Composition of Aspects Represented as UML Sequence Diagrams

by

Pantanowitz Motshegwa, B.Eng., University of Victoria, 2004

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of

Master of Applied Science

in

Electrical and Computer Engineering

Ottawa-Carleton Institute of Electrical and Computer Engineering

Department of Systems and Computer Engineering

Carleton University

Ottawa, Canada

November, 2009

Your file Votre référence
ISBN: 978-0-494-63810-1
Our file Notre référence
ISBN: 978-0-494-63810-1

Abstract

Aspect Oriented modeling (AOM) empowers software architects to isolate and work with cross-cutting concerns as aspect models, and methodically integrate them with the core system model using aspect composition techniques. The goal of this thesis is to compose aspect models represented as UML sequence diagrams using transformation models written in ATL (Atlas Transformation Language). We propose and implement a Complete Composition Algorithm implemented using three ATL transformation models; namely *JoinPointsCount*, *Instantiate*, and *Compose*. The *JoinPointsCount* transformation determines the number of join points in the primary (core system) model. The *Instantiate* transformation is used to instantiate generic aspect models in the context of the application using a set of binding rules defined in mark models to produce context specific aspect models. The *Compose* transformation then takes the primary model and context specific aspect model as inputs, and produces a composed (integrated) model. The composed model is validated using a custom Java package we created to check the model against a list of defined constraints. The implementation is tested using a pair of case studies and several test cases.

Acknowledgements

First, I would like thank my supervisors, Professors Dorina Petriu and Samuel Ajila for providing me with great support and guidance during the course of studies, and research.

Secondly, I would like to thank my family for their unconditional love and support that seems to age like wine. Your love and faith in me kept me going during the difficult times. I would also like to thank all my friends for their help and support. May the force be with you.

Last but certainly not least, I would like to thank my fiancé, Mavis, for her continuous support and infinite love that would require all the computing power in the universe to model and simulate. Adapting a quote from Professor John Nash, I would like to say:

"You are the most important discovery of my life. It is only in the mysterious algorithms of love that any logic or reasons can be found. You the reason I am. You are all my reasons. "

Thank you.

Table of Contents

List of Figures

List of Tables

List of Acronyms

AM	Aspect Model
AO	Aspect-Oriented
AOM	Aspect-Oriented Modeling
AOP	Aspect-Oriented Programming
AOSD	Aspect-Oriented Software Development
ATL	Atlas Transformation Language
BES[s]	BehaviorExecutionSpecification[s]
CPA	Critical Pair Analysis
EMF	Eclipse Modeling Framework
EOS	ExecutionOccurrenceSpecification[s]
IDE	Integrated Development Environment
LHS	Left Hand Side
MOF	Meta-Object Facility
MOS[s]	MessageOccurrenceSpecification[s]
OCL	Object Constraint Language
OMG	Object Management Group
OOP	Object-Oriented Programming
OS[s]	OccurrenceSpecification[s]
PDE	Plug-in Development Environment
PM	Primary Model
RHS	Right Hand Side
ROE[s]	ReceiveOperationEvent[s]
RSA	Rational Software Architect
SD[s]	Sequence Diagram[s]
SDK	Software Development Kit
SOE[s]	SendOperationEvent[s]
UML	Unified Modeling Language

Chapter 1 : Introduction

1.1 Introduction

Aspect Oriented (AO) techniques have been successfully applied at the code level in Aspect Oriented Programming (AOP) to help developers deal with scattered and entangled code that results from trying to implement solutions for cross-cutting concerns. A *concern* is a system requirement or feature. Cross-cutting concerns are those concerns whose solutions are spread across components or modules of the core system functionality (or architecture) [France+04]. These concerns include security, persistency, reliability, etc. Code that realizes these concerns ends up been scattered resulting in code that is not reusable, difficult to understand, and harder to maintain or modify [Colyer+05]. Object Oriented (OO) techniques have proved inadequate in overcoming this problem. Pioneers of AOP introduced a programming construct named "aspect" to enhance OO techniques and help modularize solutions to cross-cutting concerns. An aspect encapsulates code that instead of been scattered and duplicated across the application, is only invoked when needed and can be easily modified or replaced without affecting the main system. An aspect consists of a pointcut and an advice. The advice is the code that has to be executed while the pointcut is a condition that describes when and where the advice should be executed.

Modern software systems are huge, complex, and greatly distributed. In having to design and model such systems, software architects are faced with the problem of cross-cutting concerns much earlier in the development process. At this level, cross-cutting concerns result in model elements that cross-cut the structural and behavioral views of the system. Research has shown that AO techniques can also be applied to software models. This can greatly help software architects and developers to isolate, reason, express, conceptualize, and work with cross-cutting concerns separately from the core functionality [Petriu+07]. This application of AO techniques much earlier in the development process has spawned a new field of study called Aspect-Oriented Modeling (AOM). In

AOM, the aspect that encapsulates the cross-cutting behavior or structure is a model, just like the base system model it cross-cuts. A system been modeled has several views including structural and behavioral views. Therefore, a definition of an aspect depends on the view of interest. Unified Modeling Language (UML) provides different diagrams to describe the different views. Class, Object, Composite Structure, Component, Package, and Deployment diagrams can be used to represent the structural view of a system or aspect. On the other hand, Activity, State Machine, and Interaction diagrams are used to model the behavioral view. Interaction diagrams include Sequence, Interaction Overview, Communication, and Timing diagrams.

After reasoning and working with aspects in isolation, the aspect models eventually have to be combined with the base system model to produce an integrated system model. This merging of the aspect model with the base model is called Aspect Composition or Weaving. Several approaches have been proposed for aspect composition using different technologies/methodologies such as graph transformations [Whittle+07], matching and merging of model elements [Fleurey+07], weaving models [Didonet Del Fabro+06] and others.

The goal of this thesis is to compose aspect models represented as UML sequence diagrams using transformation models written in ATL (Atlas Transformation Language).

1.2 Thesis Motivation and Objectives

Composing behavioral models (views) represented as UML Sequence diagrams is more complex than composing structural views. Not only is the relationships between the model elements important but the order is equally paramount. Several approaches have been proposed for composing behavioral aspects with core system behavior, and these include graphs transformations [Whittle+07] and generic weavers [Morin+08].

In this thesis we view composition as a form of model transformation. Aspect composition can be considered a model transformation since it transforms aspect and primary models to an integrated system model. Toward this end, we propose and focus on an approach that uses model transformations to compose both primary and aspect models rep-

resented as UML Sequence diagrams (SDs). SDs modeling the primary model and generic aspect models are created using Graphical UML modeling tools like Rational Software Architect (RSA). Model transformations are then used to instantiate the generic aspect models in the context of the application to produce context specific aspect models. Binding rules used for instantiating the generic aspect are represented as mark models that conform to a metamodel. Using other model transformations, the context specific aspect models are then composed with the primary model to produce an integrated system model.

Verification and validation is performed, not only to verify that composition was successful, but also to ensure that the composed model is a valid UML model that can be processed further and shared with other researchers.

1.3 Contributions of the Thesis

The main contributions of this work are summarized as follows:

1. Proposed a formal definition of SDs (primary and aspect models) in terms of a sequence of interaction fragments.
2. Proposed three algorithms for pointcut detection, advice composition, and complete composition.
3. Designed and implemented the Complete Composition algorithm to achieve composition of the primary model and generic aspect models represented as SDs. Composition is viewed as model transformation; hence, the algorithm is implemented using model transformations written in ATL model transformation language.
4. Proposed and designed a simple metamodel for mark models. The metamodel is defined in Ecore, a metametamodel for the Eclipse Modeling Framework. The mark models are used to define the binding rules to instantiate generic aspect models.

5. Designed and implemented a custom Java package to validate the composed model. The Java classes check the composed model elements against a list of defined constraints designed to ensure that essential model elements are present in the composed model and that they are properly initialized.

1.4 Thesis Outline

This thesis consists of six chapters. Chapter 2 introduces current state of the art in AOM composition, Model Driven Engineering, ATL, Eclipse and other topics related to work done in the thesis. Chapter 3 introduces our AOM composition approach and in the process gives a definition of the primary model, generic and context specific aspect models, and mark models. The UML Interactions metamodel is also briefly looked at, and a formal notation introduced to formally define the primary and aspect models. Chapter 3 then introduces the Pointcut Detection, Advice Composition and Complete Composition algorithms used for aspect composition. Chapter 4 discusses an implementation of the Complete Composition algorithm using ATL transformations. Chapter 5 describes case studies and test cases used for testing the implementation. This chapter also introduces how the composed models were validated using a custom Java package and an off-the-shelf modeling tool (RSA). Chapter 6 summarizes the work done in the thesis and some open topics for future work.

Chapter 2 : Literature Review

This chapter looks at state of the art of Model Driven Engineering (MDE), Aspect-Oriented Modeling, Aspect Composition, Atlas transformation language, and UML tools (Eclipse based).

2.1 Model Driven Engineering/Development (MDE/MDD)

In Model Driven Engineering (MDE) everything is a model. By a model, were are referring to a simplified view of a real world system of interest; that is, an abstraction of a system [SYSC 5708]. MDE considers models as the building blocks or first class entities [Didonet Del Fabro+06]. A model conforms to a metamodel while a metamodel conforms to a metametamodel. MDE is mainly concerned with the evolution of models as a way of developing software by focusing on models. With this new paradigm of software development, the code will be generated automatically by model to code transformations. MDD is copyrighted term by Object Management Group (OMG).

2.1.1 Model Transformation

One of the most important operations in MDE/MDD is model transformation. There are different kinds of model transformations including model to code, model to text, and model to model. Our interest is in model to model transformations. Figure 2.1 shows the process of model transformation. Since every artifact in MDE is a model, the model transformation is also a model that conforms to a metamodel. The transformation model defines how to generate models that conform to a particular metamodel from models that conform to another metamodel or the same metamodel. In Figure 2.1, the transformation model M_t transforms M_a to M_b. M_t conforms to MM_t while M_a and M_b conform to MM_a and MM_b respectively. The three metamodels conform to a common metametamodel MMM.

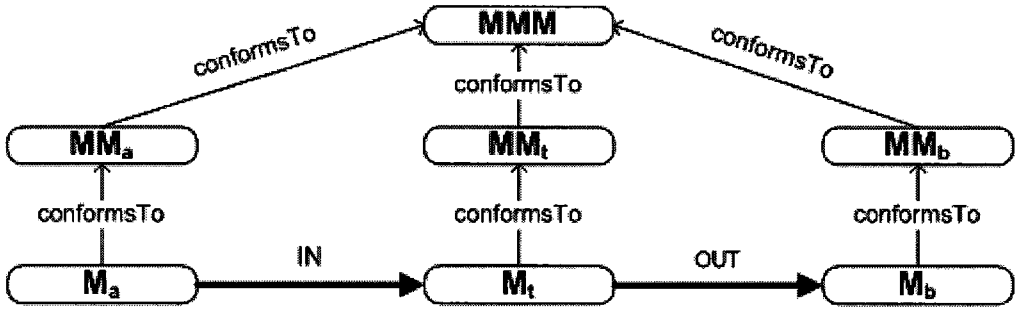


Figure 2.1: Overview of Model Transformation Adopted from [ATLUserGuide].

2.1.2 Model Driven Architecture (MDA)

OMG's Model-Driven Architecture (MDA) is a term copyrighted by OMG that describes an MDE approach supported by OMG standards; namely, UML, Meta-Object Facility (MOF), MOF-Query/View/Transformation (QVT), XML Metadata Interchange (XMI) and Common Warehouse Metamodel (CWM) [SYSC 5708]. MDA decouples the business and application logic from the underlying platform technology through the use of the Platform Independent Model (PIM), Platform Specific Model (PSM) and model transformations [SYSC 5708]. Figure 2.2 shows the relationship between these models.

Figure 2.2: MDA Models

The PIM describes a software system independently of the platform that supports it while PSM expresses how the core application functionality is realized on a specific platform. Given a specific platform, the PIM is transformed to PSM. Platform in this case refers to technological and engineering details that are independent of the core functionality of the application; For example, middleware (e.g., CORBA), operating system (e.g., Linux), hardware, etc [SYSC 5708].

2.2 Aspect-Oriented (AO) Techniques/Technologies

The size of modern software systems has increased tremendously. Software architects and developers have to design and develop systems that are not only enormous, but are more complex, and greatly distributed. These systems naturally have many cross-cutting concerns (requirements) whose solutions tend to cross-cut the base architecture and system behavior. Such concerns include security, persistence, system logging, new features in software product lines, and many others.

Aspect-Oriented techniques allow software developers and architects to reason, express, conceptualize, and work with multiple concerns separately [Groher+07] [Kienzle+09][Petriu+07]. These techniques allow us to modularize concerns that we cannot easily modularize with current Object-Oriented (OO) techniques [Whittle+07]. The final system is then produced by weaving or composing solutions from separate concerns [Petriu+07]. Klein et al. point out that dividing and conquering these cross-cutting concerns also allows us to better maintain and evolve software systems [Klein+07].

2.2.1 Aspect-Oriented Programming (AOP)

AOP applies AO techniques at code level [France+04][Kiczales+05][Petriu+07]. AOP was introduced by Gregor Kiczales [Kiczales+97] to enhance Object-Oriented Programming to better handle cross-cutting concerns that cause code scattering and tangling, which leads to code that is very difficult to maintain and impossible to reuse or modify. AOP addresses these issues by introducing a class like programming construct called an *aspect* which is used to encapsulate cross-cutting concerns. Just like a class, an aspect has attributes (state) and methods (behavior). An aspect also introduces concepts well known to AO community; namely, join points, advice, and pointcut. Join points are points in the code where the cross-cutting behavior is to be inserted. AspectJ (a popular AOP Java tool) supports join points for method invocations, initializing of attributes, exception handling, etc [Colyer+05, 38].

A *pointcut* is used to describe a condition that matches join points, that is, it is a way of defining join points of interest where we want to insert the cross-cutting functionality. This cross-cutting behavior to be inserted at a join point is defined in the *advice*. AspectJ supports *before*, *after*, and *around* advices [Colyer+05, 46].

2.2.2 Aspect-Oriented Software Development (AOSD)

AOSD applies AO techniques to the software development process. The final system is then produced by weaving or composing solutions for the separate concerns. Groher and Voelter believe that AOSD techniques improve re-usability, modularity and the ease of evolving a software system.

2.2.3 Aspect-Oriented Modeling (AOM)

Recent work in AO has focused on applying AO techniques much earlier in the development process [Kienzle+09][Klein+07][Morin+08][Petriu+07]. In AOM, Aspect-Oriented techniques are applied to models (unlike AOP). Whittle et al. define an AO model as "a model that cross-cuts other models *at the same level of abstraction*"[Whittle+06]. Aspects are considered models as well; hence, it makes sense to define (or abstract) other concepts such as pointcuts and advices as models. However, the precise definition of a joint point, pointcut or advice depends on our modeling view. For example, in a structural view, such as a class diagram, an aspect is defined in terms of classes and operations/methods whereas in a behavioral view, such as a sequence diagram, an aspect is defined in terms of messages and lifelines.

This has resulted in several approaches to AOM most of which have focused on separation and weaving (composition) of structural (class diagrams) and behavioral views (sequence, activity, and state diagrams) [France+04][Kienzle+09][Klein+07][Petriu+07] [Whittle+07][Woodside+09].

Earlier work by France et al. proposed an AOM approach that has a base/primary model, generic (reusable) aspect models, binding rules, and composition directives [France+04]. This approach focused on representing aspect models as UML class dia-

grams and UML 1.x collaboration diagrams (communication diagrams in UML 2.x). The primary model describes the base architecture of the system been modeled (without cross-cutting concerns). Generic aspect models describe the cross-cutting concerns. They are defined with templates parameters using a notation adapted from UML templates [France+04]. The main idea is that a generic aspect is first instantiated by been bound (using the binding rules) to the application context to create a context specific aspect model. The context specific aspect model is then composed with the primary model with the help of the composition directives to produce a complete system model [France+04]. If there are several context specific aspects models, the composition directives, amongst other things, should specify the order in which the aspects will be composed because there could be interference and dependencies between the aspect models [Kienzle+09] [Whittle+07].

Kienzle et al. take the approach by France et al. further, and propose a Reusable Aspect Model (RAM) approach where an aspect is defined as functionality or any concern that can be reused [Kienzle+09]. RAM combines the structural view (class diagram) and the behavioral views (sequence and state diagrams) of an aspect into one package-able model [Kienzle+09]. In both views, the aspect consists of a pointcut and advice whose definition depends on the view of interest. For example, in a state diagram a pointcut defines transitions and states that must be present in the primary model state diagram while the advice consists of a state diagram that will be composed at the join point matched by the pointcut [Kienzle+09]. However, in the sequence diagram view the pointcut is defined in terms of a sequence diagram with lifelines and a sequence of messages between the lifelines. Similarly the advice is also defined as a sequence diagram that will be weaved at the join points. This definition of a pointcut and advice in sequence diagrams is also used by [Klein+07] and it is the approach we have adapted in our work.

In RAM, the classes, operations, states, messages and lifelines that are not complete are known as mandatory instantiation parameters and are identified by a prefix character "|" to the names [Kienzle+09] similar to template parameters in [France+04]. These reusable aspects are first instantiated using specified bindings, and then weaved with

base/primary model. Class diagrams are composed using Kompose [Fleurey+07] [Jeanneret+08] while the state and sequence diagrams are weaved using GeKo [Morin+08].

Whittle et al. believe that the above approaches are not scalable because for a system with many aspects and join points, instantiating each aspect (generic or reusable) can be tedious [Whittle+06]. They represent and weave aspects using graph transformations where a graph transformation is a rule consisting of a left-hand side (LHS) or pattern graph, and a right-hand side (RHS) or replacement graph [Whittle+06]. They argued that any UML diagram can be described as a graph, therefore, transformations of UML models can be represented as a graph with model elements as nodes and associations (relationships) between the elements interpreted as links [Whittle+06].

2.2.4 Aspect Composition in AOM

Several approaches have been proposed for composing aspects in AOM. These include using graph transformations [Gong08][Whittle+07], semantics [Klein+06], executable class diagrams (ECDs) [Barais+08], weaving models [Didonet Del Fabro+06], generic approaches and frameworks [Fleurey+07][Morin+08], etc. Composition primarily involves deciding what has to composed, where to compose, and how to compose [Jeanneret+08]. Aspect composition can either be symmetric or asymmetric. In symmetric composition, there is a clear distinction between the models to be composed; that is, one model plays the role of a base model while the other is declared an aspect model [Jeanneret+08]. That distinction is absent in asymmetric composition.

Xweave supports asymmetric composition of both models and metamodels [Groher+07]. This weaver can only handle metamodels that conform to Ecore (a metametamodel for the Eclipse Modeling Framework) or instances of those metamodels since Xweave is based on Ecore [Groher+07]. Pointcuts can be specified in two ways; namely, signature matching, and explicit pointcut expressions. Name or signature matching involves comparing the name and type of a base model element with that of the aspect model before deciding to compose. Pointcut expressions are defined in openArchitectureWare [Oaw], an OCL based language. Currently Xweave cannot remove or override ex-

isting base model elements but can only add new elements.

Recent work by Whittle et al. in [Whittle+07] introduces and describes an AOM tool called MATA (Modeling Aspects Using a Transformation Approach) that uses graph transformations to define and compose aspects [Whittle+07]. MATA is built on top of Rational Software Modeler, and relies on a graph execution tool [Whittle+07]. Pointcuts are defined as a LHS graph of the graph rule while the RHS describes the new model elements to be inserted or removed [Whittle+07]. There is no distinct notion of a join point unlike in other approaches. MATA considers composition an instance of model transformation. In MATA, a rule is given in one diagram that is decorated with a profile that defines three stereotypes; namely, <<create>>, <<delete>> and <<context>>. The <<create>> stereotype identifies the new elements to be created by the rule while <<delete>> is used to mark elements that will be removed by the rule. The <<context>> is used to identify elements that should not be affected by the other two stereotypes. This way MATA packages "what", "where", and "how" to compose into a single module (rule). Jeanneret et al. point out that although this may be convenient, it is less flexible and difficult to maintain [Jeanneret+08]. MATA is able to compose both structural (class diagrams) and behavioral views (state and sequence diagrams). The use of graphs in MATA means that this approach can easily detect conflicts between interacting aspects (graphs) using the Critical Pair Analysis (CPA) technique [Whittle+07].

Kompose is a model composition tool implemented in Kermata [Kompose]. Like other symmetric approaches, Kompose does not offer mechanisms to make aspects reusable. It achieves composition in two steps: signature matching and merging [Fleurey+07]. Matching is modeling language specific whereas merging is language independent [Fleurey+07]. Kompose matches (comparing signatures) elements from the input models been processed. Matched elements are merged to create the composed model [Jeanneret+08]. The process is iterative and recursive. Kompose has pre-merge and post-merge directives to tweak the model before and after composition, and also to help resolve conflicts that might arise during composition [Fleurey+07][Jeanneret+08].

The Atlas Model Weaver (AMW) is used to manually establish links between models to create a weaving model that conforms to a weaving metamodel [Didonet Del Fabro+06][Jeanneret+08]. The weaving model is then used to create an ATL transformation that will merge the input models. The weaving model describes how to compose, whereas the ATL transformation defines where to compose [Jeanneret+08]. The exact location where elements are inserted is unclear but seem to depend on the ATL transformation [Jeanneret+08].

2.3 Atlas Transformation Language

The Atlas Transformation Language (ATL) is a model transformation language from the ATLAS INRIA & LINA research group [ATLUserManual]. The language is both declarative and imperative, and allows developers to transform a set of input models to a number of output target models [ATLUserManual]. In ATL, source or input models can only be navigated but cannot be modified [Jouault+05] whereas target models are write-only and cannot be navigated [Jouault+05]. Figure 2.3 below shows an overview of an example of an ATL transformation (Family2Person) from [ATL_Examples] that transforms a Family model to a Person model.

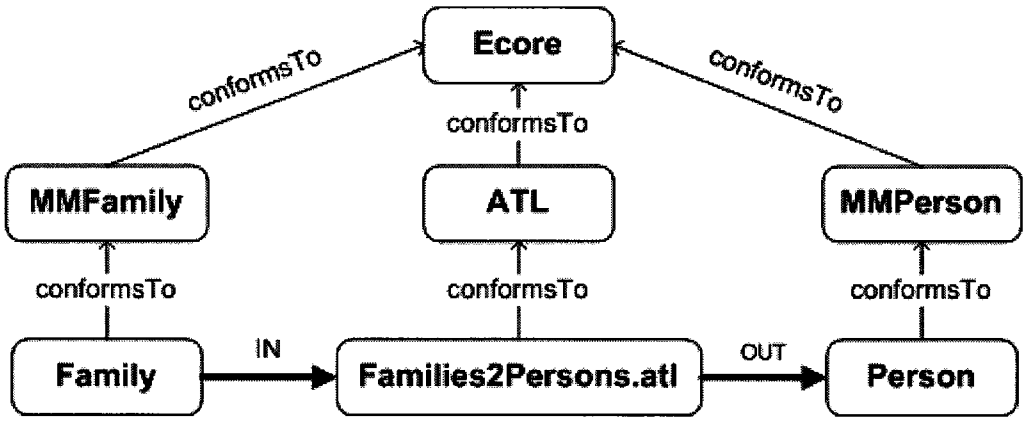


Figure 2.3: Overview of the Family to Person ATL Transformation.

The Family model conforms to a *MMFamily* metamodel whereas the Person model conforms to a *MMPerson* metamodel. The ATL, *MMFamily*, and *MMPerson* metamodels

all conform to the Ecore metametamodel which is a metamodel for the Eclipse Modeling Framework. Families2Persons.atl is an ATL model or program that transforms a Family model to a Person model.

ATL has three types of units that are defined on separate files [ATLUserGuide]; namely, ATL modules, queries and libraries. ATL has types and expressions that are based on the Object Constraint Language (OCL) from OMG [ATL_Manual]. ATL has primitive types (Numeric, String, Boolean), collection type (sets, sequences and bags) and other types, all of which are sub-types of the *OCLAny* abstract super-type. An ATL module or program, like Families2Persons.atl in the previous example, defines a model to model transformation [Jouault+05]. It consists of a header, helpers (attribute and operation helpers) and transformation rules (matched, called and lazy rules) [ATL_Manual] [Jouault+05]. The header defines the module's name, and the input and target models. ATL operation helpers are more like functions or Java methods, and can be invoked from rules and other helpers. Attribute helpers unlike operation helpers do not take any arguments. All helpers are, however, recursive and must have a return value.

Rules define how input models are transformed to target models. They are the core construct in ATL [Jouault+05]. ATL supports both declarative and imperative rules. Declarative rules include matched rules and lazy rules. Lazy rules are similar to matched rules but can only be invoked from other rules. A matched rule consists of source pattern and target pattern [Jouault+05]. A source pattern is defined as an OCL expression and defines what type of input elements will be matched [ATLUserGuide]. The target pattern defines the type of target elements to be generated, and how their attributes or features will be initialized [ATL_Manual]. The code snippet on the next page is an example of a simple matched rule. The rule is named *Classes* as shown on line 1. Line 2 shows the rule's source pattern while line 3 defines the rule's target pattern. Imperative rules are referred to as called rules. They have arguments just like procedures [Jouault+05]. The use of called rules is not encouraged and should only be explored if declarative rules cannot be used. This is because called rules can lead to performance degradation of the ATL engine [ATL_Manual][Jouault+05].

```
1.  rule Classes {
2.      from s : UML2!Class ( s.oclIsTypeOf(UML2!Class) )
3.      to t : UML2!Class  (
4.         name <- thisModule.getBinding(s.name),
5.         ownedOperation <- s.ownedOperation,
6.         visibility <- s.visibility,
7.         isAbstract <- s.isAbstract,
8.         ownedAttribute <- s.ownedAttribute
9.     )
10. }
```

An ATL model is compiled, and then executed on the ATL engine that has two model handlers; namely, EMF (Eclipse Modeling framework) and MDR (Meta Data repository) [ATL_Manual]. Model handlers provide a programming interface for developers to manipulate models [Jouault+05]. The EMF handler allows for manipulation of Ecore models while MDR allows the ATL engine to handle models that conform to the MOF 1.4 (Meta Object Facility) metametamodel [ATL_Manual]. For example, the ATL transformation in Figure 2.3 on page 12 would require an EMF model handler since the metamodels conform to Ecore.

ATL has an integrated development environment (IDE) that is built on top of Eclipse. We will briefly talk about Eclipse in the next section. The IDE has a source code editor, Console view, Error log view, Properties view, Navigator, Outline and Problems view. Most of these features are standard in Eclipse based IDEs. Figure 2.4 on the next page shows a screen shot of the ATL IDE showing some of the components of the IDE.

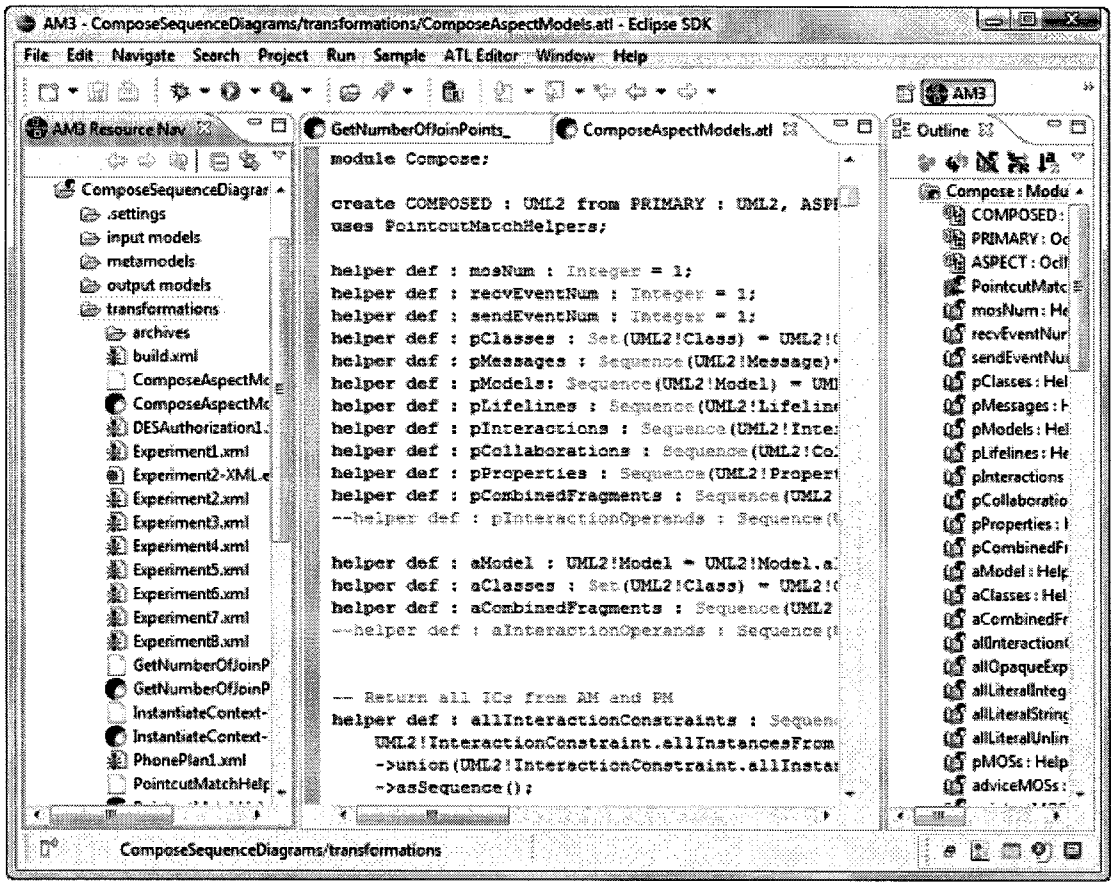


Figure 2.4: A screen shot of the ATL IDE.

2.4 Eclipse

Eclipse is an open source platform for software and tool development. Implemented in Java, it is probably best known as a Java IDE [EclipseProject]. However, Eclipse also supports development of PHP, C/C++, J2EE and other applications. Eclipse has web tools, modeling tools, source version control tools and many other tools. It has excellent support for the development of IDEs including the ATL IDE. Since Eclipse is written in Java, it is platform independent and can run on all operating systems that have Java support. It is highly extensible through the use of plug-ins like the UML2 plug-in. Plug-ins are, of course, written in Java using Eclipse's plug-in Development Environment (PDE).

Chapter 3 : Our Approach to AOM Composition

This chapter introduces our approach to AOM composition of UML interactions in the form of Sequence Diagrams (SDs). We begin by describing our definition of a Primary Model, a Generic and Context Specific Aspect Model, a join point, a pointcut, and an advice. We also discuss the instantiation and binding of a generic aspect model to the context of the application. This will be achieved with the help of an example. We will also look at some of the useful metaclasses from the UML Metamodel defined in the UML superstructure specification [OMG09].

Our approach shown in the Figure 3.1 is an adaptation of the approach proposed by Petriu et al. in [Petriu+07].

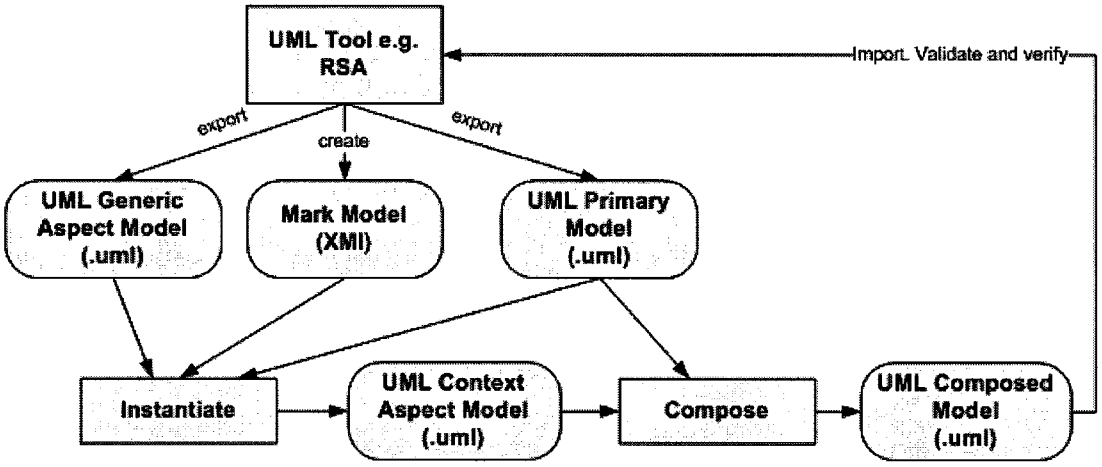


Figure 3.1: Our AOM Composition Approach.

Using a UML modeling tool, like RSA (Rational Software Architect) from IBM, the primary and generic aspect models are modeled in UML and then exported to a UML 2.1 (.uml) format file. The mark model is created in an XMI file. The *Instantiate* and *Compose* operations in Figure 3.1 are defined as ATL model transformations. We first instantiate a generic aspect model to the context of the application by using a model transformation that takes the primary, generic aspect and mark models as input, and transforms them to a context specific aspect model. We then invoke a second transformation that will take as input the newly created context specific aspect model and the primary model, and then

output a composed target model.

3.1 Example

Let us introduce a simple example to provide a better view of our approach and the definitions of the various concepts used in the approach. This example is adapted from Klein et al. [Klein+07]. It illustrates the weaving of a simple security aspect into a primary model. This is a basic example but more complex and interesting examples will be tackled in Chapter 5 when we test our implementation.

3.1.1 Primary Model

The primary model consists of a single scenario. In fact, our composition approach assumes that the primary model has only one SD and models a particular instance of a use case. This example consists of a login scenario shown in Figure 3.2 below.

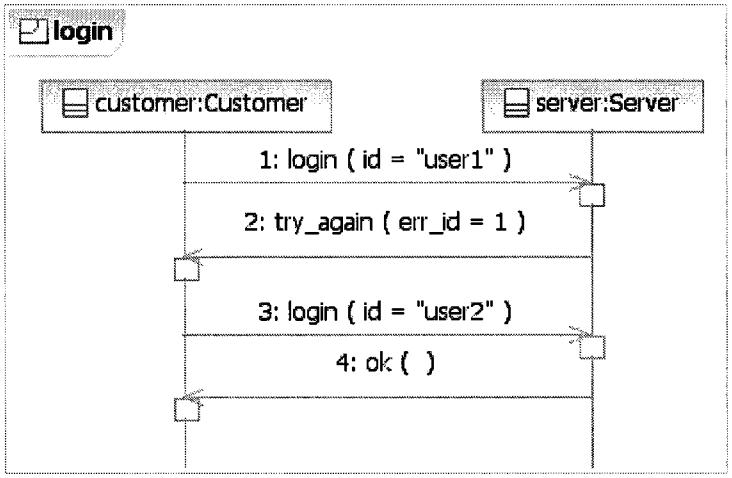


Figure 3.2: A Login Scenario for a Security Aspect Example (Primary Model).

The model shows a simple iteration between instances of Server and Customer. The Customer attempts to login into the Server by sending a *login* message. The Customer's login details are incorrect; hence, the Server sends a *try_again* message to the Customer to attempt another login. The Customer then sends a *login* message with the correct details this time and the Server responds with an *ok* message.

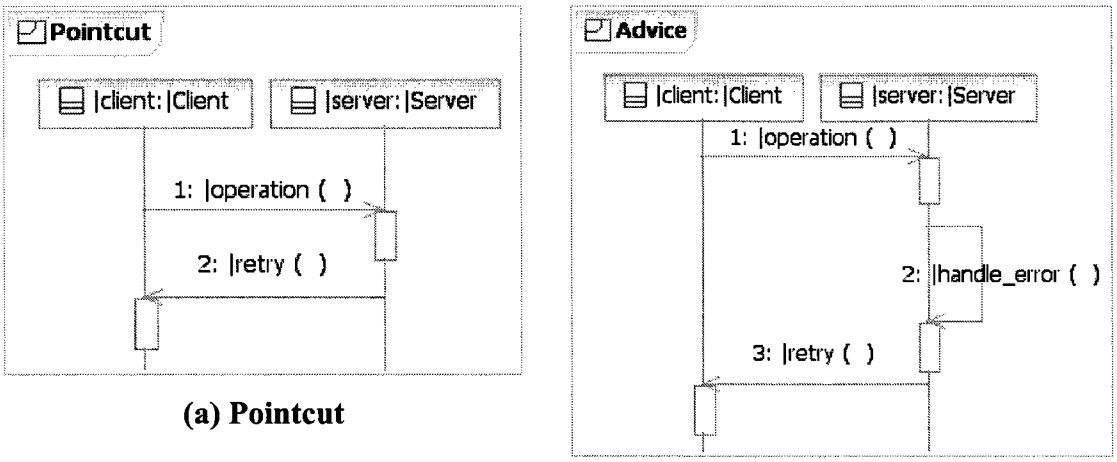
3.1.2 Generic Aspect Model

The primary model does not have any security features. We want to add some security mechanism to the scenario so that when a customer attempts to login and fails, the system should do something about that exception. We can model this security mechanism as a Security Aspect model that will detect a presence of a message from the Customer to the Server, and a reply from the Server back to the Customer. The presence of this sequence of messages or the pattern to detect is defined in the aspect's pointcut. The "something" that we want done, that is, the new behavior we want to add to the primary model to enhance security, is defined in the aspect's advice. However, to make the aspect reusable and more useful, it has to be generic but not specific to our example or situation. This way we can reuse the aspect and in different situations and scenarios.

To create a generic aspect we adopt the use of template parameters used by France et al. [France+04] and others [Kienzle+09][Petriu+07] to define generic roles played by the participants and messages in the generic aspect model. These generic roles are then bound to specific roles (names) when the generic aspect is instantiated. Figure 3.3 shows the pointcut and advice that make up our generic security aspect model. It should be noted that in case of multiple aspects, each aspect will be modeled separately. The lifelines (participants) and messages in the model are made generic. The pointcut in Figure 3.3a defines that the aspect detects any sequence of messages between a lifeline that plays the role of |client and lifeline that plays the role of |server such that |client sends a message tied to the role |operation and |server responds with |retry. During instantiation these template parameters (roles) will be set (bound) to concrete names of appropriate lifelines and messages.

As already mentioned, the advice represents the new or additional behavior we want executed if the pointcut matches, that is, if we find the sequence of messages defined in the pointcut in our primary model. An advice can also say how the existing behavior will be overwritten or removed. The advice in Figure 3.3b declares that we want |server to invoke a self call after receiving |operation and before sending |retry to |client. So our advice in this case adds new behavior (the |handle_error self call). The idea is that during

composition, as we shall see later, we replace whatever was matched by pointcut with what is defined in the advice.



(a) Pointcut

(b) Advice

Figure 3.3: Generic Aspect Model.

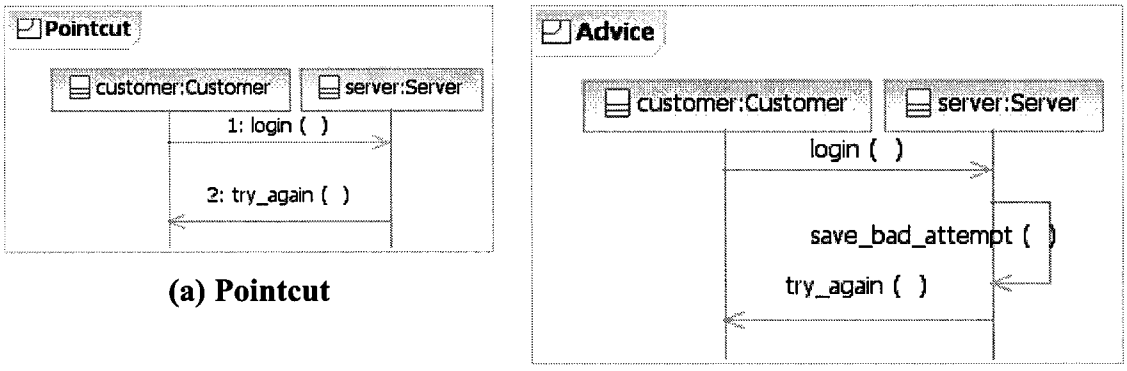
3.1.3 Context Specific Aspect Model

Before an aspect can be composed with the primary model, the generic aspect model must first be instantiated to the context of the application to produce a Context Specific Aspect Model. This is achieved by "binding" the template parameters to application specific values. For example, we want to bind "customer" to |client because in our primary model, customer plays the role of |client. In fact we can define the bindings (binding rules) as in Table 1 to instantiate our generic aspect model to our situation.

Instantiating our generic aspect model using the bindings in Table 1, we obtain the context specific aspect model shown in Figure 3.4. The pointcut from the context specific aspect will then match the sending of a *login* message from *customer* to *server* and a try_*again* message from *server* back to *customer*, which is what we want. Its advice declares that the *save_bad_attempt* self call will be added to the server hopefully for the server to do something useful and security related.

Parameter	Binding value	Comment
|client	customer	Lifeline object name.
|server	server	Lifeline object name.
|Client	Customer	The name of the type for the lifeline object.
|Server	Server	The name of the type for the lifeline object.
|operation	login	
|reply	try_again	
|handle_error	save_bad_attempt	

Table 1: Example of Security Aspect Binding Rules



(a) Pointcut

(b) Advice

Figure 3.4: Context Specific Aspect Model.

3.1.4 Binding and Mark Model

In our AOM composition approach, the instantiation of the generic aspect model is achieved through the use of ATL transformations as mentioned earlier. ATL manipulates models only. Therefore, we have to abstract the binding rules as a model. One way of achieving this is by using mark models which can be considered part of the OMG's MDA standard [Happe+09]. Happe et al. use mark models to parameterize their transformations. We use mark models to capture binding rules. We will discuss how we achieved this in next chapter.

3.1.5 Model Composition

After instantiating a context specific aspect model, a complete integrated system is obtained by composing the primary model with the context specific aspect model. We view composition as a form of model transformation as shown in Figure 3.5. Therefore, our aim is to transform the input models (Primary and Context Specific Aspect) to a target composed model.

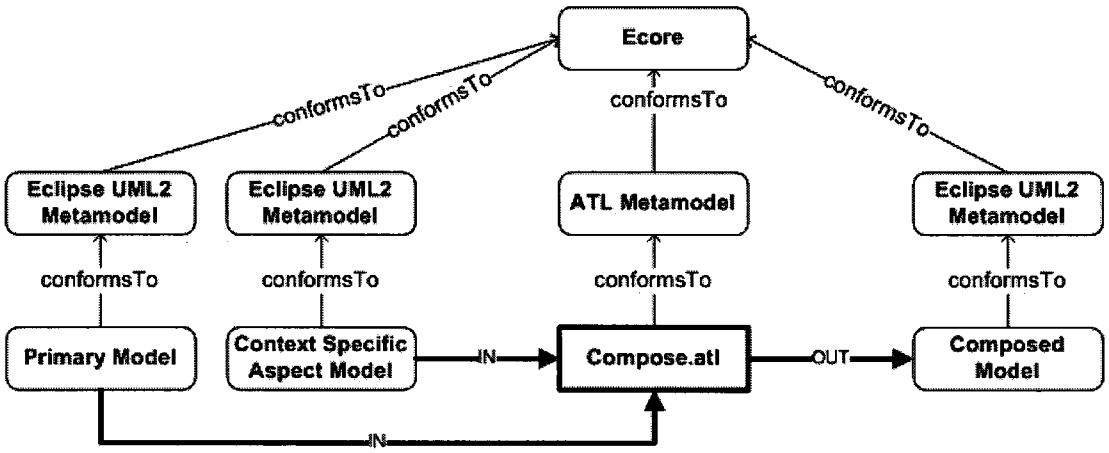


Figure 3.5: Aspect Composition as an ATL model transformation.

As shown in Figure 3.5, both the input and output models conform to an EMF implementation of the UML metamodel specification while our ATL program or model conforms to the ATL metamodel. All the metamodels conform to the EMF's Ecore metametamodel.

As in other aspect composition approaches [Barais+08][Kienzle+09][Klein+07][Morin+08][Petriu+07][Straw+04][Whittle+07], composition has to be performed on different views, that is, structural or behavioral views. Our main focus is on the behavioral view. Composition inevitably results in some model elements been replaced, removed, added or merged [Morin+08][Gong08]. Similarly in our approach, all model elements from the context specific aspect model that are not already in the primary model, will be added to the composed model but elements common to both models will be merged. All join point model elements (from primary model) are replaced by advice elements. The rest of the elements from the primary model will be added to the composed model.

3.1.6 Behavior Composition

The primary and aspect models are behavioral views of our system and cross-cutting concerns modeled as Sequence diagrams (SDs). In UML version 2.x, SDs and Communication diagrams are known as Interaction diagrams. Both diagrams can be used to model behavior, however, our interest is in SDs. Composing these SDs is viewed as a model transformation. A pattern of model elements from the primary model that matches the pointcut SD from the context specific aspect model is replaced by the advice SD from the same aspect model. This is achieved by implementing a model transformation in ATL that takes the primary and context specific aspect model as input, and produces a composed model as output as described earlier. The details of the implementation are described in next chapter.

3.2 Interactions Metamodel

A formal definition of our models and the proposed algorithm (for matching and composing) are based on UML metamodel classes. The specification for the UML metamodel [OMG09] is enormous and also includes metaclasses for other UML diagrams that we are not interested in. Therefore, it makes sense to look only at some of the important classes whose objects are used in creating SDs.

SDs, just like Interaction Overview Diagrams, Communication Diagrams and Timing Diagrams can be used to display interactions between objects [OMG09]. SDs are used to show a sequence of message exchanges or interactions among objects (so called participants or lifelines) in a system [Pilone+05][OMG09]. A sequence diagram shows the order in which messages are exchanged among participants; hence, order is crucial [Hamilton+06][Pilone+05]. The messages or interactions, to be precise, on a specific lifeline are totally ordered but interactions between two lifelines are partially ordered. The most important model elements in a SD are probably lifelines (participants), messages, message ends, and the enclosing interaction. Figure 3.6 is a simplified class diagram of the Interactions Metamodel showing the relationships among the metaclasses for these model elements.

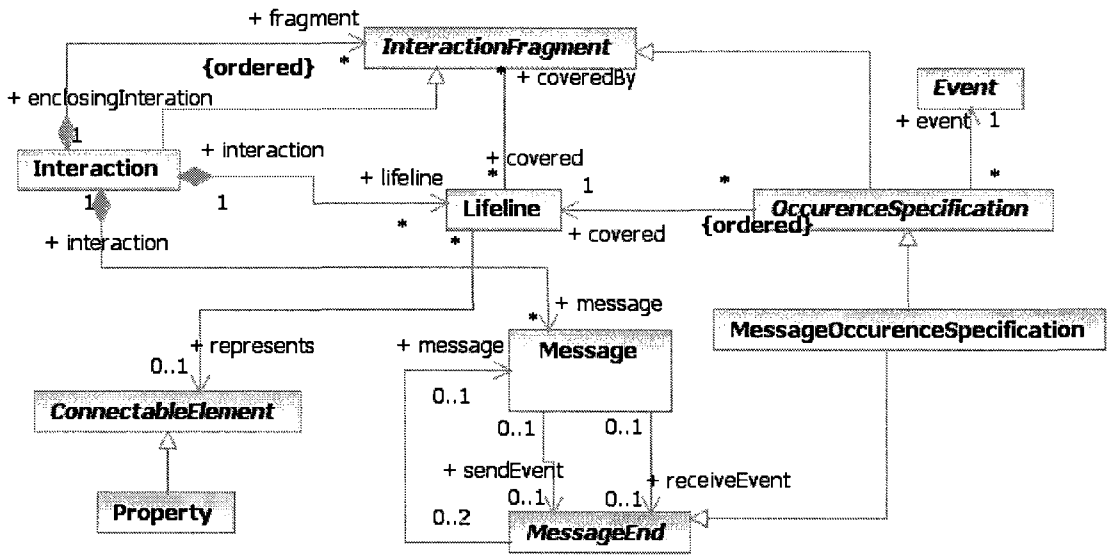


Figure 3.6: Simplified Metamodel for Sequence Diagrams.

A complete description of each metaclass can be obtained from the UML specification [OMG09]. The *InteractionFragment* abstract class represents a general concept of an interaction [OMG09]. An *Interaction* is a sub class of *InteractionFragment* that represents the modeled behavior or interactions (exchange of messages) between participants (lifelines)[OMG09]. An *Interaction* essentially encloses *Messages*, *Lifelines* and other *InteractionFragments*. The enclosed *InteractionFragments* are stored in an ordered list referenced by the *fragment* role. This ordering is exploited in our algorithms for matching and composing SDs. A *Message* models the kind of communication between participants [OMG09]. There are five main types of messages; namely, synchronous, asynchronous, delete, create, and reply messages [Hamilton+06]. Each message is accompanied by a pair of MessageOccurrenceSpecifications (MOSs). The *sendEvent* MOS represents the sending of the message while *receiveEvent* MOS models the reception of the message. Each MOS also has a reference to the lifeline for which the message is received or sent from through the *covered* association. In return, each *Lifeline* has a reference to a list of *InteractionFragments* or specializations of *InteractionFragment* (including MOSs), which cover the lifeline, through the *coveredBy* association. This list is ordered which means that if MOS1 (representing the sending of message1) is ahead of MOS2 (on the lifeline's *coveredBy* list), representing the reception of message2, then message1 is sent

before message2 is received by the lifeline. Each Lifeline has a *Property* (a *ConnectableElement*) that defines the participant (name and type) been represented by the lifeline. Many MOSs can have a reference to an event that is related to the message. Figure 3.7 shows *Event* classes and their hierarchy.

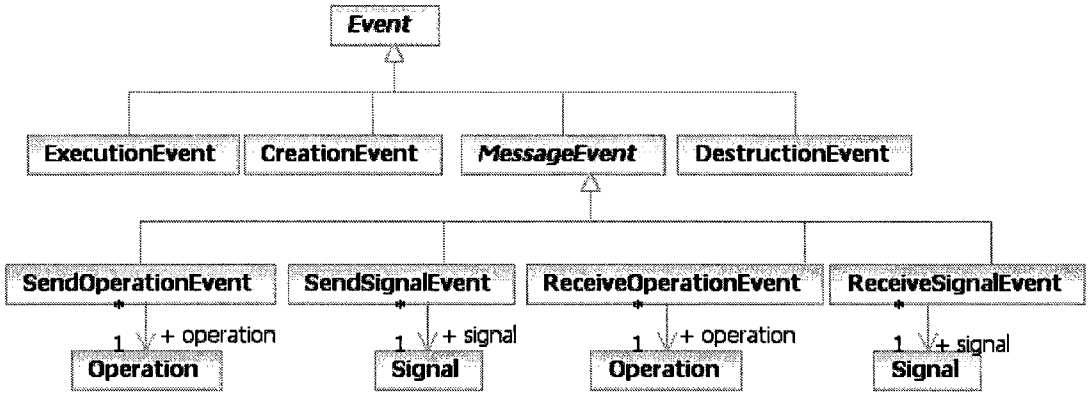


Figure 3.7: Events Classes Hierarchy.

The events that we are interested in are specializations of the *MessageEvent* abstract class mainly the *SendOperationEvent* (SOE) and *ReceiveOperationEvent* (ROE) classes. These types of events occur during the sending or receiving of a request for an operation invocation [OMG09].

Another specialization of *InteractionFragment* is the *CombinedFragment* (CF) class which was introduced in UML to help manage complex SDs [Hamilton+06]. *CombinedFragments* are essentially containers for interactions with operators [Pilone+05]. Figure 3.8 shows the relationship between the CF class and other metaclasses. There are several kinds of CFs including alt (if-else like) and loop (for-loop like). *CombinedFragments* own *InteractionOperands*. The *InteractionOperands* in return own other fragments (MOSs and *CombinedFragments*) in an ordered list. Each *InteractionOperand* has an optional guard condition (an *InteractionContraint*) which determines whether a combined fragment can be invoked [Pilone+05].

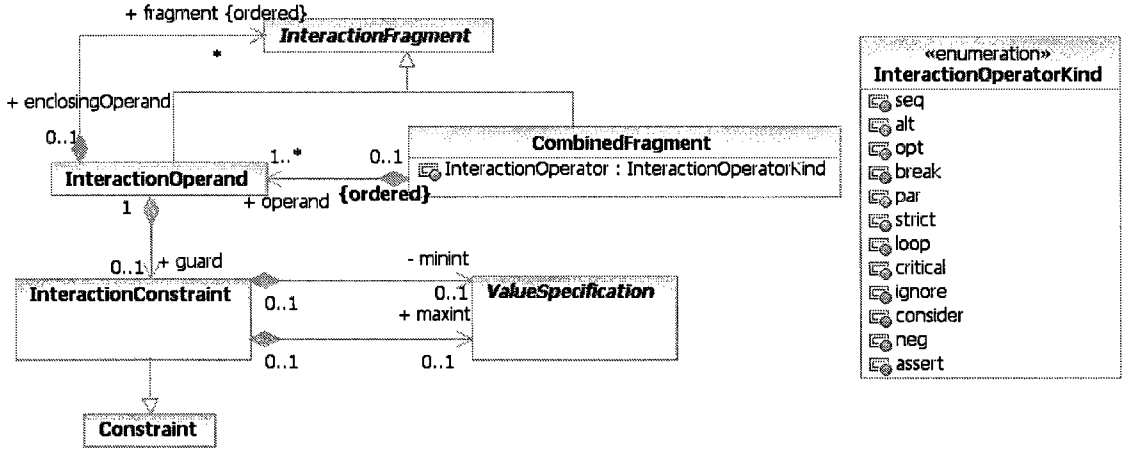


Figure 3.8: CombinedFragments and Related Metaclasses.

3.3 Sequence Diagram (SD) Composition

3.3.1 Composition Overview

As previously described, our AOM approach has a primary model, one or more generic aspect models and a mark model. The primary model describes the core system functionality (behavior) without cross-cutting concerns. The generic aspect models describes (encapsulate) cross-cutting concerns which could otherwise be scattered across our core functionality; for example, new features (in software product lines), security, persistence, etc. Before composing the primary model with an aspect model, we first instantiate the generic aspect model in the context of the application with the help of a mark model. We employ an ATL transformation model that takes the primary, generic aspect, and mark models as input, and produces a context specific aspect model as output. We would like to point out that the mark model does not necessarily have to specify all the bindings for the template parameters in cases where some of the bindings can be matched or implied from the primary model. A second ATL transformation model then takes as input the primary and context specific models to produce the composed model. Defining a generic aspect improves re-usability since the same aspect can be instantiated and then composed with the primary model multiple times until a complete integrated system model is obtained.

Since we are mainly interested in the behavioral view (of our primary and aspect models), our work is mainly focused on the composition of interactions diagrams in the form of SDs. As described earlier, the aspect model consists of a pointcut and an advice defined as SDs where the pointcut is the behavior to detect and the advice is the new behavior to compose or weave at the join points [Klein+07]. Before composing, we first have to identify all our join points by matching the pointcut SD with the primary model. The pointcut SD consists of message or a sequence of messages between lifelines; therefore, we want to find the occurrence of these sequences of messages in the primary model and then insert the defined cross-cutting behavior (defined in the advice SD) at every join point. Composition is essentially inserting this new behavior; that is, composition is achieved by replacing the join points with the advice SDs.

Before instantiating a generic aspect, we first have to ensure that the aspect can be applied to the primary model; that is, whether its pointcut matches. A formal definition of matching will be given later. Also during composition we have to find where to weave. This makes pointcut detection or finding join points a core operation. The algorithm designed for pointcut detection manipulates the SD metaclasses by exploiting the relationship between *InteractionFragments* and their ordered list of fragments in an interaction. It also makes use of the fact that a sequence of messages (and indeed a SD) is essentially a list of ordered fragments. Even though these fragments include *BehaviorExecutionSpecifications* and *ExecutionOccurrenceSpecifications*, we have realized that we can ignore these two types of fragments and still retain the essence of an SD. The pointcut detection algorithm finds join points for both instantiation and composition operations; therefore, aspect model here refers to either a generic or a context specific aspect model.

3.3.2 Useful Notation

Let us first introduce the following set like notation:

Let,

- **P** be a sequence of fragments, that is, *InteractionFragments* (CombinedFragments and MOSs), from the aspect's pointcut SD.

- **A** be a sequence of fragments from the aspect's Advice SD.
- **C** be a sequence of fragments from the primary model SD.

Note that a sequence is an ordered collection/list.

Then, **P** = Sequence$\{f_1, ..., f_\psi\}$ where ψ = number of fragments in **P** and f_i is either a *CombinedFragment* (CF) or a MessageOccurrenceSpecification (MOS), such that,

f_i =	If instance of	where
CF(O, Λ)	CF	**O** is a sequence of operands in the **CF** and each operand is also a sequence of fragments just like **P**. This is the case with nested CFs. **Λ** is a list of lifelines covered by the **CF**.
MOS(L_i,E_i,M_i)	MOS	L_i is a lifeline covered by f_i. E_i is an event associated with f_i. M_i is a message associated with f_i.

and,

C = Sequence$\{c_1, ..., c_\mu\}$ where μ = number of fragments in **C** and c_i is also either a CF or a MOS, such that,

c_i =	If instance of	where
CF(O, Λ)	CF	**O** is a sequence of operands in the **CF** and each operand is also a sequence of fragments just like **C**. This is the case with nested CFs. **Λ** is a set of lifelines covered by the **CF**.
MOS(L_i,E_i,M_i)	MOS	L_i is a lifeline covered by c_i, E_i is an event associated with c_i. M_i is a message associated with c_i.

3.3.3 Aspect and Primary Models Definition

Using the above notation, we will define an aspect model as a pair of fragment sequences, that is, **Aspect** = (**P**, **A**) where **P** and **A** are the sequences defined earlier. This definition is adapted from Klein et al. in [Klein+06] and [Klein+07]; However, Klein et al. define a simple SD as a tuple that consists of a set of lifelines, a set of events, a set of actions and partial ordering between the messages [Klein+07]. This is different from our definition of a sequence of fragments. Using our definition, the primary model = **C**, a sequence of fragments from the primary model SD.

Then our pointcut **P** matches **C** if and only if there exists two sub-sequences M_1 and M_2 in C such that,

$C = M_1 \oplus P \oplus M_2$,

where $\oplus$ denotes a union of sequences. $A \oplus B$ returns a sequence composed of all elements of A followed by the elements of B.

If the **P** matches **C** several times, say *n* times, then we can say,

$C = M_1 \oplus P \oplus M_2 \oplus \ldots \oplus M_n \oplus P \oplus M_{n+1}$.

This definition is an adaptation of the definition given by Klein et al. in [Klein+06].

3.3.4 Join Point Definition

Part of the sequence **C** that corresponds or matches **P** is the join point. In other words, a join point is a sub sequence of **C** that is equal to the sequence **P**. Equal here means that fragments at the same corresponding location in **P** and join point (same index on either sequences) are equal. For example, if elements at position 1 in **P** and in the join point are both MOSs, they can only be equal if and only if they cover similar lifelines (same name and type), have the same message, and have other features that are similar. More details for checking for equality will be given in the design and implementation chapter.

Since the size (number of fragments) of **P**, hence the size of a join point, is fixed we can afford to keep track of only fragments at the beginning of each join point. With this assumption we can define;

S = Sequence$\{s_1, \ldots, s_n\}$, a sequence of fragments at the beginning of each join point where $n > 1$ is number of join points matched by **P**.

A join point, J_i is then given by a sub sequence of **C** from index of s_i in **S** to the index of s_i plus ψ minus 1. That is, if;

x_i = **indexOf**(s_i) and

$y_i = x_i + \psi - 1$, where ψ = number of elements in **P**, then,

J_i = C->**subSequence**(x_i,y_i) for $1 \leq i \geq n$

3.3.5 Pointcut Detection Algorithm

The pseudo code for the algorithm that detects or matches pointcuts and returns **S** is given below. The algorithm begins by creating an empty sequence **S** on line 2. It then iterates over all fragments c_i in **C** checking if c_i is equal to f_1, the first element in our pointcut **P** on line 9. If the elements are **not** equal, the algorithm moves to the next c_i. However, if the fragments (c_i and f_1) are equal, it obtains $\mathbf{J_i}$, a sub sequence of **C** starting from c_i and with the same size as **P**, on line 10.

Algorithm-1 Pointcut Detection Algorithm

Input : **P** = Sequence$\{f_1, \ldots, f_\psi\}$, **C** = Sequence$\{c_1, \ldots, c_\mu\}$

where ψ = number of fragments in **P**, and μ = number of fragments in **C**

Output : **S** = Sequence$\{s_1, \ldots, s_n\}$

1. **begin**
2. S = Sequence{}
3. **foreach** c_i in C do

```
4.          //compute the location of the end of the potential join point
5.          k = i + ψ -1
6.          if k > μ then //make sure we have a valid location
7.                  break
8.          end if
9.          if c_i = f_l then
10.                 J_i = C->subSequence(i,k) // Potential join point
11.                 /* check if fragment at the same location in P is equal to the
12.                 corresponding element in the join point */
13.                 if pairWiseMatch(P, J_i) then
14.                         S->enqueue(c_i)
15.                 end if
16.         end if
17.       end loop
18.     return S
20. end
```

On line 13 the algorithm then compares **P** and $\mathbf{J_i}$, side-by-side by checking if each pair of fragments at index *j* on both sequences are equal for $1 \leq j \geq \psi$. If this is true, then indeed $\mathbf{J_i}$ is a join point. So the algorithm inserts the first element (c_i) of the join point into **S** and loops back to line 3. It continues looping until it has checked all the elements of **C** or the condition on line 6 is true to ensure we do not fall off the edge. More details on the implementation this algorithm and its functions, like *pairWiseMatch*, will be discussed in the next chapter.

Discussion of the algorithm-1 complexity:

If the algorithm-1 has to visit all fragments in **C** (when $\psi = 1$) then both functions on lines 10 and 13 will take constant time, that is, O(1) which makes the algorithm linear or O(n). If **P** is the same size as C ($\psi = \mu$), then the algorithm has to loop only once but both *subSequence* and *pairWiseMatch* functions are O(n); hence, the algorithm is again linear. However, if $\psi < \mu$ then again both functions (i.e., *Sequence* and *pairWiseMatch*) are, in the worst case, linear and the algorithm will have to loop several times each time invoking the two functions making the algorithm quadratic, that is $O(n^2)$; therefore, in general the algorithm is $O(n^2)$.

3.3.6 Assumptions and Requirements

The above algorithm and indeed the other algorithms to be introduced later, make the following assumptions:

- The input models are well formed and valid; hence, the sequences **S**, **P**, and **A** are valid. For example, we do not have empty sequences. We also assume that the aspect models (generic and context specific) consists of two interactions (SDs) named Advice and Pointcut, and that the primary model represents one interaction or scenario; therefore, consists of one instance of *Model*, one instance of *Collaboration*, and one instance of *Interaction*.
- We can correctly compare any two fragments regardless of their specialization, for example, comparing a MOS with a CF.
- Nested CFs have been properly and consistently unrolled.
- A lifeline's name is the same as that of the represented object (property).
- Message have arguments with primitive UML types (strings and integers).
- We can ignore other fragments like *BehaviorExecutionSpecifications* (BESs) and *ExecutionOccurrenceSpecifications* (EOSs) focusing only on MOSs and CFs (and their operands), and still achieve accurate pointcut detection.

3.3.7 Complete Composition Algorithm

After detecting the pointcut and obtaining our join points, the next step is to weave the advice at the join points. Since the advice has already been bound to the context of the application during aspect instantiation, weaving the advice is simplified to replacing a join point with the advice. This is trivial with only one join point. Challenges arise when we have multiple join points because we have only one advice from the aspect model. We can either duplicate the advice or work with one join point at a time. Both options were explored but duplicating the advice (without duplicating the aspect model) proved to be complex due to the inability to navigate target models in ATL, and the nested relationships between *InteractionFragments*. Focusing one join point at a time is easier and more elegant. The complete composition algorithm presented in this section achieves this. Let us first introduce abbreviations that we will use in the algorithm.

- **GAM** = Generic Aspect Model
- **CSAM** = Context Specific Aspect Model (Instantiated generic aspect model)
- **MM** = Mark Model
- **PM** = Primary Model
- **CM** = Composed Model

The pseudo code of the Complete Composition Algorithm is given on the next page. The three functions defined in this algorithm represent the ATL transformations used to implement this algorithm as we shall see in the next chapter. The algorithm begins by retrieving *n* the number of join points matched in the primary model (PM) using the *JoinPointsCount* function on line 2. This function implements algorithm-1 (Pointcut detection Algorithm) to return a sequence of fragments at the beginning of each join point, and then finds the size of that sequence. The details of the implementation of this function will be given in next chapter. The number of join points determines if the algorithm will execute lines 6 to 10, and not necessarily the number of times the loop will iterate. If $n > 0$, that is, we have at least one join point, the algorithm instantiates the GAM to create a CSAM

on line 6. This corresponds to the instantiate process shown in Figure 3.1 on page 16. It then composes PM with CSAM by weaving the advice at the first join point using the *Compose* function on line 8 to produce our composed model. This corresponds to the compose process described in section 3.1.5 on page 21 and shown in Figure 3.1.

Algorithm-2 Complete Composition Algorithm

Input :GAM, MM, PM

Output :CM

```
1.  begin
2.      n = JoinPointsCount(GAM, MM, PM)
3.      temp = PM
4.      while n > 0
5.          // instantiate our generic aspect model
6.          CSAM = Instantiate(GAM, MM, temp)
7.          // compose advice and first join point
8.          CM = Compose (temp,CSAM)
9.          temp = CM
10.         n = JoinPointsCount(GAM, MM,temp)
11.     end while
12.     return CM
13. end
```

The algorithm then checks the CM for more join points on line 10. If more are found, it returns to line 6 to instantiate the GAM using CM (new primary model). It then creates another CM and checks for more join points. The algorithm continues looping until no join points are found.

As it is, this algorithm has a potential nasty flaw in the form of positive feedback, which, if left unattended, can cause the algorithm to loop indefinitely in some cases! The problem is rooted on the fact that composing an aspect in most cases results in the addition of new model elements (fragments, messages and lifelines) which in turn can produce more join points. This means that after composition on line 8, the algorithm may find more join points on line 10 causing the algorithm to iterate again and again. For example, if the pointcut is defined as a single message MSG1, and the primary model has two invocations of this message, then we have two join points. If the advice adds three instances of the same message MSG1, then after composition (1^{st} iteration) we will have four join points. After the second iteration well have six, then eight, etc. With the number of join points increasing all the time the algorithm will never terminate.

This problem is easily solved by tagging model elements from advice during instantiation on line 6. To be precise we only have to tag MOSs (fragments). Then when pointcut matching during the invocation of *JoinPointsCount* (implementing algorithm-1), we check for that tagging. If a potential join point has at least one tagged fragment, then we know that this join point emerged only after composition; therefore, it is immediately disqualified.

Discussion of the algorithm-2 complexity:

The complexity of algorithm-2 is difficult to analyze because on the surface the algorithm appears to be linear on the number of join points. However, the algorithm is not necessarily linear on the number fragments. We have already seen that detecting the number of join points is quadratic. Therefore, if that is nested within a loop, we could say that (in general) the algorithm is cubic, that is, $O(n^3)$

3.3.8 Advice Composition Algorithm

At the core of the *Compose* function, used by the Complete Composition Algorithm described above, is the Advice Composition algorithm that weaves the advice at the join point. Recall the definition of an **Aspect** = (**P**, **A**). We will use definition again where by "**Aspect**" we are referring to a context specific aspect model. Our main interest is mainly

on the advice sequence **A**. Recall that,

- **A** = Sequence$\{a_1, ..., a_\omega\}$, a sequence of fragments from the aspect model advice SD, where ω = number of fragments in **A**.

Recall also that,

- **C** = Sequence$\{c_1, ..., c_\mu\}$, a sequence of fragments from the primary model, where μ = number of fragments in **C**.
- **S** = Sequence$\{s_1, ... ,s_n\}$, a sequence of fragments at the beginning of each join point, where $n > 1$ is number of join points matched by **P**.
- A join point, $\mathbf{J_i}$ is then given by a sub sequence of C from index of s_i in **S** to the index of $s_i + \psi - 1$; That is, If, x_i = indexOf(s_i) and $y_i = x_i + \psi - 1$, then, $\mathbf{J_i}$ = C->subSequence(x_i,y_i) for $0 \leq i \geq n$

Since our Complete Composition Algorithm is concerned with one join point at a time, our Advice Composition algorithm needs to work with only one join point; that is, the join point that begins with s_1 (the first element in **S**). Then, let $\mathbf{C_{CM}}$ be a sequence of fragments from the composed model. Recall again that;

- **P** is a sequence of fragments from the pointcut SD.

With the notation defined, we can now describe our Advice Composition algorithm. Its pseudo code is given on the next page. In a nut shell, the algorithm simply replaces the join point with the advice. The algorithm first checks if we have a join point. If so, it obtains the first element of **S**, on line 5. Using that element, the algorithm finds the location (x_1) at the beginning and at the end (y_1) of the join point, as shown on line 6 and 7. The algorithm then obtains a sub sequence of fragments from **C** (primary model) before the start of the join point, on line 12. Please note indexing for our sequence data structure starts at 1 instead of zero as in Java lists or arrays. On line 16, the algorithm returns a sequence of fragments after the last element of the join point to the end of **C**. The composed model is then given by $\mathbf{C_{CM}}$ = *sub_before* $\oplus$ **A** $\oplus$ *sub_after*, that is, the union of *sub_before*, **A** and *sub_after*.

Algorithm-3 Advice Composition

Input : **C, A, P, S -** where ψ = number of fragments in **P**

Output : $\mathbf{C_{CM}}$

```
1.  begin
2.      if S->isEmpty() then
3.          C_CM = {}
4.      else
5.          s_1 = S->first()        // fetch the tail of the 1st join point
6.          x_1 = C->indexOf(s_1)   // find its location in C
7.          y_1 = x_1 + ψ–1         // find the location of the join point's head
8.          sub_before = {}
9.          sub_after = {}
10.         if x_1 > 1 then
11.             // get all fragments before the join point
12.             sub_before = C->subSequence(1,x_1-1)
13.         end if
14.         if y_1 < μ then
15.             // get all fragments after the join point
16.             sub_after = C->subSequence(y_1+1, μ)
17.         end if
18.         //Insert the advice in place of the join point
19.         C_CM = Sequence {sub_before, A, sub_after}
20.     end if
```

21. **return C_{CM}**

22. **end**

Discussion of the algorithm-3 complexity:

Creating *sub_before* and *sub_after* is linear in the worse case. Creating C_{CM} is also O(n) in the worst case; hence, the above algorithm is clearly linear.

Chapter 4 : Design and Implementation

In the previous chapter, we introduced our definition of primary, aspect and mark models. We also introduced our approach to AOM composition, and discussed our Complete Composition Algorithm that uses two other algorithms to detect join points, and compose the primary and aspect models. In this chapter we describe how the Complete Composition Algorithm was implemented using ATL transformations to realize the functions *JoinPointsCount(...)*, *Instantiate(...)* and *Compose(...)* employed by the algorithm. These functions were implemented as ATL transformation models used to transform several input models to desired target models to achieve composition of SDs. Before giving implementation details of these transformation models, we would like to first justify some of our design decisions and also describe how we designed our mark model.

4.1 Design Decisions

Several key decisions were taken in this work. These include:

- The use of ATL transformation models for composition instead of, say, graph transformations or general programming languages (e.g., Java).
- The use of RSA 7.5 as a modeling tool of choice.
- The use of a mark model.
- Ignoring BehaviorExecutionSpecifications (BESs) and ExecutionOccurrenceSpecifications (EOSs) model elements during pointcut detection and composition.

4.1.1 Using ATL Transformation Models for Composition

Aspect composition or weaving can be considered a form of model transformation because we take at least two input (primary and aspect) models and produce at least one target model (composed). Therefore, model transformation approaches can be used for aspect composition. There are several techniques that can be used for transforming models

(model-to-model). Jézéquel in [Jézéquel06] classifies these techniques into five categories; namely, general purpose programming languages (e.g., Java), generic transformation tools (e.g., Graph transformations), CASE tools scripting languages (e.g., Fujaba), dedicated model transformation tools (e.g., ATL and QVT) and Meta-modeling tools like Kermata [Jézéquel06]. Graph transformations are powerful and well known but can be complex, have a steep learning, and might require a longer period of sustained effort [Jézéquel06] which is beyond the scope of this work. General purpose programming languages are easier to learn but not suitable for complex transformations [Jézéquel06]. Dedicated transformation languages, like ATL and OMG's QVT, provide for powerful expressions, and makes maintenance and development of model transformations easier since relationships between the models are encoded in rules [Jézéquel06]. ATL was chosen because it is mature and has a rich set of development tools that are built on top of flexible and versatile Eclipse platform. ATL is based on OCL; therefore, it is not difficult for a developer with some OCL experience to learn. ATL was also chosen because no work on behavioral aspect composition, that we are aware of, has been attempted using ATL.

4.1.2 Using RSA 7.5 as a Modeling tool of choice

Several open source modeling tools such as Eclipse's MDT-UML2Tools plug-in, Papyrus, and EclipseUML from Omondo on Eclipse 3.3 (Free edition) were tested for creating SDs. We will briefly look at why these tools fell short.

When we started our research, the Eclipse's MDT-UML2Tools plug-in did not have support for SDs so it was quickly rejected. However, the recently released version now supports SDs.

Papyrus (Version: 1.11.0) is not easy to use. For example, we cannot change message type attribute (e.g., from a synchronous to asynchronous) once the message has been placed in the drawing area. Also self calls are difficult to create, and once the application is restarted, messages get entangled.

EclipseUML on Eclipse 3.3 is better than Papyrus when it comes to drawing SDs but its Achilles heel is that the UML model is saved in a *.usd* file where the model representation is combined with the layout information.

RSA 7.5 is not free but we already have a license for it. It is a great UML modeling tool. It has excellent support for SDs. It is easy and intuitive to use. It allows for easy model migration. We can export or import UML models as .uml or XMI files. It allows for model validation (not available in some of the tools) which we found very useful. RSA can also generate a sequence diagram from an imported model. This makes verification of our composed model easy and less error prone.

4.1.3 Using Mark Models

ATL Transformations only work with models; therefore, our binding rules have to be in the form of a model that has a metamodel. Mark models are a convenient way to work with parameterized transformations. MDA, certainly, allows for use of mark models in model transformations [Happe+09]. Happe et al. use mark models to annotate performance models in their work on performance completions [Happe+09].

4.1.4 Ignoring BehaviorExecutionSpecifications and ExecutionOccurrenceSpecifications

As stated in the previous chapter, we are convinced that we can ignore BehaviorExecutionSpecifications (BESs) and ExecutionOccurrenceSpecifications (EOSs) (focusing only on MOSs and CFs) and still achieve accurate pointcut detection. This is because BESs and EOSs are used to define the duration of execution of behavior [OMG09] of say, a message invocation. Our work is focused on detecting the occurrence of a sequence of messages (interactions between participants) and doing something when we find the sequence. We are not concerned about how long the participant will execute for after a message invocation. During early stages of system modeling or at high levels of system abstraction, BESs and EOSs are not really applicable or useful; therefore, our decision to ignore them is reasonable.

4.2 Designing a Mark Model Metamodel

A mark model helps define binding rules for instantiating generic aspect models. These rules are merely template parameter and value pairs stored in mark model instances. In MDE, a model must have a metamodel that it conforms to, and our mark model is no exception. Since the mark model is to be used in ATL transformations (with an EMF model handler), its metamodel must conform to a metametamodel that is the same as the ATL metamodel, that is, Ecore as shown in Figure 4.2 on page 43 and Figure 4.4 on page 51. ATL development tools include KM3 (Kernel MetaMetaModel) which is textual notation for defining metamodels [ATL_Manual]. The code snippet below shows a KM3 definition of the metamodel for our mark model which we named *BindingDirectives*. The metamodel has one class with a *parameter* and *binding* value attributes of type *String*. This essentially means that the instances of the mark model will be a collection of objects with initialized parameter and binding attributes.

```
package BindingDirectives {
        class BindingDirective {
                attribute parameter : String;
                attribute binding : String;
        }
        package PrimitiveTypes {
                datatype String;
        }
}
```

The metamodel is defined in a *.km3* file which is then converted (injected) to an Ecore format encoded in XMI 2.0 using injectors in the ATL IDE [ATL_Manual]. Figure 4.1 shows the generated *.ecore* file viewed in Eclipse's Ecore Model Editor.

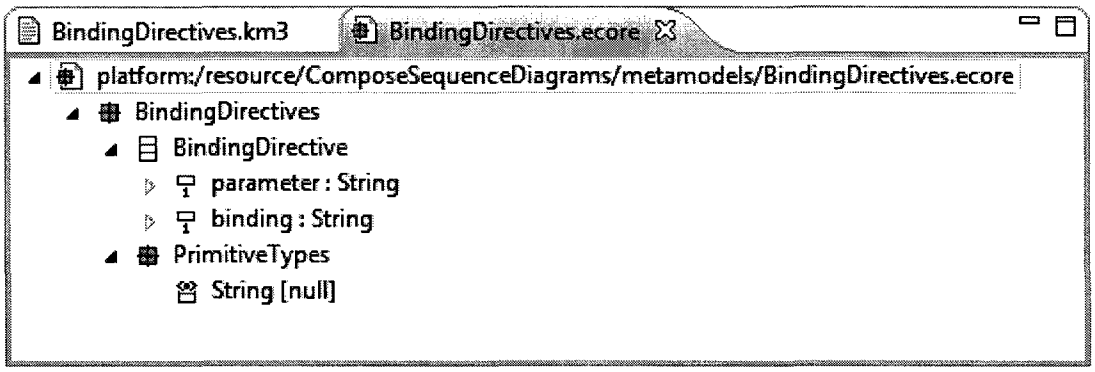


Figure 4.1: A Metamodel for Mark Model defined in Ecore.

Once the metamodel has been defined, we can begin creating mark models in an XMI format. The code snippet below shows a sample mark model we created for one of the test cases.

```
1.  <?xml version="1.0" encoding="ISO-8859-1"?>
2.  <xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
3.  xmlns="BindingDirectives">
4.   <BindingDirective parameter="|client" binding="customer"/>
5.    <BindingDirective parameter="|server" binding="server"/>
6.    <BindingDirective parameter="|Client" binding="Customer"/>
7.    <BindingDirective parameter="|Server" binding="Server"/>
8.    <BindingDirective parameter="|reply" binding="try_again"/>
9.    <BindingDirective parameter="|operation" binding="login"/>
10.   <BindingDirective parameter="|handle_error" binding="save_bad_attempt"/>
11. </xmi:XMI>
```

The XML header on line 1 provides the encoding and version information while the root XMI tag provides name space information [ATL_Manual]. The elements of the mark model are defined inside the XMI tag from lines 4 to 10.

4.3 Implementation of the Complete Composition Algorithm

As mentioned earlier, the Complete Composition Algorithm uses the *JoinPointsCount*, *Instantiate*, and *Compose* transformations to produce a composed model (SD). These transformations in return implement the other two algorithms (Pointcut detection and Advice Composition algorithm) to achieve their objectives.

4.3.1 Getting the Number of Join Points

The *JoinPointsCount* transformation is implemented by the ATL transformation model shown in Figure 4.2. It returns the number of join points found in the primary model given a pointcut defined in a generic aspect, and binding rules defined in a mark model.

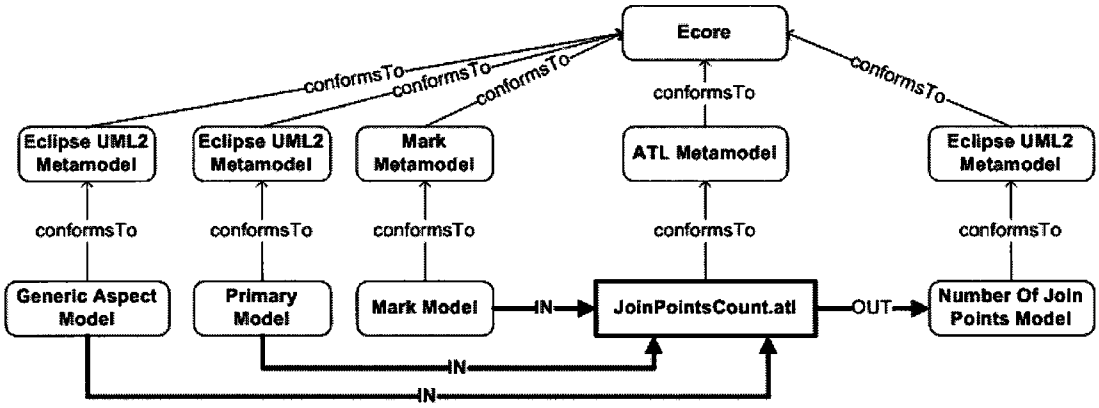


Figure 4.2: An Overview of the *JoinPointsCount* ATL Transformation.

The transformation produces a simple UML target model that contains the number of join points found. The number of join points must be returned in a model because an ATL transformation (module) has to produce a model but not a string or integer. The *JoinPointsCount* transformation is implemented in an ATL module creatively named **JoinPointsCount** as shown below by its header definition.

```
module JoinPointsCount;
create NUMJOINPOINT:UML2 from PRIMARY:UML2, ASPECT:UML2, BIND:BD;
uses PointcutMatchHelpers;
                                    ...
```

The header declares that the transformation takes as input two UML2 models (bound to variables PRIMARY and ASPECT), a model that conforms to the BD (Binding Directive) metamodel, that is, the mark model bound to the variable BIND. The transformation then produces a UML2 target model bound to the variable NUMJOINPOINT. The header also declares that the transformation uses the *PointcutMatchHelpers* ATL library. This is where common or general purpose helpers such as the ones used for pointcut detection, used also by other transformations, are defined. This helps reduce code duplication and allows for better code maintenance.

4.3.2 JoinPointsCount helpers

The transformation employs several helpers listed in Table 2 below. It also uses some of the helpers defined in the *PointcutMatchHelpers* library listed in Appendix E. Please note that aspect model here refers to the generic aspect model (not context specific) which is one of the input models to the transformation.

Helper Name	**Return type**	**Purpose**
messagesMatch (pMsg, cMsg)	Boolean	Returns true if message *pMsg* from the pointcut has the same name and type (when bound) as message *cMsg* from the primary model.
lifelinesMatch (pL, cL)	Boolean	Returns true if lifeline *pL* from the pointcut has the same name and represents the same object as lifeline *cL* from the primary model.
numJoinPoints	Integer	Returns the number of join points.
firstJPIndex	Integer	Returns the index of the first element from the first join point.
aClasses	Sequence	Returns all classes from the aspect model.
aCollaborations	Sequence	Returns all collaborations from the aspect model.

Helper Name	Return type	Purpose
aMessages	Sequence	Returns all messages from the aspect model.
aOperations	Sequence	Returns all class operations from the aspect model.
aLifelines	Sequence	Returns all lifelines from the aspect model.
aSendEvents	Sequence	Returns all SOEs from the aspect model.
aRecvEvents	Sequence	Returns all ROEs from the aspect model.
aProperties	Sequence	Returns all lifeline properties the aspect model.
allBindings	Sequence	Returns all binding rules from the mark model.

Table 2: JoinPointsCount Helpers

We will briefly look at a few core helpers used by our transformation to give some appreciation of ATL helpers. The *numJoinPoints* attribute helper is used to return the number of join points by finding the size of **S**, the sequence that has all the fragments at the beginning of each join point, as shown in the code snippet below.

```
                                    ...
helper def: numJoinPoints : Boolean =
    thisModule.joinPointsFragments()->size();
                                    ...
```

S is returned by the *joinPointsFragments()* operation helper that implements the Pointcut Detection Algorithm described in the previous chapter. This helper, which is defined in the *PointcutMatchHelpers* library, is shown on the next page. By leaving out the context in the helper's definition on line 2, this helper adopts the default context of the ATL module. The helper gets a sequence of fragments from the primary model (the sequence **C**) and a sequence of fragments from the aspect model's pointcut (the sequence **P**) by calling the *getPrimaryFragments()* and *getAspectFragments('Pointcut')* helpers respectively. The helper selects all fragments in **C** such that the first selected fragment is equal to the first

element in **P**, and the sub sequence from that fragment, with the same size as **P**, matches **P** pair wise. This matching is performed by the *PairwiseMatchFragments* on line 5. Checking fragments for equality is performed by the overridden *equals* helper with the appropriate context.

```
1.  -- Return a sequence of the Fragments at the start of the join points
2.  helper def:joinPointsFragments() : Sequence(UML2!InteractionFragment) =
3.  thisModule.getPrimaryFragments()->select(e |
4.     thisModule.getAspectFragments('Pointcut')->first().equals(e) and
5.        thisModule.PairwiseMatchFragments(
6.              thisModule.getAspectFragments('Pointcut'),
7.              thisModule.getPrimaryFragments()->subSequence(
8.              thisModule.getPrimaryFragments()->indexOf(e),
9               (thisModule.getPrimaryFragments()->indexOf(e)+
10.             thisModule.getAspectFragments('Pointcut')->size()-1)
11.                   )
12.             )
13.       );
```

The *getPrimaryFragments* operation helper has a very important role of returning the sequence **S**, fragments (CFs and MOSs) from the primary model, whether for pointcut detection above or for advice composition. The code snippet on the next page shows a definition of this helper. The helper begins by obtaining all the fragments owned (referenced via the *fragment* attribute) by the first (and only) interaction from the primary SD on lines 4 and 5. The *allInstancesFrom(metamodel: String)* operation is an ATL built-in operation that returns all the instances of a given context type that belong to a metamodel whose name is supplied as a string parameter. In this case, we want all instances of type UML *Interaction* from the source model bound to the variable 'PRIMARY' as defined in the module's header. The *getPrimaryFragments* helper then iterates and accumulates fragments that are either of type CombinedFragment (CF) or MessageOccurrenceSpecification (MOS) on lines 6 to 13 using ATL's iterative expression.

```
1.  -- Return all MOSs and CFs from the primary model.
2.  helper def : getPrimaryFragments() :
3.  Sequence (UML2!InteractionFragment) =
4.     UML2!Interaction.allInstancesFrom('PRIMARY')
5.         ->first().fragment->asSequence()
6.  ->iterate (e; frg:Sequence(UML2!InteractionFragment)= Sequence {} |
7.         if e.oclIsTypeOf(UML2!CombinedFragment) then
8.                frg.append(Sequence{e, e.getFragments()})
9.         else
10.           if e.oclIsTypeOf(UML2!MessageOccurrenceSpecification) then
11.                   frg.append(e)
12.                else
13.                   frg.append(Sequence{})
14.                                    ...
```

This expression consists of an iterator, an accumulator and a body [ATLUserGuide] as shown on line 6. The accumulator is a sequence of *InteractionFragments* that is initialized to an empty sequence. The body is the if-else expression. The helper iterates over the fragments from the primary SD paying attention to only CFs and MOSs. If the fragment is a CF then it has at least one operand that might contain MOSs or other nested CFs. Recall from Figure 3.8 on page 25 that a CF owns *InteractionOperands* which in turn own other *InteractionFragments* (including CFs and MOSs). Therefore, if the fragment that we are currently looking at is a CF, we need to unroll it recursively and return its contained fragments, and preserve the order of those fragments. This is achieved by adding, to the accumulator, a sequence containing the CF and its nested fragments obtained by calling the *getFragments()* helper on the CF on line 9. However, if the fragment we are currently iterating over is a MOS, then we just add it to the accumulator else it is ignored. As an illustration, given the SD in Figure 4.3 on the next page, we expect the helper (*getPrimaryFragments*) to return the sequence {MOS1, MOS2, CF1, MOS3, MOS4, CF2, MOS5, MOS6}. CF1 contains 2 MOSs (MOS3 and MOS4) and nested CF2 which

in turn contains 2 MOSs (MOS5 and MOS6). The helper adds MOS1 and MOS2 first, and then adds CF1 and its fragments as shown on lines 8 and 11 of the code snippet on the previous page. The *getFragments()* does the retrieving of the CF's fragments.

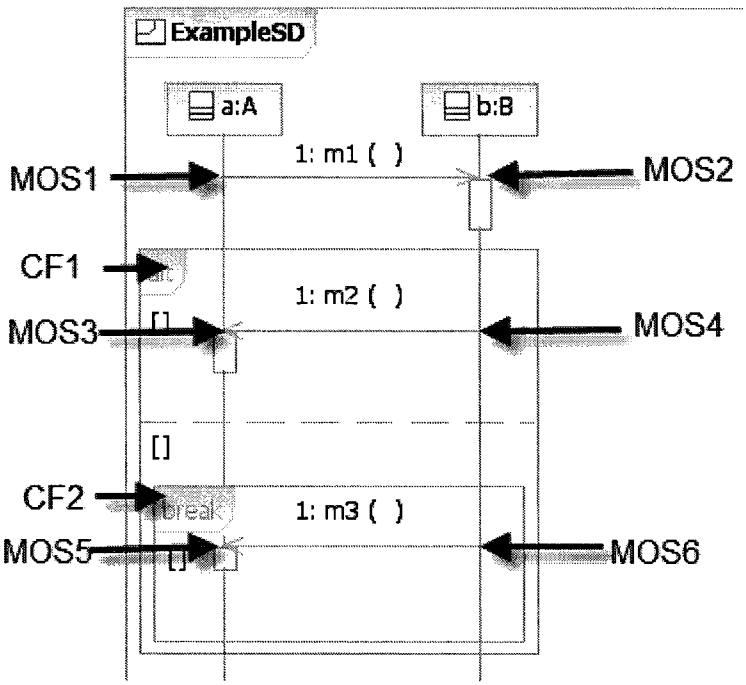


Figure 4.3: An Example SD.

The *getAspectFragments* is similar to *getPrimaryFragments* except that it returns fragments from an SD (interaction) of the aspect model, that is, pointcut SD or Advice SD. The name of the SD is passed as a parameter. The helper passes the name of the SD to *getApectsSD(sd : String)* to fetch the interaction, on line 6 of *getAspectFragments's* code snippet on the next page. The helper (*getAspectFragments*) then iterates over the interaction's fragments accumulating CFs and MOSs just like *getPrimaryFragments*.

The *equals* helper is overridden several times by changing the defined context. However, all the *equals* helpers are used to compare self (context object) with the given model element and return true if they are equivalent given some criteria. In fact, we have *equals* with *MessageOccurrenceSpecification*, *CombinedFragment*, and *InteractionOperand* contexts. We also have *equals* helpers for messages lifelines, messages, etc, but the *equals* helpers we talking about here are the ones invoked by the *joinPointsFragments*.

```
1.  helper def : getAspectFragments(sd : String) :
2.  Sequence (UML2!InteractionFragment) =
3.  if thisModule.getAspectSD(sd).oclIsUndefined()then
4.        Sequence{}
5.  else
6.     thisModule.getAspectSD(sd).fragment->asSequence()
7.     ->iterate (e; frg:Sequence(UML2!InteractionFragment)= Sequence {} |
8.                                    ...
```

Recall that *joinPointsFragments* selects all fragments in **C** such that the selected fragment is "equal" to the first element in **P** as shown by the code snippet below.

```
                                    ...
e | thisModule.getAspectFragments('Pointcut')->first().equals(e) and
thisModule.PairwiseMatchFragments(
                                    ...
```

Depending on what type of *InteractionFragment* the first element of **P** is, the ATL engine will invoke the appropriate *equals* helper (with the right context); For example, if the first element is a CF, the engine will invoke the *equals* helper with a CF context shown in the code snippet below. This *equals* helper returns true only if the argument fragment is also a CF with the same operator (alt, loop, break, etc) and has matching operands.

```
1.   helper context UML2!CombinedFragment def:
2.   equals(f : UML2!InteractionFragment) : Boolean =
3.   if f.oclIsUndefined() then
4.     false
5.   else
6.     if f.oclIsTypeOf(UML2!CombinedFragment) then
7.       if self.interactionOperator = f.interactionOperator then
9.         thisModule.PairwiseMatchOperands(self.operand, f.operand)
```

```
10.                                    ...
```

If the first element is a MOS, then another *equals* helpers with a MOS context is called.

4.3.3 JoinPointsCount Rules

Rules are used to generate the target model in ATL. Our *JoinPointsCount* transformation has two simple declarative rules (one matched rule and one lazy rule) that generate a UML model to store the number of join points found. A proper UML model should have a *Model* container element that packages all the other modeling elements. The list of contained objects is then referenced by the *packagedElement* attribute or association. The *Model* matched rule is the main rule that generates a *Model* element. We want the rule to match only one element. Therefore, its source pattern defines that the rule should match an element of type **UML2 Model** from the input aspect model as it can be seen on line 2 in the code snippet for the rule below. The rule's target pattern defines that a **UML2 Model** element will be generated.

```
1.  rule Model {
2.      from s : UML2!Model(s.fromAspectModel() )
3.      to t : UML2!Model (
4.        name <- 'NumberOfJoinpoints',
5.         packagedElement <- Sequence {
6.        thisModule.CreateLiteralInteger(thisModule.numJoinPoints,
7.                        'NumberOfJoinpoints'),
8.                                    ...
```

The attributes of the target elements will be initialized as defined from line 4. A *Model* element has a name and a collection of packaged elements. The *name* attribute is set to 'NumberOfJoinpoints'. The *packagedElement* attribute will be set to a sequence containing a **UML2 LiteralInteger** element generated by the invoked *CreateLiteralInteger* lazy rule. This lazy rule is passed the number of join points and a string (name) as parameters. The number of join points is, therefore, returned in a **UML2 LiteralInteger** model ele-

ment packaged in a **UML2 Model** element. The code snippet for the *CreateLiteralInteger* lazy rule is shown below. The rule creates a *LiteralInteger* model element and initializes its name and value attributes with a string (desired name) and an integer (number of join points found by our transformation) respectively.

```
1. lazy rule CreateLiteralInteger  {
2.      from count : Integer,  name :String
3.      to t: UML2!LiteralInteger  (
4.         name <- name,
5.        value <- count
6.                                    ...
```

4.3.4 Instantiating A Generic Aspect Model

The *Instantiate* transformation is implemented by the ATL transformation shown in Figure 4.4. This transformation instantiates a generic aspect model to produce a context specific aspect model. It inputs a primary model, generic aspect model and a mark model, and outputs a context specific aspect model.

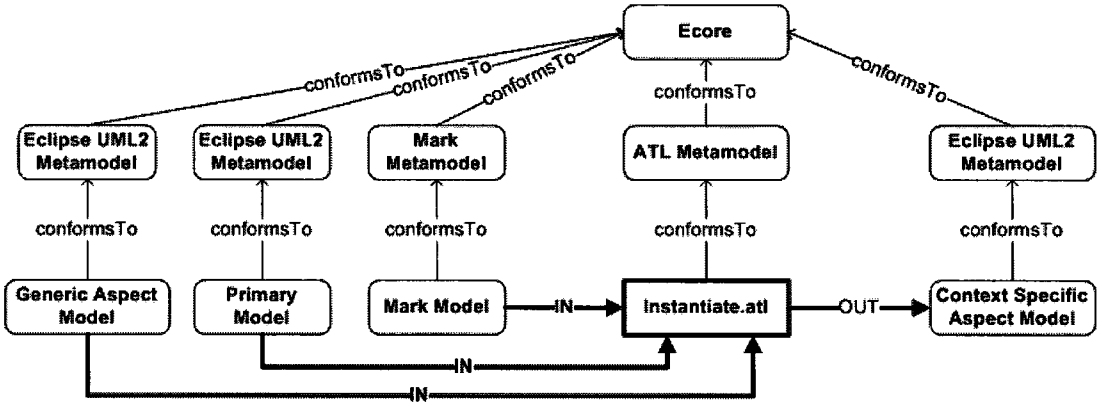


Figure 4.4: An overview of the *Instantiate* ATL Transformation.

The *Instantiate* ATL module, whose header is shown below, implements the *Instantiate* transformation. The header declares that the module creates a target **UML2 model** bound to the CONTEXTSPECIFIC variable.

```
module Instantiate;
create CONTEXTSPECIFIC : UML2 from PRIMARY : UML2, ASPECT : UML2, BIND : BD;
uses PointcutMatchHelpers;
...
```

The module has two UML2 source models (bound to variables PRIMARY and ASPECT) and one source model bound to the variable BIND that conforms to our *Binding Directives* metamodel (BD); that is, a mark model. The header also declares that the module imports the *PointcutMatchHelpers* library.

4.3.5 Instantiate Helpers

Just like the *JoinPointsCount*, this module also uses some of the helpers defined in the *PointcutMatchHelpers* library listed in Appendix E. This transformation also uses helpers defined within its module. These helpers are listed in Table 3.

Before we can generate the context specific aspect model, we have to ensure that we have a join point where we can weave. The *pointcutMatched* attribute helper returns true if we have at least one join point. It is used as a guard condition for all the rules that generate the target model elements as we shall see later. This ensures that no model element will be generated if there are no join points. The details of this helper are shown below.

```
helper def: pointcutMatched : Boolean =
    thisModule.joinPointsFragments()->notEmpty();
```

The helper returns true if the sequence that contains all the fragments at the beginning of each join point (returned by *joinPointsFragments()*) is not empty. Since *pointcutMatched* is defined as an attribute helper, it is evaluated once and the result cached. This means that successive calls to the helper will be faster which improves performance especially in our case where the helper is called many times by all the rules.

Helper name	Return type	Purpose
mosNum	Integer	Global id for naming MOS.
aMOSs	Sequence	Returns a sequence of MOSs from the aspect model (AM).
aInteractionOperands	Sequence	Returns interaction fragments from AM.
aInteractionCon-straints	Sequence	Returns interaction constraints from AM
aOpaqueExpressions	Sequence	Returns opaque expressions from AM.
pMOSs	Sequence	Returns MOSs from the primary model (PM).
pOperations	Sequence	Returns operations from PM.
pMessages	Sequence	Returns messages from PM.
pLifelines	Sequence	Returns a sequence of all lifelines from PM.
pProperties	Sequence	Returns lifeline properties from PM.
allLifelines	Boolean	Returns all lifelines from the PM plus those from AM (allowing duplicates).
equals (cMsg) (context = Message)	Boolean	Returns true if *cMsg* is equivalent to self.
bind () (context = Message)	String	Binds the context message to a supplied binding value or to a matching message from PM.
bind () (context = Lifeline)	String	Binds a lifeline to a supplied binding value or to a matching lifeline from PM.
equals (cPty) (context = Property)	Boolean	Returns true if self matches the given PM property (cPty).
getLifeline (p)	Lifeline	Returns a lifeline that represents the property *p*.
bind ()	String	Binds a context property to a supplied binding

Helper name	Return type	Purpose
(context = Property)		value or to a matching property from PM.
bind() (context = Operation)	String	Binds an operation to a supplied binding value or to a matching operation from PM.
matches(o)	Boolean	Returns true if self, a pointcut operation, matches a PM operation.
getMessage()	Message	Returns a message associated with an operation.
fromAspectModel() (context = Model)	Boolean	Returns true if a model is from AM.
bind () (context = Property)	String	Binds a context property to a supplied binding value or to a matching property from PM.
bind() (context = Operation)	String	Binds an operation to a supplied binding value or to a matching operation from PM.
fromAspectModel() (context = Interaction)	Boolean	Returns true if an interaction is from AM.
getArguments	Sequence	Returns arguments of given a message.
getParameters()	Sequence	Returns parameters of an operation.
getMOSByLifeline(l)	Sequence	Returns a sequence of MOSs that cover a given lifeline.
getMOSLifeline (mos)	Sequence	Returns a sequence of lifelines that is covered the given MOS.
createAspectName()	String	Generates a name for the context specific AM.

Table 3: Instantiate Helpers

4.3.6 Instantiate Rules

Several rules are required to generate a complete context specific aspect model. In fact, we have a rule for every model element type required for a well formed UML sequence diagram. These rules include several matched rules and a handful of lazy rules. Just like in the previous transformation, our target UML model should have a *Model* container element that packages all the other modeling elements. The rule that generates the target *Model* element is shown below.

```
1.  rule Model {
2.      from s : UML2!Model (
3.               s.fromAspectModel() and thisModule.pointcutMatched
4.          )
5.      to t : UML2!Model (
6.               name <- s.createAspectName(),
7.               packagedElement <- s.packagedElement
8.          )
9.  }
```

The source pattern specifies that the rule matches elements of type **UML2 Model**. It also has a condition that the matched element should come from the aspect model (using *fromAspectModel()* helper), and also that *pointcutMatched* must be true, as mentioned earlier. There is only one *Model* element from the aspect model. If at least one join point was found, then only one **UML2 Model** element will be created on the target model since the target pattern declares that the rule creates an instance of **UML2 Model**. Its packaged elements will be initialized to the list from the matched element as defined on line 7 above. The name will be initialized with the string returned by the *createAspectName()* helper on line 6 above.

The UML specification describes that an *Interaction* can be contained in a *Collaboration.* Collaborations are used to show the structure of cooperating elements with a particular purpose [OMG09]. Indeed, the primary and aspect models created using RSA have

interactions contained within collaborations. The *Collaborations* matched rule has the task of generating *Collaboration* objects that enclose the interactions for the advice and pointcut. Recall that the aspect model consists of the advice and pointcut SDs (interactions). The rule is described by the code snippet shown below.

```
1.  rule Collaborations {
2.      from s : UML2!Collaboration (
3.               thisModule.aCollaborations->includes(s) and
4.               thisModule.pointcutMatched
5.          )
6.      to t : UML2!Collaboration (
7.               name <- s.name,
8.               ownedBehavior <- s.ownedBehavior,
9.               ownedAttribute <- s.ownedAttribute,
10.              ownedConnector <- s.ownedConnector
11.         )
12. }
```

The guard condition for this rule's source pattern ensures that only collaborations from the aspect model (and not from primary model) are matched. It checks if a collaboration is included in the collection of collaborations from the aspect model returned by the *aCollaborations* attribute helper. The attributes of the generated collaboration, including the enclosed interactions (*ownedBehavior*), are initialized from those of the matched collaboration as shown on lines 7 to 10 above.

The *aInteractions* and *pInteractions* rules are used to create *Interaction* target elements for the advice and pointcut respectively. The rules are almost identical with slight differences in the source pattern guard. The code on the next page gives details of the *aInteractions* rule. The difference between the rules is in line 3. The guard for *aInteractions* rule ensures that the rule matches the interaction from the aspect's advice which has the name "Advice".

```
1.  rule aInteractions {
2.      from s : UML2!Interaction (
3.              s.name = 'Advice' and thisModule.pointcutMatched
4.        )
6.      to t : UML2!Interaction  (
7.         name <- s.name,
8.              lifeline <- s.lifeline,
9.              fragment <- s.fragment,
10.             message <- s.message,
11.             ownedAttribute <- s.ownedAttribute,
12.             ownedConnector <- s.ownedConnector,
13.             generalOrdering <- s.generalOrdering,
14.             ownedBehavior <- s.ownedBehavior,
15.             covered <- s.covered
16.       )
17. }
```

The guard for the *pInteractions* rule matches the interaction from the aspect's pointcut which has the name "Pointcut". Both rules then initialize the attributes of the generated interactions using the values from the attributes of the matched source elements as it can be seen from lines 7 to line 15.

The *Lifelines* rule generates lifelines for both the advice and pointcut SDs. The rule matches all lifelines from the advice model as shown on line 3 of rule's code snippet on the next page. The *aLifelines* helper returns all lifelines from the generic aspect model (advice and pointcut SDs). The generated lifeline's attributes are then initialized as shown from lines 6 to 8. This rule probably best shows how helpers are used to assist rules in creating the target models other than been used as guard conditions in the source pattern. We can see on line 6 that binding is achieved by using the *bind()* helper to initialize the name of the generated lifeline.

```
1.  rule Lifelines {
2.      from s : UML2!Lifeline (
3.         thisModule.aLifelines->includes(s) and thisModule.pointcutMatched
4.         )
5.      to targetLifeline : UML2!Lifeline (
6.         name <- s.bind(),
7.               coveredBy <- thisModule.getMOSByLifeline(s),
8.               represents <- s.represents
9.         )
10. }
```

Recall that instantiating a generic aspect involves binding of template parameters to values defined in the mark model. Our transformation achieves instantiation by using the *bind()* helper defined with several contexts for the different model elements that can be parameterized. The *bind()* helper used in the above rule is defined in the context of a lifeline. It is defined by the code on the next page. The helper begins by checking if the lifeline's name is a template parameter because not all lifelines will be parameterized. If the name is not a template parameter, the helper just returns the lifeline's name, on line 30. Otherwise it checks if there is binding directive defined in the mark model. If no binding directive is defined, the helper attempts to find a matching lifeline from the primary model and return its name (lines 22 to 24) otherwise it returns the string "UNKNOWN" (on line 26). However, if a binding exists for self, the helper checks (on line 7) if self is to be bound to "*", that is, to any lifeline. That been the case, the helper searches the primary model for a lifeline that is *equal* to self, on lines 10 to 12. If a matching lifeline is found, its name is returned else the string "UNKNOWN" is returned, on line 16. However, if a binding is defined and is not a wild card, then that binding is returned by calling *getBinding* helper, on line 19.

```
1.  helper context UML2!Lifeline def : bind () : String =
2.  if self.name.startsWith('|') then
```

```
3.  -- check if a binding has been defined in our Mark model
4.    if thisModule.bindingDefined(self.name) then
5.    -- Binding is defined, now check if the binding does not contain
6.    -- wildcards for now we can only check for '*'
7.          if thisModule.getBinding(self.name).startsWith('*') then
8.          -- A binding was defined but is wildcard. Bind to matching
9.          -- lifeline in PM
10.               if thisModule.pLifelines->select (e |
11.                     self.equals(e))->notEmpty() then
12.               thisModule.pLifelines->any (e | self.equals(e)).name
13.               else
14.               -- We should never get here unless we call this helper
15.               -- without ensuring that our pointcut matches
16.                     'UNKNOWN!!!'
17.               endif
18.         else
19.               thisModule.getBinding(self.name)
20.         endif
21.   else
22.         if thisModule.pLifelines->select (e |
23.               self.equals(e))->notEmpty() then
24.               thisModule.pLifelines->any (e | self.equals(e)).name
25.         else
26.               'UNKNOWN!!!'
27.         endif
28.    endif
29. else
30.         self.name
31. endif;
```

As explained previously, our transformation has a rule for every type of model element that will exist in the composed model. Table 4 gives a summary of all the rules defined in the *Instantiate* transformation.

Rule	**Purpose**
Model	Generates the *model* elements.
Collaborations	Generates advice and pointcut collaborations to contain advice and pointcut interactions respectively.
aInteractions	Generates the advice interaction.
pInteractions	Creates the pointcut interaction.
CFs	Generates combined fragments.
IOs	Creates all interaction operands
ICs	Generates interaction constraints that serve as guard conditions for operands.
Lifelines	Generates lifelines.
Messages	Generates messages.
MOSs	Generates MOSs.
SOEs	Generates SOEs for the MOSs in the target model.
ROEs	Generates ROE for the MOSs.
Operations	Generates operations for classes.
Classes	Generates classes.
Properties	Generates *Properties* for the target lifelines.
OEs	Generates *OpaqueExpressions* used to express guard conditions for interaction operands.
LiteralStrings	Creates *LiteralStrings* that are used for guard conditions.

Rule	Purpose
LUNs	Creates *LiteralUnlimitedNaturals* that are used for guard conditions.
CreateLiteralString (lazy)	Creates strings that are used as arguments for messages.
CreateLiteralInteger (lazy)	Creates integers that are used for guard conditions and arguments for messages.
CreateParameter (lazy)	Creates string parameters for operations.

Table 4: Instantiate Rules

4.3.7 Composing Aspect Models

After obtaining a context specific aspect model from the previous transformation (*Instantiate*), the next step is to compose the context specific aspect model with the primary model. This is achieved by the *Compose* ATL transformation whose overview is shown in Figure 3.5 on page 21. The transformation inputs the primary and context specific source models, and produces a composed target model. Both the source models and the output model conform to the UML2 metamodel. This transformation is implemented by the *Compose* ATL module. The code snippet below shows a description of the module's header.

```
module Compose;
create COMPOSED : UML2 from PRIMARY : UML2, ASPECT : UML2;
uses PointcutMatchHelpers;
                                    ...
```

As expected, the header declares that the module creates a UML2 Model bound to the variable COMPOSED from two UML2 source models bound to the variables PRIMARY and ASPECT for the primary and aspect models respectively. The module also uses some helpers from the *PointcutMatchHelpers* library.

4.3.8 Compose Helpers

Our *Compose* transformation has a number of helpers listed in Table 5. All the helpers have a necessary role to play but some roles are, certainly, more important than others. For example, the *getTargetFragments* attribute helper has the privilege of returning the composed sequence of fragments, that is, *sequence {sub_before, A, sub_after}* from algorithm-3. The code definition of this helper is shown below. The helper begins by ensuring that there is, at least, one join point by calling the *pointcutMatched* helper on line 2, which we have described earlier. If there exists a join point, *getTargetFragments* then obtains a sequence of fragments before the join point by invoking the *lowerFragSub* helper on line 4, a sequence of fragments from the advice (by invoking *getAspectFragments*) on line 5, and a sequence fragments after the join point on line 6. It returns a flattened sequence consisting of those sequences. The fragments returned by *getTargetFragments* are used to initialize the fragment attribute of the interaction generated by our transformation.

```
1.  helper def : getTargetFragments : Sequence(UML2!InteractionFragment) =
2.          if thisModule.pointcutMatched then
3.              Sequence {
4.                thisModule.lowerFragSub(thisModule.firstJPIndex),
5.                thisModule.getAspectFragments('Advice'),
6.                thisModule.upperFragSub(thisModule.firstJPIndex +
7.                thisModule.numPCTFs-1)
8.              }->flatten()->asSequence()
9.      else
10.             Sequence{}
11.     endif;
```

The *getTargetFragments* helper also serves as the base of our composition process. Almost all the other elements to be used in generating the target model are rooted from this helper. The *targetCFs* helper, which returns all combined fragments to be used for generating combined fragments in the target model, iterates through fragments returned

by *getTargetFragments* returning all instances of *CombinedFragment*. The *targetOperands* helper, in return, iterates through the sequence of combined fragments returned by *targetCFs* to obtain all instances of *InteractionOperand*.

A summary of all the helpers used by our transformation is given in Table 5. Please note that aspect model here, refers to the context specific aspect model.

Helper name	**Return type**	**Purpose**
mosNum	Integer	A global variable for numbering MOSs.
recvEventNum	Integer	A global variable for numbering ROEs.
sendEventNum	Integer	A global variable for numbering SOEs.
aModel	Model	Returns the model instance from the aspect model.
pModels	Sequence	Returns instances of model from the primary model.
aCFs	Sequence	Returns CFs from the aspect model.
pCFs	Sequence	Returns CFs from the primary model.
pProperties	Sequence	Returns all lifeline properties from the primary model.
pInteractions	Sequence	Returns a sequence of interaction model elements from the primary model.
pLifelines	Sequence	Returns lifelines from the primary model.
pMessages	Sequence	Returns a sequence of messages from the primary model.
pClasses	Sequence	Returns all classes from the primary model.
allICs	Sequence	Returns interaction constraints from both input models.
allLiteralIntegers	Sequence	Returns literal integers from both input models.

Helper name	Return type	Purpose
allOEs	Sequence	Returns opaque expressions from both input models.
allLiteralStrings	Sequence	Returns literal strings from both input models.
allLUNs	Sequence	Returns Literal Unlimited Naturals from both input models.
pMOSs	Sequence	Returns MOSs from the primary model.
adviceMOSs	Sequence	Returns MOSs from the aspect model advice.
pointcutMOSs	Sequence	Returns MOSs from the aspect model pointcut.
numPointcutMOS	Integer	Returns the number of MOSs in the pointcut SD.
numPCTFs	Integer	Returns the number fragments (MOSs and CFs) from the aspect model.
numPrimaryMOS	Integer	Returns the number of MOSs from the primary model.
numPrimaryFragments	Integer	Returns the number of fragments (MOSs and CFs) from the primary model.
pointcutMatched	Boolean	Returns true if at least one join point is found.
getAllMOSs()	Sequence	Returns a union of MOSs from the primary model and MOSs from the aspect model.
lowerFragSub(j)	Sequence	Returns the sub sequence of fragments before the join point. This is the *sub_before* sequence in the algorithm-3.
upperFragSub(k)	Sequence	Returns the sub sequence of fragments after the join point. This is the *sub_after* sequence in the algorithm-3.

Helper name	Return type	Purpose
getTargetFragments	Sequence	Returns the composed sequence of fragments, that is, Sequence {sub_*before*, **A**, sub_*after*}.
targetCFs	Sequence	Returns CFs to be used for generating target model combined fragments.
targetOperands	Sequence	Returns operands to be used for generating target model interaction operands.
getTargetOperand()	Sequence	Returns operands for a given CF.
operandTgtFrags()	Sequence	Returns all fragments to be enclosed by the given operand. If the operand includes the join point, then this helper returns all fragments from *getTargetFragments* that were enclosed by the operand and are before the join point plus all fragments from the advice plus all fragments in *getTargetFragments* that were in the operand but after the join point.
targetConstraints	Sequence	Returns all constraints which form the guard conditions for operands in *targetOperands*.
tgtOEs	Sequence	Returns all opaque expressions for constraints in *targetConstraints*.
getTargetMOSs	Sequence	Returns all MOSs from the composed sequence of fragments to be used from generating target MOS model elements.
tgtMOSByLifeline(L)	Sequence	Returns MOSs from *getTargetMOSs* that cover the given lifeline *L*.
mosLifeline (mos)	Sequence	Returns lifelines covered the given MOS.
cfTgtLifelines()	Sequence	Returns lifelines covered by the context CF.

Helper name	Return type	Purpose
getTargetFragments()	Sequence	Returns a list of composed fragments that are enclosed by the CF.
notInPrimaryModel() context = Model	Boolean	Returns true if the context MOS is not from the primary model.
samePropertyType (p1, p2)	Boolean	Checks if the two supplied lifeline properties are instances of the same class.
notInPrimaryModel() context = Lifeline	Boolean	Checks if the lifeline exists in the primary model.
targetLifelines	Sequence	Returns lifelines from the primary model and aspect model's advice to be used for generating lifelines in the target model.
messagesMatch (pMsg, cMsg)	Boolean	Returns true if the given pointcut message matches the primary model message. Similar to *messagesMatch* from *Instantiate* but not does have to check for binding.
lifelineMatch (pL, cL)	Boolean	Returns true if the supplied pointcut lifeline is equivalent to the given lifeline from primary model. Similar to *lifelineMatch* from *Instantiate* but does have to check for binding.
allMatchedMessages		Returns messages from the primary model and the advice allowing duplicates.
targetMessages	Sequence	Returns messages to be used for generating the target model. These are messages for which there exists a MOS in *getTargetMOSs*.
allOperations	Sequence	Returns a combination of operations from both

Helper name	Return type	Purpose
		input models.
allClasses	Bag	Returns a union of classes from the aspect and primary models.
targetClasses	Set	Returns a union of classes from both input models with duplicates removed.
allEvents	Sequence	Returns a union of *MessageEvents* from the primary and aspect models with duplicates removed.
getTargetEvents()	Sequence	Returns all *MessageEvents* (from *allEvents*) that will be used to generate events in the target model. These are events from which there exists a related operation in *allOperations*.
operationExists(op)	Boolean	Returns true if the given operation exists in *allOperations*.
getOwner()	Class	Returns a class that owns the operation.
aspectCol(name)	Collaboration	Returns the first collaboration (if any) from the aspect model that has the supplied name.
allProperties	Sequence	Returns lifeline properties from both input models removing duplicates.
targetProperties	Sequence	Returns properties to be used for generating the target model.
getType ()	Class	Returns the class that defines the type for the given property.
getProperty (name)	Property	Returns the property with the given name.
getModelName(n, md)	String	Generates a name for our composed model.

Table 5: Compose Helpers

4.3.9 Compose Rules

Several rules are defined for creating the composed target model. Rules in the *Compose* transformation probably use more helpers compared to the two previous transformations, mainly because in this transformation more elements are removed or added. This requires modifications to many associations between model elements. The code below is that for the *Model* matched rule which is used to create the **UML2 Model** element.

```
1.  rule Model {
2.      from s : UML2!Model (
3.         s.fromPrimaryModel()  and thisModule.pointcutMatched
4.         )
5.      to t : UML2!Model (
6.                 name <- thisModule.getModelName(s.name, thisModule.aModel),
7.                 packagedElement <- Sequence {
8.                         thisModule.targetClasses,
9.                         thisModule.pCollaborations,
10.                        thisModule.getTargetEvents()
11.                                    ...
```

The rule matches elements of type **UML2 Model** from the primary model, and provided the pointcut matches as defined by the source pattern on lines 2 and 3. The rule creates instances of *Model* as declared on line 5. Since the primary model consists of one instance of the *Model* class, this rule will generate only one instance. It then initializes the created instance with the use of several helpers as defined on lines 6 to 11. The *getModelName* helper generates a string used to initialize the name attribute. The *packageElement* list attribute is initialized to a sequence of classes returned by *targetClasses*, a collaboration from the primary returned by *pCollaborations*, and a sequence of events returned by the *getTargetEvents()* operation helper. These three helpers are defined in the context of the module; hence, the use of the keyword *thisModule*.

The *Messages* rule shown below is used to generate messages for the target model. This rule is more interesting since it calls a few lazy rules to help initialize some of the attributes of the target messages to be created. The rule matches all messages included in *targetMessages* and creates messages for the target model. The generated message is initialized as defined from line 7. On line 12, the *argument* attribute is initialized by calling a suitable lazy rule. We are only interested in primitive type message arguments (integers and strings); therefore, we have two lazy rules for creating an instance of *LiteralInteger* or *LiteralString* depending on the argument type for the matched message.

```
1.  rule Messages{
2.      from s : UML2!Message (
3.          thisModule.targetMessages->includes(s) and
4.          thisModule.pointcutMatched
5.          )
6.      to t : UML2!Message (
7.          name <- s.name,
8.          sendEvent <- s.sendEvent,
9.          receiveEvent <- s.receiveEvent,
10.         messageSort <- s.messageSort,
11.         argument <-  s.argument->collect (e |
12.                     if  e.oclIsTypeOf(UML2!LiteralString) then
13.                             thisModule.CreateLS(e)
14.                     else
15.                             if  e.oclIsTypeOf(UML2!LiteralInteger) then
16.                                     thisModule.CreateLI(e)
17.                             else
18.                                     OclUndefined
19.                             endif
20.                                     ...
```

The rule uses ATL's built-in *oclIsTypeOf(t: oclType)* operation to check the type of the argument for the matched message. If it is a *LiteralString* then the *CreateLS* lazy rule is called but if it is a *LiteralInteger* the *CreateLI* lazy rule is called instead. If the argument is neither an integer nor a string, the message's argument attribute is initialized to *OclUndefined*, ATL's equivalent of null.

All the rules that are used by the Compose transformation to generate the composed model are listed in Table 6 below.

Rule	**Purpose**
Model	Generates a *Model* element that contains all the other target model elements.
Collaborations	Creates a *Collaboration* that contains the composed model interaction.
Interactions	Generates the *Interaction* that owns fragments, lifelines, messages, etc.
CombinedFragments	Generates all *CombinedFragments* including nested ones.
InteractionOperands	Generates all *InteractionOperands*.
InteractionConstraints	Creates all the constraints.
OpaqueExpressions	Generates *OpaqueExpressions* for *InteractionConstraints*.
Lifelines	Generates all lifelines.
Messages	Produces messages for the target model.
MOSs	Creates all *MessageOccurrenceSpecifications* (MOSs).
ROEs	Generates *ReceiveOperationEvents* for *receiveEvent* MOSs.
SOEs	Produces *SendOperationEvents* for *sendEvent* MOSs.
Operations	Generates operations for classes.
Classes	Generates classes.
Properties	Generates all *Properties* for lifelines.

lazy rule CreateLUN	Creates *LiteralUnlimitedNaturals* that are used for guard conditions.
lazy rule CreateLS	Creates strings that are used for guard conditions and arguments for messages.
lazy rule CreateLI	Creates integers that are used for guard conditions and arguments for messages.
lazy rule CreateParam	Creates parameters for operations in the target model.

Table 6: Compose Rules

4.4 Running an ATL transformation

A launch configuration must be set up first before executing an ATL transformation manually. The configuration allows us to specify information required to execute an ATL transformation [ATLUserGuide]. Such information includes the location of the models and metamodels involved in the transformation, the location of the ATL file and libraries, and the type of model handlers (EMF or MDR). Figure 4.8 shows an example of a launch configuration.

Once the launch configuration information has been specified, the transformation can then run as many times as required [ATLUserGuide]. Running an ATL transformation using this method is appropriate if we have only one transformation to execute but if one has a chain of transformations, like in our composition approach, this method will not suffice. Recall that algorithm-2 first executes the *JoinPointsCount* transformation (function) to get the number of join points. If at least one join point exists, it then executes the *Instantiate* transformation to produce a context specific model which is then fed, together with the primary model, into the *Compose* transformation to produce a composed model. If there are multiple join points, this chain of transformations will have to be repeated several times. Fortunately, there exists ant tasks which allow for execution of several transformations.

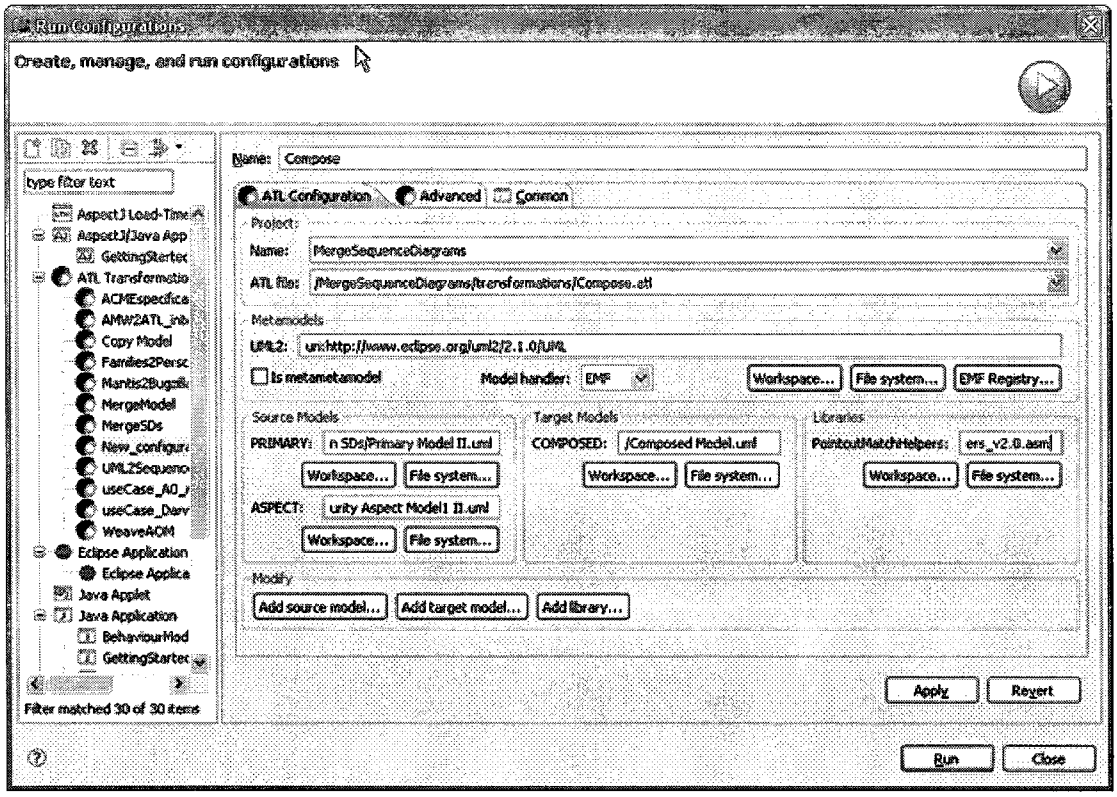


Figure 4.8: An ATL Transformation Launch Configuration.

4.4.1 AM3 Ant Tasks

Ant (Apache Ant) is a make tool that is based on Java but unlike other make tools, it is much more user-friendly, and its configuration files are based on XML not shell commands [Ant_Manual]. We used ant tasks supplied by the *org.eclipse.gmt.am3.tools.ant* plug-in of the ATLAS MegaModel Management (AM3) project [AM3_Tasks_Wiki] [AM3_Wiki] to achieve execution of consecutive transformations. On the next page, we have a code snippet of ant file we used to compose the security aspect discussed in Chapter 3.

The file starts with a *project* root tag which has a name and target attributes. The target attribute declares the target to execute, and depends on other targets [AM3_Tasks_Wiki]. Our script defines two targets; namely, *ComposeSequenceDiagram* (on line 3) and *loadMetamodels* (on line 26). *ComposeSequenceDiagram* depends on

loadMetamodels as declared on line 3 below. A *target* defines a task which is the code or transformation to be executed [AM3_Tasks_Wiki]. The *am3.loadModel* task can load a model using a model handler where the model can even be a metamodel [AM3_Tasks_Wiki]. For example, on lines 4 to 7 the *am3.loadModel* task loads the primary, aspect and mark models using the EMF model handler. However, on lines 26 to 28, the task loads the UML2 and BD metamodels. The *am3.saveModel* task, on the other hand, saves a model to disk. As we can see on line 24, this task has two attributes; namely, model and path. The *model* attribute captures the name of the model (on the script) to save while *path* defines the location and file to save the model [AM3_Tasks_Wiki]. The *am3.atl* task executes an ATL transformation. It uses models loaded by the *am3.loadModel* task for execution and references them by name [AM3_Tasks_Wiki]. Lines 12 to 22 show how this task is defined in our example. It references the UML2 model loaded in line 27, and the PRIMARY and ASPECT models loaded on lines 4 to 8. The produced models are saved to disk by the *am3.saveModel* task on line 24.

```
1.  <project name="Experiment2" default="ComposeSequenceDiagram">
2.          <property name="outPath" value="../output/" />
3.          <target name="ComposeSequenceDiagram" depends="loadMetamodels">
4.            <am3.loadModel modelHandler="EMF" name="PRIMARY"
5.  metamodel="UML2"  path="../input/Primary Model.uml" />
6.          <am3.loadModel modelHandler="EMF" name="ASPECT" metamodel="UML2"
7.  path="../input/Generic Security Aspect Mode.uml" />
8.  <am3.atl path="Instantiate.asm" allowInterModelReferences="true">
9.                  ...
10.           <inModel name="PRIMARY" model="PRIMARY"/>
11.          <inModel name="ASPECT" model="ASPECT"/>
12.           <inModel name="BIND" model="BIND"/>
13.      <outModel name="CONTEXTSPECIFIC" model="CSAM1" metamodel="UML2"/>
14.               <library path="PointcutMatchHelpers_v2.0.asm"/>
```

```
15.      </am3.atl>
16.          <am3.atl path="Compose.asm" allowInterModelReferences="true">
17.              <inModel name="UML2" model="UML2"/>
18.              <inModel name="PRIMARY" model="PRIMARY"/>
19.              <inModel name="ASPECT" model="CSAM1"/>
20.              <outModel name="COMPOSED" model="CM1"  metamodel="UML2"/>
21.              <library path="PointcutMatchHelpers_v2.0.asm"/>
22.          </am3.atl>
23.          ...
24.  <am3.saveModel model="CM1"  path="${outPath}Composed Model.uml"/>
25.        </target>
26.        <target name="loadMetamodels">
27.        <am3.loadModel modelHandler="EMF" name="UML2" metamodel="MOF"
28.   nsUri="http://www.eclipse.org/uml2/2.1.0/UML"/>
29.                                          ...
```

Ant tasks, however, have a slight limitation. We cannot really navigate models in an ant script. We can only execute transformations and serialize the produced target models. But in our Complete Composition Algorithm we want to be able to execute the *JoinPointsCount* transformation to find how many join points are there before executing the other two transformations. To do this we must be able to navigate the model produced by the *JoinPointsCount* transformation and retrieve the value of the *NumberOfJoinpoints LiteralInteger* model element packaged in *Model* container instance. If we have one or two join points then this is not a huge problem. We can manually execute *JoinPointsCount* by launching its configuration, as described previously, to find the number of times the algorithm will loop. We can then unroll the loop and create an ant script that will execute that chain of transformations. For example, say our primary model has two join points. Then unrolling the algorithm would give the following statements (chain of transformations) from the algorithm.

1.	// instantiate our generic aspect model for 1st join point
2.	**$CSAM_1$** = Instantiate(**GAM, MM, PM**)
3.	// compose advice and first join point
4.	**CM_1** = Compose (**PM, $CSAM_1$**)
5.	// instantiate our generic aspect model for 2nd join point
6.	**$CSAM_2$** = Instantiate(**GAM, MM, CM_1**)
7.	// compose advice and 2nd join point
8.	**CM_2** = Compose (**CM_1, $CSAM_2$**) // final composed model.

Where;

- **GAM** = Generic Aspect Model
- **$CSAM_i$** = Context Specific Aspect Model (Instantiated generic aspect model)
- **MM** = Mark Model
- **PM** = Primary Model
- **CM_i** = Composed Model

and $1 \leq i \geq 2$

We can then easily encode these statements in an ant script. In fact this is the approach we took in the experiments we conducted described in the next chapter. Clearly this approach does not scale well and is only useful for cases with a few join points. Getting the number of join points offline and manually is not ideal. Having a completely automatic composition process that requires no loop unrolling or manually getting the number of joints would require invoking the three ATL transformations from a program (e.g., in Java) that can navigate produced models. This program would provide the glue between the transformations.

4.4.2 Invoking ATL from Java

Java can potentially be used to glue our transformations and help automate our entire composition process. Invoking ATL from Java is beyond the scope of this work. However, we can provide an overview of how such Java program or plug-in would achieve this task. Figure 4.8 shows a flowchart for the proposed program.

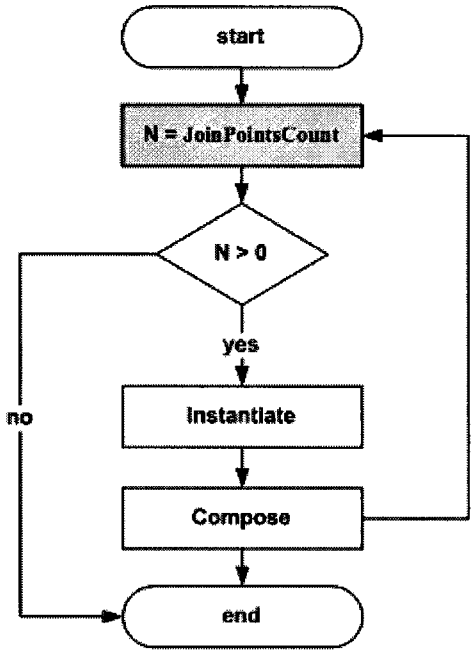


Figure 4.8: A Flow Chart for Our Proposed Java Program.

The plug-in would obviously implement our composition algorithm. It would pass transformation configuration information to the ATL engine, execute the engine, and save output models to disk. Some work by [Minanovic07] and other developers has been done on executing ATL from Java. However, the part that we are interested is how such a Java plug-in would navigate the model produced by the *JoinPointsCount* transformation, and extract the number of join points. This step is highlighted in Figure 4.8.

The UML2 Java plug-in is an EMF implementation of the UML specification [UML2_Project] [UML2_Wiki]. The plug-in provides functionality for manipulating UML models whether in memory or on disk. Therefore, using the UML2 plug-in we can write a class or methods that will read the model produced by the *JoinPointsCount* transforma-

tion, extract the number of join points, and pass the number to other parts of the main Java program to decide if it should execute the other two transformations. The code below shows a class named *ModelUtil* that can be used to get the number of join points given a reference to a model. A client that uses *ModelUtil* will create an instance of this class passing a model instance obtained by reading the target model (from disk) produced by *JoinPointsCount*. After creating an instance of *ModelUtil* the client will invoke its *getNumJoinpoints()* method to return the actual number of join points. The method *getNumJoinpoints()* first ensures that the model object to be manipulated is not null (line 22). It then obtains the instance of *LiteralInteger* which contains the number of join points in its *value* attribute. This instance of *LiteralInteger* is retrieved from the model object by the *getLiteralInteger*("NumberOfJoinpoints"), on line 23. The *getLiteralInteger (String name)* method iterates the packaged elements in the model object (returned by the model's *getPackagedElements method*) looking for objects that are instances of *LiteralInteger* and have the supplied name (e.g., "NumberOfJoinpoints"). To get the actual integer stored in the appropriate instance of *LiteralInteger*, *getNumJoinpoints()* then calls the object's *getValue()* method, on line 25. This is the number (number of join points) that is returned to the client.

```
1.  public class ModelUtil {
2.          private Model model;
3.          public ModelUtil(Model model) { this.model = model; }
4.
5.          public LiteralInteger getLiteralInteger(String name){
6.          for (Iterator<PackageableElement>
7.              itr = model.getPackagedElements().iterator();itr.hasNext();){
8.                      PackageableElement e = itr.next();
9.
10.                     if (e instanceof LiteralInteger){
11.                             if (e.getName().equals(name)){
12.                                     return (LiteralInteger)e;
```

```
13.                         }
14.                     }
15.             }
16.             return null;
17.         }
18.
19.         public int getNumJoinpoints(){
20.             int count = 0;
21.             LiteralInteger numJoinpoints = null;
22.             if (model != null){
23.             numJoinpoints = this.getLiteralInteger("NumberOfJoinpoints");
24.                     if (numJoinpoints != null){
25.                         count = numJoinpoints.getValue();
26.                     }
27.             }
28.             return count;
29.                                 ...
```

The code snippet on the next page shows how we may use *ModelUtil*. We first create a URI (Uniform Resource Identifier) for the file containing the model (with the number of join points) using the file's location on disk, its name, and the *.uml* extension, on lines 6 to 8. We then create (on line 11) an instance *ModelValidationUtil*, a utility class that has methods to load a model file from disk (using supplied URI) and cast it to a UML package (*Model* object). It also has methods to serialize an instance of *Model* to disk, and other housekeeping operations. A model object is then obtained by calling the load method from the newly created instance of *ModelValidationUtil* (util). On line 12, we then ensure that the model is not null before creating an instance (mU) of our *ModelUtil* class, on line 16. We finally invoke *mU*'s *getNumJoinpoints()* method to get the number of join points and print this number to the standard output as shown on lines 17 and 18.

```
1.                                          ...
2.   String modelFilename = "GetNumberOfJoinpoints1";
3.   String location = "file:/d:/ComposeSequenceDiagrams/";
4.                                          ...
5.   try {
6.         URI uri = URI.createURI(location)
7.         .appendSegment(modelFilename).appendFileExtension(
8.         UMLResource.FILE_EXTENSION);
9.         ModelValidationUtil util = new ModelValidationUtil();
10.                                        ...
11.        Model model = util.load(uri);
12.        if (model == null){
13.              throw new Exception ("Unable to load model");
14.        }
15.
16.        ModelUtil mU = new ModelUtil(model);
17.        System.out.println("Number of join points = "
18.                                          + mU.getNumJoinpoints());
19.                                        ...
```

This code snippet could be part of larger program where the code is executed (with slight modifications) when the larger program needs the actual number of join points before launching the *Instantiate* and *Compose* transformations.

Chapter 5 : Testing, verification and Validation

We introduced our approach to AOM composition in Chapter 3, and in the process discussed (and defined) concepts such as the primary model, generic and context specific aspect models, mark model, advice, pointcut and join points. We also discussed composition of sequence diagrams and the algorithm (Complete Composition Algorithm) to achieve this composition. In Chapter 4, we described how this algorithm was implemented using ATL transformations to get the number of join points, to instantiate a generic aspect, and to compose a context aspect model with the primary model. In this chapter, we will put our implementation to the test with a number of test cases (TCs) of varying and increasing complexity. We will, however, focus on two case studies (CS). The first case study is adapted from Whittle and Jayaraman. The second case study adapted from Petriu et al. and Gong shows a possible limitation of our approach. Before diving into our test cases and case studies, let us first discuss how the input models (primary and generic aspect models) for our TCs and case studies were produced, and how we established that the composed models were well formed and valid.

5.1 Creating Models and Scripts

For all the test cases and case studies, the primary and generic aspect models were created using Rational Software Architect (RSA) 7.5, exported to *.uml* files, and copied to Eclipse and the ATL IDE for composition. The ant scripts and mark models were created as XML and XMI files respectively using Eclipse.

5.2 Composed Model Validation

Validation is obviously important. Our main objective is to compose two UML SDs and produce another SD; therefore, we must ensure that we have produced a valid UML SD. We must also verify that the composition was correctly done and that the produced SD can be manipulated by our modeling tools (like RSA) allowing for further analysis and sharing with other researchers. Two options were explored in the quest for validating

our composed model; namely, using a custom Java package and using RSA 7.5.

5.2.1 Using a Custom Java Validator Package

A Java package named *validator* consisting of several classes was created to navigate a supplied model and check if it meets a set of defined criteria or constraints. Tables 7 and 8 list a number of constraints that our package checks the model against. Table 7 shows constraints for some of the objects that might be present in our composed model (UML SD) while Table 8 lists general model constraints. Clearly these are not all the constraints defined in the UML specification [OMG09] for interaction diagrams. Some of our defined constraints are not even there in the specification. For example, in UML an interaction can have no lifelines, no messages, and no fragments. However, after composition it doesn't make sense for our sequence diagram to be empty. This is one of the main advantages of implementing custom Java classes to validate the composed model using specified criteria. We can choose what features we expect in the composed model. If designed properly, more constraints can easily be added to our *validator* package.

Table 7 shows that for the *Model* package object, we check if the object has a name, a message event (related to a message) and a collaboration (enclosing the interaction). The UML specification does not have the last two constraints for the *Model* class but we have included them in our checklist because we know that the composed model must have a collaboration enclosing the interaction, and that there should be events associated with the sending and receiving of messages. A collaboration must a have a name and an enclosed interaction. Every message has a pair of MOSs; therefore, we must have twice as many MOSs as there are messages. Table 8 shows a general constraint that is not tied to a specific element in the model but affects several elements. We know that a context specific aspect model is instantiated from a generic aspect model; therefore, if the instantiation is not properly done, some elements of the context specific aspect model advice may still have template parameters and wildcards. These will then appear in the composed model after composition; hence, we want to guard against this. Therefore, it also makes sense to check for our constraints much earlier and before composition by validating the context

specific aspect model too.

Class	Constraint
Model	• Must have at least one collaboration. • Must have at least one event (packaged elements). • Must have a name.
Collaboration	• Must have a name. • Should contain at least one *Interaction*. • Must have a property for each *Lifeline* in the *Interaction*. • Must be contained in a *Model* element.
MOS	• There should be twice as many MOSs as there are messages. • Should have a message. • Should have an *Event* (Send or Receive). • Should have covered *Lifeline*.
Interaction	• Must have a name. • Must have messages. • May contain MOSs directly or within optional CFs. • Must have at least one covered *Lifeline*.
Message	• Must have a name. • Must have two MOSs (*SendEvent* and *ReceiveEvent*).
Lifeline	• Must have a name. • Must have a *Property* . • Must be covered by MOS and/or CFs.

Class	Constraint
Property	• Should have name.
SOE and ROE	• Must have an *Operation*. • Must be contained in a *Model* package element. • Could have a name.

Table 7: Custom Model Validation Constraints

Cross-cutting Constraints
1. If instantiated from a generic aspect model then each *Class*, *Message*, *Class Operation*, and *Lifeline* should be properly bound; i.e., it should not contain the template parameter symbol "|" or wildcards (e.g. "*") or the string "UNKNOWN.

Table 8: Constraints for the Entire Model

Figure 5.1 shows the class diagram for our *validator* package that checks our model against the constraints given in the previous two tables. The classes in this package use the UML2 plug-in [UML2_Tutorial]. We have a class for each model element that we want to validate. For example, the *LifelineValidator* class checks a lifeline against the defined constraints. We have the *MessageValidator* for messages, *MOSValidator* for MOSs, etc. All these classes that validate individual model elements specialize the *Validator* abstract class; therefore, provide implementation for the *isValid* abstract method. This method returns true if the model element being checked meets the defined constraints.

The code snippet on the next page shows how the *CollaborationValidator* provides implementation for this method. The method checks that a collaboration has a name, on lines 5 to 8. On lines 10 to 13, it ensures that the collaboration contains an interaction. It finally checks if the collaboration is contained in a model element on line 15. It returns true if all the three constraints are met. Otherwise it returns false and an appropriate error

message as well.

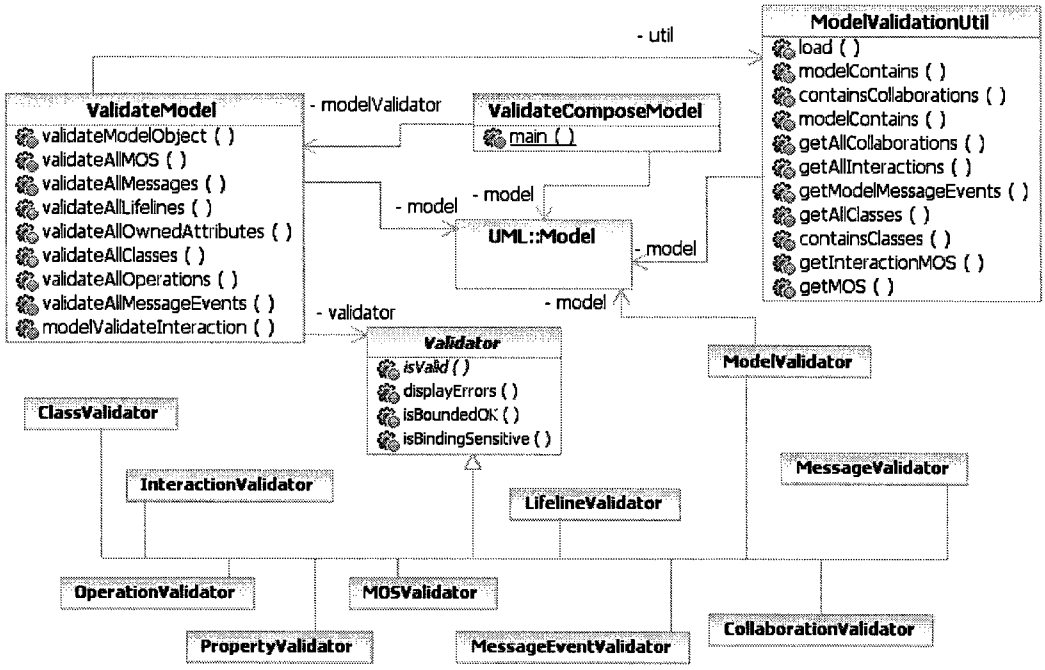


Figure 5.1: Validator Package Class Diagram.

```
                                    ...
1.   boolean isValid() {
2.      boolean valid = true;
3.      if (col != null){
4.          //Has a name
5.          if (col.getName().isEmpty() || !col.isSetName()){
6.                  valid = false;
7.                  this.errors.add("Must have name.");
8.          }
9.          //Contains at least one Interaction
10.         if (col.getOwnedBehaviors().isEmpty()){
11.                 valid = false;
12.                 this.errors.add("Must have at least one interaction.");
```

```
13.         }
14.         //Contained in a Model
15.         if (col.getModel() == null){
16.                 valid = false;
17.                 this.errors.add("Must be contained in a Model element.");
18.         }
19.     }
20.     return valid;
21. }
22.                                     ...
```

The *Validator* class provides implementation for the *isBoundedOK* method which ensures that a model element was bound properly by checking for the absence of "|" and "*". All the classes that extend *Validator* only validate one model element at a time. Our model may contain several instances of the same type element. The task to validate all the instances is delegated to the *ValidateModel*. This class has several methods including methods for validating all classes, all MOSs, all interactions, and others, as it can be seen in Figure 5.1. The *ValidateComposedModel* class in the entry point to our package. It loads the model from a file on disk into memory, and then makes a request to an instance of the *ValidateModel* class to validate all model elements.

The code snippet on the next page shows *ValidateComposedModel* validating an interaction, contained messages and lifelines. On line 2 it checks that the interaction retrieved is not null before validating. The class then validates the interaction model element on line 9. It then moves on to validate all the messages in the interaction by calling the *validateAllMessages* method on an instance of *ValidateModel*. It also checks all the lifelines in the interaction by invoking the *validateAllLifelines* method and printing a message to the standard output if all the lifelines are valid.

```
1.                                          ...
2.   if (sd == null){
3.      throw new Exception ("Unable to start Interaction validation");
4.   }
5.   sdName = sd.getName();
6.   if (modelValidator.modelValidateInteraction(sd)) {
7.         modelValidator.out("Interaction model element "+ sdName +
8.      " is valid :)");
9.      if (modelValidator.validateAllMOS(sd)){
10.           modelValidator.out("All Message Occurrence Specifications
11.           in " + sdName + " are valid :)");
12.     }
13.     else{
14.         modelOk = false;
15.     }
16.     if (modelValidator.validateAllMessages(sd)){
17.           modelValidator.out("All Messages in " + sdName +
18.            " are valid :)");
19.     }
20.     else{
21.         modelOk = false;
22.     }
23.     if (modelValidator.validateAllLifelines(sd)){
24.         modelValidator.out("All lifelines in " + sdName +
25.        " are valid :)");
26.     }
27.                                     ...
```

On the next page is a trace returned after running *ValidateComposedModel* on a composed model that violates a few constraints.

```
1.  Reading model Security Aspect Composed2 from disk …
2.  Validating Security Aspect Composed2
3.  Model container is valid
4.  All Classes are valid :)
5.  All Class Operations are valid :)
6.  All Message events are valid :)
7.  Collaboration model element Login is valid :)
8.  All Owned attributes in Login are valid :)
9.  Interaction model element Login is valid :)
10. There should be twice as many as MOS as there are messages
11. MessageOccurrenceSpecification(11) - Should have a lifeline. :(
12. Message(1) - Must have send message end MOS. :(
13. Lifeline(2) - Must have a represented object. :(
```

A few errors are returned on lines 10 to 13. Line 11 says there is MOS that does not cover any lifeline while line 13 says that the model has a lifeline (2nd lifeline) that does not have referenced property. On line 12, there is a message that does not have a *sendEvent*.

5.2.2 Using RSA 7.5

Validating a model in RSA is very simple. A tutorial of how this is done is given in Appendix C. After importing a model into RSA, we select the model, select **Modeling**, and then **Run validation**. Below is a trace from RSA when validating the same model validated by our *validator* previously.

```
Validation - 0 error(s), 0 warning(s), 0 informational message(s)
```

RSA says that the model does not violate any UML constraint which is probably true because a lifeline does not necessarily need to have an associated property or that every MOS should cover a lifeline. These are constraints we invented because in our sequence diagram models it does not make sense not to have such constraints. For example, in our

case a message cannot be sent without been received or received without been sent. We assume there are no lost or found messages.

5.3 Test Cases

Test cases (TCs) 1 to 9 involve composition of security and logging aspects with a primary model adapted from Klein et al. [Klein+09]. The aspects and the primary model are modified on each experiment to make the composition more complex, vary the number of join points (JPs), and also to test certain features. The Table 9 below gives a summary of these TCs. In all the test cases the composed models were all valid UML models and composition was properly achieved. The validity of the composed models was tested using RSA and our *validator* Java package, as mentioned earlier. Verifying composition was performed by generating a SD from the composed model, visually inspecting it, and by inspecting the XML code from the composed model file.

TC #	# JPs	Aim	Execution Time (sec)
1	2	Composition with multiple join points.	2
2	1	Messages with simple arguments.	2
3	1	Composition when we have ***alt*** CFs in both models.	1
4	1	Composition of ***loop*** CFs in both models.	1
5	0	Distinguishing between Synchronous and Asynchronous messages.	0.47
6	1	Composition of nested CFs.	2
7	2	Using the wildcard "*" for defining pointcuts.	5
8	2	Logging Aspect followed by a Security Aspect.	2
9	1	Security Aspect followed by a Logging Aspect.	2

Table 9: Summary of Test Cases

5.4 Case Studies

5.4.1 Phone Call features as Aspects

This case study of a cell phone application was adapted from Whittle and Jayaraman in [Whittle+07]. The application has three use cases but we are only interested in two; namely, *Receive a Call* and *Notify Call Waiting* [Whittle+07]. The *Receive a Call* use case is considered to be the base model and the *Notify Call Waiting* is considered the aspect [Whittle+07]. Figure 5.2 shows a dynamic view of the *Receive a Call* use case

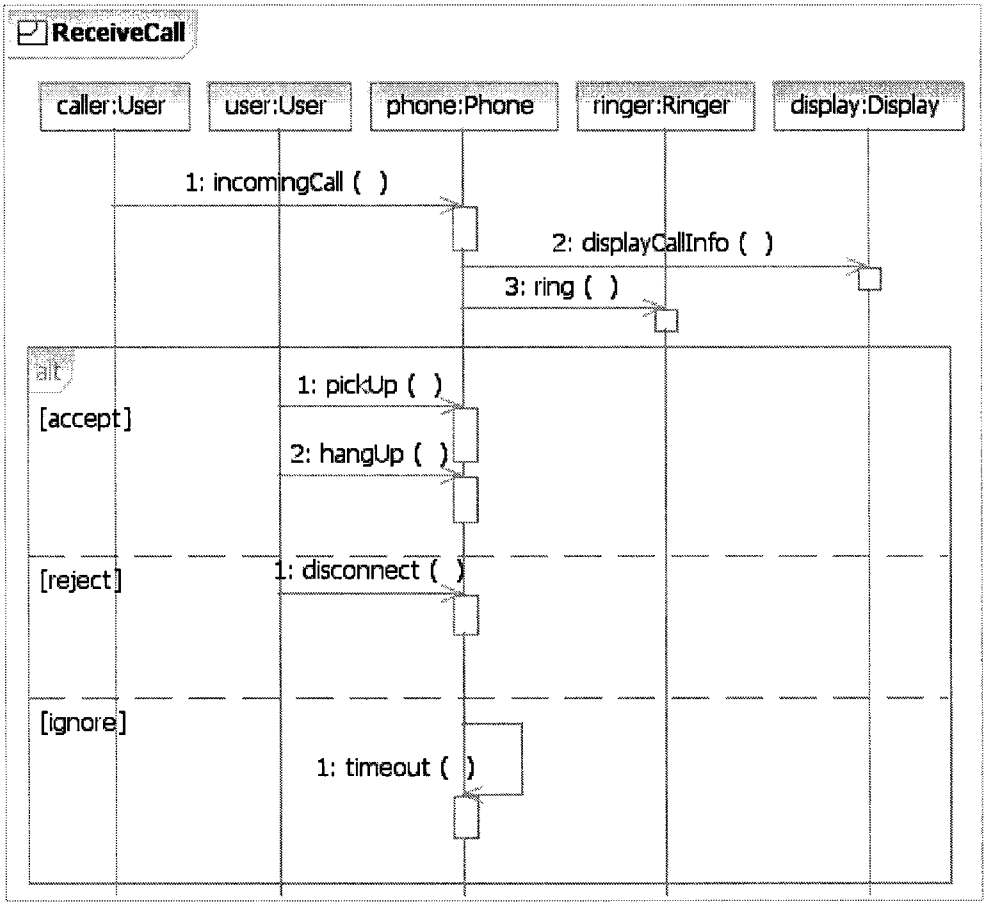
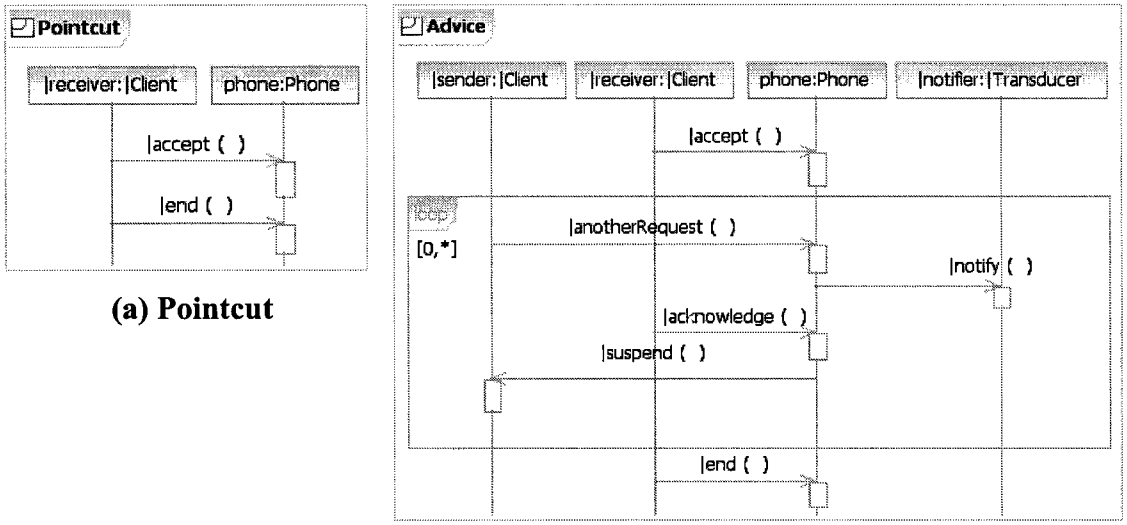


Figure 5.2: Receive a Call Primary Model Adapted from [Whittle+07].

modeled as a sequence diagram. This will be our primary model. When the user's phone receives a call (*incomingCall* message), it alerts the user by displaying the appropriate information about the caller on the phone's display [Whittle+07] by sending a *displayCall-*

Info message to the display. The phone also sends a *ring* message to the ringer. The user then has several options captured by an **alt** combined fragment. The user can either accept the call by sending a *pickUp* message to the phone and later end the call by sending a *hangUp* message [Whittle+07]. Alternatively, if the user chooses not to accept the call, the user can send a *disconnect* message to the phone. If the user elects to ignore the call, the phone will ring for a specified amount of time and then time out ending the scenario. As mentioned, the *Notify Call Waiting* scenario or feature is considered an aspect. The approach (graph transformations) taken by Whittle and Jayaraman does not have the notion of generic or context specific models like our approach. Therefore, Figure 5.3 shows our representation of the behavioral model of the *Notify Call Waiting* scenario as a generic aspect model.



(a) Pointcut

(b) Advice

Figure 5.3: Notify Call Waiting Generic Aspect Model.

The pointcut is defined as a sequence of parameterized |accept and |end messages from the |receiver lifeline to the phone lifeline. This will match a sequence of two messages that will be bound to |accept and |end from the lifeline bound to |receiver. The advice shown in Figure 5.3b is slightly more complex. It introduces messages that if bound properly, will place the current call on hold [Whittle+07]. The behavior defined by the advice is only applicable when the user is currently on call; therefore, we must ensure that the advice is weaved between the *pickUp* and *hangUp* messages on the primary mod-

el [Whittle+07]. To achieve this, we will bind |accept and |end to *pickUp* and *hangUp* respectively, as shown on lines 9 and 10 of the mark model below. The |receiver parameter is bound to *user* so the pointcut matches *pickUp* and *hangUp* from the user to the phone.

```
1.  <?xml version="1.0" encoding="ISO-8859-1"?>
2.  <xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
3.  xmlns="BindingDirectives">
4.    <BindingDirective  parameter="|receiver"    binding="user"/>
5.    <BindingDirective  parameter="|sender"      binding="caller"/>
6.    <BindingDirective  parameter="|anotherRequest" binding="incomingCall"/>
7.    <BindingDirective  parameter="|notify" binding="displayCallInfo"/>
8.    <BindingDirective  parameter="|acknowledge" binding="ok"/>
9.    <BindingDirective  parameter="|accept"      binding="pickUp"/>
10.   <BindingDirective  parameter="|end"         binding="hangUp"/>
11.   <BindingDirective  parameter="|suspend"     binding="putOnHold"/>
12.   <BindingDirective  parameter="|Client"      binding="User"/>
13.   <BindingDirective  parameter="|notifier"    binding="display"/>
14.   <BindingDirective  parameter="|Transducer"  binding="Display"/>
15. </xmi:XMI>
```

The rest of the parameters are bound as defined by the mark model. Recall that by binding these parameters from the generic aspect model, we are actually instantiating it to produce a context specific aspect model. This is done by the *Instantiate* ATL transformation as discussed in earlier chapters. However, before going into the trouble of instantiating a context specific aspect (and composing it with the primary model), we must first determine if the pointcut matches, and if so, how much join points were found. Getting the number of join points is performed by the *JoinPointsCount* transformation which takes the generic aspect, the primary and mark models as input, and produces a target model that contains the number of join points. Executing this transformation produces the model shown in Figure 5.4 (when viewed on Eclipse's uml editor). The model contains a *LiteralInteger* object with the name *NumberOfJoinpoints* and which has a value of one, that is,

our primary model has one join point.

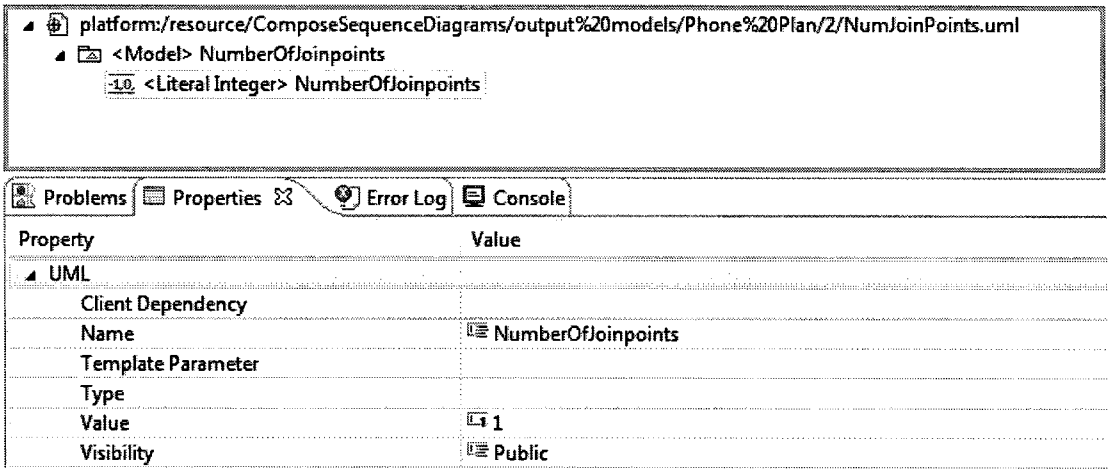


Figure 5.4: JoinPointsCount Output Model.

With only one join point, our composition algorithm only has to loop once; hence, no loop unrolling is required. We can then create an ant script, shown in the Appendix A.1, to execute the *Instantiate* and *Compose* transformations to generate the context specific aspect and composed models respectively. Below is a trace for the ant script.

```
loadMetamodels:
[am3.loadModel] INFO: Loading of model BD
PhonePlan:
[am3.loadModel] INFO: Loading of model PRIMARY
[am3.loadModel] INFO: Loading of model ASPECT
[am3.loadModel] INFO: Loading of model BIND
     [am3.atl] INFO: Executing ATL transformation Instantiate.asm
     [am3.atl] INFO: Executing ATL transformation Compose.asm
[am3.saveModel] INFO: Saving model CSAM1
[am3.saveModel] INFO: Saving model CM
BUILD SUCCESSFUL
Total time: 8 seconds
```

As it can be seen, the script takes about 8 seconds to execute the two transformations and save the output models to disk. The next step is to validate our models using both RSA and our custom *validator* package. Running the *ValidateComposedModel* class from our custom package to validate the composed model (CM) produces the output shown below.

```
Reading model CM from disk ...Validating CM
Model container is valid
All Classes are valid :)
All Class Operations are valid :)
All Message events are valid :)
Collaboration model element ReceiveCall is valid :)
All Owned attributes in ReceiveCall are valid :)
Interaction model element ReceiveCall is valid :)
All Message Occurrence Specifications in ReceiveCall are valid :)
All Messages in ReceiveCall are valid :)
All lifelines in ReceiveCall are valid :)
Our model and all its model elements meet our validation requirements
```

Running the *ValidateComposedModel* to validate the context specific aspect model (CSAM) also shows that the model is valid. The trace is given in the Appendix A.1.

Both models are valid according to the checklist defined in our *validator* package. Both models are then imported into RSA for validation and visual inspection. Validating both the composed model and CSAM in RSA gives the same output shown below. Therefore, we can confidently say that our models are valid.

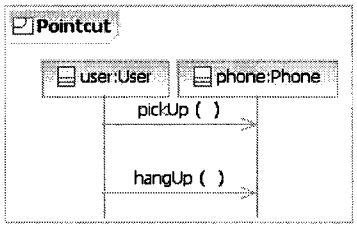
```
Validation - 0 error(s), 0 warning(s), 0 informational message(s)
```

Unfortunately, the tools cannot tell us that the composition was done properly. This will have to be done by visual inspection. We have several options to verify our composition. We can open the model file in text editor and inspect the *.uml* file which is in an XML format. We could also view the model file in the Eclipse uml editor which gives us

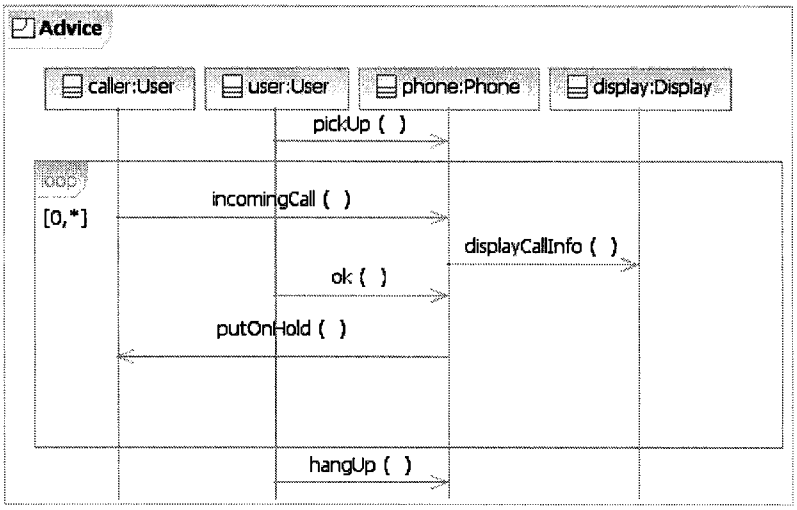
a view of the model elements hierarchy and properties as we saw in Figure 5.4. This is more user-friendly than code inspection. However, the best option would be to build a sequence diagram from the composed model *.uml* file. This way we can immediately see where the advice has been weaved. Given an imported valid UML model, RSA can create a sequence diagram for us. A tutorial on how to do this is given in Appendix D.

Figure 5.5 and 5.6 show the context specific aspect and composed model SDs created on RSA after importing the models. The pointcut was properly bound. We can see in Figure 5.5a that |accept and |end are bound to *pickUp* and *hangUp* while |receiver and |Client have been bound to *user* and *User* respectively. Figure 5.5b shows that the advice has also been bound as specified by the mark model.

Figure 5.6 shows a sequence diagram for our composed model. We can see that the advice has been properly weaved into the **alt** combined fragment's first operand. This way if the user chooses to accept a call and another *incomingCall* message is received by the phone, the phone's display will show the new caller's information. The user can then send an *ok* message to phone to put the caller on hold.



(a) Pointcut



(b) Advice

Figure 5.5: Context Specific Aspect Model.

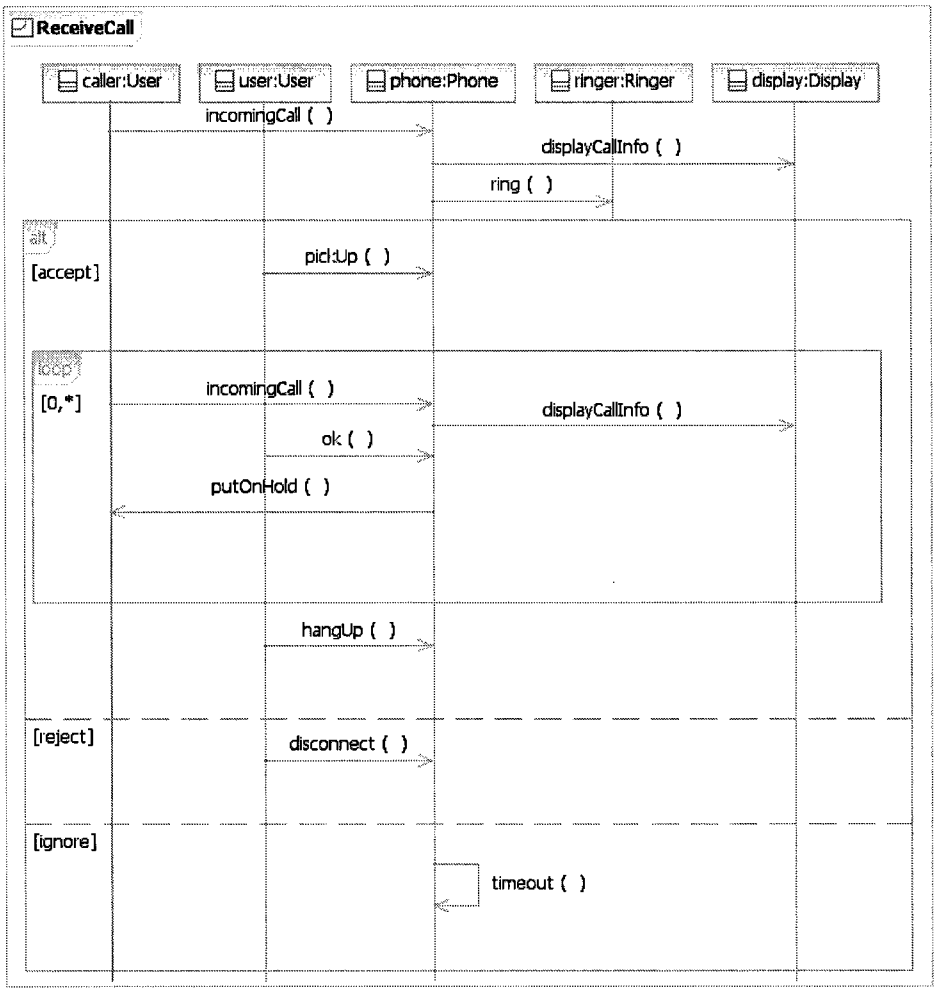


Figure 5.6: Composed Model.

5.4.2 Document Exchange Server (DES)

This case study of a DES system was adapted from Petriu et al. and Gong [Gong08]. Clients query and obtain documents from the server as it can be seen in Figure 5.7. The original case study focuses on a key scenario named *RetrieveDoc* [Petriu+07]. This scenario represents the retrieval of a document (from the server). It was initially modeled as a UML activity diagram by the researchers. However, in this work, we naturally used SDs, and in the process also made a few modifications. The sequence diagram for the *RetrieveDoc* scenario in Figure 5.7 is our primary model.

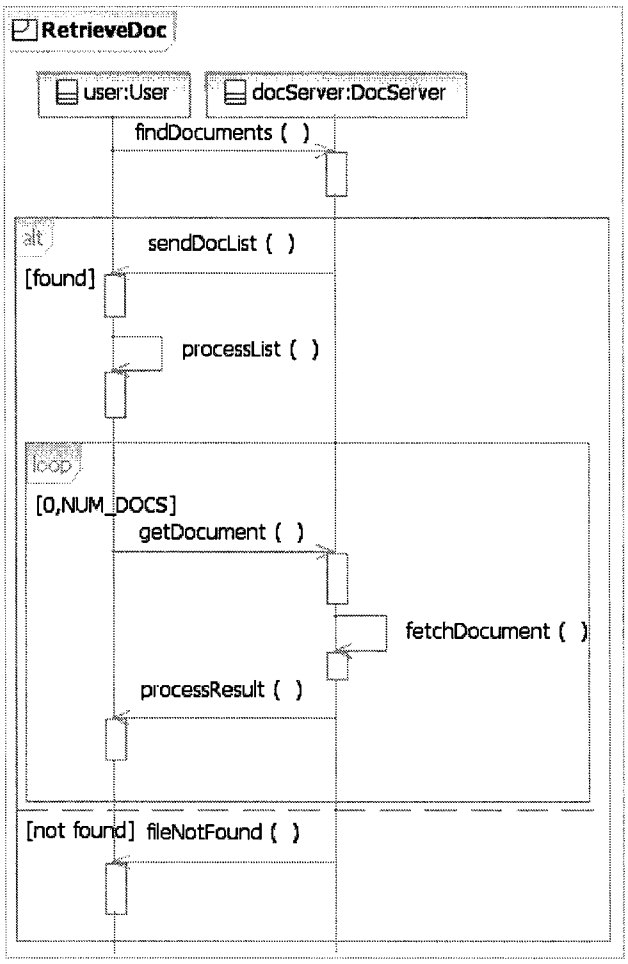
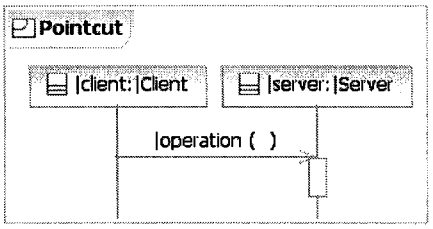


Figure 5.7: DES Primary Model.

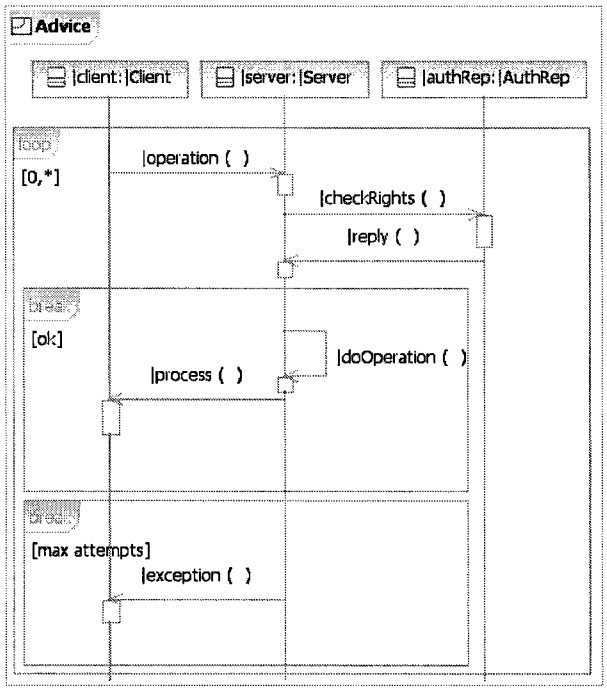
The user (client) begins by searching for a number of documents. If no documents are found the document server responds with a *fileNotFound* message, otherwise the server sends the user the list of file names. The conditional flow is modeled by an **alt** combined fragment. If the user receives the *sendDocList* message, the user processes that list and begins to retrieve the documents from the server by repeatedly sending a *getDocument* message to the server within the **loop** combined fragment. The server responds by fetching the required document and sending it the user via the *processResult* message.

As it is, our primary model (*RetrieveDoc* scenario) has no security implemented. Any user can request for documents from the DES server. Therefore, we want to extend the primary model with a security cross-cutting concern such that only authorized users are

permitted to fetch documents from the server [Petriu+07]. A possible solution to this security concern is an authorization aspect [Petriu+07]. Figure 5.8 shows the behavioral model (advice and pointcut SDs) of our generic authorization aspect model.



(a) Pointcut



(b) Advice

Figure 5.8: Generic Authorization Aspect Model.

The pointcut is designed to match a generic |operation message that requires authorization. We can then bind it to the *getDocument* message that we want secure. The advice in Figure 5.8b shows how the aspect implements authorization. It loops keeping track of the number of iterations. This is the number of times the |operation message is sent from the object that plays the role of |client to the object with the role of |server. On receiving |operation, |server checks if |client is authorized to request |operation by invoking |checkRights on an instance of |AuthRep. |AuthRep, which has user rights information, checks its database, and then replies to |server. If |user is authorized to invoke the |operation, |server can then run |doOperation, send a |process message back to |user and breaks out of the loop combined fragment. If |user does not have rights and has reached the maximum number of attempts for invoking |operation, |server sends a |exception message to |user

denying further access. Below is the mark model for instantiating the generic aspect model in the context of the scenario.

```
1.  <?xml version="1.0" encoding="ISO-8859-1"?>
2.  <xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
3.  xmlns="BindingDirectives">
4.   <BindingDirective  parameter="|operation"    binding="getDocument"/>
5.    <BindingDirective parameter="|reply"        binding="acceptResult"/>
6.   <BindingDirective parameter="|doOperation"   binding="fetchDocument"/>
7.   <BindingDirective parameter="|process"       binding="processResult"/>
8.   <BindingDirective parameter="|exception" binding="processException"/>
9.   <BindingDirective parameter="|checkRights" binding="checkUserRights"/>
10.  <BindingDirective parameter="|Server"       binding="DocServer"/>
11.  <BindingDirective  parameter="|Client"      binding="User"/>
12.  <BindingDirective parameter="|AuthRep" binding="DocAuthorizationRep"/>
13.  <BindingDirective  parameter="|authRep"     binding="authRep"/>
14.  <BindingDirective parameter="|server" binding="docServer"/>
15.  <BindingDirective  parameter="|client"      binding="user"/>
16. </xmi:XMI>
```

It can be seen on line 4 that |operation is bound to the *getDocument* message that we want to secure. On lines 10 and 11, |Server and |Client are bound to *DocServer* and *Client* while |AuthRep is bound *DocAuthorizationRep*, on line 12. The rest of the parameters are bound as defined by the mark model above.

Before authoring an ant script to execute the *Instantiate* and *Compose* transformation, we must first find the number join points. Our primary model only has one *getDocument* message, so it not surprising that the *JoinPointsCount* transformation returned one join point as shown by its output model shown in Figure 5.9.

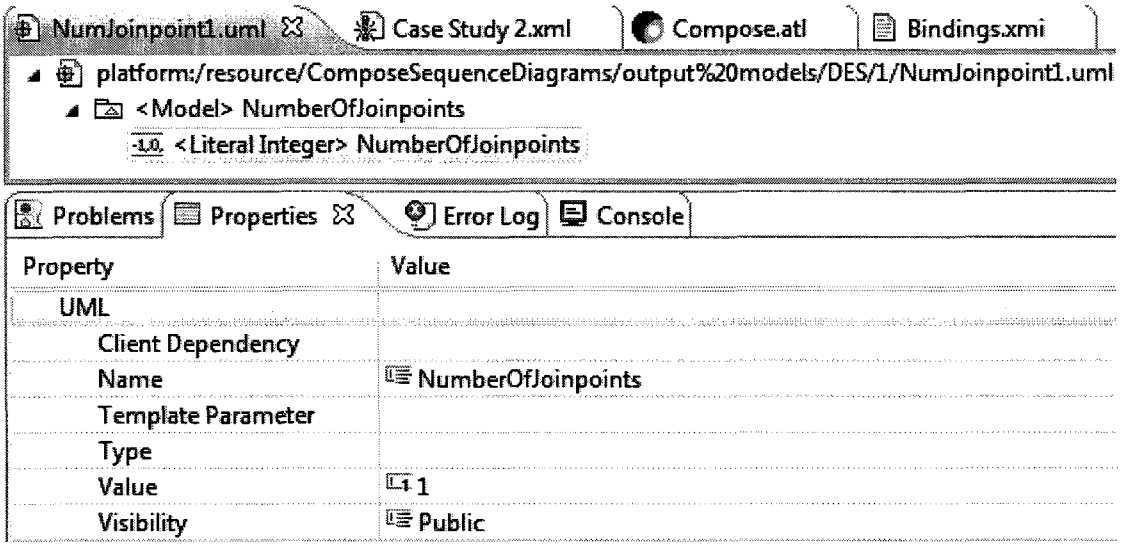
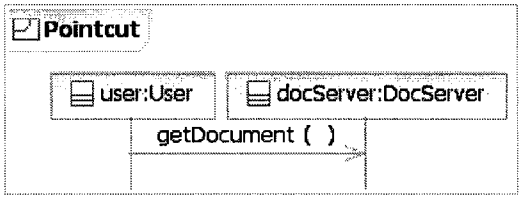


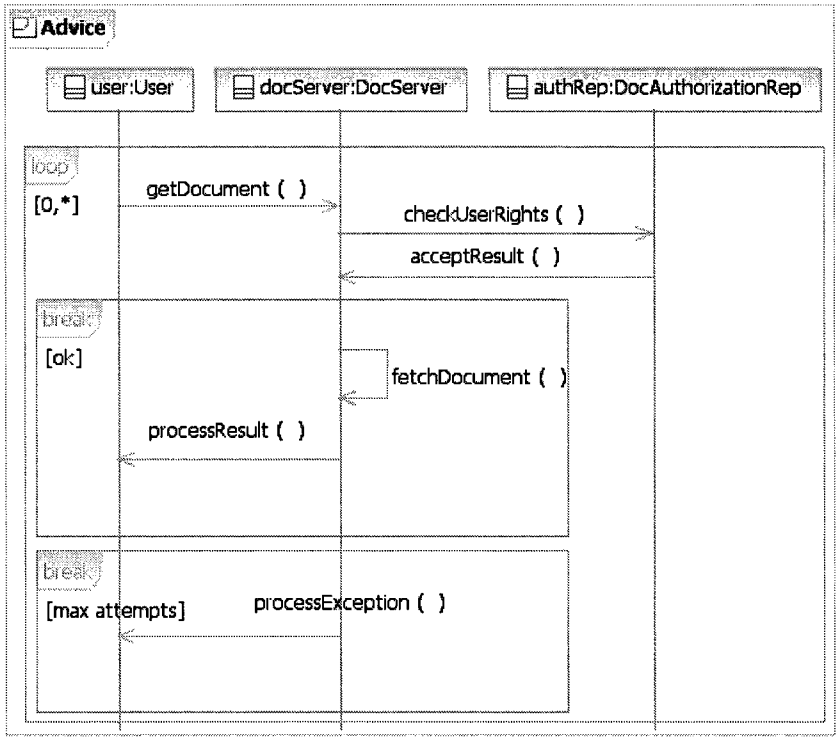
Figure 5.9: JoinPointsCount Output Model.

Just like in the previous case study, no loop unrolling is required. So a simple ant script in Appendix A.2 executes both *Instantiate* and *Compose* transformations to produce the context specific aspect model and composed model respectively. Validating both models using the *ValidateComposedModel* class from our *validator* package yields the traces in Appendix A.2. Both models meet our defined constraints. Importing and validating the two models in RSA returns no errors or warnings; therefore, both models are valid UML SDs. Generating SDs from these models yields the aspect models SDs in Figure 5.10, and the composed model in Figure 5.11. Visual inspection verifies that the generic aspect was properly instantiated as defined by the mark model.

The composed model in Figure 5.11 shows a slight problem with the composition. The aspect weaves the *fetchDocument* and *processResult* messages in the **break** combined fragment nested within the **loop** combined fragment. Therefore if *user* has rights, the *docServer* will fetch the requested document, send it to user and break out of the inner loop. The problem is the second pair of *fetchDocument* and *processResult* messages highlighted in Figure 5.11. This pair from the primary model should have been deleted.



(a) Pointcut



(b) Advice

Figure 5.10: Authorization Context Specific Aspect Model.

The cause of this slight anomaly arises from the nature of our composition approach. If we want a message or any model element from primary model to be deleted during composition (like *fetchDocument* and *processResult* messages) then those model elements should be part of the pointcut and not present in the advice. Therefore, to fix the problem, we must have both *fetchDocument* and *processResult* messages in the pointcut before composition.

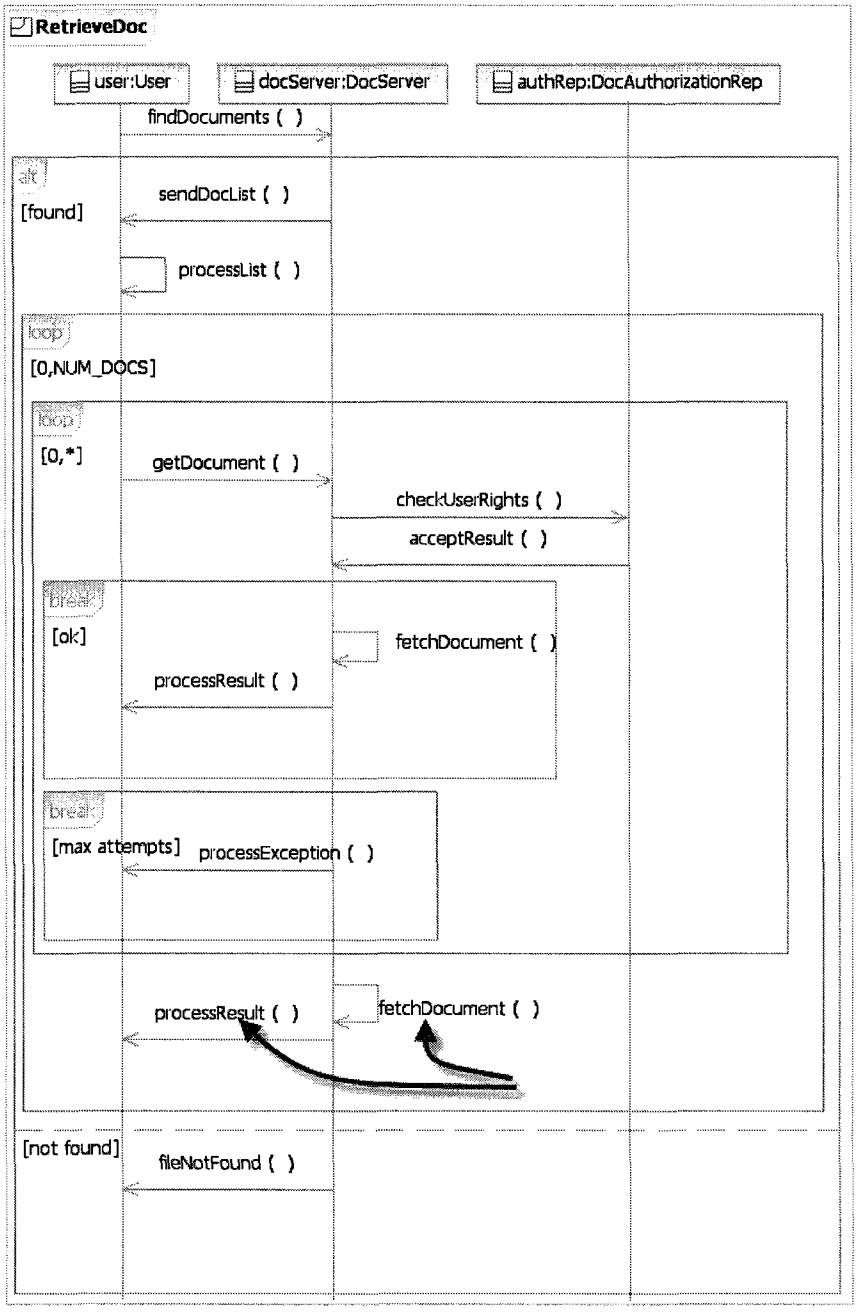


Figure 5.11: Retrieve Doc with Authorization Composed Model.

Figure 5.12 shows how the generic aspect model pointcut is updated to accommodate the new changes. That is the only change that is needed. The mark model remains the same.

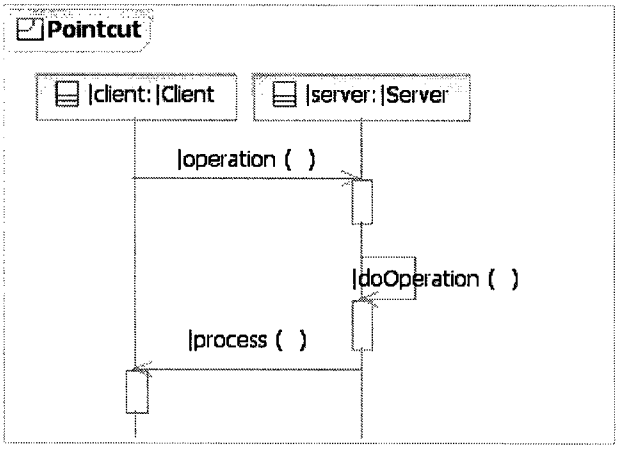


Figure 5.12: Modified Authorization Generic Aspect Pointcut.

The |doOperation and |process are to be bound to *fetchDocument* and *processResult* just like in the advice. With the new pointcut we still get one join point. Running the previous ant script gives valid context specific aspect and composed models. Generating a sequence diagram for the context specific model gives the pointcut SD in Figure 5.13.

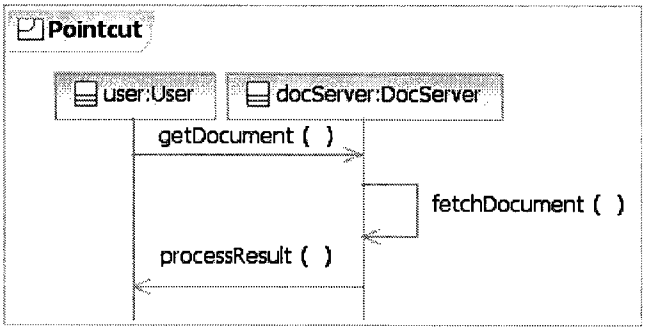


Figure 5.13: Modified Authorization Context Specific Aspect Pointcut.

The |doOperation and |process messages are bound to *fetchDocument* and *processResult* while |operation is bound to *getDocument*, like before. The advice SD remains unchanged, and is identical to the previous one in Figure 5.10b.

Figure 5.14 shows the final composed model. The authorization security aspect has been successfully composed with our primary model. Now only authorized users will be permitted to invoke the *getDocument* message. Otherwise, after a defined number of attempts (max_attempts) the user will be sent a *processException* message terminating the request for a specific document.

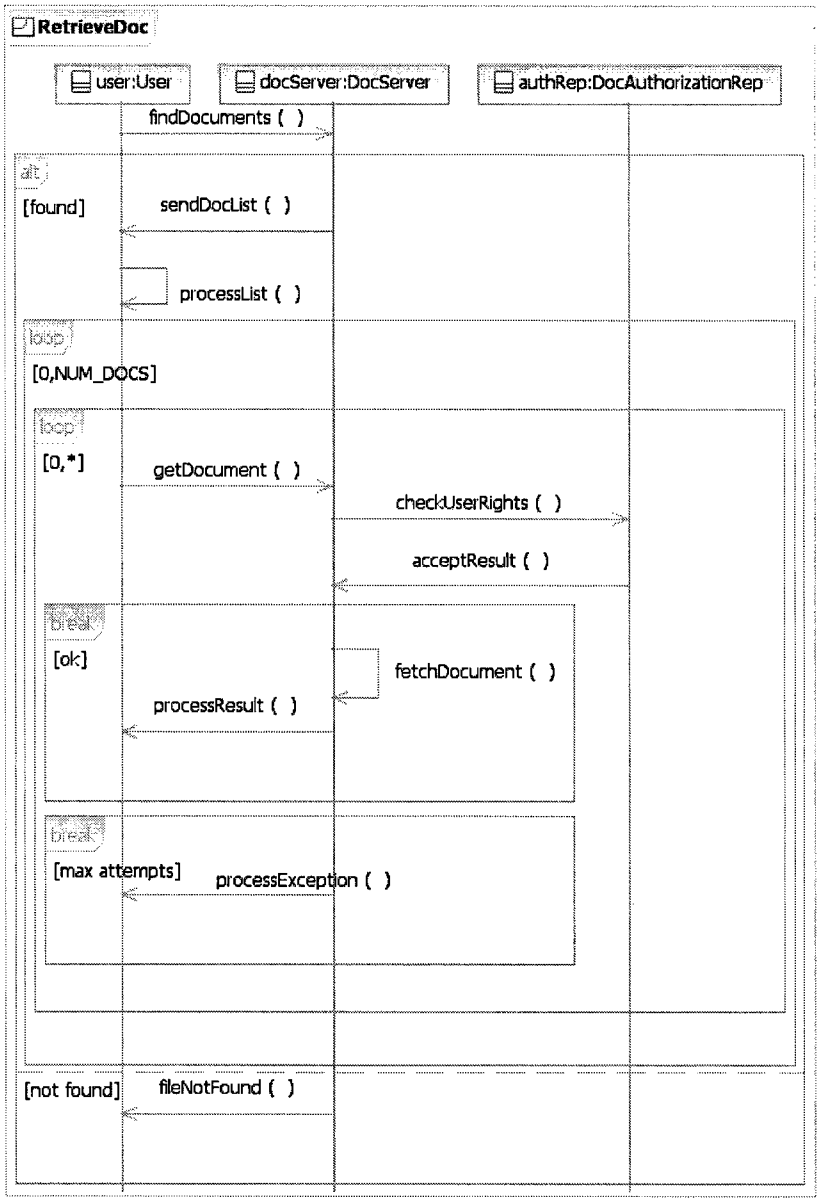


Figure 5.14: Retrieve Doc with Authorization Composed Model.

Modularizing the cross-cutting security concern into an aspect gives us the flexibility to change our authorization policy or implementation without having to redesign the core system. This is the essence of aspect oriented techniques like AOM.

Chapter 6 : Conclusions and Future Work

6.1 Conclusions

The main objective of this thesis is to compose aspect models represented as UML sequence diagrams (SDs) using the Atlas Transformation Language (ATL). Toward this end, we proposed a formal definition of SDs in terms of an ordered list of interaction fragments, and in the process defined three algorithms for pointcut detection, advice composition and complete composition. We designed and implemented the Complete Composition algorithm to achieve composition of the primary model and generic aspect models. We consider aspect composition as a form of model transformation; therefore, the algorithm is implemented using model transformations written in the ATL model transformation language. We also designed a simple metamodel in Ecore for mark models used to define binding rules which are used to instantiate generic aspect models. We finally designed and implemented a custom Java package to help validate the composed model. The Java classes check the composed model elements against a list of defined constraints designed to ensure that essential model elements are present in the composed model and are properly initialized.

The Complete Composition Algorithm proposed and implemented in this thesis composes behavioral views of both primary and aspect models represented as UML sequence diagrams. The primary model defines the core system behavior without cross-cutting concerns while the aspect models represent behavior that cross-cuts the primary model. The models are described in UML Sequence diagrams created using an eclipse-based modeling tool (RSA) and then exported to a UML2.1 file for composition and validation.

Using ATL, a mature model transformation language, the Complete Composition Algorithm is implemented using three transformation models; namely *JoinPointsCount*, *Instantiate*, and *Compose*. The *JoinPointsCount* transformation determines the number of join points in the primary model given the pointcut from the aspect model. The aspect models are made generic so that they can be more reusable; therefore, they must first be

instantiated before they can be composed with the primary model. The *Instantiate* transformation is used to instantiate generic aspect models in the context of the application using a set of binding rules defined in mark models to produce context specific aspect models. The *Compose* transformation then takes the primary model and context specific aspect model as inputs, and produces a composed model. This process is repeated as many times as there are join points and aspect models until a complete integrated system is obtained.

To test our design and implementation, several test cases and case studies were successful conducted. Validation and verification of the composed model is vital. Validation was achieved by using custom Java classes to check the model against a set of defined constraints. The composed model was also validated using RSA's built-in validation feature. To verify composition, a sequence diagram was generated from the model's UML2.1 file using RSA. The generated sequence diagram was then visually inspected to see if the composition was performed properly.

Using ATL for composing models does have its challenges. The inability to navigate target models or modify input models makes intricate weaving of aspects a hard problem. However, the benefits of using a versatile language that allows for powerful expressions may outweigh the challenges.

6.2 Limitations and Future Work

The work done in this thesis is part of an ambitious quest for a complete AOM composition framework and a set of tools that can allow software architects and developers to easily apply AO techniques to model driven software development. There are several limitations that must be addressed, and new features to be added before our composition approach can be more useful. These include:

- Improved string pattern definition and matching for template parameters.
- Non primitive message and operation argument types.
- Support for Interaction Occurrences.

- Support for BehaviorExecutionSpecifications (BESs), ExecutionOccurrenceSpecifications (EOSs) and other model elements.
- Structural view composition.
- Invoking ATL from Java.
- An eclipse plug-in to help in the creation of the mark model.

Our approach currently supports the use of the wildcard "*" for defining template parameters. This gives some flexibility when defining generic aspect models. However, to allow for powerful expressions, we need to use regular expressions. Currently ATL (version 2.0.x) does not support the use of regular expressions for comparing or matching strings. ATL only uses regular expressions for replacing and splitting strings. Future research may include development of a custom string ATL library that will provide helpers that implement regular expression matching operations.

As mentioned in Chapter 3, we have assumed that messages in the primary and aspect models have simple arguments that are either strings or integers. However, messages can have arguments that are instances of classes defined in the structural view of the system. Therefore, the current use of lazy rules to create message arguments (and class operation parameters) may not be ideal. Future work would look at a more efficient and elegant way of creating message arguments in the target model.

Interaction Occurrences provide a way to reuse and manage complex SDs. They are a notation for copying one SD (basic) into another one, which may be larger [Pilone+05]. Our current composition approach has no support processing interaction occurrences; therefore, future research would look into including interaction occurrences in pointcut detection and composition. This will provide challenges because the use of interaction occurrences means that the primary model SD or the aspect model SDs may contain more than one instance of *Interaction* depending on the number of interaction occurrences.

Recall that BESs and EOSs were ignored in our composition approach. Future work could look into how these model elements can be processed with other interaction frag-

ments. We would also add support for other model elements that are not currently supported; like, connectors, signals (and related events), gates, etc.

Future research would also include composition of other behavioral views (Statechart and Activity diagrams) and structural views (class diagrams) of the primary model and aspect models. Our current approach does achieve some composition of structural model elements from the primary and aspect models but does ignores associations between the structural elements.

As mentioned in Chapter 4, invoking ATL from Java would help automate the composition process. It would also allow for easy integration with a custom validation Java package or plug-in like the one that was developed to validate the composed models. Future research would involve designing, building, and integrating an Eclipse plug-in that can achieve this. Another plug-in would be built to allow architects and developers to easily create mark models without having to deal with XMI code. The plug-in could have a user interface that can allow users to pair up template parameters with bindings.

References

[AM3_Tasks_Wiki] Eclipse ATLAS MegaModel Management Ant Tasks website, http://wiki.eclipse.org/AM3_Ant_Tasks, last accessed on September 22, 2009.

[AM3_Wiki] Eclipse ATLAS MegaModel Management website, http://wiki.eclipse.org/AM3, last accessed on September 22, 2009.

[Ant_Manual] Apache Ant website, http://ant.apache.org/, last accessed on September 22, 2009.

[ATL_Examples] ATL Basic Examples and Patterns website, http://www.eclipse.org/m2m/atl/basicExamples_Patterns/ last accessed on September 22, 2009.

[ATLHome] ATL Project website, http://www.eclipse.org/m2m/atl/, last accessed on September 22, 2009.

[ATLUserGuide] http://wiki.eclipse.org/ATL/User_Guide, last accessed on September 22, 2009.

[ATL_Manual] ATLAS group LINA & INRIA Nantes, "ATL User Manual version 0.7," Online resource available at : http://www.eclipse.org/m2m/atl/doc/ATL_User_Manual[v0.7].pdf , last accessed on September 22, 2009.

[Barais+08] O. Barais, J. Klein, B. Baudry, A. Jackson, and S. Clarke, "Composing Multi-View Aspect Models," In Proceedings of the Seventh International Conference on Composition-Based Software Systems, pages 43-52, 2008.

[Colyer+05] A. Colyer, A. Clement, G. Harley, M. Webster, "Eclipse AspectJ. Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools,"Addison Wesley Professional, Decem-

ber 14, 2004, ISBN : 0-321-24587-3

[Didonet Del Fabro+06] M. Didonet Del Fabro, J. Bézivin, and P. Valduriez, "Weaving Models with the Eclipse AMW plug-in," In: Eclipse Modeling Symposium, Eclipse Summit Europe 2006, Esslingen, Germany, 2006.

[EclipseProject] Eclipse project website, http://www.eclipse.org, last accessed on September 22, 2009.

[Fleurey+07] F. Fleurey, B. Baudry, R. France, and S. Ghosh, "A Generic Approach for Automatic Model Composition," Lecture Notes In Computer Science archive Models in Software Engineering: Workshops and Symposia at MoDELS 2007, Nashville, TN, USA, September 30 - October 5, 2007,pages 7–15, 2008, Springer-Verlag Berlin, Heidelberg.

[France+04] R. France, I. Ray, G. Georg and S. Ghosh, "Aspect-Oriented Approach to Design Modeling," IEEE Proceedings – Software, Special Issue on Early Aspects: Aspect –Oriented Requirements Engineering and Architecture Design, 151(4):173-185, August 2004.

[Gong08] H. Gong, "Composition of Aspects Represented as UML Activity Diagrams.", Master's thesis, Carleton University, 2008.

[Groher+07] I. Groher, and M. Voelter. "XWeave: models and aspects in concert," In Proceedings of the 10th international workshop on Aspect-oriented modeling, March 2007.

[Hamilton+06] K. Hamilton, and R. Miles, "Learning UML 2.0", O'Reilly, April 2006, ISBN-10: 0-596-00982-8 .

[Happe+09] J. Happe, S. Becker, C. Rathfelder, H. Friedrich, and R. H. Reussner, "Parametric Performance Completions for Model-Driven Performance Prediction," Journal of Systems and Software, Volume 82 Issue 1, January 2009, Elsevier Science Inc.

[Jacobson+04] I. Jacobson, and P. Ng, "Aspect-oriented software development with use case," Addison-Wesley Professional, December 30, 2004, ISBN-10: 0321268881.

[Jeanneret+08] C. Jeanneret, R. France, and B. Baudry, "A reference process for model composition," In Proceedings of the 2008 AOSD workshop on Aspect-oriented modeling, April 2008.

[Jézéquel06] J.M. Jézéquel, "Model Transformation Techniques Model Techniques," Online resource available at: http://modelware.inria.fr/static_pages/slides/ModelTransfo.pdf, last accessed in August 30, 2009

[Jouault+05] F. Jouault, and I. Kurtev, "Transforming Models with ATL," In proceedings of the Model Transformation in Practice Workshop, October 3rd 2005.

[Kiczales+97] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-Oriented Programming," In ECOOP'97: In Proceedings of the 11th European Conference on Object-Oriented Programming, volume 1241, pages 220–242, Berlin, Heidelberg, and New York, 1997, Springer-Verlag.

[Kiczales+05] G. Kiczales, and M. Mezini, "Aspect-Oriented Programming and Modular Reasoning," In ICSE '05: Proceedings of the 27th international conference on Software engineering, May 15–21, 2005, pages: 49-58, St. Louis, USA

[Kienzle+09] J. Kienzle, W. A. Abed, and J. Klein, "Aspect-oriented multi-view modeling," In Proceedings of the 8th ACM international conference on Aspect-oriented software development, March 2009.

[Klein+06] J. Klein, L. Hélouët, and J.M. Jézéquel, "Semantic-based Weaving of Scenarios," AOSD 06, March 20-24, Bonn, Germany, 2006

[Klein+07] J. Klein, F. Fleurey, and J.M. Jézéquel, "Weaving Multiple Aspects

in Sequence Diagrams," Transactions on AOSD III, LNCS 4620, pp. 167–199, 2007

[Kompose] Kompose website, http://www.kermeta.org/kompose/ last accessed on September 23, 2009.

[Minanovic07] M. Milanovic, "Complete ATL Bundle for launching ATL transformations programmatically." Online resource available at : http://milan.milanovic.org/download/atl.zip, last accessed on September 10, 2009.

[Morin+08] B. Morin, J. Klein, O. Barais, and J.M. Jézéquel, "A generic weaver for supporting product lines," In proceedings of the 13th international workshop on Software architectures and mobility, Leipzig, Germany, May 2008.

[OMG07] Object Management Group, "UML Superstructure Specification, v2.1.1," Online resource available at : http://www.omg.org/docs/formal/07-02-05.pdf, last accessed last accessed on July 20, 2009.

[OMG09] Object Management Group, "UML Superstructure Specification, v2.2," Online resource available at : http://www.omg.org/docs/formal/09-02-02.pdf, last accessed on August 22, 2009.

[Oaw] OpenArchitectureware website, http://www.openarchitectureware.org/ last accessed on September 23, 2009.

[Petriu+07] D.C. Petriu, H. Shen, and A. Sabetta, "Performance Analysis of Aspect- Oriented UML Models", Software and Systems Modeling (SoSyM), Vol. 6, No. 4., pages. 453-471, 2007, Springer Berlin / Heidelberg

[Pilone+05] D. Pilone, and N. Pitman, "UML 2.0 in a Nutshell", O'Reilly, (June 2005) ISBN: 0-596-00795-7

[Straw+04] G. Straw, G. Georg, E. Song, S. Ghosh, R. France, and J. M. Bieman, "Model Composition Directives," LNCS 3273, pages 84-97, 2004, Springer Berlin / Heidelberg

[SYSC 5708] SYSC 5708 (ELG 6178) Model-Driven Development of Real-Time and Distributed Software course website,

http://www.sce.carleton.ca/courses/94578/index.html, last accessed on September 30, 2009.

[UML2_Project] UML2 Eclipse project website, http://www.eclipse.org/modeling/mdt/?project=uml2, last accessed on September 22, 2009.

[UML2_Tutorial] Getting Started with UML2 website, http://www.eclipse.org/modeling/mdt/uml2/docs/articles/Getting_Started_with_UML2/article.html, last accessed on September 22, 2009.

[UML2_Wiki] Eclipse MDT/UML2 website,

http://wiki.eclipse.org/MDT-UML2, last accessed on September 22, 2009.

[Whittle+06] J. Whittle, J. Araújo, and A. Moreira, "Composing aspect models with graph transformations," In Proceedings of the 2006 international workshop on Early aspects at ICSE, pages 59-65, Shanghai, China, 2006

[Whittle+07] J. Whittle, and P. Jayaraman, "MATA: A Tool for Aspect-Oriented Modeling Based on Graph Transformation," At MoDELS'07: 11th International Workshop on Aspect-Oriented Modeling, Nashville TN USA, Oct 2007.

[Woodside+09] M. Woodside, D.C. Petriu, D.B. Petriu, J. Xu, T. Israr, G. Georg,

R. France, J.M. Bieman, S.H. Houmb, and J. Jürjens, "Performance analysis of security aspects by weaving scenarios extracted from UML models," Journal of Systems and Software, Volume 82 , Issue 1 (January 2009), pp 56-74, 2009

Appendix A Case Studies

A.1 Case Study 1

Composition ant script – CaseStudy1.xml

```
<project name="CaseStudy1" default="PhonePlan">
  <property name="outPath" value="../output models/Phone Plan/2/" />
  <property name="inPath" value="../input models/Phone Plan/" />
  <target name="PhonePlan" depends="loadMetamodels">
      <am3.loadModel modelHandler="EMF" name="PRIMARY" metamodel="UML2"
path="${inPath}Primary Model.uml" />
      <am3.loadModel modelHandler="EMF" name="ASPECT" metamodel="UML2"
path="${inPath}Generic Notify Call Waiting Aspect Model.uml" />
      <am3.loadModel modelHandler="EMF" name="BIND" metamodel="BD" path="${in-
Path}Bindings.xmi" />
            <!-- Instantiate aspect -->
  <am3.atl path="Instantiate.asm" allowInterModelReferences="true">
      <inModel name="UML2" model="UML2"/>
      <inModel name="BD" model="BD"/>
      <inModel name="PRIMARY" model="PRIMARY"/>
      <inModel name="ASPECT" model="ASPECT"/>
      <inModel name="BIND" model="BIND"/>
      <outModel name="CONTEXTSPECIFIC" model="CSAM1" metamodel="UML2"/>
      <library path="PointcutMatchHelpers_v2.0.asm"/>
  </am3.atl>
            <!-- Compose -->
  <am3.atl path="Compose.asm" allowInterModelReferences="true">
      <inModel name="UML2" model="UML2"/>
      <inModel name="PRIMARY" model="PRIMARY"/>
      <inModel name="ASPECT" model="CSAM1"/>
```

```
      <outModel name="COMPOSED" model="CM" metamodel="UML2"/>
            <library path="PointcutMatchHelpers_v2.0.asm"/>
      </am3.atl>
      <!-- Save models to disk -->
      <am3.saveModel model="CSAM1" path="${outPath}CSAM.uml"/>
      <am3.saveModel model="CM" path="${outPath}CM.uml"/>
      </target>
      <target name="loadMetamodels">
            <!-- Load UML metamodel -->
            <am3.loadModel  modelHandler="EMF"  name="UML2"  metamodel="MOF"
nsUri="http://www.eclipse.org/uml2/2.1.0/UML"/>
            <am3.loadModel  modelHandler="EMF"  name="BD"  metamodel="MOF"
path="../metamodels/BindingDirectives.ecore" />
      </target>
</project>
            <!-- Load UML metamodel -->
<am3.loadModel      modelHandler="EMF"      name="UML2"      metamodel="MOF"
nsUri="http://www.eclipse.org/uml2/2.1.0/UML"/>
<am3.loadModel modelHandler="EMF" name="BD" metamodel="MOF" path="../metamod-
els/BindingDirectives.ecore" />
      </target>
</project>
```

Receive a Call Scenario Validation Trace from *Validator*

```
Reading model CM from disk ...Validating CM
Model container is valid
All Classes are valid :)
All Class Operations are valid :)
All Message events are valid :)
Collaboration model element ReceiveCall is valid :)
```

```
All Owned attributes in ReceiveCall are valid :)

Interaction model element ReceiveCall is valid :)

All Message Occurrence Specifications in ReceiveCall are valid :)

All Messages in ReceiveCall are valid :)

All lifelines in ReceiveCall are valid :)

Our model and all its model elements meet our validation requirements
```

Notify Call Waiting Validation Trace from Validator.

```
Reading model CSAM from disk ...

Validating CSAM

Model container is valid

All Classes are valid :)

All Class Operations are valid :)

All Message events are valid :)

Collaboration model element Pointcut is valid :)

All Owned attributes in Pointcut are valid :)

Interaction model element Pointcut is valid :)

All Message Occurrence Specifications in Pointcut are valid :)

All Messages in Pointcut are valid :)

All lifelines in Pointcut are valid :)

Collaboration model element Advice is valid :)

All Owned attributes in Advice are valid :)

Interaction model element Advice is valid :)

All Message Occurrence Specifications in Advice are valid :)

All Messages in Advice are valid :)

All lifelines in Advice are valid :)

Our model and all its model elements meet our validation requirements
```

A.2 Case Study 2

Composition ant script - CaseStudy2.xml

```
<project name="CaseStudy2" default="DES">
	<property name="outPath" value="../output models/DES/1/" />
	<property name="inPath" value="../input models/DES/" />
	<target name="DES" depends="loadMetamodels">
	<am3.loadModel modelHandler="EMF" name="PRIMARY" metamodel="UML2"
path="${inPath}Primary Model.uml" />
	<am3.loadModel modelHandler="EMF" name="ASPECT" metamodel="UML2"
path="${inPath}Generic Authorization Aspect Model.uml" />
	<am3.loadModel modelHandler="EMF" name="BIND" metamodel="BD" path="${in-
Path}Bindings.xmi" />
	<am3.atl path="Instantiate.asm" allowInterModelReferences="true">
		<inModel name="UML2" model="UML2"/>
		<inModel name="BD" model="BD"/>
		<inModel name="PRIMARY" model="PRIMARY"/>
		<inModel name="ASPECT" model="ASPECT"/>
		<inModel name="BIND" model="BIND"/>
<outModel name="CONTEXTSPECIFIC" model="CSAM1" metamodel="UML2"/>
			<library path="PointcutMatchHelpers_v2.0.asm"/>
		</am3.atl>
	<am3.atl path="Compose.asm" allowInterModelReferences="true">
			<inModel name="UML2" model="UML2"/>
			<inModel name="PRIMARY" model="PRIMARY"/>
			<inModel name="ASPECT" model="CSAM1"/>
		<outModel name="COMPOSED" model="CM" metamodel="UML2"/>
			<library path="PointcutMatchHelpers_v2.0.asm"/>
		</am3.atl>
```

```
<am3.saveModel model="CSAM1" path="${outPath}Authorization CSAM1.uml"/>

<am3.saveModel model="CM" path="${outPath}Authorization CM1.uml"/>

      </target>

      <target name="loadMetamodels">

            <!-- Load UML metamodel -->

            <am3.loadModel  modelHandler="EMF"  name="UML2"  metamodel="MOF"
nsUri="http://www.eclipse.org/uml2/2.1.0/UML"/>

            <am3.loadModel  modelHandler="EMF"  name="BD"  metamodel="MOF"
path="../metamodels/BindingDirectives.ecore" />

      </target>

</project>
```

Case Study 2.xml trace

```
loadMetamodels:

[am3.loadModel] INFO: Loading of model BD

DES:

[am3.loadModel] INFO: Loading of model PRIMARY

[am3.loadModel] INFO: Loading of model ASPECT

[am3.loadModel] INFO: Loading of model BIND

     [am3.atl] INFO: Executing ATL transformation Instantiate.asm

     [am3.atl] INFO: Executing ATL transformation Compose.asm

[am3.saveModel] INFO: Saving model CSAM1

[am3.saveModel] INFO: Saving model CM

BUILD SUCCESSFUL

Total time: 8 seconds
```

Primary Model Validation Trace from *Validator*

```
Reading model Authorization CM1 from disk ...

Validating Authorization CM1

Model container is valid
```

```
All Classes are valid :)

All Class Operations are valid :)

All Message events are valid :)

Collaboration model element RetrieveDoc is valid :)

All Owned attributes in RetrieveDoc are valid :)

Interaction model element RetrieveDoc is valid :)

All Message Occurrence Specifications in RetrieveDoc are valid :)

All Messages in RetrieveDoc are valid :)

All lifelines in RetrieveDoc are valid :)

Our model and all its model elements meet our validation requirements
```

Context Specific Aspect Model Validation Trace from *Validator.*

```
Reading model Authorization CSAM1 from disk ...

Validating Authorization CSAM1

Model container is valid

All Classes are valid :)

All Class Operations are valid :)

All Message events are valid :)

Collaboration model element Pointcut is valid :)

All Owned attributes in Pointcut are valid :)

Interaction model element Pointcut is valid :)

All Message Occurrence Specifications in Pointcut are valid :)

All Messages in Pointcut are valid :)

All lifelines in Pointcut are valid :)

Collaboration model element Advice is valid :)

All Owned attributes in Advice are valid :)

Interaction model element Advice is valid :)

All Message Occurrence Specifications in Advice are valid :)

All Messages in Advice are valid :)
```

```
All lifelines in Advice are valid :)

Our model and all its model elements meet our validation requirements
```

Appendix B Test Cases

B.1 Test Case 1 - Composition with multiple join points

Test Case 1 Primary Model

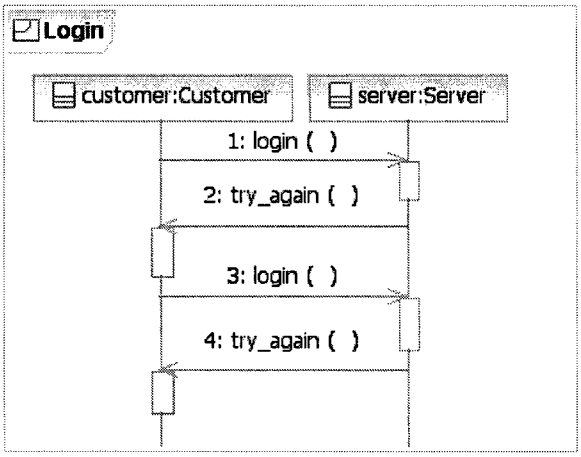
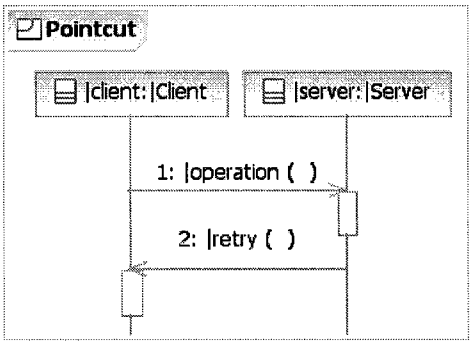
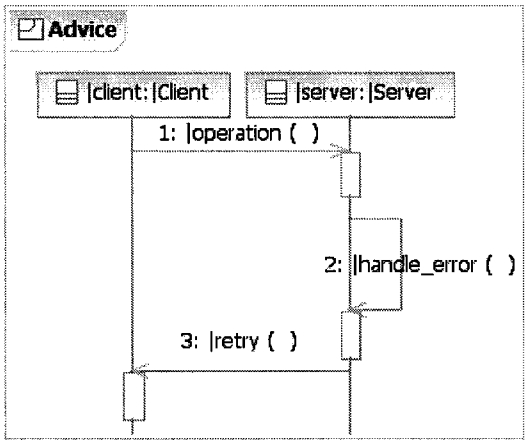


Figure B.1.1: Test Case 1 Primary Model

Test Case 1 Generic Aspect Model (GAM)



(a) Pointcut



(b) Advice

Figure B.1.2: Test Case 1 Generic Aspect Model

Mark Model

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

```
<xmi:XMI  xmi:version="2.0"  xmlns:xmi="http://www.omg.org/XMI"  xmlns="Binding-
Directives">
 <BindingDirective parameter="|client" binding="customer"/>
  <BindingDirective parameter="|server" binding="server"/>
  <BindingDirective parameter="|Client" binding="Customer"/>
  <BindingDirective parameter="|Server" binding="Server"/>
  <BindingDirective parameter="|reply" binding="try_again"/>
 <BindingDirective parameter="|operation" binding="login"/>
  <BindingDirective parameter="|handle_error" binding="save_bad_attempt"/>
</xmi:XMI>
```

Ant Script - TestCase1.xml

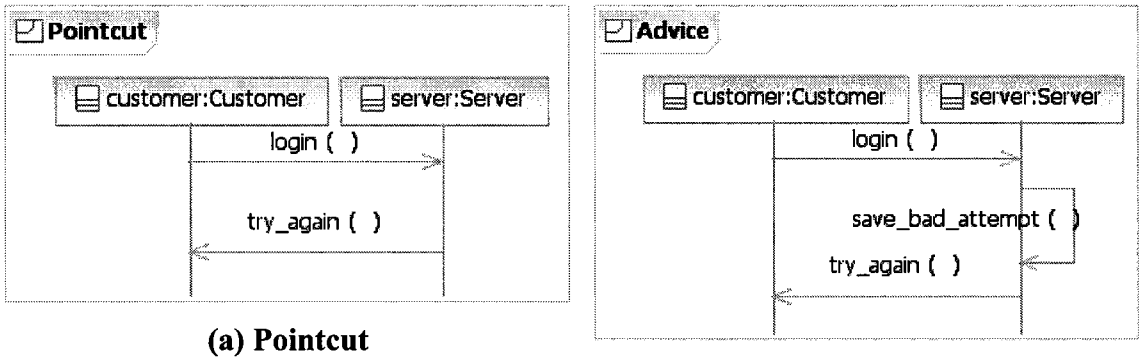
```
<project name="TestCase1" default="ComposeSequenceDiagram">
      <property name="outPath" value="../output models/WMASDs/1/" />
      <target name="ComposeSequenceDiagram" depends="loadMetamodels">
            <am3.loadModel modelHandler="EMF" name="PRIMARY" metamodel="UML2"
path="../input models/WMASDs/Primary Model II.uml" />
            <am3.loadModel modelHandler="EMF" name="ASPECT" metamodel="UML2"
path="../input models/WMASDs/Generic Security Aspect Model.uml" />
            <am3.loadModel modelHandler="EMF" name="BIND" metamodel="BD"
path="../input models/WMASDs/Security-Bindings.xmi" />

      <am3.atl path="Instantiate.asm" allowInterModelReferences="true">
                  <inModel name="UML2" model="UML2"/>
                  <inModel name="BD" model="BD"/>
                  <inModel name="PRIMARY" model="PRIMARY"/>
                  <inModel name="ASPECT" model="ASPECT"/>
                  <inModel name="BIND" model="BIND"/>
      <outModel name="CONTEXTSPECIFIC" model="CSAM1" metamodel="UML2"/>
                  <library path="PointcutMatchHelpers_v2.0.asm"/>
```

```
        </am3.atl>
        <am3.atl path="Compose.asm" allowInterModelReferences="true">
            <inModel name="UML2" model="UML2"/>
            <inModel name="PRIMARY" model="PRIMARY"/>
            <inModel name="ASPECT" model="CSAM1"/>
            <outModel name="COMPOSED" model="CM1" metamodel="UML2"/>
            <library path="PointcutMatchHelpers_v2.0.asm"/>
        </am3.atl>
        <am3.atl path="Instantiate.asm" allowInterModelReferences="true">
            <inModel name="UML2" model="UML2"/>
            <inModel name="BD" model="BD"/>
            <inModel name="PRIMARY" model="CM1"/>
            <inModel name="ASPECT" model="ASPECT"/>
            <inModel name="BIND" model="BIND"/>
    <outModel name="CONTEXTSPECIFIC" model="CSAM2" metamodel="UML2"/>
            <library path="PointcutMatchHelpers_v2.0.asm"/>
        </am3.atl>
        <am3.atl path="Compose.asm" allowInterModelReferences="true">
            <inModel name="UML2" model="UML2"/>
            <inModel name="PRIMARY" model="CM1"/>
            <inModel name="ASPECT" model="CSAM2"/>
        <outModel name="COMPOSED" model="FinalCM" metamodel="UML2"/>
            <library path="PointcutMatchHelpers_v2.0.asm"/>
        </am3.atl>
    <am3.saveModel model="CSAM1" path="${outPath}TestCase1CSAM1.uml"/>
    <am3.saveModel model="CM1" path="${outPath}TestCase1 CM1.uml"/>
    <am3.saveModel model="CSAM2" path="${outPath}TestCase1 CSAM2.uml"/>
    <am3.saveModel model="FinalCM" path="${outPath}TestCase1 CM2.uml"/>
    </target>
```

```
      <target name="loadMetamodels">
            <!-- Load UML metamodel -->
            <am3.loadModel modelHandler="EMF" name="UML2" metamodel="MOF"
nsUri="http://www.eclipse.org/uml2/2.1.0/UML"/>
            <am3.loadModel modelHandler="EMF" name="BD" metamodel="MOF"
path="../metamodels/BindingDirectives.ecore" />
      </target>
</project>
```

Test Case 1 Context Specific Aspect Model (CSAM) 1

Produced after 1st Iteration.



(a) Pointcut

(b) Advice

Figure B.1.3: Test Case 1 Context Specific Aspect Model

Test Case 1 Composed Model 1 (after 1st Iteration)

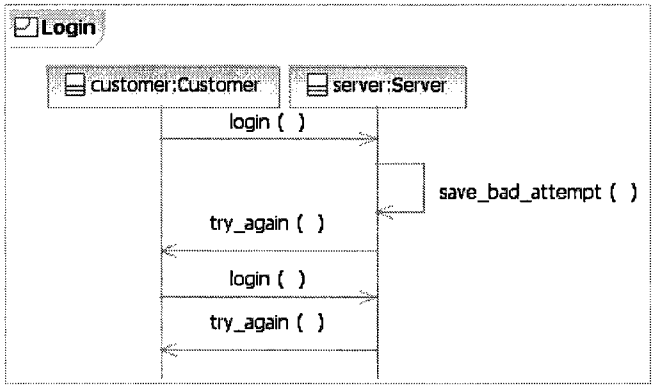
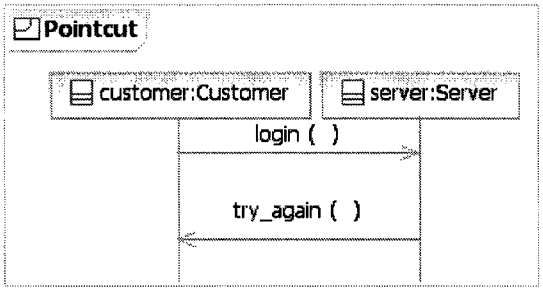
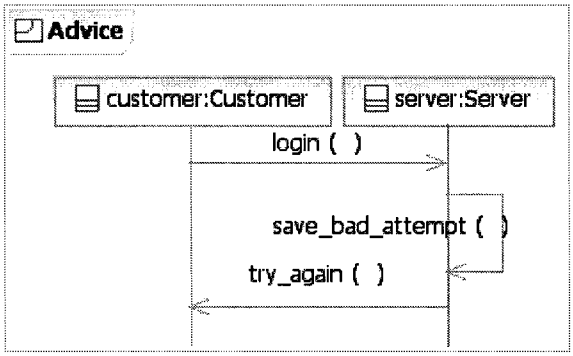


Figure B.1.4: Test Case 1 Composed Model 1st Iteration

Test Case 1 CSAM 2 after 2nd (final) Iteration



(a) Pointcut



(b) Advice

Figure B.1.5: Test Case 1 CSAM final Iteration

Test Case 1 Final Composed Model

Produced after 2nd (final) Iteration.

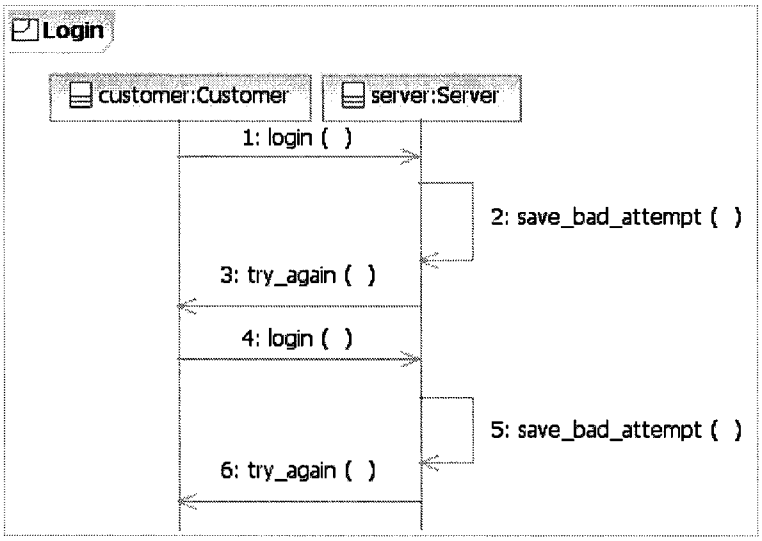


Figure B.1.6: Final Composed Model

B.2 Test Case 2 - Messages with simple arguments

Test Case 2 Primary Model

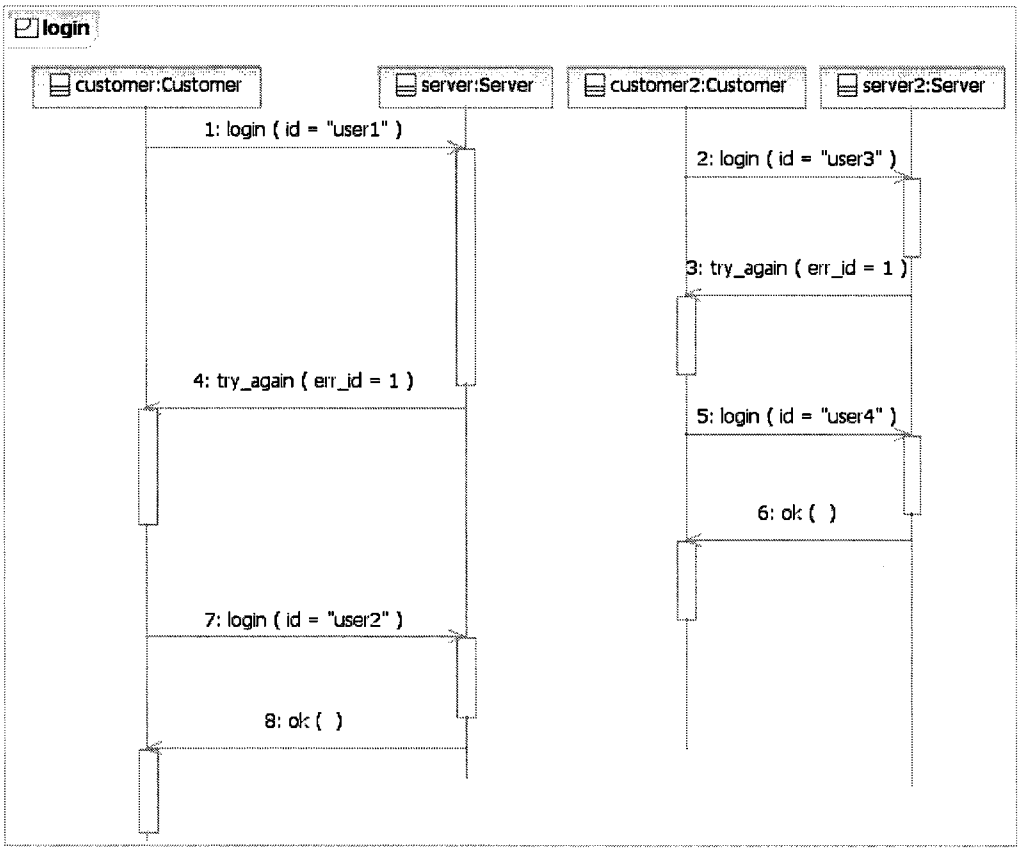
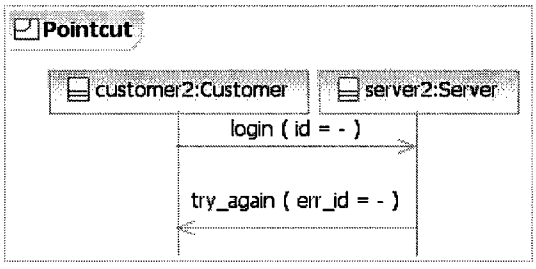
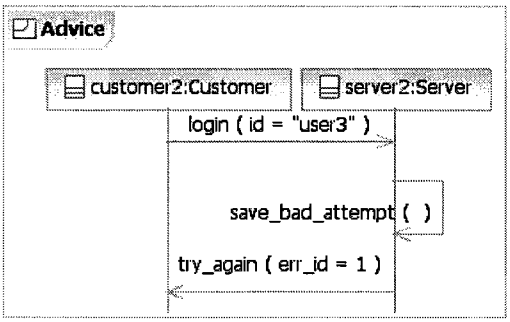


Figure B.2.1: Test Case 2 Primary Model

Test Case 2 GAM



(a) Pointcut



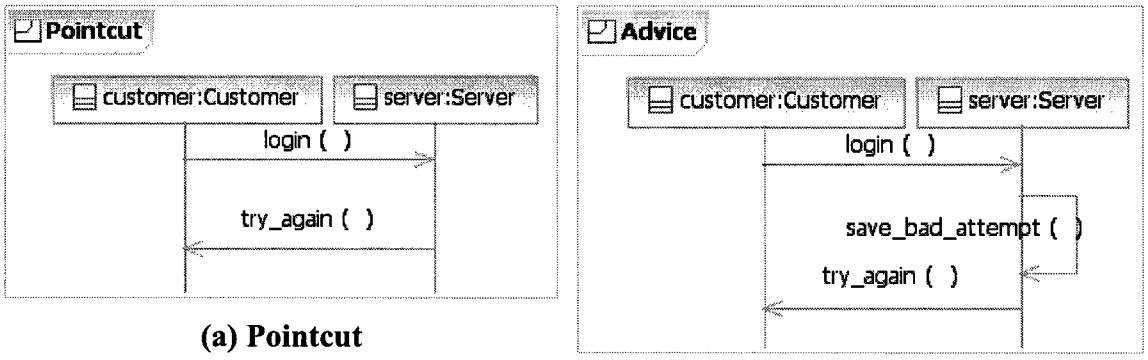
(b) Advice

Figure B.2.2: Test Case 2 GAM

Mark Model

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns="Binding-
Directives">

 <BindingDirective parameter="|client" binding="customer2"/>

 <BindingDirective parameter="|server" binding="server2"/>

  <BindingDirective parameter="|Client" binding="Customer"/>

  <BindingDirective parameter="|Server" binding="Server"/>

  <BindingDirective parameter="|reply" binding="try_again"/>

  <BindingDirective parameter="|operation" binding="login"/>

  <BindingDirective parameter="|handle_error" binding="save_bad_attempt"/>

</xmi:XMI>
```

Test Case 2 CSAM



(a) Pointcut

(b) Advice

Figure B.2.3: Test Case 2 CSAM

Test Case 2 Composed Model

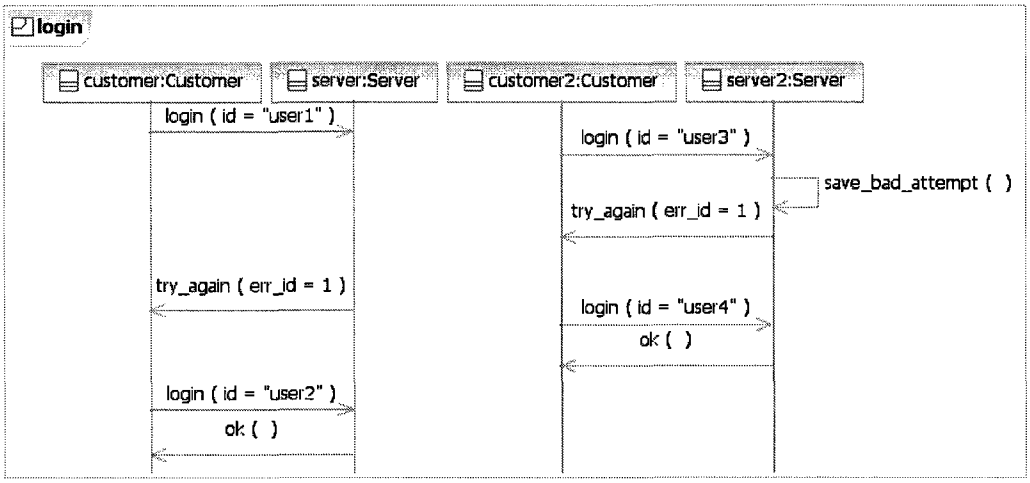


Figure B.2.4: Test Case 2 Composed Model

B.3 Test Case 7 - Pointcuts with wildcards

Primary Model

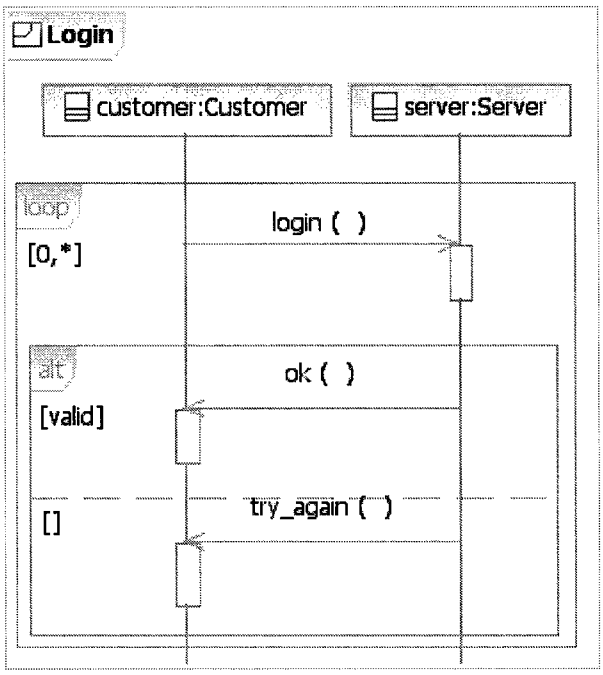
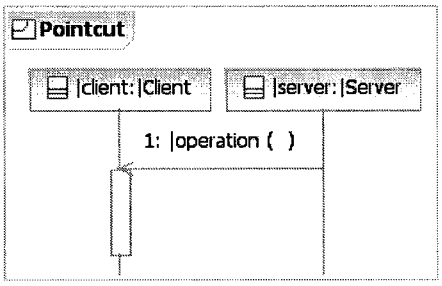
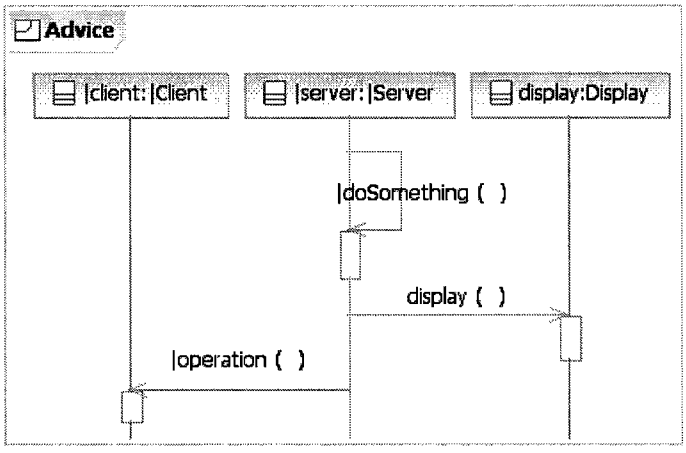


Figure B.3.1: Test Case 7 Primary Model

Test Case 7 GAM



(a) Pointcut



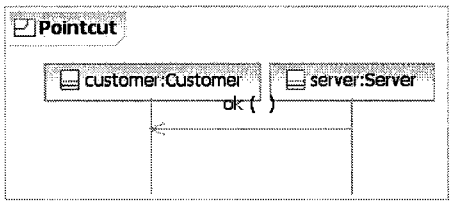
(b) Advice

Figure B.3.2: Test Case 7 GAM

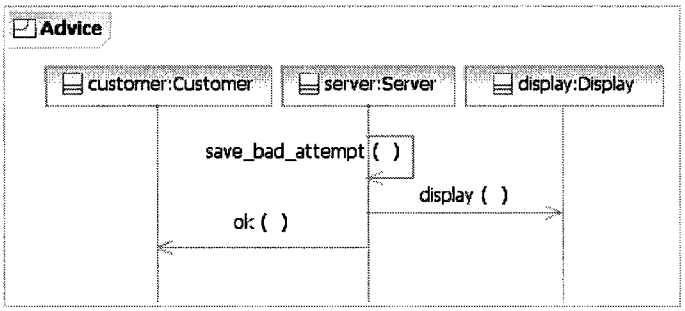
Mark Model

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<xmi:XMI  xmi:version="2.0"  xmlns:xmi="http://www.omg.org/XMI"  xmlns="Binding-
Directives">

 <BindingDirective parameter="|client" binding="customer"/>

  <BindingDirective parameter="|server" binding="server"/>

  <BindingDirective parameter="|Client" binding="Customer"/>

  <BindingDirective parameter="|Server" binding="Server"/>

  <BindingDirective parameter="|operation" binding="*"/>

  <BindingDirective parameter="|doSomething" binding="save_bad_attempt"/>

</xmi:XMI>
```

Test Case 7 CSAM 1



(a) Pointcut



(b) Advice

Figure B.3.3: Test Case 7 CSAM 1

Test Case 7 Composed Model 1

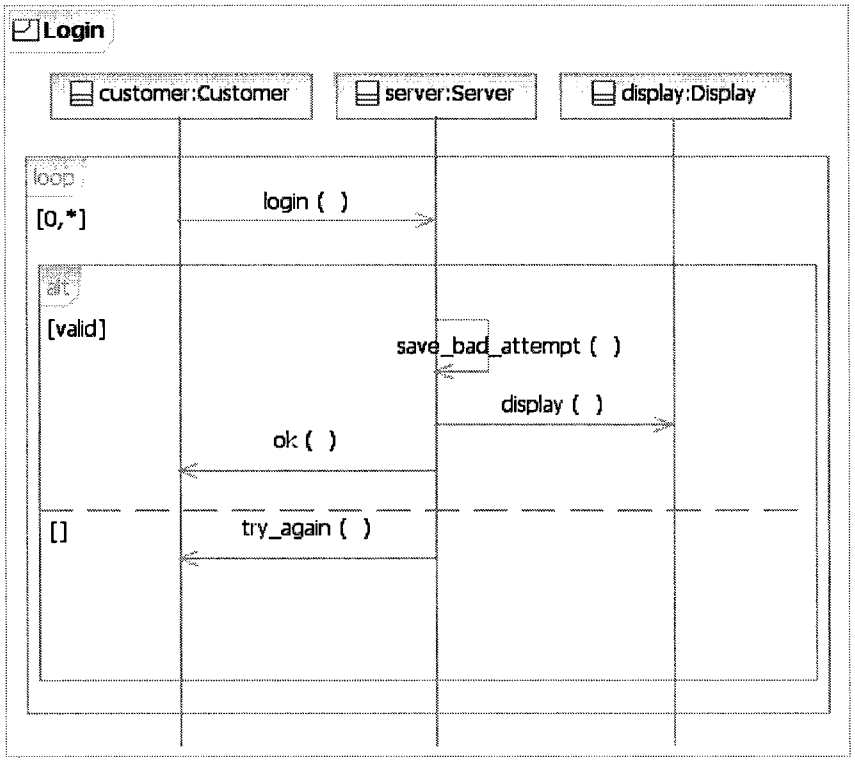
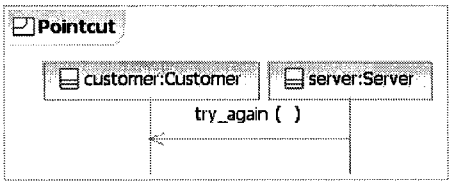
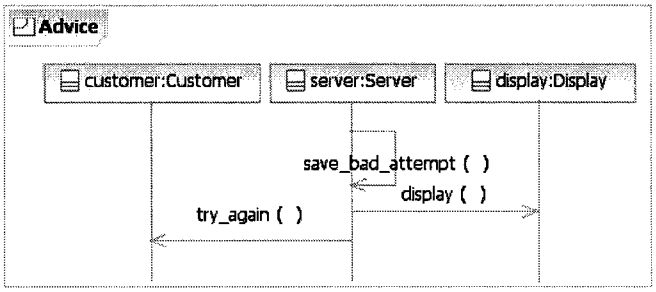


Figure B.3.4: Test Case 7 Composed Model 1

Test Case 7 CSAM 2



(a) Pointcut



(b) Advice

Figure B.3.5: Test Case 7 CSAM 2

Test Case 7 Composed Model 2 (final)

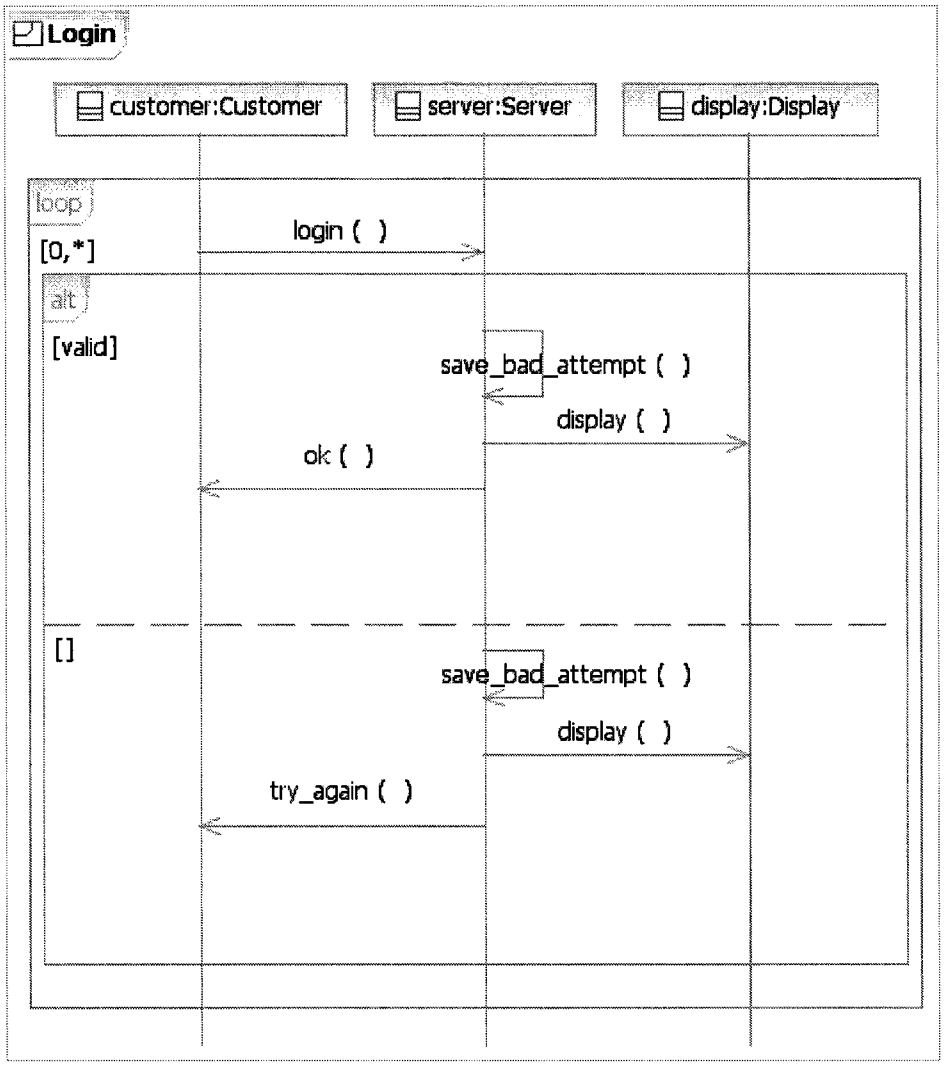


Figure B.3.6: Test Case 7 Final Composed Model

Appendix C Model Validation on RSA 7.5

Rational Software Architect 7.5 allows for validation of UML models. It checks the model and model elements against UML constraints (hopeful all of them). To validate a model we first have to import the model into the tool. After creating a project:

1. Right-click on the project and select **Import**.
2. The tool will present an **Import** dialog as shown below.

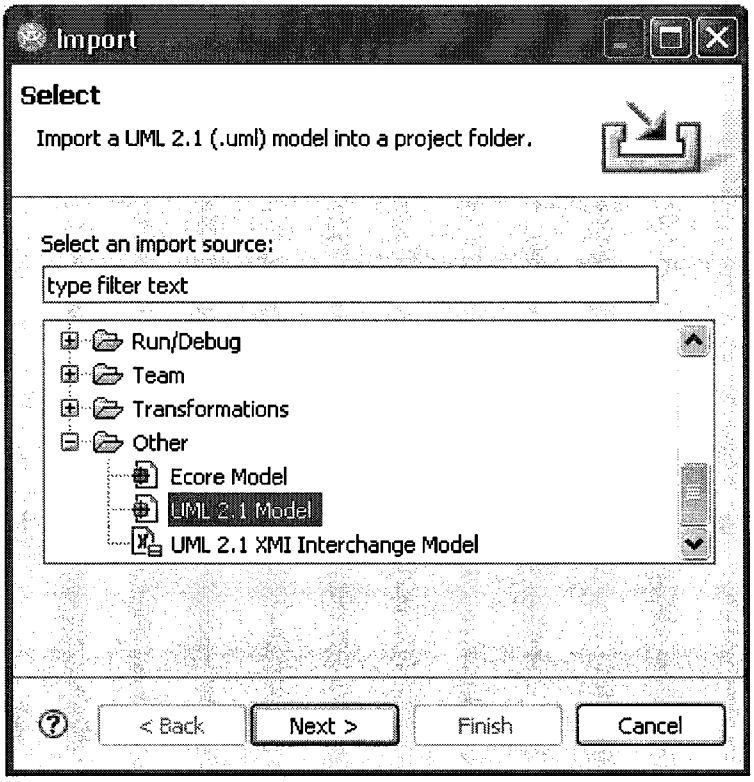


Figure C.1: Start Import UML 2.1 Model

3. Select to import a **UML 2.1 Model** and click the next button.
4. On the next screen shown in the next page, click on the **Browse** button to browse to the *.uml* file and click the **Finish** button to import the model.

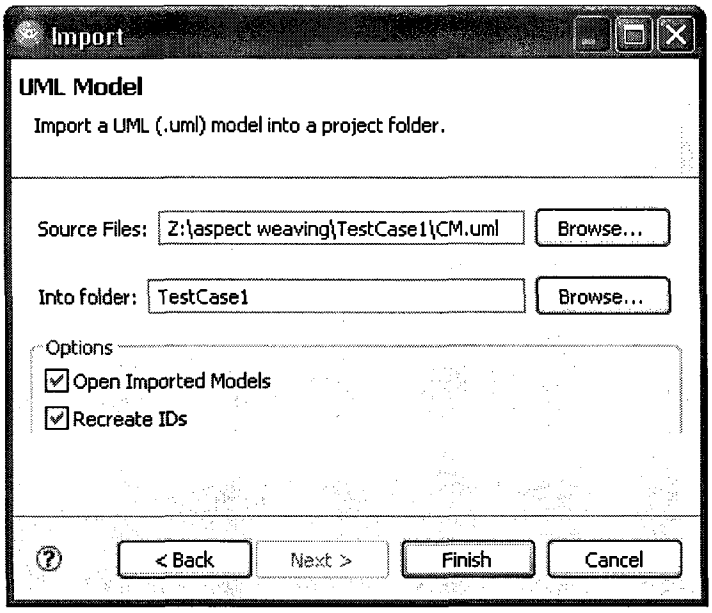


Figure C.2: Finish Import UML 2.1 Model

5. Your model should then be imported and listed in the Models folder like as shown below.

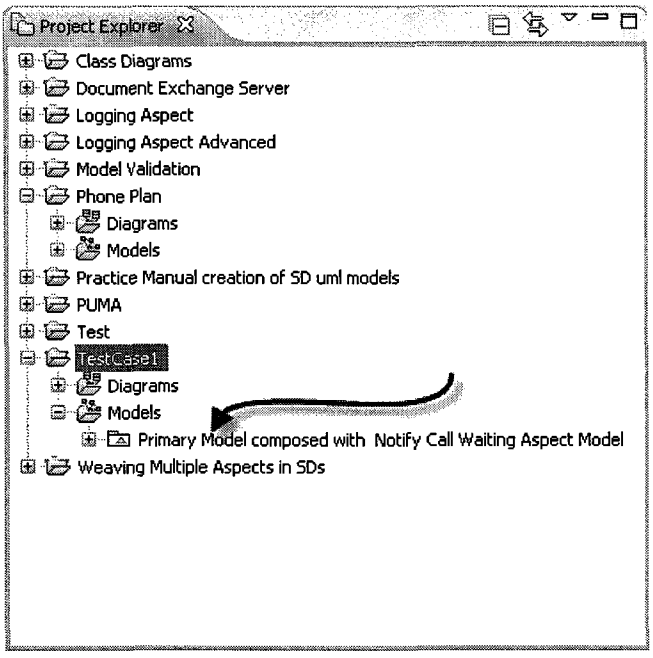


Figure C.3: View Imported Model

6. Right click on the model and select **Validate** as shown on the screen shot on the next page.

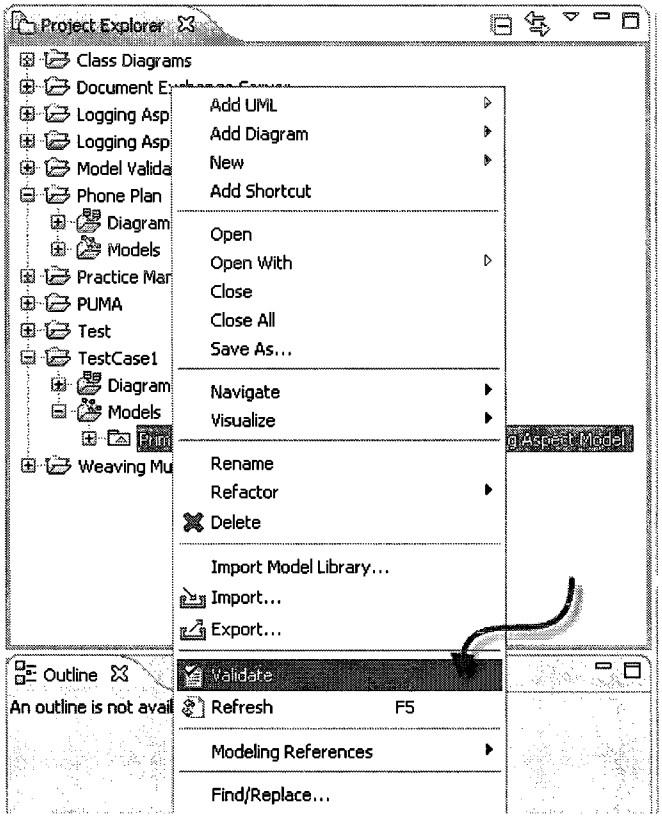


Figure C.4: Validate Imported Model

7. The tool will validate your model and return errors or warnings, if any, on the **Problems** view.

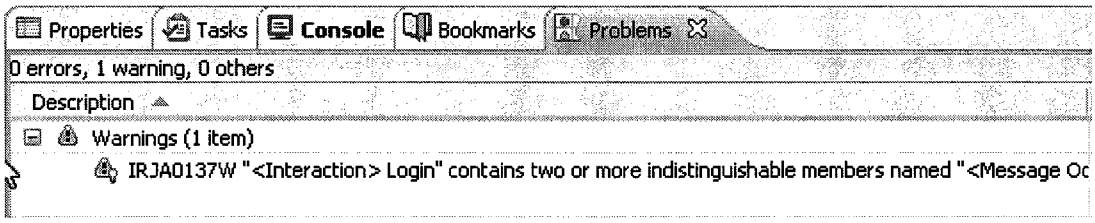


Figure C.5: Validation Warnings Example

8. You can right click on the warning to view what constraint was violated.

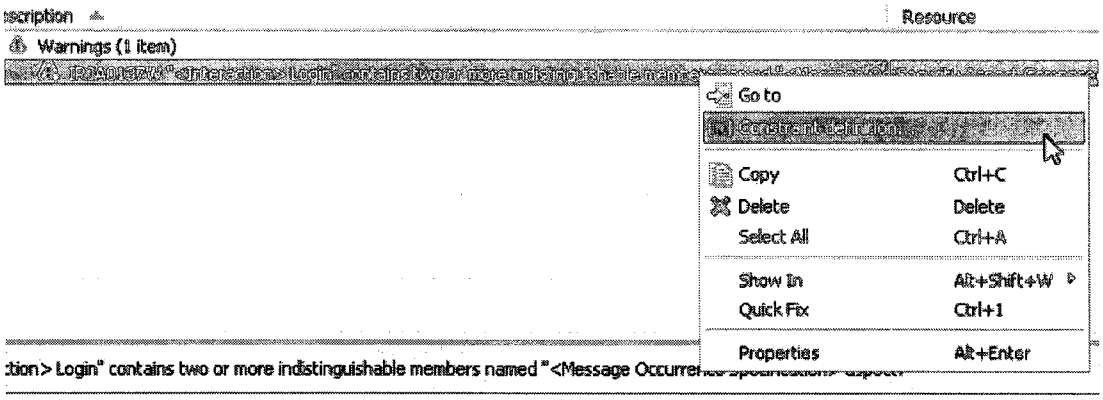


Figure C.6: View Validation Warnings Details

9. RSA will return a Constraint **Definition** like the one shown below giving details of the violated constraint.

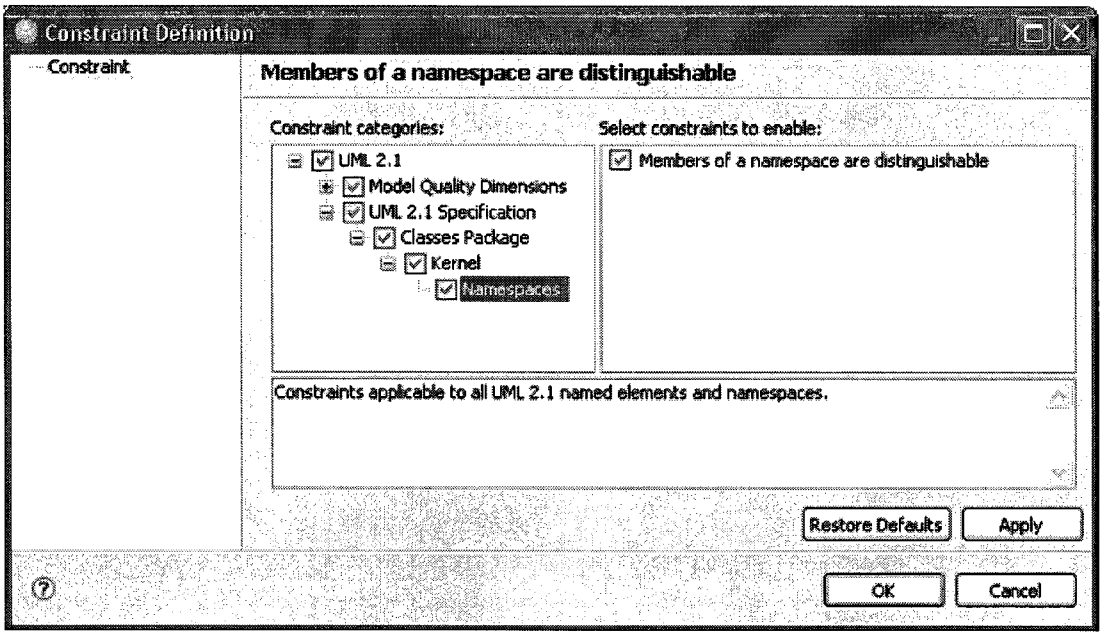


Figure C.7: Violated Constraint Details

In this case the model violates the constraint which says that all named elements must have a unique name. Our MessageOccurrenceSpecifications from the advice were given the same name.

Appendix D Creating a Sequence Diagram from a UML model in RSA

1. After importing the UML model into RSA, navigate to the model's **Interaction**.
2. Right click on the **Interaction** and select Add Diagram → Sequence Diagram.

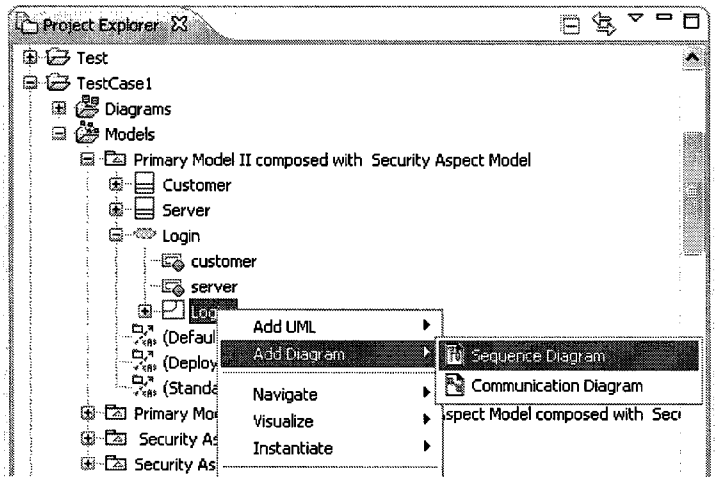


Figure D.1: Add Sequence Diagram.

3. RSA will generate a sequence diagram and prompt you for a name for the sequence diagram.

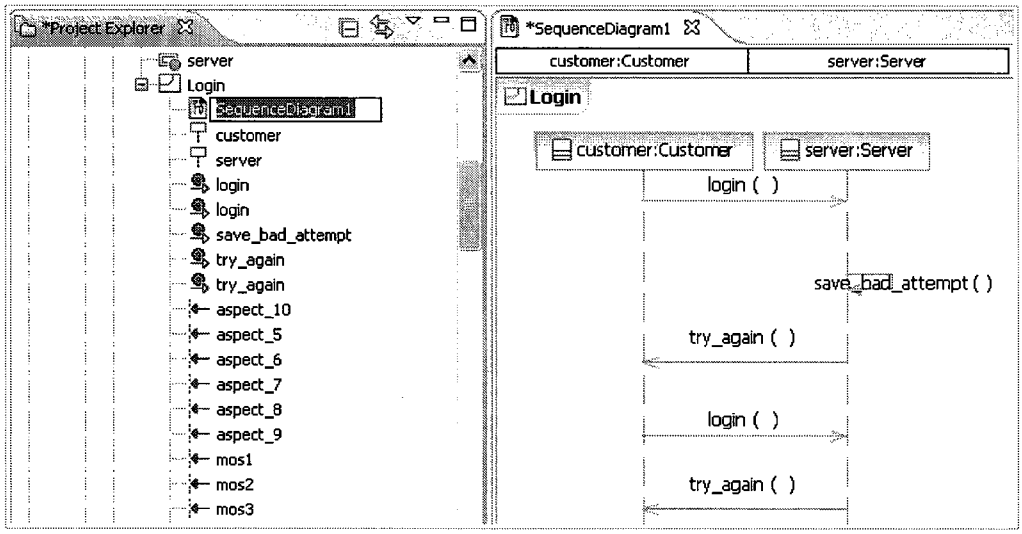


Figure D.2: Generated SD.

4. Enter the name for your sequence diagram and save.

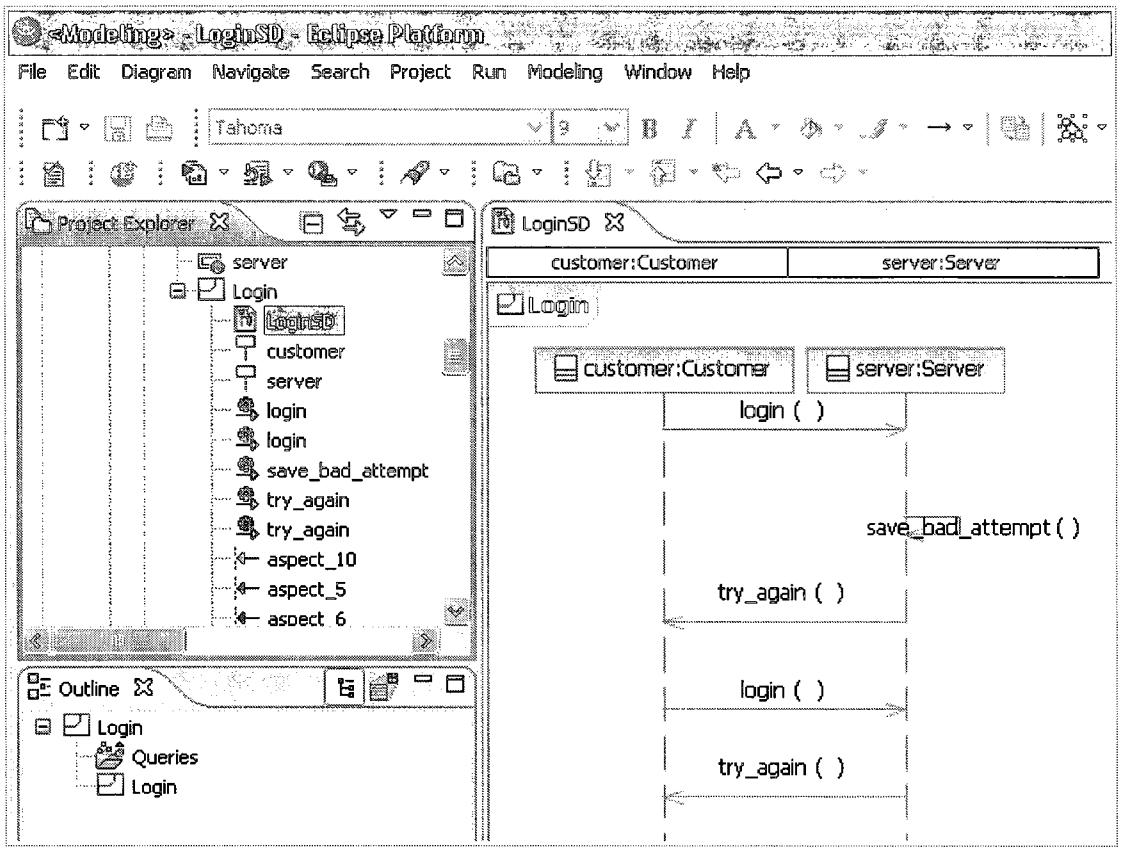


Figure D.3: Final Generated Sequence Diagram.

Appendix E PointcutMatchHelpers Library

Helper name	Return type	Purpose
adviceMOSEncoding()	String	A tagging string for advice MOSs
getAspectFragments(sd)	Sequence	Returns MOSs and CFs from a given SD from the aspect model.
getPrimaryFragments()	Sequence	Returns MOSs and CFs from a given SD from the primary model.
getFragments()	Sequence	Returns MOSs and CFs within the context CF.
getFragments()	Sequence	Returns MOSs and CFs within the context Interaction Operand.
sameEventType(e1, e2)	Boolean	Returns true if the given *MessageEvents* are both ROE or SOE
PairwiseMatchFragments (src, tgt)	Boolean	Checks if the fragments at the same index from *src* and *tgt* sequences are "equal".
getBinding(name)	String	Retrieves the binding value from the mark model for the given template parameter.
bindingDefined(String)	Boolean	Returns true if a binding value exists for the given parameter.
samePropertyType(pct , core)	Boolean	Returns true if the given *Properties* have the same type (class).
equals(mos) Context = MOS	Boolean	Checks if the context and supplied MOSs are equivalent.

Helper name	Return type	Purpose
invalidMOSName()	Boolean	Returns true if the context MOS has been tagged meaning it was added from a previous composition iteration
getLifelineMOS()	Sequence	Returns MOSs that cover the context lifeline.
getAspectSD (name)	Interaction	Returns an SD (Advice or Pointcut) from the Aspect Model.
getMessageLifelines (m)	Sequence	Returns the sender and receiver lifelines for the given message.
getSDLifelines (sd)	Sequence	Returns all lifelines in a given SD.
getSDMessages (sd)	Sequence	Returns all messages in a given SD.
createMOSName (code, idx)	String	Generates a name for the context MOS given our tagging string and MOS index in the sequence of MOS.
fromAspectModel() Context = Model	Boolean	Returns true if the context model is from the aspect model.
fromAspectModel() Context = Interaction	Boolean	Returns true if the context Interaction is from the aspect model.
fromPrimaryModel() Context = Model	Boolean	Returns true if the context model is from the primary model.
fromPrimaryModel() Context = Message	Boolean	Returns true if the context message is from the primary model.
notInPrimaryModel() Context = Class	Boolean	Returns true if the context class is from the primary model.

Helper name	Return type	Purpose
notInPrimaryModel() Context = MessageEvent	Boolean	Returns true if the context MessageEvent is from the primary model.
notInPrimaryModel() Context = Operation	Boolean	Returns true if the context operation is from the primary model.
equals(c) Context = Class	Boolean	Returns true the context class is the same as the supplied class; that is, if they have the same name.
classMatch (c1 , c2)		Returns true if the two classes are equal.
equals(o) Context = Operation	Boolean	Returns true if the context operation is the same as the supplied operation.
equals(e) Context = MessageEvent	Boolean	Returns true if self is the same as the supplied event; that is, they are of the same type and have the same operation.
equals(f) Context = CombinedFragment	Boolean	Returns true if the context CF is the same as the given interaction fragment which also has to be CF.
equals(f) Context = InteractionOperand	Boolean	Returns true if the context interaction operand is equal to the given interaction fragment .
getMOSs() Context = InteractionOperand	Sequence	Returns all MOSs enclosed by the context interaction operand.
getMOSs() Context = CombinedFragment	Sequence	Returns all MOSs enclosed by the context CF.
PairwiseMatchOperands (src,tgt)	Boolean	Returns true if the given sequences of operands have matching pair of oper-

Helper name	Return type	Purpose
		ands, that is, the operands at the same index are equal.
PairwiseMatchMOSs (src, tgt)	Boolean	Checks if the elements at the same index from the *src* and *tgt* sequence are equal.
joinPointsFragments()	Sequence	Returns fragments at the start of the join points.
getMaxInt()	ValueSpecificationAction	Returns the *maxInt* value of the given Interaction constraint.
getMinInt()	ValueSpecificationAction	Returns the *minInt* value of the given Interaction constraint.

Table 10: Helpers from the PointcutMatchHelpers Library