_addAccount()`

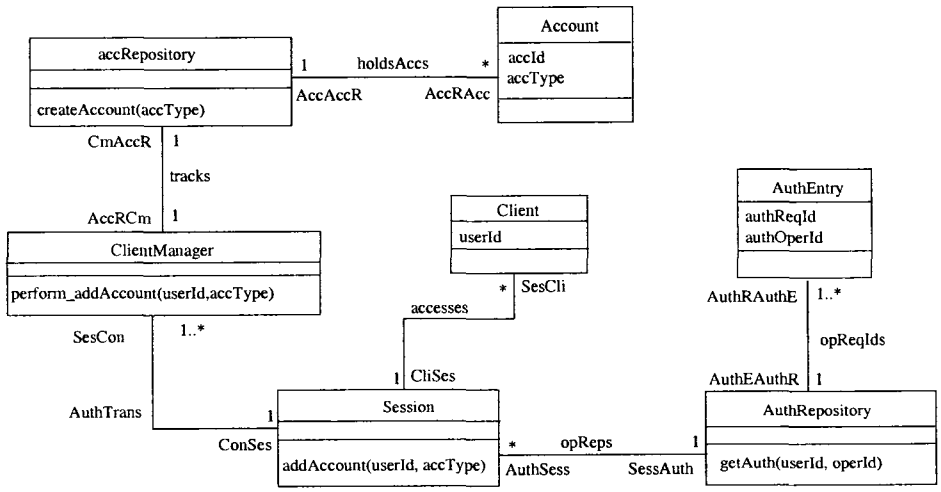


Figure 7.22: Primary class model composed with Authorization aspect class model

The composed model obtained by composing the authorzation aspect class model

with the primary model using a default signature type (*name*) for classes, operations and a complete signature types for attributes, parameters and association ends is shown in Figure 7.22.

The composed model is then integrated with the instantiation of replicated repository aspect class diagram template. The context-specific aspect class model shown in 7.23 is obtained by applying the bindings specified in table 7.3.

Table 7.3: Bindings for the authorized banking application and replicated repository aspect

Aspect model parameter	**Application specific element**
|RepUser	ClientManager
|Entity	Account
|ReplicatedRepository	accRepository
|Entity::|entityId	Account::accId
|ReplicatedRepository::|doTask	accRepository::createAccount
|doTask::|params2*	createAccount::accType
|updates	tracks
|containsEntity	holdsAccs
|a	*
|b	2
|m	1

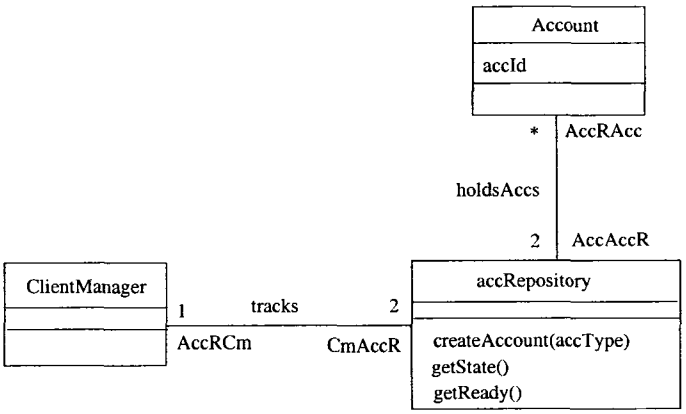


Figure 7.23: Context-specific replicated repository aspect class model

The composed model obtained by composing the authorzation aspect class model

with the primary model using a default signature type (*name*) for classes, operations and a complete signature types for attributes, parameters and association ends is shown in Figure 7.22.

The context-specific aspect class model has a `holdsAccs` association with a multiplicity value "2" on the `accRepository` end. Similarly, the `tracks` associtation has a multiplicity value "2" on the `accRepository` end. However, in the primary model the associations `holdsAccs` and `tracks` have multiplicity value "1" on the `accRepository` end. A default merge using the tool will apply the default merge semantics by choosing the stronger of the two multiplicities. A modeler may choose a weaker muliplicity using the following composition directives if needed:

1. **association end multiplicity rule**

   `primary::accRepository::CmAccR; aspect::accRepository::CmAccR;`

   weaker

2. **association end multiplicity rule**

   `primary::accRepository::AccAccR; aspect::accRepository::AccAccR;`

   weaker

Alternatively, the default composition rule can be overriden for all the elements using the composition directive:

association end multiplicity rule weaker

7.3.5 Applying Interaction model Composition

The primary sequence model shown in 7.16 describes a partial banking application that allows addition of accounts to the account repository. The primary sequence model in which `addAccount` operation call message is protected (i.e., placed under

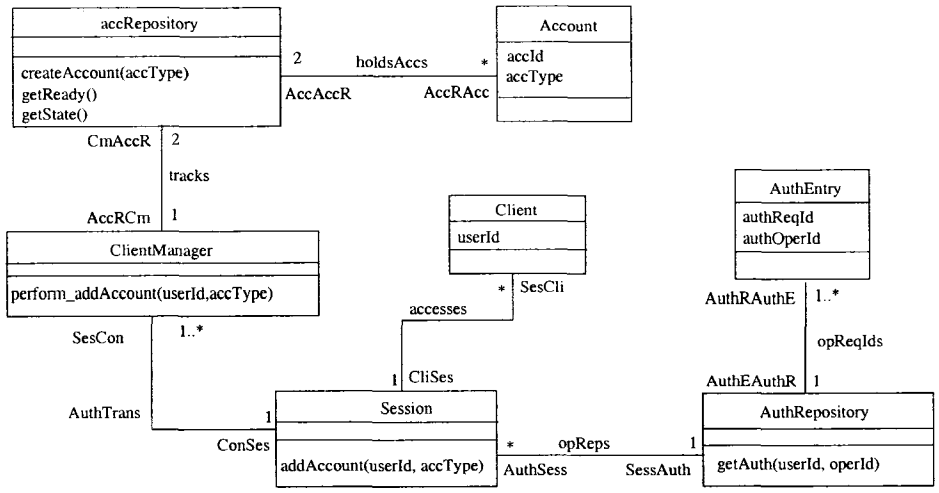


Figure 7.24: Primary class model composed with Authorization and Replicated Repository aspect class models

access control) and the account repository is fault-tolerant (i.e., repositories are replicated) can be obtained by composing the primary sequence model with the instantiations of authorization aspect sequence diagram template shown in Figure 7.18 and replication aspect sequence diagram template shown in Figure 7.20.

The `addAccount` message in the primary sequence model is tagged using an authorization `compositeAspect` tag. The system should replicate all messages sent to the account repository. The intent is to broadcast the messages to all the repositories. Hence the `createAccount` message is tagged using the compositeAspect `replication` (see Figure 7.25).

The composition of the instantiation of the aspect interaction diagram template with the primary sequence model is done using the explicit binding semantics for the authorization aspect since the binding are explicitly specified in the tagged primary sequence model. The implicit binding semantics are applied for the replication aspect. The resultant composed model is shown in Figure 7.26.

Using the explicit bindings specified in the tagged primary sequence model, the `Requestor` is bound to a `Client` object and the `Transact` to `ClientManager`.

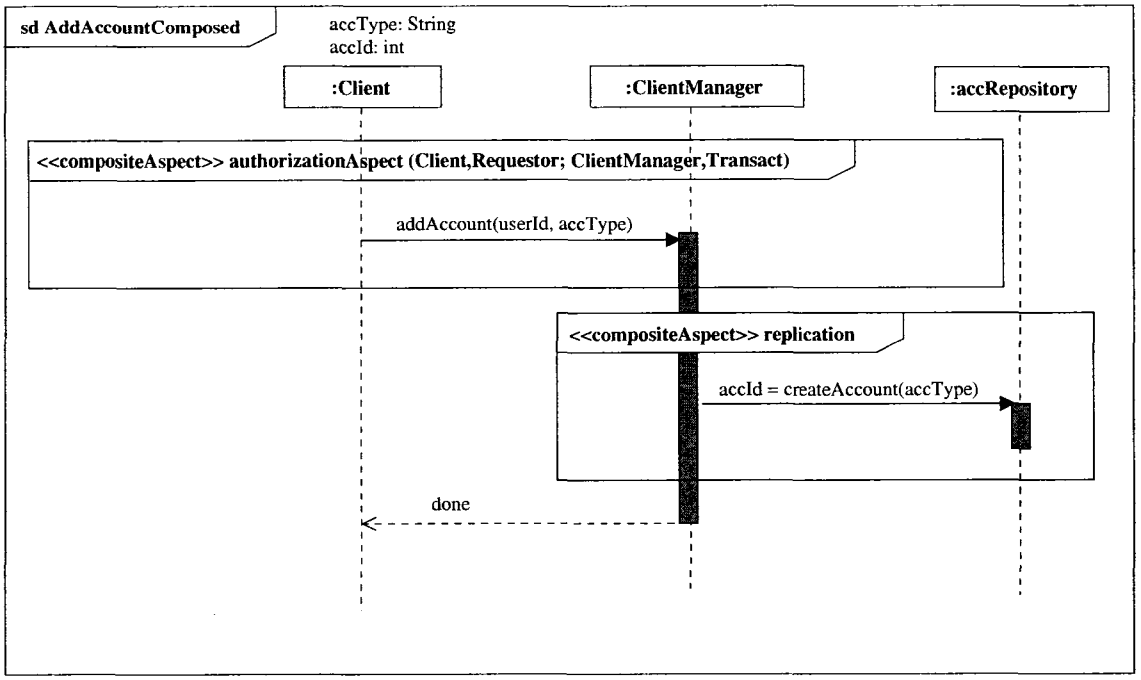


Figure 7.25: The tagged AddAccount sequence model

Based on the rules defined for a `compositeAspect` tag, the message template `Operation(reqId,params1*)` sent by the `Requestor` lifeline template and received by the `Authorizer` template can be bound to messages that origate from a `Client` object (application specific value of `Requestor` lifeline template) and received by `ClientManager` objects (application specific value of `Transact` lifeline template). This results in an association between the `Client` object and `ClientManager` object, which is <<`prohibited`>> (see Figure 7.21). The presence of such a message call will make the class model and sequence model view inconsistent with each other. Hence a message call from `Client` object to the `ClientManager` object needs to be removed in the composed sequence model.

The implicit bindings semantics of the `replication` aspect bind the `ClientManager` to `RepUser` and `accRepository` to `ReplicatedRepository`. Since replication requires atleast two repositories, the account repository is replicated twice.

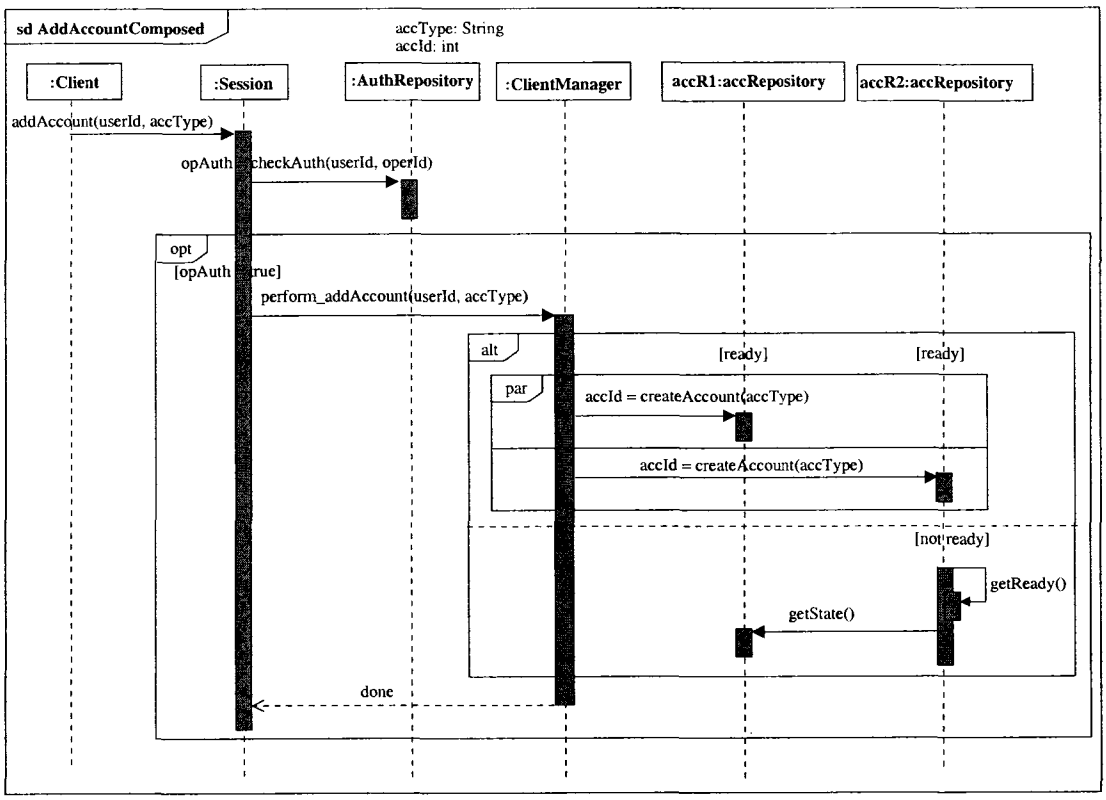


Figure 7.26: AddAccount sequence model composed with Authorization and Replicated Repository aspect sequence models

The sequence in the <<body>> fragment of the compositeAspect `authorizationAspect` refines the `addAccount` message by breaking it up into `addAccount` and `perform_addAccount` messages where the `addAccount` just requests the `Session` object to perform an authorization check for the given operation `operId`. If the operation is authorized, the `perform_addAccount` adds the account to the repository objects using `createAccount`. The `perform_addAccount` behavior exhibits the semantic properties of the `addAccount` message defined in the primary model.

The sequence in the <<body>> fragment of the compositeAspect `replication` replicates the `createAccount` message. The message is sent to both the account repositories. If an account repository is in a *not ready* state, it updates itself by

getting the current state from another repository.

Chapter 8

Conclusion and Future work

The class and interaction model composition techniques described in this thesis are a part of the broader AOMDF framework that is currently being developed at CSU. The AOMDF is an MDD framework that supports (1) separation of crosscutting features from other features and (2) transformation of features across different levels of abstractions. The AOMDF is based upon the aspect oriented modeling (AOM) approach [15]. The AOM approach supports modeling of crosscutting features as aspects and other features (core business functionality) as a primary model. The support for transformation is provided through the application QVT standards.

The aspect models are described using template RBML and the primary model is described using UML. Class diagram templates are used to describe the static structure of an aspect model and sequence diagram templates are used to describe the behavior. The template aspect models are bound to application-specific values before they are composed with the primary model. The bindings are specified as (aspect model parameter, application specific element) pairs for class models. For sequence models, the application-specific values are bound depending upon the type of aspect sequence model (compositeAspect or simpleAspect). Composition of aspect and primary model results in a composed model that describes the integrated view of the system design. The class models are composed to obtain an integrated structural view of the system design. The sequence models are composed to obtain an integrated

behavioral view of the system design.

The class model composition technique uses a signature-based approach that allows one to vary how class models are composed using composition directives. Composition directives give added flexibility by providing the means to alter model elements and override default composition rules to obtain desired composed models. The directives described in this thesis are based on the experience gained by using the composition approach to compose aspects modeling security features with primary models.

A composition metamodel that describes the static and behavioral properties needed to support class model composition is also presented. The metamodel describes the static relationships among composition concepts and provides specifications of behaviors that are needed to support model composition. As proof of concept, we implemented the class composition metamodel using the KerMeta language. The prototype implementation currently supports the composition of UML class model elements and can be extended to support additional features that appear in the composition metamodel. Since KerMeta is a metamodeling language compatible with EMF, the composition metamodel was directly added to the ecore model. The implementation in KerMeta has a one-one correpondence with the composition metamodel.

The interaction model composition technique uses tags to compose the aspect and primary sequence models. Tags are placed on the primary model to specify where in the primary model the aspect model sequence needs to be composed. Two types of aspects are described in this thesis: simpleAspect and compositeAspect. The simpleAspect is used when the aspect sequence just needs to be inserted in the primary model. The compositeAspect is used when the primary model sequence needs to be modified, or the elements in the primary model sequence need to be replaced/integrated with the elements in the aspect model.

A composition metamodel that describes the static properties of the interaction

model composition has been presented. The behavioral properties needed to implement the sequence model composition have not been addressed in this thesis. A set of atomic operations that can be used to implement the interaction model composition have been described.

The composition techniques have been applied on pilot studies (described in this thesis) and other secutiry features (role based access control features) to study the applicability of the techniques. The composition techniques can be used as a foundation for developing model analysis and transformation techniques. The composition techniques also have some limitations and issues that need to be addressed in future to be more effective.

8.1 Contribution

Many researchers [2, 4, 22, 31, 39, 56] have developed support for composing aspect-oriented programs. There is very little work on composition of design models.

Ossher *et al.*, Aldawud *et al.*, and Clarke *et al.* have proposed composition techniques at the design level. They compose design models using model element names but do not provide mechanisms to handle conflicts that may occur during composition. As shown in this thesis, composition using model element names is not sufficient. The approach by Clarke *et al.* supports overriding of model elements but this is not sufficient to adddress conflicts that may occur during composition. For example, model elements may need to be added or deleted to avoid conflicts.

In this thesis we have provide composition directives that are more extensive and can be used to resolve composition problems. Although the list of composition directives is not complete, the directives described in this theis are more extensive and well-defined. Further, we have described a composition metamodel that has been used to implement a general-purpose composition algorithm. The general-purpose composition algorithm can be varied using composition directives. We have also

described an interaction model composition technique that can be used to compose sequence models.

The properties of the composed model (for example, pre and postconditions) can be analyzed to determine if the primary and aspect model properties have been preserved. The composition technique described in this thesis enables one to build such an analysis technique by providing mechanisms to compose the pre and postconditions. Techniques for carrying out such analysis are not discussed in this thesis.

In the context of AOMDF, the composition techniques described in this thesis can be used to compose the aspect and primary models at a particular level of abstraction. The composed model can then be transformed to produce models at another level of abstraction. The models may also be transformed individually at a particular level of abstraction and then composed at another.

8.2 Composing hierarchical structures

The design models used in this thesis for applying the composition technique do not have any hierarchical structure. The classes do not have any superclass/subclass relationships. For example, consider the models, Model 1 that has two classes (*Shape* and *Rectangle*) with a superclass operation (*move*) that is overriden in the subclass *Rectangle* (see Figure 8.1(a)). The move operation is executed using set of coordinates (*x1, y1, x2, y2*). The model (*Model 2*) has two classes (*Shape* and *ShapeEditor*) associated with each other. The (*Shape* class has an operation (*move*) that moves a particular shape using the a single set of coordinates (*x1, y1*) and the angle of movement (*angle*)(see Figure 8.1(b)).

The composition of the models with a default signature (*name*) for classes, operations, parameters and association ends using the tool will result in a the a composed model shown in Figure 8.1(c). Using the specified signature, the two shape classes match and the the merge is invoked on the constituent elements. The operations

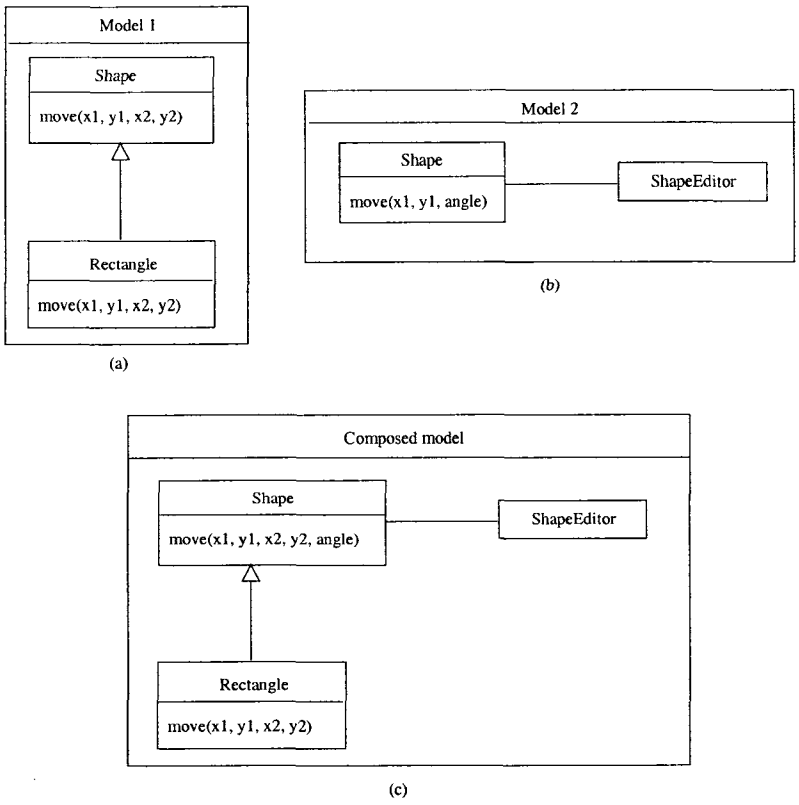


Figure 8.1: Composition of classes with hierarchical structures

move match but the in the parameter list only *x1 and y1* match. This results in a composed model with a move operation with the parameter list *x1, y1, x2, y2, angle* (see Figure 8.1. In the composed model, the *move* method in *Rectangle* class does not override the superclass *move* method. To resolve this issue, one may have to modify the parameter list using a set of composition directives. It needs to be determined if the existing composition directives can be used to composed class models with heirarchical structures.

The class model composition implemented in the tool can detect conflicts. This detection of conflicts is strictly styntactic. For, example if two attributes are matched using a complete signature type, then all the properties of the attributes should match. So an attribute of type *float* will not match an attribute of type *double* although one

is the subtype of the other.

8.3 Selection of signatures

In the class model composition technique, the class models can be composed using a default signature type or specify a subset of the complete signature type. Using a default signature type can increase the model elements that match with each while a complete signature type can restrict the number of model elements that match. A selection of a particular signature type may result in the application of more composition directives or vice versa. The approach does not define a systematic approach or a criteria for selecting a particular signature type. It's upto the modeler to define a signature type that can best reduce the usage of composition directives. A set of guidelines that help a modeler to choose particular signatures for particular model element compositions are needed.

8.4 Validation of the composition techniques

Empirical evaluation is needed to validate the composition techniques. Such an evaluation requires the existence of an integrated robust tool set that can be used to perform the various tasks in AOMDF. The lack of such a tool set has limited the validation of the techniques described in this thesis. Further, an independent validation limits bias. Such studies can determine the amount of effort required to apply the compositions in complex designs.

The studies can also be used to determine whether the composition directives match the requirements of a complex project. The insights gained from the studies may be used to develop a tractable method for selecting, defining, and applying composition directives and signatures in class model composition. Work in this respect could result in the specification of some common composition strategies to ease the task of specifying and using composition directives.

The studies can show the relationship between the class and sequence model composition techniques. This will help in determining whether the class models need to be composed before the sequence model or vice versa. The experience gained from such studies can help in defining a common composition metamodel that establishes a clear relationship between the class and sequence models. Further, this common metamodel may be used to guide the implementation of tools that can compose any UML model type.

8.5 Tool Support

In the context of AOMDF, following tool support is available at Colorado State Univerisity:

- A prototype editor for creating aspect class diagram templates.
- A tool, built on top of Rational Rose, that generates instantiations from UML class diagram templates.
- A prototype model composer (developed as part of this thesis) implemented in KerMeta that takes in primary class model and context-specific aspect class model and outputs a composed class model.

The prototype tool implemented for composing class models does not allow composition directives. Hence the tool needs to be extended to provide support for composition directives. Implementing the functionality to handle composition directives in the tool will enable one to specify the order of composition if multiple aspect models are to be composed with the aspect model. In cases where the composed model is expected to exhibit undesirable properties, one can just input the composition directives to the tool to obtain the desired results. For example, if a primary model needs to be modified, one has to change the primary model manually and then

input the changed primary model to the tool. In cases where the composition results in conflicts, currently a developer needs to explicitly handle these conflicts using the composition directives. For example, if the aspect model needs to override primary model elements in all the cases, one may have to specify the composition directive multiple times. The tool needs to handle this by allowing a global specification of the overriding characteristics for the aspect and primary model elements.

The prototype tool needs to be extended with interaction model specific features. This will enable the composition of sequence models. To accomplish this, the atomic operations specified in this thesis need to implemented. The implementation will help the developer to determine whether the atomic composition operations are sufficient to perform composition on different variants of sequence models.

8.6 Consistency of Framework

A systematic approach to maintain the consistency between class and sequence model views needs to be developed. Such an approach will enable one to explore the relationships between the class and interaction model composition. It's not enough, if one develops an approach to maintain consistency of design views. A developer also needs to ensure the consistency of aspect, primary or composed models when they are transformed across different levels of abstraction.

In AOMDF, the composition can be performed at the source level, target level or at both levels. Depending on where the composition is performed, the target level composed model can be obtained by using two different paths: (1) Composition of transformed aspect and primary models and (2) Transformation of composed model. In the first path, the target level aspect and primary models are obtained by transformation and then the aspect and primary models are integrated to obtain a composed model. In the second path, the source level aspect and primary models are integrated at the source level to obtain a composed model. The source level composed model is

then transformed to obtain a target level composed model. The AOMDF supports using either paths and thus, one has to ensure that the composition techniques produce consistent results for both paths. Consistency implies the preservation of properties through refinement [14].

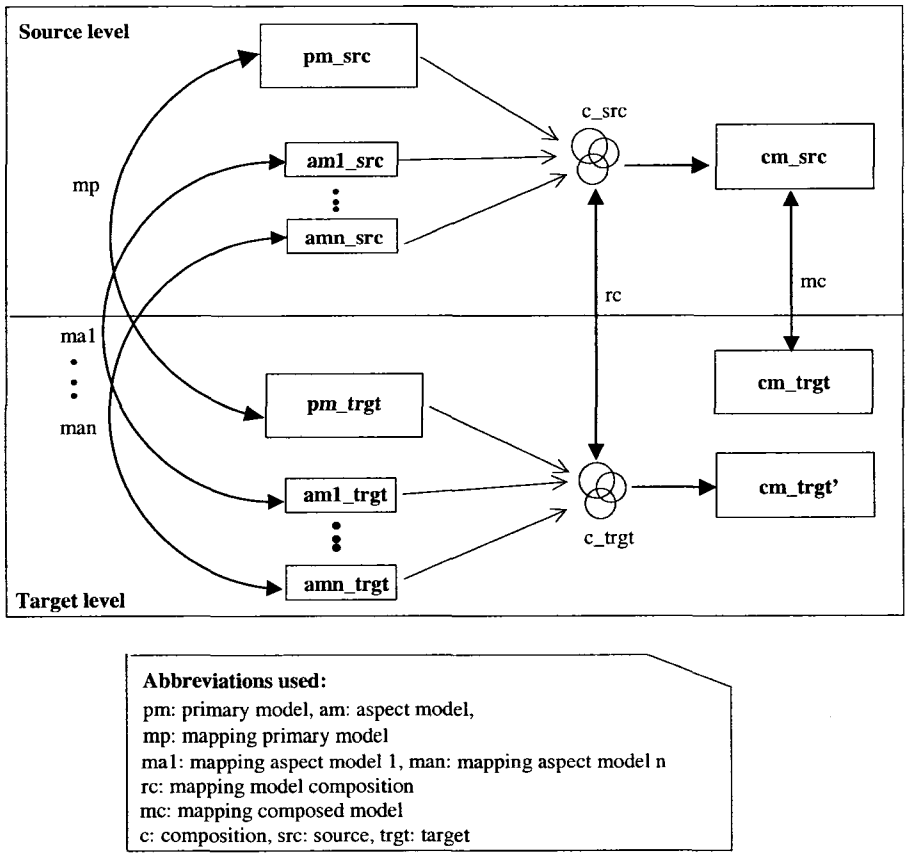


Figure 8.2: Two paths to obtain composed model

In Figure 8.2, *cm_src*, a composed model at the source level is created by composing a source level primary model *pm_src* and a set of source level aspect models *am_src* using a composition technique c_src. To establish consistency, we need to show that for any mapping specification *mc* that maps *cm_src* to a corresponding model at the target level *cm_trgt*, there exists an equivalent mapping consisting of:

- Separate mappings (*mal...man*) for each of the aspects (*aml_src...amn_src*)

that can be used to derive a set of corresponding aspects ($am1\_trgt...amn\_trgt$) at the target level.

- A mapping (mp) of the primary model pm_src to a corresponding target level primary model (pm_trgt).

- A composition technique at the target level that produces a composed model (cm_trgt'), where cm_trgt is consistent with cm_trgt.

We identify the following three necessary conditions for these two paths to be consistent:

1. The composition at the target level (c_trgt) is a refinement (rc) of the composition at the source level (c_src).

2. pm_trgt is consistent with pm_src, $am1\_trgt...amn\_trgt$ is consistent with $am1\_src...amn\_src$.

3. cm_trgt is consistent with cm_src.

A proof can be derived to show that condition 1-3 are sufficient to make the equivalence of the two paths true. This follows since if condition 1, 2 and 3 holds cm_trgt is consistent with cm_src. Proofs that establish the consistency of framework can lead to a more effective usage of the framework and need to be addressed in the framework.

Appendix A

Money Transfer Service with Transaction management

The Figure A.1 shows expanded version of money transfer sequence model composed with transaction aspect sequence model shown in 7.9. The <<end>> fragment has been expanded in this figure. The end fragment shows how the process is committed or aborted depending the result of the voting phase of the transaction.

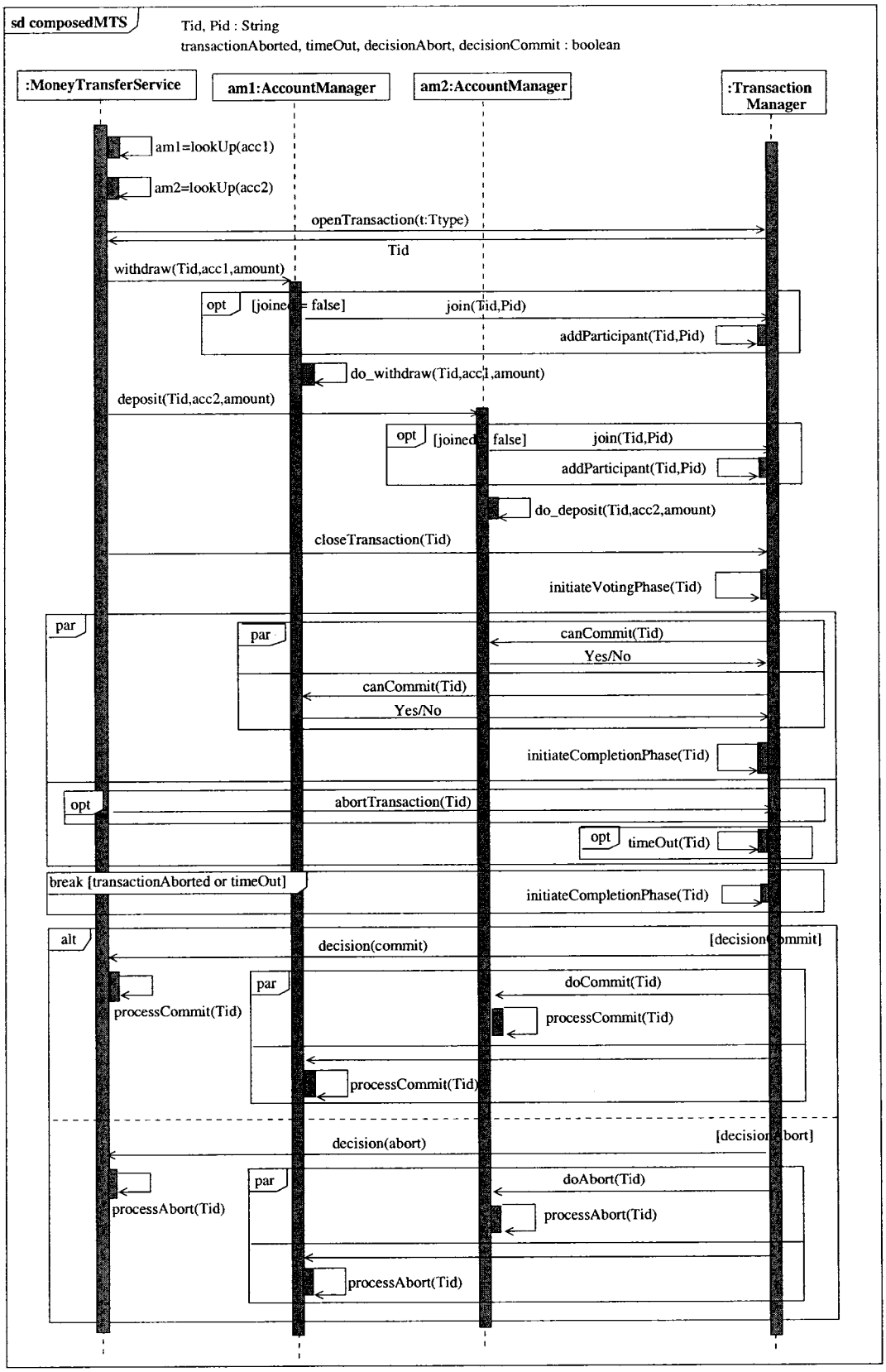


Figure A.1: Expanded Money transfer sequence model composed with transaction aspect sequence model

Appendix B

Merge algorithm

The merge part of the composition algorithm is shown below. Notice the algorithm is deviod of composition directives. The algorithm performs a recursive merge of the given model elements based on the specified signature.

```
**************************************************************
// e1 and e2 are the model elements that need to be merged
//precondition : e1.sigEquals(e2) returns true

e1.merge(e2 : ModelElement)
**************************************************************
// create the merged instance in the context of e1
result := e1.getMetaClass.new

// Iterate on all properties of the objects to be merged
// e1 and e2 have the same metaclass.
// So they have me same set of properties.

foreach Property p in e1.getMetaClass.getAllProperties
  if type of p is primitive
```

```
// Primitive type is the basic datatype like string, int, etc,.
// If an object does not have a value for a property then
// the value val is taken from the other object and vice versa.
// This is not a conflict.
// If neither object has values,
// then val is null in the merged object.

  if e1.get(p) is null or e2.get(p) is null then
       result.set(p, val)
   else
       // if the values are the same then it is ok
       // otherwise a conflict has been detected.
       if e1.get(p) = e2.get(p) then
            result.set(p, e1.get(p))
       else
            A conflict has been detected
else
 // Type of p is not primitive.
 // If the property refers to a single object

   if the property upper bound is 1
       if e1.get(p) is null or e2.get(p) is null then
           result.set(p, val)      // val is the same as above
       else
           if sigEquals(e1.get(p), e2.get(p)) then
           // If the object e1.get(p) is contained by e1
           // and same for e2
```

```
            // (p.isComposite=true) then the objects should be merged,
            // otherwise, one is choosen.
            // Either one can be chosen because they both
                        // have the same signature
                  if p.isComposite is true then
                        result.set(p, merge(e1.get(p), e2.get(p)))
                  else
                        result.set(p, e1.get(p).clone())
            else
                  A conflict has been detected
else
 // The property refers to a collection of objects.
 // The merged object should contain property values that are only
 // in e1 or only in e2,
 // and the merged version of objects that are in both e1 and e2.
    for each value v1 in e1.get(p)
        for each matching element v2 in e2.get(p)
            if p.isComposite then
                result.get(p).add(merge(v1, v2))
             else
                result.get(p).add(v1.clone())
                if no element found
                   result.get(p).add(v1.clone())
        for each value v2 in e2.get(p)
            if NO matching element found in e1.get(p)
                result.get(p).add(v2.clone())
```

REFERENCES

[1] M. Aksit and B. Tekinerdogan. Solving the modeling problem of object-oriented languages by composing multiple aspect using composition filters. In *AOP 1998 workshop paper*, 1998.

[2] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting Object Interactions Using Composition Filters. In Rachid Guerraoui, Oscar Nierstrasz, and Michel Riveill, editors, *Proceedings of the ECOOP'93 Workshop on Object-Based Distributed Programming*, volume 791, pages 152–184. Springer-Verlag, 1994.

[3] O. Aldawud, A. Bader, and T. Elrad. Weaving with statecharts. In *Workshop on Aspect-Oriented Modeling (held with AOSD-2002)*, Enschede, Netherlands, 2002.

[4] J. Araujo and P. Coutinho. Identifying aspectual use cases using a viewpoint-oriented requirements method. In *Early Aspects 2003: Aspect Oriented Requirements Engineering and Architecture Design, Workshop of the 2nd International Conference on Aspect-Oriented Software Development*, Boston, MA, March 2003.

[5] J. Araujo, J. Whittle, and D.K. Kim. Modeling and composing scenario-based requirments with aspects. In *Proceedings of 12th IEEE conference on Requirements Engineering*, pages 53–62, Kyoto, Japan, September 2004.

[6] E. Baniassad and S. Clarke. Theme: An approach for aspect-oriented analysis and design. In *Proceedings of the International Conference on Software Engineering*, pages 158–167, 2004.

[7] B. Baudry, F. Fleury, R. B. France, and R. Reddy. Exploring the relationship between model composition and model transformation. In *Aspect Oriented Modeling workshop held with MODELS/UML 2005*, Montego Bay, Jamaica, October 2005.

[8] J. Bosch. *Design and Use of Sofware Architectures: Adopting and evolving a product-line approach.* Addison Wesley Professional, 2000.

[9] J. Brichau and M. Haupt. Survey of aspect-oriented languages and execution models. Technical Report AOSD-Europe-VUB-01, AOSD - Europe, May 2005.

[10] I. Brito and A. Moreira. Towards a composition process for aspect-oriented requirements. In *Proceedings of the Early-Aspects Workshop at AOSD2002*, 2002.

[11] R. Chitchyan, A. Rashid, P. Sawyer, A. Garcia, M. Alarcon, J. Bakker, B. Tekinerdogan, S. Clarke, and A. Jackson. Survey of aspect-oriented analysis and design approaches. Technical Report ULANC-9, AOSD - Europe, May 2005.

[12] S. Clarke. Extending standard UML with model composition semantics. *Science of Computer Programming*, 44(1):71–100, July 2002.

[13] S. Clarke and R. J. Walker. Composition Patterns: An approach to Desigining Reusable Aspects. In *Proceedings of 23rd International Conference on Software Engineering (ICSE)*, pages 5–14, Toronto, Canada, May 2001.

[14] D. DSouza and A.C. Wills. *Objects, Components, and Frameworks with UML: The Catalsysis Approach.* Addison-Wesley Object Technology Series, 1998.

[15] R. France, I. Ray, G. Georg, and S. Ghosh. An aspect-oriented approach to early design modeling. *IEE Proceedings - Software, Special Issue on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design*, 151(4):173–185, August 2004.

[16] R. B. France, D. Kim, S. Ghosh, and E. Song. A UML-Based Pattern Specification Technique. *IEEE Trans. on Software Eng.*, 30(3):193–206, March 2004.

[17] G. Georg, R. B. France, and I. Ray. Composing aspect models. In *4th AOSD Modeling with UML workshop*, San Francisco, CA, October 2003.

[18] G. Georg, I. Ray, and R. B. France. Using aspects to design a secure system. In *Proceedings of the Eighth International Conference on Engineering of Complex Computer Systems*, pages 117–126. IEEE Computer Society, 2002.

[19] Geri Georg, Indrakshi Ray, and Robert France. Using Aspects to Design a Secure System. In *Proceedings of the Interational Conference on Engineering Complex Computing Systems (ICECCS 2002)*, pages 117–126, Greenbelt, MD, December 2002. ACM Press.

[20] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering.* Prentice Hall, Upper Saddle River, New Jersey, second edition, 2003.

[21] H. Giese and A. Vilbig. Separation of Non-Orthogonal Concerns in Software Architecture and Design. *Software and Systems Modeling (SoSym)*, To appear.

[22] J. Gray, T. Bapty, S. Neema, and J. Tuck. Handling crosscutting constraints in domain-specific modeling. *Communications of the ACM*, 44(10):87–93, October 2001.

[23] J. Greenfield, K.Short, S. Cook, S. Kent, and J. Crupi. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools.* Wiley, 1st edition, August 2004.

[24] J.C. Grudy. Supporting aspect-oriented component-based systems engineering. In *Proceedings of the 11th International Conference on Software Engineering and Knowledge Engineering*, pages 388–395, Kaiserslautern, Germany, June 1999.

[25] J. C. Grundy. Multi-perspective specification, design and implementation of software components using aspects. *International Journal of Software Engineering and Knowledge Engineering*, 20(6):713–734, December 2000.

[26] W. Harrison and H. Ossher. Subject oriented programming (a critique of pure objects). In *Proceedings of the 8th Annual Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA '93)*, pages 411–428, Washington, D.C., September 1993.

[27] W. Harrison, H. Ossher, and P. Tarr. Asymmetrically vs. symmetrically organized paradigms for software composition. Technical report, IBM - RC22685 (W0212-147), December 30 2002.

[28] R. Hilliard. Aspects, concerns, subjects, views, ... In *First Workshop on Multi-Dimensional separation of Concerns in Object-Oriented Systems*, Denver, Colorado, November 1999.

[29] W. Ho, F. Pennaneach, J. Jézéquel, and N. Plouzeau. Aspect-Oriented Design with the UML. In *Proceedings of Multi-Dimensional Separation of Concerns Workshop at ICSE*, pages 60–64, 2000.

[30] W. Ho, F. Pennaneach, and N. Plouzeau. UMLAUT: A Framework for Weaving UML-Based Aspect-Oriented Designs. In *TOOLS '00: Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 33)*, pages 324–334. IEEE Computer Society, 2000.

[31] I. Jacobson. Case for Aspects - Part I. *Software Development Magazine*, pages 32–37, October 2003.

[32] I. Jacobson. Case for Aspects - Part II. *Software Development Magazine*, pages 42–48, November 2003.

[33] J. Jézéquel, A. Guennec, F. Pennaneach, G. Sunyé, and K. Vinceller. The UMLAUT Web Page. *URL* `http://www.irisa.fr/UMLAUT/`, 2004.

[34] J. Jürgens. Constructing tool-support for sophisticated analysis of uml models: A hands-on introduction. In *Tutorial*, Lisbon, Portugal, October.

[35] J. Jürjens, B. Rumpe, R. France, and E. Fernandez. Critical Systems Development with UML. In *Proceedings of UML03 Satellite workshop Critical Systems Development with UML*, number TUM-10317.

[36] M. Katara and S. Katz. Architectural views of aspects. In *Proceedings of International Conference on Aspect-Oriented Software Development*, pages 1–10, Boston, USA, 2003.

[37] R. Kazman, L. J. Bass, M. Webb, and G.D. Abowd. SAAM: A method for analyzing the properties of software architectures. In *International Conference on Software Engineering*, pages 81–90, Sorrento, Italy, May 1994.

[38] G. Kiczales, E. Hilsdale, M. Kersten, J. Palm, and W. Griswold. An overview of aspectj. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 327–353. Springer Verlag LNCS, June 2001.

[39] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingier, and J. Irwin. Aspect oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Springer Verlag LNCS 1241*, pages 220–242, Finland, June 1997.

[40] J. Kienzle, Y. Yu, and J Xiong. On composition and reuse of aspects. In *Proceedings of the Foundations of Aspect-Oriented Languages Workshop*, Boston, MA, USA, March 2003.

[41] D. K. Kim, R. B. France, S. Ghosh, and E. Song. A UML-Based Metamodeling Language to Specify Design Patterns. In *Workshop on Software Model Engineering (WiSME) with UML 2003*, San Francisco, California, October 2003.

[42] D.K. Kim. *A Meta-Modeling Approach to Specifying Patterns.* PhD thesis, Colorado State University, 2004.

[43] J. Kienzle M. Kande and A. Strohmeier. From aop to uml - a bottom-up approach. In *Aspect Oriented Modeling workshop held with Aspect Oriented Software Development conference*, Enschede, The Netherlands, April 2002.

[44] P. Muller, F. Fleury, and J. Jézéquel. Weaving executability into object-oriented meta-languages. In *Proceedings of MODELS/UML 2005*, pages 264–278, Montego Bay, Jamaica, October 2005.

[45] B. Nuseibeh. Crosscutting requirements. In *Proceedings of the 3rd International Conf. on Aspect-Oriented Software Development (AOSD 2004)*, pages 3–4, Lancaster, UK, 2004.

[46] B. Nuseibeh, J. Kramer, and A. Finkelstein. A framework for expressing the relationships between multiple views in requirements specification. *IEEE Transactions on Software Engineering*, 20(10):760–773, 1994.

[47] OMG Adopted Specification ptc/03-10-04. The Meta Object Facility (MOF) Core Specification. Version 2.0, OMG, http://www.omg.org.

[48] OMG Document ad/2003-01-07. Response to the UML2.0 OCL RFP(ad/2000-09-03). Version 1.6 Revised Submission, OMG, http://www.omg.org, January 2003.

[49] H. Ossher, M. Kaplan, A. Katz, W. Harrison, and V. Kruskal. Specifying subject-oriented composition. *Theory and Practice of Object Systems, Wiley and Sons*, 2(3), 1996.

[50] H. Ossher and P. Tarr. Multi-Dimensional Separation of Concerns and The Hyperspace Approach. In *Symposium on Software Architectures and Component Technology: The State of the Art in Software Development.* Kluwer, 2000.

[51] D. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[52] S.L. Pfleeger and J.M. Atlee. *Software Engineering - Theory and Practice.* Prentice Hall, Upper Saddle River, New Jersey, third edition, 2006.

[53] I. Porres. *Modeling and Analyzing Software Behavior in UML.* PhD thesis, Turku Center for Computer Science, 2001.

[54] QVT-Merge Group 1.8. Revised submission for MOF 2.0 Query/Views/Transformations RFP (ad/2002-04-10). Technical report, OMG, http://www.omg.org.

[55] A. Rashid, A. Moreira, and J. Araujo. Modularization and composition of aspectual requirements. In *2nd International Conference on Aspect-Oriented Software Development*, pages 11–20, Boston, March 2003. ACM.

[56] A. Rashid, P. Sawyer, A. Moreira, and J. Araujo. Early aspects: A model for aspect-oriented requirements engineering. In *IEEE Joint International Conference on Requirements Engineering*, pages 199–202, Essen, Germany, September 2002.

[57] R. Reddy, R. B. France, and G. Georg. Aspect oriented modeling approach to analyzing dependability features. In *Aspect Oriented Modeling workshop held with Aspect Oriented Software Development conference*, Chicago, March 2005.

[58] R. Reddy, R. B. France, S. Ghosh, F. Fleury, and B. Baudry. Model composition - a signature based approach. In *Aspect Oriented Modeling workshop held with MODELS/UML 2005*, Montego Bay, Jamaica, October 2005.

[59] Y. R. Reddy, R. B. France, and G. Georg. An aspect-based approach to modeling and analyzing dependability features. Technical Report CS04 - 109, Colorado State University, November 2004.

[60] Y. R. Reddy, S. Ghosh, R. B. France, G. Straw, J.M. Bieman, N. McEachen, E. Song, and G. Georg. Directives for composing aspect-oriented design class models. *Transactions on Aspect-Oriented Software Development 1*, 3880:75–105, Febraury 2006.

[61] K.G. Saha. Beyond the conventional techniques of software fault tolerance. *ACM ubiquity*, 4(47):87–93, 2004.

[62] D. Simmonds, A. Solberg, R. Reddy, R.B. France, and S. Ghosh. An aspect oriented model driven framework. In *Proceedings of the Ninth IEEE "The Enterprise Computing Conference" (EDOC 2005)*, pages 119–130, Enschede, Netherlands, September 2005. IEEE Computer Society Press.

[63] A. Solberg, D. Simmonds, R. Reddy, R.B. France, S. Ghosh, and J. Aagedal. Develop service oriented systems using An Aspect Oriented Model Driven Framework. *International Journal of Cooperative Information Systems (IJCIS)*, To Appear.

[64] A. Solberg, D. Simmonds, R. Reddy, S. Ghosh, and R.B. France. Using aspect oriented technologies to support separation of concerns in model driven development. In *Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC 2005)*, volume 1, pages 121–126, Ediburgh, Scotland, July 2005. IEEE Computer Society Press.

[65] E. Song, R. Reddy, R. France, I. Ray, G. Georg, and R. Alexander. Verifiable composition of access control features and applications. In *Proceedings of the 10th ACM Symposium on Access Control Models and Technologies (SACMAT 2005)*, pages 120–129.

[66] D. Stein, S. Hanenberg, and R. Unland. A UML-based Aspect-Oriented Design Notation For AspectJ. In *Proceedings of the 1st International Conf. on Aspect-oriented software development*, pages 106–112, Enschede, The Netherlands, 2002. ACM Press.

[67] D. Stein, S. Hanenberg, and R. Unland. On representing join points in the uml. In *Aspect Oriented Modeling workshop held with UML 2002*, Dresden, Germany, October 2002.

[68] S. M. Sutton and I. Rouvellou. Modeling of Software Concerns in Cosmos. In *Proceedings of the 1st International Conf. on Aspect-oriented software development*, pages 127–133, Enschede, The Netherlands.

[69] S.M. Sutton. Early stage concern modeling. In *Early Aspects Workshop, Held with Aspect Oriented Software Development*, 2002.

[70] J. Suzuki and Y. Yamamoto. Extending UML with aspects: Aspect support in the design phase. In *3rd ECOOP Aspect-Oriented Programming Workshop*, Lisbon, Portugal, June 1999.

[71] P. Tarr, H. Ossher, W. Harrison, and S.M. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering (ICSE '99)*, pages 107–119, May 1999.

[72] B. Tekinerdogan. ASAAM: Aspectual Software Architecture Analysis Method. In *Fourth Working IEEE/IFIP Conference on Software Architecture (WICSA'04)*, Oslo, Norway, June 2004.

[73] B. Tekinerdogan. The Aspect-Oriented Software Architechture Design Portal. *URL http://trese.cs.utwente.nl/taosad/publications.htm*, 2004.

[74] The Object Management Group (OMG). Common Warehouse Metamodel (CWM) Specificaton. Version 1.0, OMG, http://www.omg.org, Febraury 2001.

[75] The Object Management Group (OMG). Unified Modeling Language. Version 1.4, OMG, http://www.omg.org, 2001.

[76] The Object Management Group (OMG). OMG MDA Guide. Version 1.0.1, OMG, http://www.omg.org, 2003.

[77] The Object Management Group (OMG). Unified Modeling Language Specification: Infrastructure. Version 2.0, Final Adopted Specification, OMG, http://www.omg.org, September 2003.

[78] The Object Management Group (OMG). Unified Modeling Language: Superstructure. Version 2.0, Final Adopted Specification, OMG, http://www.omg.org, August 2003.

[79] M. Tkatchenko and G.Kiczales. Uniform support for modeling crosscutting strcuture. In *Proceedings of MODELS/UML 2005*, pages 508–521, Montego Bay, Jamaica, October 2005.

[80] TRISKELL. The KerMeta Project Home Page. *URL* `http://www.kermeta.org`, 2005.