Composition of Aspects
Represented as UML Activity Diagrams

By

Hongyan Gong

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

Master of Science in
Information and Systems Science

Department of Systems and Computer Engineering
Carleton University
Ottawa, Canada

April 2008

©Copyright 2008, Hongyan Gong
NOTICE:  
The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Avis:  
L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.
Abstract

Aspect-Oriented Modeling (AOM) techniques allow software designers to address separately solutions for crosscutting concerns. The thesis proposes and implements an aspect composition method that takes UML activity diagrams as input (describing both primary model and aspect model), composes them according to the user's directives, and generates the composed activity diagram as output. The composition of activity diagrams is approached as a graph rewriting problem, where the graph transformation rules are automatically generated from the UML aspect model. The goal is to provide tool support to software developers who would like to compose aspect and primary models, without requiring them to know about graph rewriting.

The thesis describes the proposed transformation rules both at the UML graphical notation level, which is more intuitive, and at the UML metamodel level, which corresponds to the actual implementation. A UML profile (AspectProfile) and a plug-in for IBM Rational Software Architect were built in order to realize the proposed transformation.
Acknowledgement

First and foremost, I would like to thank my supervisor, Dorina Petriu, for providing me the interesting topic, valuable advice, and guidance during my study at Carleton University.

I would like to thank my wife, Yongfang, for her constant support and unconditional love. Without her support, I could have not finished finish my studies.

I also thank my parents and my sisters from the bottom of my heart for your constant support and everything you have done for me. I am quite certain that my success can be credited to all of you!

I also would like to thank all of my friends – especially Bo Wang, Bo Zhan, Zhenmo Wu, Wenhao Deng, and Yichao Zheng – for your help and advice. Life becomes better because of you!

Finally, I wish my son Alex all the best! I just hope you know how much I love you!
# Table of Contents

Chapter 1: Introduction ........................................................................................................... 1  
1.1 Thesis Motivation ............................................................................................................. 1  
1.2 Scope of the Research ...................................................................................................... 1  
1.3 Contributions of the Thesis ............................................................................................ 3  
1.4 Thesis Outline ................................................................................................................ 4  

Chapter 2: Literature Review ............................................................................................... 6  
2.1 Aspect-Oriented Modeling (AOM) ................................................................................... 6  
2.2 The Unified Modeling Language ...................................................................................... 8  
  2.2.1 UML Extension Mechanisms: Profiles ................................................................... 10  
2.3 Graph Transformation Concepts .................................................................................... 11  
2.4 Eclipse-based Design Tools ............................................................................................ 14  
  2.4.1 Eclipse ....................................................................................................................... 14  
  2.4.2 Eclipse Plug-in ........................................................................................................... 16  
  2.4.3 UML2 Plug-in ............................................................................................................ 17  
  2.4.4 Rational Software Architect (RSA) ....................................................................... 18  

Chapter 3: Approach to AOM composition ......................................................................... 20  
3.1 AOM Example ................................................................................................................ 20  
  3.1.1 Document Exchange System .................................................................................... 20  
  3.1.1.1 DES Primary Model ............................................................................................ 21  
  3.1.1.2 Generic Aspect Model .......................................................................................... 24  
  3.1.1.3 Context-Specific Aspect Model ......................................................................... 27  
  3.1.1.4 Model Composition .............................................................................................. 29  
    3.1.1.4.1 Structure Composition .................................................................................... 29  
    3.1.1.4.2 Behavior Composition .................................................................................... 29  
3.2 Activity Metamodel ....................................................................................................... 32  
  3.2.1 Aspect Profile ............................................................................................................ 35  
  3.2.1.1 Aspect Profile Definition ...................................................................................... 36  
  3.2.1.2 Advice Stereotype ............................................................................................... 37  
  3.2.1.3 EdgePointCut Stereotype .................................................................................... 38  
  3.2.1.4 NodePointCut Stereotype .................................................................................... 38  
  3.2.1.5 GroupPointCut Stereotype .................................................................................. 39  
3.3 Activity Diagram Composition ...................................................................................... 40
# List of Figures

Figure 1: Scope of the Research ............................................................. 2  
Figure 2: AOM Composition ................................................................. 8  
Figure 3: Graph Rewriting Example ....................................................... 13  
Figure 4: Architecture of Eclipse Platform ............................................. 15  
Figure 5: RSA Screen Shot: Activity Diagram ....................................... 19  
Figure 6: User Case of DES ................................................................. 21  
Figure 7: Primary Model - Component Diagram .................................... 22  
Figure 8: Primary Model - Deployment Diagram .................................... 22  
Figure 9: Primary Model: High Level Activity Diagram for Retrieve Document Scenario ............................................. 23  
Figure 10: DES Primary Model: nested activity diagram doGetDocument .................................................. 24  
Figure 11: Generic Authorization Aspect Model - Component Diagram ........ 25  
Figure 12: Generic Authorization Aspect Model - Deployment Diagram .......... 25  
Figure 13: Generic Authorization Aspect Model: High-level Activity Diagram ............................................. 26  
Figure 14: Generic Aspect Model: Nested |checkAuthorization Activity Diagram ............................................. 27  
Figure 15: Aspect Activity Model with more than Two Output Points ........ 31  
Figure 16: Composed Model ................................................................. 32  
Figure 17: Simplified Metamodel for UML Activity Diagram .................... 33  
Figure 18: Partial Metamodel for ControlNode ........................................ 34  
Figure 19: Partial Metamodel of ActivityGroup ...................................... 35  
Figure 20: Aspect Profile Package ......................................................... 37  
Figure 21: A Generic Aspect Model with Binding Rules ......................... 42  
Figure 22: Context-specific Model from Figure 21 .................................. 43  
Figure 23: Primary model  Figure 24: Context-specific Aspect Model .......... 45  
Figure 25: Composed Model from Figure 23 and Figure 24 ....................... 46  
Figure 26: Primary Model  Figure 27: Context-specific Aspect Model .......... 47  
Figure 28: Composed Model from Figure 26 and Figure 27 ....................... 48  
Figure 29: Primary Model  Figure 30: Aspect Model ............................... 49  
Figure 31: Composed Model from Figure 29 and Figure 30 ....................... 50  
Figure 32: Primary Model ................................................................. 51  
Figure 33: Context-specific Aspect Model ............................................. 52  
Figure 34: Composed Model from Figure 32 and Figure 33 ....................... 53  
Figure 35: AD Composition Flowchart ................................................. 55
List of Tables

Table 1: UML Four-layer Metamodeling Architecture................................................................. 9
Table 2: Advice Stereotype........................................................................................................... 37
Table 3: EdgePointCut Stereotype............................................................................................... 38
Table 4: NodePointCut Stereotype............................................................................................... 38
Table 5: GroupPointCut Stereotype............................................................................................. 39
Table 6: Summary of Test Cases .................................................................................................. 86
List of Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AO</td>
<td>Aspect-Oriented</td>
</tr>
<tr>
<td>AOM</td>
<td>Aspect-Oriented Modeling</td>
</tr>
<tr>
<td>AOP</td>
<td>Aspect-Oriented Programming</td>
</tr>
<tr>
<td>AOSD</td>
<td>Aspect-Oriented Software Development</td>
</tr>
<tr>
<td>API</td>
<td>Application Program Interface</td>
</tr>
<tr>
<td>CVS</td>
<td>Concurrent Version System</td>
</tr>
<tr>
<td>DES</td>
<td>Document Exchange System</td>
</tr>
<tr>
<td>EMF</td>
<td>Eclipse Modeling Framework</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>JAR</td>
<td>Java Archive</td>
</tr>
<tr>
<td>JDT</td>
<td>Java Development Tooling</td>
</tr>
<tr>
<td>LQN</td>
<td>Layered Queueing Network</td>
</tr>
<tr>
<td>MOF</td>
<td>Meta Object Facility</td>
</tr>
<tr>
<td>OCL</td>
<td>Object Constraint Language</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group</td>
</tr>
<tr>
<td>OOP</td>
<td>Object-Oriented Programming</td>
</tr>
<tr>
<td>PDE</td>
<td>Plug-in Developer Environment</td>
</tr>
<tr>
<td>RSA</td>
<td>Rational Software Architect</td>
</tr>
<tr>
<td>SWT</td>
<td>Standard Widget Toolkit</td>
</tr>
<tr>
<td>UI</td>
<td>User Interface</td>
</tr>
</tbody>
</table>
UML  Unified Modeling Language
XMI  XML Metadata Interchange
XML  eXtensible Markup Language
XSL  XML Stylesheet Language
Chapter 1: Introduction

1.1 Thesis Motivation

The Aspect-Oriented Modeling (AOM) approach is a technique that allows software designers to isolate crosscutting concerns (such as security, confidentiality, and data integration) and address them separately from the core functionally captured in the design models. The crosscutting concerns are described in distinct UML aspect models, while the core functionality is described in the primary model. An integrated system view can then be obtained by weaving (composing) the aspect models with the primary model [France+04]. Among the benefits of applying AOM during the software design phase is the fact that AOM can ease the task of exploring alternative for the crosscutting functionality and that AOM promotes the reuse of generalized solutions by modeling patterns as aspects, and allows software developers to replace easily some features with other alternatives [Georg+02] [Georg+03] [France+04]. A review of AOM research is given in Capter 2 of the thesis.

1.2 Scope of the Research

The main objective of the thesis is to automate the process of composing (weaving) aspects in UML models. We focus on the composition of behavior models expressed as activity diagrams. Such models can be viewed as directed graphs, where the nodes are UML model elements and the edges are links between model elements. The input models represent the UML primary model and one or more aspect models created by a software
designer using a graphical editor, such as IBM Rational Software Architect [RationalRSA] or programmatically using an Eclipse plug-in named UML2 [UML2Plugin]. The binding rules and composition directives are represented by a UML profile developed in this thesis to support the automatic composition. Figure 1 shows the information flow and the processing steps for the aspect composition implemented in the thesis.

Figure 1: Scope of the Research

The graphical editor tool exports the primary and generic aspect models in standard XMI format, which in turn is read by an Eclipse XMI reader and creates the internal data structure using UML Meta Model Library (Realized by UML2 Plug-in). The next step is to instantiate the generic aspect model, producing a context-specific one as in [France+04], by following a set of binding rules provided by the designer through stereotypes and tagged values of the Aspect Profile proposed in the thesis, then the
context-specific aspect model(s) is (are) composed with the primary model, according to a set of composition directives, which are also Aspect Profile stereotypes and tagged values. Finally, the composed in-memory model is output using the UML2 API in standard XMI format.

Various UML graphical editors can be used to generate UML models, and all UML tools are supposed to produce UML models in standard XMI format. In principle, the composition engine implemented in the thesis should be able to accept UML models in standard XMI format from any UML tool. However, this is not the case, as in reality different UML tools produce models in different XMI variants. The transformation engine realized in this thesis works with UML models in XMI format produced by RSA version 7, and may need to be revised if the XMI generated is from a different tool.

1.3 Contributions of the Thesis

The goal of the thesis is to define and implement a transformation process that can compose a primary activity model with one or more aspect activity models and generates an integrated activity diagram, which is then serialized in the XMI format. The contributions of the thesis are summarized as follows:

- Designed and implemented a UML profile named *Aspect Profile* by borrowing Aspect-Oriented Programming concepts, such as advice, join point, and pointcut. The Aspect Profile allows the user to add binding and composition information required for automated aspect composition.

- Proposed a method for the automatic composition (weaving) of aspects with a primary UML 2.0 model those were annotated with the Aspect Profile introduced in
the thesis. The composition of activity diagrams is treated as a graph rewriting problem, where the graph transformation rules are automatically generated from the UML aspect model annotated with the Aspect Profile. The goal is to provide tool support for weaving aspect models into the primary model, without requiring the users to have knowledge of graph transformations.

- Designed, implemented, developed and tested an aspect composition engine as an RSA 7.0 plug-in, which composes aspect models with primary model represented as activity diagrams. (The UML tool used in the thesis is RSA 7.0). The plug-in automatically extract Java objects representing UML model elements and the composition directives from UML activity diagrams annotated with the Aspect Profile; the composed model is generated in memory and is output in standard XMI format.

1.4 Thesis Outline

The thesis includes six chapters and the contents of each are described below:

Chapter 2 introduces the background literature for the thesis. Some topics related to the thesis – such as AOM, Graph transformation concepts, and Eclipse-based design tools – are also discussed.

Chapter 3 gives more details on the AOM approach. It explains the architecture and elements of the AOM, such as generic aspect model and context specific model and binding rules. The high-level composition algorithm is also given in this chapter.

Chapter 4 describes the Java design and implementation of the proposed transformation
algorithm. It also describes the detailed process of producing the Aspect Profile, which is important for the composition.

Chapter 5 contains an investigation of a case study and discusses the verification of the Java application.

Chapter 6 summarizes the thesis work and details some open issues for future study.
Chapter 2: Literature Review

This chapter provides background information related to the thesis that covers the following topics: Aspect-Oriented Modeling, UML and extension mechanisms, Eclipse-based tools, and Graph transformation concepts.

2.1 Aspect-Oriented Modeling (AOM)

Before introducing AOM, we do a quick review of Aspect-Oriented Software Development (AOSD). Compared with the Object-Oriented Programming (OOP), AOSD is a new approach that addresses limitations inherent from other approaches, including OOP. AOSD aims to address those limitations by providing means for software developers to conceptualize, represent, and reason about multiple concerns separately; and then compose them automatically to form a working system. This approach results in better modularization support and reduces the development, maintenance and evolution costs [AOSD].

The application of the concept of Aspect-Oriented techniques to code level resulted in the introduction of Aspect-Oriented Programming (AOP) [Kiczales+97]. AOP, proposed by Gregor Kiczales, et al., from the group at Xerox PARC (Palo Alto Research Center, Inc), was first applied to the implementation phases of the software lifecycle. AOP aims to provide means for creating code in which crosscutting concerns (such as logging) are encapsulated as first class elements, and the final deliverable is obtained by weaving the aspects with the base program. One of the best known tools for AOP is AspectJ, which was developed at Xerox PARC and is a popular AOP language. In the Aspect-Oriented
(AO) community, there are four concepts that are widely used: *aspect, join point, pointcut* and *advice*. *Aspects* are used to specify the non-core functionalities that crosscut core concerns; an *aspect* is usually specified as a class-like module and can have attributes and methods. The term *join point* refers to a point in the execution of the base program where an aspect will be inserted (weaved). *Pointcut* is used to specify the conditions for identifying the join points, and *advice* is used to indicate the new code inserted by the aspect.

AOP is not introduced to replace the OOP, but is a technique that removes some of the shortcoming of OOP by modularizing the crosscutting concerns into aspect and weaving those concerns dynamically; those crosscutting concerns such as security and performance is scattered in multiple component and hard deal with in the traditional OOP. The main advantages of applying the AO technique to the earlier phase of software development are improved modularity and the fact that the notation of the aspects preserved throughout the development ensures traceability.

AOM is the result of applying AO techniques to models used in earlier software lifecycle phases. The main topics of current AOM research are addressing the following problems:

- Crosscutting concerns localization and description. [France+04][Ray+04][Mahoney+04]
- Model composition or weaving various aspect models to obtain a working system [France+04][Reddy+05][Reddy+06][Georg+02][Georg+03][Whittle+06][Mahoney+05].
- Tools support for AOM--providing tools that automate the process of weaving / composing aspect models with the primary model to obtain the composed models
Composed model analysis to identify and resolve conflicts and verify different model properties [France+04] [Georg+02].

- Applying aspects for analyzing different non-functional properties such as security and performance [Shen+05] [Petriu+07].

According to [France+04], an AOM model contains a primary model, aspect models, the binding rules, and the composition directives. An aspect model is generic at first, and therefore binding rules are used to instantiate the aspect model in the application context, obtaining a context-specific aspect model. Composition directives are used to indicate how context-specific aspect models should be composed with the primary model in order to obtain the composed model. Figure 2 shows the AOM composition [France+04].

![Figure 2: AOM Composition](image)

### 2.2 The Unified Modeling Language

In the field of software engineering, the UML (Unified Modeling Language), standardized by the Object Management Group (OMG), is a widespread modeling and
visualization language used for software development. The OMG defines UML modeling language as a four-layer architecture: M0 contains instances of a model, M1 defines a UML model, M2 defines the UML metamodel and M3 the meta-metamodel. [www.omg.org]. The meta-metamodel level is defined by the standard Metaobject Facility (MOF), and the UML metamodel is defined by using MOF Table 1 shows the four-layer architecture of UML.

<table>
<thead>
<tr>
<th>Layer</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Meta-metamodel (M3)</td>
<td>The infrastructure for metamodeling architecture. Defines the language for specifying metamodels.</td>
<td>MetaClass, MetaOperation</td>
</tr>
<tr>
<td>Metamodel (M2)</td>
<td>An instance of a meta-metamodel. Defines the language for specifying a model.</td>
<td>Class, Attribute, Operation</td>
</tr>
<tr>
<td>Model (M1)</td>
<td>An instance of a metamodel. Defines a language to describe an information domain.</td>
<td>Person, Order</td>
</tr>
<tr>
<td>User objects (M0)</td>
<td>An instance of a model. Defines a specific information domain.</td>
<td>person_001, order_001,</td>
</tr>
</tbody>
</table>

Table 1: UML Four-layer Metamodeling Architecture

UML is designed to be a general purpose modeling language. It can be used to specify, visualize, construct, simulate, test and document software-intensive system and it provides a graphical notation used to create an abstract model of a system. The UML metamodel and UML models may be serialized in XMI. [UMLHome]
2.2.1 UML Extension Mechanisms: Profiles

Although UML provides a rich set of modeling concepts and notations to meet the need of modeling a typical software system, software designers may wish to attach non-semantic information to the models or to have additional features or notations that are not defined in the standard UML. To facilitate the different needs of different users, UML provides an extension mechanism named Profile, composed of the following built-in extensions to provide a flexible mechanism that allows software designers to tailor UML to their own needs.

**Stereotypes:** Stereotypes defines how an existing metaclass can be extended and allows the usage of the platform or domain specific terminology or notation in addition to ones used for the extended metaclass. They group constraints and tagged values and can be given descriptive names and applied to model elements.

**Tagged Values:** Stereotypes, like classes, may have properties. These properties are referred to as tag definitions and the values of these properties are called tagged values. Tagged values are one of the extensibilities that UML allows to attach arbitrary information to model elements.

**Constraints:** Constraints place conditions or restrictions on natural language or in Object Constraint Language (OCL) which declare some of the semantics of a particular design element.

The extension mechanisms are used to define UML profiles that specialize the UML for different application domains. This mechanism allows us to extend the concepts of UML and to introduce some new or customized concepts. In the thesis, a new *Aspect Profile* is proposed to enable aspect composition.
2.3 Graph Transformation Concepts

The term "graph transformation" is used to represent both graph grammars and graph rewriting systems. Although these are two different kinds of techniques, they are often considered synonymous. However, strictly speaking, a graph grammar is a set of production rules that generates a construct from a list of terminal graphs, produces non-terminal graphs as intermediate results, and recognizes that a construct obeys the grammar's rules. On the other hand, a graph rewriting system is a set of rules that transforms one instance of a given class of graphs into another instance of the same class of graphs without distinguishing between terminals and non-terminals [Schurr90].

A graph rewriting system is a tool that is used to rewrite an input graph according to graph grammars. It also included an algorithm for selecting the rules to be applied when performing the rewriting. The graph grammar is a specification of the family of graphs that can be constructed from a given graph instance. PROgrammed Graph Rewriting Systems (PROGRES) is such a system. The main elements of PROGRES are the following: typed nodes, labeled directed edges, a graph schema, graph rewriting rules, and algorithms that are used to select and apply the graph rewriting rules [Schurr90] [Schurr97].

Nodes and Edges: The nodes and edges are the primitive elements of a graph. A node is considered to be a data structure that stores application information as attributes. A directed and labeled edge represents a binary relationship between the source and target nodes. Edges do not have attributes.

Graph Schema: A graph schema captures the application domain knowledge. It describes
how a host graph can be constructed because it identifies what node types and edge types can be connected and the direction of the edge relationships between the nodes. It is very similar to a database schema defined in a binary entity relationship notation or object-oriented analysis class diagram.

*Graph Rewriting Rules:* A graph-rewriting rule is understood as an operation of host graph traversal and update/replacement. There are two main graph-rewriting mechanisms in PROGRES: production and transaction. A production is the most primitive graph rewriting operation. It modifies a fragment of the host graph, as illustrated in Figure 3. A transaction selects and applies one or more productions to rewrite a host graph, and it is atomic to ensure the consistency of the host graph after applying the rules. A transaction is made up of imperative control statements and declarative rules to specify, select, and apply a set of graph rewriting rules.

Figure 3 illustrates the concept of the application of the production rules from Figure 3(a) to the host graph in Figure 3(b). The left-hand side (LHS) is a sub-graph pattern that will be replaced by its right-hand side (RHS). The production rules also need to indicate in detail how the RHS will be embedded in the host graph. In this example, node 1 still exists after applying the production rule, whereas node 2 and its edges will be deleted and replaced by two new nodes 3 and node 4. A subgraph that matches the LHS is found in the host graph (if more such subgraphs exist, one will be chosen in a non-deterministic fashion) and will be replaced by the RHS. The result after the transformation is shown in Figure 3(c).
In this thesis, the composition of activity diagrams is approached as a graph-rewriting problem, as discussed in detail in Chapter 3.
2.4 Eclipse-based Design Tools

2.4.1 Eclipse

Eclipse is an excellent open source project, originally created by IBM. Eclipse has built-in Java support, but it is more than just a Java IDE, as it provides a variety of tools to support application development. The tools support C/C++, PHP, JSP/Servlets, etc. It has also been ported to different versions for different operating systems such as Windows and Linux. One of the most exciting features of Eclipse is its extensibility through the plug-in mechanism that will be discussed later in this chapter. The Eclipse Platform’s principal role is to provide mechanisms for tool builders to use and rules to follow, that lead to seamlessly integrated tools [Erickson01]. These mechanisms are exposed via well-defined API interfaces, classes, and methods. The Eclipse platform also provides useful building blocks and frameworks that facilitate developing new tools [Erickson01]. Eclipse operates under an open source paradigm, with a common public license that provides royalty-free source code and worldwide redistribution rights for tool developers with flexibility and control over their software technology [EclipseProject]. Figure 4 shows the architecture of the Eclipse platform.

The Platform Runtime: is the core kernel of the Eclipse platform. It discovers what plug-ins are installed at Eclipse start-up and creates a registry of information about those plug-ins. Since Eclipse is running on the top of a Java Virtual Machine, it will consume lots of system resource and the time spent on the start-up is significant. Eclipse, thus, uses a mechanism called “Lazy loading” to load the plug-ins and reduce the start-up time and resource usage. This means it does not load any plug-in until it is actually needed. Except for the Platform Runtime, the rest of the Eclipse platform is implemented as a plug-in.
The Workspace: the main responsibility of the workspace is to manage the user's resources. The user's resources include everything the user creates, such as projects, files, etc. Another important responsibility of the workspace is detecting and keeping track of other resource changes—such as files that are created, deleted, or changed—to prevent accidental data loss. The workspace also provides a mechanism for annotating resources, such as annotations that are used to show compiler error messages, search hits, and debugger breakpoints.

The Workbench: provides Eclipse with a user interface (UI). It is built using the Standard Widget Toolkit (SWT) and a higher-level API JFace. JFace, also built on the top of SWT, provides an easy-to-use application programming interface (API) for developers to extend the Eclipse platform and is widely used in the plug-in development. SWT is used for the entire Eclipse Platform UI and any tools that plug into it.
The Team Support Component: the main function of the team support component is to provide support for version control and configuration management. It integrates the concurrent versions system (CVS) to allow developers to work on the same project in a distribution environment.

The Help Component: provides a mechanism to allow tools to add documentation to the Eclipse platform in the form of HTML format in the same way that plug-ins add functionality to Eclipse. It also supports generating help documents in the HTML format.

2.4.2 Eclipse Plug-in

Plug-in is the smallest piece of the Eclipse platform, coded in Java, which can be developed and delivered separately. It usually consists of Java code in a JAR library. The code libraries and read-only content of a single plug-in are located together in a directory in the file system or at a base URL on a server. Eclipse has a special mechanism that allows a plug-in to be synthesized from other plug-ins, each in its own directory or URL. This is the mechanism used to deliver separate language packs for an internationalized plug-in.

Each plug-in has a manifest file named plugin.xml. The file contains information such as the dependency to other plug-ins, the extension points, and extensions to one or more extensions points in other plug-ins. Any plug-in is free to define new extension points and to provide new API for other plug-ins to use.

Eclipse provides a Plug-in Development Environment (PDE) that contains a wizard to start the plug-in projects with the basic functionalities and user interfaces to the Platform. PDE also provides a wizard to export the plug-in to installable JAR files, making plug-in development easier to distribute. Through the plug-in mechanism, Eclipse is not a closed
IDE any more; it is open to any tools vendor and allows them to seamlessly integrate their own tool into the Eclipse platform to provide a customized tool.

### 2.4.3 UML2 Plug-in

UML2 (www.eclipse.org/uml2) project is an open source project based on Eclipse Modeling Framework (EMF). It is an EMF-based implementation of the UML 2.0 metamodel for the Eclipse platform and is widely used by IBM Rational tools. The object of the UML2 project is to provide a useable implementation of the metamodel to support the development of modeling tools, to provide various test cases to support the validation of the specification, and to specify the validation rules to define and enforce levels of compliance [EclipseProject]. The UML2 also provides a common XMI schema to assist the interchange of semantic models. Models constructed in the standard XMI schema can be imported into various UML modeling tools, such as Rational Software Architect. With the help of UML2 plug-in, UML models can be programmatically created in the Eclipse using the API provided by UML2. The UML2 API can also be used to create UML models as well as parse and manipulate already created models (model objects). This in-memory model can then be transformed as per the requirements of the users and the transformed model can then be serialized in XML format (actually in an XMI file) for easier interoperability.

The UML2 plug-in can also be used to create a user-defined UML profile programmatically. In the chapter 4, we will show how to create a UML profile in detail with the help of Rational Software Architect (RSA).
2.4.4 Rational Software Architect (RSA)

RSA is an Eclipse-based tool that offers a powerful, integrated design and construction environment. RSA offers extended support for UML and many powerful visual modeling feathers to improve productivity and ease the design-to-code experience for Java, J2EE. The latest version of RSA is built on the top of open source project Eclipse platform and based on UML2.1. It supports a variety of diagram types, such as use case, class, activity, components, interactions (sequence and communication diagrams), and state machine. This allows the software developer to capture and communicate all the aspects of an application architecture using the industry standard.

RSA also offers powerful engineering ability. The powerful ability that it provides can transform from UML models to code and from code to UML model. That is, it provides both forward engineering – (model-to-code) support from UML to Java, C++, EJB, etc. – and reverse engineering – (code-to-model) support generation UML diagrams from Java and C++.

In addition to its modeling and engineering capability, RSA can also export user-created UML 2.0 models in UML2 format (XMI format); thus, this exported model can be imported into the Eclipse platform (with the help of UML2 plug-in) for further analysis and transformation. This powerful feature is extremely useful since this can produce the metaobjects of any given diagrams for further transformations.
Figure 5: RSA Screen Shot: Activity Diagram
Chapter 3: Approach to AOM composition

This chapter illustrates our approach to AOM composition by using an example, describes the metamodel of UML2.0 activity diagrams, and introduces the composition algorithm.

3.1 AOM Example

This section will use an example to illustrate the AOM composition concept and approach; the example describes the composition at the UML graphical notation level. The example used in this section is an enhanced Document Exchange System that was introduced in [Petriu05] and analyzed in [Shen+05].

3.1.1 Document Exchange System

The original Document Exchange System (DES) is a client/server system without any authorization verification. It consists of one document exchange server and multiple clients. The server will run constantly waiting for the requests from clients. Documents are stored on the server machine. There are two types of users: general users and the system administrator. A general user can get the document directory on the server, upload documents to the server, and retrieve the documents stored at the server. The system administrator can update the existing documents, access the document directory, and access the log files of the server. Each time a client sends a request to the server, the log file in the server will be updated to record this request; this log file can also be used as a trace record. The use case diagram of the original DES system is shown in Figure 6.
In this example, we will focus on the scenario for retrieving a document as the key primary scenario used for composition.

3.1.1.1 DES Primary Model

The DES primary UML model contains a structural view and a behavioural view. The structural view contains: a) the high-level software architecture represented by one or more components diagrams showing the concurrent (distributed) component instances, User and DocServer, and their required and provided interfaces (see Figure 7), and b) the
deployment of high-level software components to hardware devices. DocServer component is multithreaded, containing a Dispatcher thread that accepts the requests and dispatches them to a number of Worker threads, organized in a thread pool. As illustrated in Figure 8, the original DES is deployed on a distributed system over a local area network. The shared documents are stored on the server disk, SDisk.

Figure 7: Primary Model - Component Diagram

Figure 8: Primary Model - Deployment Diagram
Scenario

The behavioural view contains one or more scenarios modeled by activity diagrams. In this case study, we will focus on the scenario for retrieving a document, RetrieveDoc, represented in Figure 9 and 10.

From Figure 9, we can see that the dispatcher loops indefinitely, accepting a new request once the previous one was dispatched to the worker. The detailed processing of the operation doGetDocument() performed by a Worker thread, which is encapsulated in the structured activity with the same name, is given in Figure 10.
Figure 10: DES Primary Model: nested activity diagram doGetDocument

3.1.1.2 Generic Aspect Model

The original DES does not contain any security mechanisms. We intend to add an authorization mechanism which will be modeled as an aspect. In the enhanced DES, when a user makes a call for a certain server operation, an authorization check is performed and the server will perform the operation only for authorized clients. The generic authorization aspect model (unrelated to DES) is shown in Figures 11 to 14. The aspect model can be considered as a “template”, which describes a generic authorization solution. The following views are given: software architecture (Figure 11) deployment
(Figure 12) and behaviour (Figures 13 and 14) for checking whether a request from |Client to |Server for executing a generic |operation is authorized or not. First of all, the component roles, processors and some operations bears generic (formal) names that begin with ‘|’. All generic names (component roles, processors or operations) will be replaced with actual names specific to the context for which the generic aspect model is instantiated.

![Generic Authorization Aspect Model - Component Diagram](image1)

**Figure 11: Generic Authorization Aspect Model - Component Diagram**

![Generic Authorization Aspect Model - Deployment Diagram](image2)

**Figure 12: Generic Authorization Aspect Model - Deployment Diagram**
From Figure 13, we can see that the aspect behavior model contains two kinds of activities. The activities marked with darker grey color represent new functionality associated with the aspect (which will be inserted into the primary model during the composition), whereas those with a lighter background represent actions that already exist in the primary model. The nested |checkAuthorization activity is shown below in Figure 14. It is interesting to note that this structured activity has one incoming edge and two outgoing ones: "ok" for a positive result and "exc" for an exception.
Figure 14: Generic Aspect Model: Nested |checkAuthorization Activity Diagram

3.1.1.3 Context-Specific Aspect Model

The generic aspect model can be instantiated many times in different contexts. In our case, we instantiate the “template” aspect model by “binding” the formal parameters to our DES application-specific values. The binding rules in this thesis are similar to [Shen+05], and are listed below as (formal parameter, actual parameter) pairs.

- Operation bindings:

  (|operation, getDocument); (|doOperation, doGetDocument);
(checkAuthorization, checkDocAuth)

• Component bindings:

(Client, User); (Server, DocServer); (Dispatcher, Dispatcher);
(Worker, Worker) (AuthorizationRep, DocAuthorizationRep)

• Interface bindings:

(ServerAccess, DocServerAccess);

In the structural view, some of the component (operation) names are bound to actual counterparts that exist in the application context, while others are bound to new components/interfaces/operations. For instance, DocAuthorizationRep, along with its interfaces and operations, is a new component that does not have a counterpart in the primary model of the application, so it has to be added to the composed model.

The deployment diagrams in UML 2.0 show the relationships between nodes, artifacts and components. Node and artifact bindings are shown similarly to component binding:

• Node bindings:

(ClientCPU, UserCPU); (ServerCPU, DocServerCPU);

• Artifact bindings:

(ClientArtifact, UserArtifact); (ServerArtifact, DocServerArtifact);
(AuthorizationRepArtifact, DocAuthorizationRepArtifact);

Since the binding is realized by renaming the model elements, we will not show the instantiated model here.
3.1.1.4 Model Composition

In order to obtain the complete system, the context-specific aspect model needs to be composed with the primary model. In principle, the composition involves all the three relevant views: architecture, deployment and behavior view. The composition means addition, merging, or overriding of elements. The elements that exist only in the aspect model need to be added to the composed model; whereas the elements that exist in both the aspect model and the primary model need to be merged (or overridden) in the composed model.

3.1.1.4.1 Structure Composition

Structure composition is not as difficult as behavior composition. The components either exist only in the context-specific aspect model (new components) or exist in both the context-specific aspect model and primary model. The composed model will contain the union of all the components from the two models.

The composition at the deployment level, which is also a structural view, can be tackled in a similar way, by adding new nodes from the context-specific aspect model to the ones that already exist in the primary model, if necessary. In the thesis we decided to focus on the behaviour composition, which is more complex, so the structure composition was not implemented.

3.1.1.4.2 Behavior Composition

The composition of activity diagrams is approached as a graph-rewriting problem, where a subgraph $X$ found inside of a larger host graph $H$ is isolated and replaced by another
subgraph \( Y \). \( X \) is described by the left-hand-side of a transformation rule and \( Y \) by the right-hand side; the rule also specifies how to embed (i.e., connect) \( Y \) within the host graph \( H \). In our case, the host graph is the activity diagram of the primary model, \( Y \) is the subset of activities from the context-specific model that bring new functionality to the whole, and \( X \) is an element of the host \( H \) that pinpoints the insertion place. The following are some concepts that help understanding the composition:

- In a behaviour model, the direction of the control or data flow is important, so we will use the direction of the flow to define the composition.

- The aspect model contains at least one subgraph (but may contain more than one) that is new and will be inserted into the primary activity model. For instance, Figure 15 contains two subgraphs that should be inserted into the primary activity model. Each such subgraph will be converted into a graph transformation rule.

- In the composed activity model, each aspect subgraph is connected with the primary model in at least two points (but possible in more than two). If there are more than two connection points, we need to identify the input point and the output point that form the "main" flow. For example, in Figure 15, the edge \( a \) and \( b \) represent the in and output points for the "main" flow, whereas \( c \) is an extra output point.

The detailed behavior composition of this example will be explained in Chapter 5. In this section, we only give the result of the composition as an illustration. Figure 16 shows the composed model: the activities shaded in darker grey are new, while the ones with lighter background already existed in the primary model.
One issue that has to be resolved before the composition is the identification of the insertion point. In our example, there is only one place that the checkAuthorization action can be inserted, but sometime this might not be the case, which means the new part inserted may have many possible insertion places, the design has the responsibility to indicate which edge is the insertion point.

Generally speaking, there are two strategies that can be used to identify the insertion point:

- The designer indicates directly the place where the new functionality will be inserted by using special stereotypes from the Aspect Profile.

Figure 15: Aspect Activity Model with more than Two Output Points
Using pointcut specification and name based pattern matching to find the place. In this thesis, we used both strategies.

The detailed activity diagram composition algorithm that produces the composed activity diagram shown in Figure 16 will be presented in detail in Chapter 5.

3.2 Activity Metamodel

The proposed algorithm considers the UML model at the metaobject level and we will manipulate the metaobjects representing model elements to obtain the composed model.
This section will summarize the most important UML metamodel elements used in the composition of activity diagrams. For a complete metamodel description please refer the UML specification [UML2Infra] [UML2Super].

According to the UML standard, an activity diagram model shows the flow of control from one activity to another activity. It emphasizes the sequence and conditions for coordinating low-level behavior, rather than which classifiers own those behaviors [UML2Super]. Key model elements in the activity metamodel include: ActivityNode, ActivityEdge, and ActivityGroup. Figure 17 represents a simplified activity metamodel.

Activity: typically represents an invocation of an operation or a step in a business process. An activity may contain nested subordinate behaviors. An activity diagram is used to describe the workflow of the whole system. The execution sequence is modeled as activity nodes connected by activity edges. Activity nodes are specialized either as object nodes or as control nodes. In turn, control nodes are specialized as many other nodes types, as shown in Figure 18 [UMLSuper].

![Figure 17: Simplified Metamodel for UML Activity Diagram](image-url)
Activity Node: is an abstract class for the steps of an activity, and may be specialized as executable node, control node, and object node.

Activity Group: consists of a set of nodes and edges which may belong to more than one group. The activity group has no inherent semantics and can be used for various purposes, such as grouping the contents of a partition. Figure 19, which is taken from [UMLSuper], shows a partial metamodel for the activity group.

Activity Edge: is an abstract class for the connections between activity nodes; tokens can flow along the activity edges between different activity nodes. Activity edges can be control or data flow edges. A control flow edge connects two activity nodes and starts the execution of the target node once after the source node has finished its execution. A control flow edge cannot have objects or data pass along it. However, objects or data can pass along a data flow edge.
3.2.1 Aspect Profile

The focus of the thesis is an algorithm for composing an activity diagram representing an aspect with the activity diagram representing the primary model. The composition involves traversing the aspect model to find the new functions/parts that need to be added to the primary model in order to obtain the composed model. One of the key processes during composition is to find the new functionality in the aspect model that needs to be added to the primary model and to insert them into the primary model. The Aspect Profile proposed in this thesis is used to add annotations to activity diagrams elements in order to guide the composition.

More UML profiles to guide the composition of aspects can be found in literature:
However, the profiles proposed in these papers are not for composing activity diagrams. For example, the profile from [Fuentes+06] is for composing sequence diagrams and the one from [Aldawud+03] for composing State Machine diagrams.

A brief introduction to the proposed Aspect Profile and its properties will be discussed in the following section. The detailed process of how to produce it using IBM Rational Software Architect 7.0 will be described in Chapter 4.

3.2.1.1 Aspect Profile Definition

The Aspect Profile is used for attaching information to activity diagram model elements which will be used by the composition engine to perform the composition. The following annotations are provided:

- Annotations attached to Activities to distinguish between activities in the Primary model and the Aspect model.
- Annotations attached to partitions to identify new partitions that appear only in the aspect model and need to be added to the composed model.
- Annotations attached to activity nodes to distinguish between nodes that represent new functionality added by the aspect from nodes that are matching functionality existing in the primary model.
- Annotations attached to activity edges to indicate insertion points and connection points, as explained later in this section.

The Aspect Profile proposed in the thesis is a package shown in Figure 20 containing stereotypes and their attributes (also known as tagged values). The following subsections describe the stereotypes and their attributes, as well as their usage.
3.2.1.2 Advice Stereotype

<table>
<thead>
<tr>
<th>Stereotype</th>
<th>Metaclass</th>
<th>Attribute</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;&lt;Advice&gt;&gt;</td>
<td>Activity</td>
<td>Kind</td>
<td>‘primary’; ‘aspect’</td>
</tr>
</tbody>
</table>

Table 2: Advice Stereotype

The Advice stereotype is used to specify the type of the model. Like the classes, it has one attribute (property) named Kind, which has one of the two enumeration values: primary and aspect. The Advice stereotype will be used to indicate the primary model by setting
the value of Kind to primary and the aspect model by setting the value of Kind to aspect. The Advice stereotype extends the Activity metaclass, meaning that it can be applied to any activity diagram. The default value of Kind is aspect.

3.2.1.3 EdgePointCut Stereotype

<table>
<thead>
<tr>
<th>Stereotype</th>
<th>Metaclass</th>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;&lt;EdgePointCut&gt;&gt;</td>
<td>ActivityEdge</td>
<td>Kind</td>
<td>'matching'; 'in'; 'out'; 'exc';</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>'altin'; 'new', 'insertionpoint'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Name</td>
<td>Any String</td>
</tr>
</tbody>
</table>

Table 3: EdgePointCut Stereotype

EdgePointCut stereotype is used to specify the kind of an activity edge; the value of the attribute Kind can be one of the seven enumeration values: matching, in, out, new, exc, altin or insertionpoint. Later in this Chapter, some rules are given showing how to apply the EdgePointCut stereotype for some simple examples. The value of attribute Name can be any string; usually a non-empty string is required for the edges whose Kind attribute is in or out. The EdgePointCut stereotype extends the ActivityEdge metaclass and can be applied to activity edges (control flow and object flow).

3.2.1.4 NodePointCut Stereotype

<table>
<thead>
<tr>
<th>Stereotype</th>
<th>Metaclass</th>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>NodePointCut</td>
<td>ActivityNode</td>
<td>Kind</td>
<td>'new'; 'matching'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>BindingName</td>
<td>Any String</td>
</tr>
</tbody>
</table>

Table 4: NodePointCut Stereotype
The *NodePointCut* stereotype can be used to indicate the type of the activity node or to instantiate the generic aspect model in order to obtain the context-specific model. The *Kind* attribute can have one of the two values: *matching* or *new*. *Matching* means that in the primary model, there is an activity node with the same name as the activity node in the aspect model. The value *new* means the activity node is a new part that should be inserted into the primary model. *Matching* nodes are used to verify whether a proper insertion point has been found in the primary model. More exactly, all the *matching* activity nodes in the aspect model must have a counterpart with the same name in the primary model. Otherwise, the composition will not be allowed to continue.

The value of the *BindingName* attribute can be used to instantiate a context-specific aspect model from a generic one. The activity node in the composed model will be renamed to the value of the *BindingName* attribute, if it exists. The value of the *BindingName* can be any string; by default the value is a blank string and no rename action will be performed during the composition.

The *NodePointCut* stereotype extends from the activity node metaclass and can be applied to any activity node, including control nodes and object nodes.

### 3.2.1.5 GroupPointCut Stereotype

<table>
<thead>
<tr>
<th>Stereotype</th>
<th>Metaclass</th>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>GroupPointCut</td>
<td>ActivityGroup</td>
<td>BindingName</td>
<td>Any String</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Kind</td>
<td>'new'; 'matching'</td>
</tr>
</tbody>
</table>

**Table 5: GroupPointCut Stereotype**

The *GroupPointCut* stereotype is used to indicate if the activity group is a new one and
needs to be inserted into the primary model or it already exist in the primary model. It can also be used to instantiate a generic activity. The value of the *Kind* attribute can be one of the two enumeration values: *matching or new*. The *matching* means that in the primary model there is a partition with the same name as the one in the aspect model whereas *new* means the partition is a new one that should be added to the primary model. The value of the *BindingName* attribute can be any string; by default, it is a blank string. The value of the *BindingName* attribute is the name of the partition after the instantiation, if it is not blank.

The stereotype *GrouptPointCut* extends the *ActivityGroup* metaclass and can be applied to any partition.

### 3.3 Activity Diagram Composition

#### 3.3.1 AD composition overview

As we have mentioned, an AOM contains a primary model and some aspect models as well as the binding rules to instantiate them in the context of the application. A primary model reflects core design decisions and it describes core concern solutions that determine the base structure of the architecture model. Each aspect model reflects a concern that crosscuts the primary model, and each aspect model describes the crosscutting concern solution. The Aspect Profile is applied to both the aspect model and the primary model to guide the composition in order to obtain the final composed system.

The bindings are used to instantiate a generic aspect model in the context of the application. An aspect model can be instantiated many times, following different binding rules. The thesis is focused on composition of activity diagrams, which is treated as a
graph transformation problem. The following examples show the principles of the proposed AOM composition, which mimic graph rewriting principles.

3.3.2 Examples of AD composition by applying the Aspect Profile

This section will show how to instantiate the generic aspect model to obtain the context-specific model and how to compose the model by simple examples, with the help of the Aspect Profile. Figure 21 shows a simple generic aspect model before instantiation. According to [France+04], we can instantiate the generic aspect model for a given application-context by binding the generic parameters to application-specific values. A generic aspect model can be instantiated many times to obtain many context-specific aspect models based on different binding rules. In our case, the binding rules are given by using stereotypes of the Aspect Profile.

In Figure 21 the generic aspect model is instantiated by using the following binding rules:

*Group/Partition Binding:* (|Client, User), (|Server, WebServer)

*Activity Node binding:* (call |operation, call getDoc), (|doOperation, doGetDoc)

The process of instantiation occurs by expressing the binding rules with the help of the Aspect Profile. For instance, apply the GroupPointCut stereotype to partition |Client and set the value of BindingName attribute to User. After instantiation, the name |Client will be replaced with User. Please note that if there is no value attached to the BindingName attribute, then the name of the element will not be changed. The instantiated model obtained by using the above binding rules is shown in Figure 22.
We need to mention here that, in our implementation, the context specific aspect model is obtained in memory by replacing the formal parameters with actual values. The generated context-specific model can be exported to XMI by using the RSA tool, and can be read back into the tool and then seen in the model browser; however, it cannot be visualized on the tool canvas as shown in Figure 22. The activity diagram from Figure 22 has been redrawn from scratch for visualisation purposes only. The inability to visualize graphically with the RSA tool any generated model is a limitation due to the fact that the standard XMI format does not contain any layout information. In other words, the standard XMI format captures only the abstract syntax, but not the concrete syntax of UML models. (In practice, different tool vendors are adding in fact layout information in proprietary format to the XMI format. This allows for reading and displaying diagrams produced in XMI by the same tool, but “breaks down” the universality of the XMI format).
This limitation regarding layout information could be eliminated by a future diagram interchange standard, which will allow for expressing layout information in a standard manner.

The next step after instantiation is the composition of the context-specific aspect model with the primary model. In principle, we consider that the behaviour represented by the aspect model is a flow graph (either control-flow or data-flow) that has to be inserted in a larger flow graph representing the primary model. In order to make the insertion
manageable, we have put some limitations on the structure of the aspect flow graph, considering the following four cases:

1. The new functionality introduced by the aspect constitutes one flow graph connected and maximal, with a single input edge and a single output edge;
2. The new functionality introduced by the aspect constitutes one flow graph connected and maximal, with a single input edge and more than one output edges;
3. The new functionality introduced by the aspect constitutes more than one flow graph per aspect model, each one verifying the conditions for the case 1 or 2 above;
4. One of the above cases, where part of the flow graph is in a new partition.

For the sake of simplicity, in the following examples, we will not show the process of instantiation of the aspect model, only the composition. All the aspect models shown in the figures are context-specific aspect models rather than generic ones.

3.3.2.1 Case 1:

Only one pair of edge joining into and/or flowing out of the new part of the context-specific aspect model (represented in dark shade) is shown in Figure 24:

In Figure 24, action D is the new part that needs to be inserted in to the primary model to obtain the composed model. There is only one edge b, that flows into the activity node D from other node (in this case it is the activity node B), and one edge c flows out from the node D to the other node (the node C). By applying the Aspect Profile, we set the value of the Kind attribute of edge b to ‘in’ and gave it a name ‘1’ (actually, it can be any non-blank string). Similarly, we set the value of the Kind attribute of edge c to ‘out’ and gave it the name ‘1’. (Note that for a pair of ‘in’ and ‘out’ edges, the name should be the same).
This is shown in Figure 23. Assume that the new node D will be inserted between node B and node C; we indicate this by setting the value of the Kind attribute of edge a to 'insertionpoint'. We then set the value of the Name attribute of edge a to '1', which is the same as the activity edge b and activity edge c. After the composition, in the composed activity diagram, the edge b will connect with the source node (node B in Figure 23) of edge a; edge c will connect with the target node (node C in Figure 23) of edge a, and edge a will be removed. The composition can be accomplished by generating the following graph rewriting rule:

**Rule:**

- The LHS rule is edge a in Figure 23
The RHS rule is edge b, node D, and edge c in Figure 24

By applying the rule, which means replace LHS by the RHS, we would obtain the composed activity model shown in Figure 25:

Figure 25: Composed Model from Figure 23 and Figure 24.

3.3.2.2 Case 2:
One edge flows in but more than one edge flows out of the new part in the aspect model.
In Figure 27, two new activity nodes – D and E – are the new part that should be inserted into the primary model (indicated by setting their Kind attribute to 'new'). Activity node E has two edges (d and e) that flow from E to other activity nodes. One of them - let's say d - is considered the main out flow (indicated by setting Kind to 'out') and the other edge e represents a secondary flow out (indicated by setting the value of Kind to 'exit').
The nodes B and C have matching counterparts in the primary model, so we set the value of their Kind attribute to 'matching'. The composition can be accomplished by generating the following graph rewriting rule:

Rule:

- The LHS rule is edge a in Figure 26,
- The RHS rule is edge b, node D, edge c, node F, edge d, and edge e in Figure 27.

By using the RHS to replace LHS the primary model, we can get the model shown in Figure 28.
3.3.2.3 Case 3:

More than one graph transformation rule can be generated from the context aspect model. In the context-specific model shown in Figure 30, activity node F and activity node G are new parts that do not exist in the primary model (Figure 29). Thus they need to be inserted into the primary model to obtain the whole integrated model. So we can generate two graph rewriting rules as follows:

Figure 28: Composed Model from Figure 26 and Figure 27
Rule 1:
- The LHS rule of Rule 1 is edge a in the primary model (Figure 29).
- The RHS of Rule 1 is edge c, activity node F and edge d in the context-specific aspect model (shown in Figure 30)

Rule 2:
- The LHS rule of Rule 2 is edge b (Figure 29) in the primary model
- The RHS rule of Rule 2 is edge e, activity node G and edge f in the context-
specific aspect model (shown in the Figure 30).

After applying Rule 1 and Rule 2, we get the composed model, shown in Figure 31.

Figure 31: Composed Model from Figure 29 and Figure 30

3.3.2.4 Case 4:

More than one graph transformation rule can be generated and with new partitions that need to be added in the primary model. This case just adds new partitions for other cases.
In Figure 33 there is a new partition, B, which does not exist in the primary model, and in partition B there is an activity node F. The new partition B, node E and F are new activity nodes that will be inserted into the primary model. We set the value of the Kind attribute of partition B to 'new'. So the graph transformation rules for this scenario are:

**Rule 1:**

- The LHS rule is edge a in the primary model
- The RHS rule is edge a, node F, edge b, and the partition B in the context-specific
model.

Rule 2:

- The LHS rule is edge b in the primary model
- The RHS rule is edge e, node E, edge d in the context-specific model.

Applying the rules, we get the composed model, shown in Figure 34.

We should mention that after the composition, the newly inserted node F should remain in the original partition which is partition B (which is also inserted into the primary model).

Figure 33: Context-specific Aspect Model
3.4 Complete Composition Algorithm

The algorithm described here is used to generate the transformation rules at the abstract level. The detail design and implementation will be described in Chapter 4.

Before giving the algorithm, we introduce the following notations:

- \( P \) is the set of nodes in the Primary Model.
- \( A \) is the set of nodes in the context-specific aspect model.
- \( N \subseteq A \) is the set of nodes that are created by composing the context-specific aspect model with the primary model. Depending on the aspect definition, \( N \)
may be partitioned in subsets of nodes $N_i$, $1 \leq i \leq k$, so that each subset $N_i$ is contained in a maximal disjoint sub-graph $G_i$.

- **B** is the set of nodes from $A$ that are directly connected to the nodes from $N$. We also call $B$ as the *border* of $N$.

- **M** is the set of matching nodes ($M \subseteq A$) that must be present in both $A$ and $P$.

For each $N_i$, a pair of edges is identified as *in* and *out* edge for the behavior represented by that subgraph. This is consistent with the idea that a subgraph $N_i$ will substitute an edge that is identified as an *insertion point* in $P$.

Now we can use the notation to describe our algorithm for generating the graphic rewriting rule from a given aspect model:

- For each subgraph $G_i$, such that $G_i$ is connected and maximal:

  Create a graph rewriting rule $R_i$ such that

  **The LHS of** $R_i$ ($\text{LHS} (R_i)$) **contains the set of nodes** $B \subseteq P$ **such that** $B$ **is the subset of nodes of** $P$ **that are adjacent to elements added by the aspect. The activity partition nodes to which the elements of** $B$ **belong to are also included in** LHS ($R_i$).

  **The RHS of** $R_i$ ($\text{RHS} (R_i)$) **contains the elements of** $N \cup B$, **the partitions included in** LHS ($R_i$), **and the partitions which the elements of** $N$ **belong to.**

The composition algorithm described above is at an abstract level, the following flowcharts show the details of applying the composition algorithm.
Figure 35: AD Composition Flowchart

"Perform Model check" in the above flowchart verifies whether the insertion point in the primary model satisfies the conditions for performing the composition. First it finds out all the nodes, partitions in the Aspect model that are modeled as 'matching' and tries to find their counterparts in the Primary activity model. If matching nodes and partitions can be found (based on name-based matching), then it will continue to check the 'border' nodes of the new part in the Aspect model, to check if the border nodes can be found in the Primary model. If any of these nodes that are either 'matching' or as the "border" of
the new parts cannot be found in the primary model, the check/validation failed and the composition should not be allowed to continue. Basically, we do not perform a rigorous check/validation on the Aspect models, i.e. if there no new parts exist in the aspect model, the validation can still be passed and as a result no new part will be inserted in the primary model. (The model composition is performed, but actually no new part is inserted)

The following diagram, Figure 36 shows the process of the model check/validation described above.

Figure 36: Nested AD Composition flowchart – Perform Model Check
The transformation engine which will be described later in this thesis takes the activity diagrams as input model, looking for the primary model and the aspect model. It traverses the first founded aspect model to extract and store the maximal and connected graph representing new parts that will be inserted into the primary model, after that performs the model composition. If the composition succeeds, it will search for another aspect model to continue the composition until all the aspect models are weaved into the primary model. When the composition is completed, the composed model is serialized in XMI format. If any error or exception arises during the composition process, the transformation engine will show the error message to the model designer and the composition process will be terminated.

![Flowchart of Nested AD Composition](image)

**Figure 37: Nested AD Composition Flowchart – Perform Model Composition**
In the next chapter, as we present the implementation, we will describe the algorithm in more detail.

3.4.1 Assumptions and requirements

The success of our AD composition relies on the input model and the correctness of applying the Aspect Profile, thus there are some assumptions or requirements about the aspect model and primary model:

- Our proposed algorithm performs name-based composition by graph rewriting. The transformation engine will perform the default composition by name-based pattern matching if the model designer does not explicitly indicate the insertion point, so we require that every node should have a unique name in the input model (including the control nodes such as merge nodes, decision nodes). If there are multiple nodes with the same name in the same activity diagram, the correction of the composition is not guaranteed.

- The aspect models are restricted to contain one or more subgraphs of new activities, so that only one edge (marked with “in”) flows into the new part, and at least one edge marked “out” (but possible additional edges marked “exit”) are flowing out of the new part, as described in section 3.3.2. It is very important to identify the ‘in’, ‘out’ and ‘exit’ edge for the aspect model, and it is the responsibility of the model designer to identify and indicate them correctly.

- It is also very important to identify the ‘insertionpoint’ edge in the primary model. Although our composition algorithm is able to perform name-based pattern matching, we still recommend that the model designer indicates the specific insertion point by applying the Aspect Profile to guide the model composition.
Chapter 4: Implementation of an AD Composition

4.1 Introduction

This chapter describes the details of how to produce the Aspect Profile, which was discussed in Chapter 3, and how to apply the Aspect Profile to the activity diagram in RSA. The implementation of the composition is also described in this chapter.

4.2 Producing the Aspect Profile in RSA7.0

UML provides the profile mechanism that allows software designers to extend the UML for various purposes. By using the UML2 Plug-in we reviewed in Chapter 2, we could create programmatically the UML model as well as the profiles. However, creating programmatically such a UML model and profile is a tedious and error-prone process. Some tools, such as IBM Rational Software Architect, which was reviewed in Chapter 2, provide a graphical editor that allows us to easily create UML diagrams as well as profiles. This section will illustrate the process of creating the Aspect Profile in RSA7.0 in detail; the process described here is also applicable for creating other profiles.

4.2.1 Create a Profile

To create a UML profile in the Rational Software Architect 7.0, one can either use the New UML Profile Project wizard, as shown in Figure 38, or the new UML Profile wizard. To launch the New UML Profile Project wizard, click File -> New -> Other and then Modeling -> UML Extensibility -> UML Profile Project.
There are some differences between the New UML Profile and the New UML Profile Project. The New UML Profile Project wizard creates a project and a profile in it at the same time, while the New UML Profile creates a profile only and the user has to select a folder to save the profile. The New UML Profile wizard is more flexible, since sometimes the profile that you are going to create does not have to be stored in a profile project; it can be placed in any project. In our case, we choose to use the New Profile Project wizard.
### General Information

<table>
<thead>
<tr>
<th>Name:</th>
<th>AspectProfile</th>
</tr>
</thead>
<tbody>
<tr>
<td>Location:</td>
<td>C:\IBM\rationalsdp7.0\workspace\Profile.epx</td>
</tr>
<tr>
<td>Size:</td>
<td>1677</td>
</tr>
<tr>
<td>Last modified:</td>
<td>09/07/07 2:45 PM</td>
</tr>
<tr>
<td>Editable:</td>
<td>true</td>
</tr>
</tbody>
</table>

### Versions

<table>
<thead>
<tr>
<th>Version</th>
<th>Release Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 39: Profile Project Shows on the RSA7.0**

Once the profile is created, it will automatically show in the profile editor. This graphic editor will describe the general information about this profile – such as name, location, version, etc. – as shown in Figure 39 above.

**4.2.2 Create Stereotypes**

The next step is to create stereotype information for our Aspect Profile. To create a stereotype, select the Aspect Profile in the Project Explorer and right-click it, then click Add UML -> Stereotype on the context menu. We name this stereotype Advice. This result shows in Figure 40.
A stereotype can be used to attach additional semantics to the element when it is applied to the element. However, the real power of a stereotype is that it can extend one or more metaclasses through the extension mechanism. By default, the extension mechanism will not be automatically applied to all the instances of the metaclass (metaclasses). The model designer has to explicitly apply the stereotype to a modeling element that is an instance of the extended metaclass, and then the designer can attach the additional semantics to that model element. However, you can choose to make the extension *required* when the extension mechanism is defined, in which case it will be automatically applied to all model elements of the respective metaclass when the profile is applied to...
The next step is to indicate the extension metaclass for the stereotype *Advice*. The metaclasses being extended by the stereotype *Advice* are specified on the **Extensions** tab in the **Properties** view. Our *Advice* stereotype extends the Activity metaclass. The Metaclass Extensions table lists all existing extensions for the stereotype *Advice*. This is shown in Figure 41.

![Figure 41: Properties of Stereotype](image)

Basically, there are no limits on how many extensions a stereotype can extend. A new extension can be added by clicking the **Add Extension** button in Figure 41, which will bring up the **Create Metaclass Extension** dialog shown in Figure 42. The metaclasses available for extension are those defined by the UML 2.0 metamodel. Each extension can be marked as *required* or not, while the default is *not required* as explained above.
Figure 42: Create Metaclass Extension for Stereotype

The stereotype Advice in our Aspect Profile extends the Activity metaclass and sets it to not required, so that the model designer has to explicitly apply it to the model elements. This gives the maximum flexibility to the model designer.

4.2.3 Create Stereotype Attributes

When a profile is applied to a model and a stereotype is applied to the element instance of the metaclass/metaclasses, the attributes of the stereotype are added to the element instance as its properties. For example, the stereotype Advice of the Aspect Profile has an
attribute name *Type*. When this stereotype *Advice* is applied to an activity in a model, the activity’s properties will include *Type* in addition to all properties specified in the Activity metaclass, such as name, visibility, and so on.

To create a stereotype attribute for our *Advice* stereotype, select the *Advice* stereotype in **Project Explorer**, right-click on it, and choose **Add UML -> Attribute**. The result is shown in Figure 43.

![Figure 43: Attribute for Stereotype Advice](image-url)

- **Name**: Type
- **Visibility**: Public
- **Qualifiers**: Leaf
- **Type**: <Primitive Type> String
- **Default Value**: 
- **Multiplicity**: 1
- **Suppressed**: 
- **Aggregation**: None

**General Information**
- This section describes
- **Name**: 
- **Location**: 
- **Size**: 
In the RSA’s Properties view, we can find the properties for an attribute. For our Aspect Profile, the stereotype Advice has only one attribute with the name Type. Other properties for an attribute are Type, Default Value, Multiplicity, and Qualifiers.

Type: all attributes of a stereotype must have a type. The attribute of a stereotype cannot be just anything, since the attribute will eventually become extended metaclass properties when the stereotype is applied to the model elements. It has to be a type recognized by the system; otherwise the system cannot provide proper information of these properties. Three kinds of types can be chosen from in the RSA: Primary types from the model libraries, Enumerations type, and Classes type defined by the user in the profile.

If the type of an attribute is not specified when the profile is released, the RSA will indicate warnings or errors in the Problems view, as shown in Figure 44. If the warnings or errors are not resolved, the tool will normally ignore that attribute and will not add the stereotype-contributed properties to the element when the stereotype is applied to a model element.

![Figure 44: Warnings of the Profiles](image)

Default Values: the default value does not have to be specified. In case the default value
is missing, the system will use its own default value, depending on the type. For example, the default for a String typed attribute is a blank string. But for safety, we suggest the default value should be indicated.

Multiplicity: the multiplicity is used to indicate that the property should be treated as a single value or as a collection of values (multiplicity greater than 1). The property is a collection of values when the multiplicity is set to greater than 1. If the property is set to as a collection of value, we need to pay attention to the Unique field of the Qualifiers. The Unique field is used to indicate if the elements in the collection have to be unique or not. By default, the element is set to unique.

For our Advice stereotype, the type is set to an Enumeration, which was defined in our Aspect Profile. The Enumeration also predefined some values called Enumeration Literals. Enumerations have to be defined in the same profile as the stereotype itself, if one wants to use Enumerations to specify the type of stereotype attributes.

4.2.4 Create Enumeration Type and Enumeration Literals

To create an enumeration, right click the AspectProfile in the Project Explore and select add UML; in the context menu choose Enumeration. We create an Enumeration named Type, which contains two Enumeration Literals: aspect and primary. This will be used to distinguish the type of an activity. When the type value is set to primary; it means the activity is a primary activity model.
Now we can set the value of the Type attribute of the stereotype Advice to the Enumeration type we have just created, as shown in Figure 46.
The attribute of the Type property has only two values: primary and aspect. When the user applied the Aspect Profile to a model and applied the stereotype Advice to an activity, the activity can be distinguished by the Type property.

4.2.5 Version Problems of a Profile

Each profile file can contain one or more versions of the profile. Every time the profile is saved, a new version will be created and saved. The old versions will not be removed; they still remain in the profile file.

Normally, only the latest version will be used, since it can completely and accurately reflect the UML definition of the profile. To see the current (latest) version number of the profile, select it in the Project Explorer and then click on the General tab in the Properties View.

4.2.6 Release a Profile

After creating all stereotypes and defining their attributes, we need to release the profile before we can use it. Releasing a profile in RSA we created is very simple. Choose the AspectProfile in the Project Explore, right-click it, and choose release in the context menu, as shown in Figure 47.
Figure 47: Release a profile

This will bring up the **Release Profile** dialog shown in Figure 48, where the user can specify the release label for the profile. The release label can be any string.

Figure 48: Profile Release Information

Figure 49 shows how all released versions of the profile are listed in the **Releases** field on the **General** tab in the **Properties View**.
A profile that is not released should be applied to a model for testing purposes only. The model that was applied with a non-released profile can be migrated into the released version later. However, it may lose some data during the migration; for instance, it may lose stereotypes or values of the stereotype and thus it may become inconsistent. Releasing a profile can enforce the tool to check the consistency of the model. RSA will discover any inconsistency in the profile definition and show a warning if any inconsistency is found. Releasing a profile before it is applied to any model is a good practice that is always suggested.

4.3 Applying a Profile in RSA

The profiles that are already added to a model show in the Profiles tab of the Property view, as shown in Figure 50.
Figure 50: Applied Profiles to a Model

To apply a given profile to this model, select the model in the Project Explorer, choose the Profiles tab in the Property view, and click the add profile button. This will bring up a dialog that allows a profile to be selected and added to the model, as shown in Figure 51.
Once the profile has been added to the model, stereotypes and tagged values can be added to metaelements. To add a stereotype to an element, select the element in the canvas, click the Stereotypes tab in the Property view, and click the Add Stereotypes button. This will bring up a dialog box showing the available stereotypes that can be applied to that model element.
Figure 52: Apply Stereotype for a Model Element

Figure 53: Available Stereotype Properties for a Model Element
As shown in Figure 53, when a model element is selected in the **Modeling** view, the **Stereotypes** tab shows the applied stereotypes for this model element. For each stereotype, the properties are listed below with a “Value” column, which allows a value to be assigned to the attribute.

Once a profile is applied to a model, the stereotypes are available for the appropriate element. When the element is selected, the attributes of these stereotypes for the applied element are then visible in the **Stereotypes** tab and the values can be set accordingly. When it is saved, the model elements contain all the stereotype information for future use.

### 4.4 Implementation of the Transformation Algorithm

Our transformation algorithm relies on the correctness of the application of the Aspect Profile. Applying the profile must follow the pattern discussed in Chapter 3; otherwise, we cannot guarantee that the generated model will be correct.

#### 4.4.1 Design Decisions

Before implementing our transformation algorithm, the following issues were considered:

* **Tools Selection**: the tool used in our implementation is IBM RSA7.0. Some reasons why we chose to use RSA7.0 are listed below:
  - RSA7.0 is based on the latest UML2 plug-in, which implements the UML2 specification.
  - RSA7.0 has many features that were discussed in Chapter 2; these features can save lots of time and reduce the errors during the implementation.
  - Our implemented result – the plug-in can be integrated seamlessly with RSA7.0,
which will simplify the transformation process by removing the process of reading
the models into the tool.

Composition method: we use an approach similar to graph rewriting to perform the
diagram composition, which involves reconstructing the input graphs. Using RSA, we
can build the activity diagrams easily as the inputs, but we cannot change the input
diagrams directly. This means that we cannot manipulate the in-memory objects of the
input diagrams. For example, we cannot change the source object and target object of an
activity edge, or RSA will crash. The only solution is to reconstruct a copy of the input
diagrams, which means manipulating a copy of the in-memory objects of the original
input diagrams. In our implementation, we create a copy of these objects as well as the
profiles applied to these objects and manipulate the copied objects to generate the
composed activity diagram. Making a copy of these objects is not an easy job since in
Java; one has to implement the clone method in each class. However, with the help of the
Eclipse Modeling Framework (EMF), cloning UML elements is easier than cloning other
Java objects.

To summarize, our transformation algorithm will extract the metaobjects that exist in
memory as well as their stereotypes and tagged values that contain user directives and
make a copy of all the meta-objects as well as their stereotypes and tagged values. The
copied model will be used as the inputs to the transformation engine. The transformation
engine will also check/validate the input model, follow the user directives, if they exist,
for composition, transform it into a composed activity diagram, and, finally, serialize the
composed model in an XMI format. Some important steps in the composition – such as
model serialization, extracting in-memory meta-objects, and model transformation – are
encapsulated in classes. To help understand the implementation, we give the class diagrams in the Figure 54. We will examine some of the important classes in details in the following sections.

![Class Diagrams of the Algorithm](image)

**Figure 54 : Class Diagrams of the Algorithm**

### 4.4.2 The SaveHelper Class and SaveImpl Class

The transformation algorithm is given the input UML diagrams representing the primary and aspect models, and it transforms them to generate the composed model according to the user directives. Finally, the generated model is written to a file in XMI format. The
SaveHelper class offers a mechanism to save the generated model using the UML2 API built in RSA. By using this API, the generated model is produced in standard XMI format and can be readily imported into other tools that use the same API. The following code snippet shows two of the most important methods in this class.

```java
public class SaveHelper{
    protected static final ResourceSet RESOURCE_SET = new ResourceSetImpl();
    protected static void registerResourceFactories() {
        Resource.Factory.Registry.INSTANCE.getExtensionToFactoryMap().put
        (UMLResource.FILE_EXTENSION, new XMIResourceFactoryImpl());
    }
    protected static void registerPathmaps() {
        ...
    }
    protected static void saveAll(org.eclipse.uml2.uml.Package package_, URI uri) {
        Resource resource = RESOURCE_SET.createResource(uri);
        EList contents = resource.getContents();
        contents.add(package_);
        Iterator all = UMLUtil.getAllContents(package_, true, false);
        while (all.hasNext()) {
            EObject eObject = (EObject) all.next();
            if (eObject instanceof Element) {
                contents.addAll(((Element) eObject).getStereotypeApplications());
            }
        }
        try {
            resource.save(null);
        } catch (IOException ioe) {
            System.out.println(ioe.getMessage());
        }
    }
}
```

In this class, there is only a protected final field Resource_Set. The entire UML package (model) is treated as a kind of resource and stored in the Resource_Set. The registerResourceFactories() and registerPathmaps() are used to configure the Resource_Set so that it can serialize its content according to the configuration. The SaveAll() method is used to serialize the content of the Resource_Set. When it is invoked, it will output everything in the contents of the Resource, which was created by
Resource_Set using the user supplied parameter URI (which can be imaged as the path address of the serialized model) according to the configuration of the Resource_Set. The `registerResourceFactories()` and `registerPathmaps()` is to tell how to instantiate a Resource_set. For example, the `registerResourceFactories()` will instantiate the Resource_set based on the file extension. If the file extension is uml, then it will serialize the content of the resource in an XMI standard way. Many file extensions are defined in the UML2 API, but we only used this one.

The `SaveHelper` class is the super class of the `SaveImpl` Class. `SaveHelper` class provides some generic methods that will be used in the `SaveImpl` Class, such as resource factory register. The `SaveImpl` class then provides the details implementation of saving models; it provides some Graphical User Interface (GUI) methods to allow the user to save the generated model.

One of the main methods of `SaveImpl` is the `savePackage()` method; the code snippet is shown below.

```java
public class SaveImpl extends SaveHelper {
    Shell shell;
    private org.eclipse.uml2.uml.Package package_

    ..

    public void savePackage() {
        FileDialog fileDialog = new FileDialog(shell, SWT.SAVE);
        fileDialog.setText("Save created model");
        String[] filterExtension = { ".uml", "*.*" };
        fileDialog.setFilterExtensions(filterExtension);
        if (fileDialog.open() != null) {
            if (fileDialog.getFileName() != null
```
The FileDialog class is a class from the Eclipse SWT/JFace, which was reviewed in Chapter 2. It provides a mechanism that allows the user to select a directory path to save files or retrieve documents. The filterExtension is used to filter documents based on the predefined file extension, when it is supplied; it only shows documents with the specified file extension in the dialog.

The savePackage() method utilized the three methods inherited from the SaveHelper class to perform the task of serializing the model.

### 4.4.3 Extract and Manipulate Model elements from the RSA

Before we perform the transformation, we need to extract the model elements that exist in memory. We use the model elements to feed into our transformation engine and manipulate the model elements to perform the transformation. The key to extracting model elements from the memory in Eclipse-based tools is the IAdaptable interface;
**IAdaptable** is an import interface of the Eclipse platform. We know that Java is a strongly typed language; each instance has a type associated with it. There are actually two kinds of types: the declared type and the runtime type. Every instance has a runtime type. The *IAdaptable* interface provides such a mechanism to add functionality to an existing class at run-time, and dynamically convert one type of object to another without change the existing APIs. The following example will illustrate the idea of the *IAdaptable* interface in the Eclipse platform.

To convert an Array of string into XML node, we want the XML that looks like:

```xml
<Array>
  <Entry> First </Entry>
  <Entry> second </Entry>
  <Entry> Third </Entry>
</Array>
```

We cannot directly use the Array’s `toString()` method, since the result produced is not the form we need. Instead, we can attach the *IAAdapterFactory* to the *Array*:

```java
public class NodeArrayFactory implements IAdapterFactory {
    ...
    ...

    public Object getAdapter(Object array, Class clazz) {
        if (clazz == Node.class && array instanceof Array) {
            Element root = new Element("Array");
            For (int i=0; i<array.length; i++) {
                Element item = new Element("Entry");
                item.appendChild(array[i].toString());
                root.appendChild(item);
            }
            return root;
        } else {
            return null;
        }
    }
    ...
    ...
}
```

We then need to register our factory with the Eclipse platform adapter manager. When completed, we can ask any Array instance for a node. Registering an adapter with Eclipse
is also as easy as shown in the following code:

```java
Platform.getAdapterManager().registerAdapters(
    new NodeArrayFactory (), Array.class);
```

Now we can safely ask for the nodes from any Array; this is as simple as shown in the following code snippets:

```java
Node getNodeFrom(IAdaptable array) {
    Object adaptable = array.getAdapter(Node.class);
    if (adaptable != null) {
        Node node = (Node)adaptable;
        return node;
    }
    return null;
}
```

With the help of the `IAdaptable` interface, getting the reference of the UML model is relatively simple, as shown in the following code snippets.

```java
if (obj instanceof IAdaptable) {
    IAdaptable adaptable = (IAdaptable) obj;
    EObject object = (EObject) adaptable.getAdapter(EObject.class);
    if (object instanceof org.eclipse.uml2.uml.Package) {
        package_ = (org.eclipse.uml2.uml.Package) object;
        ...
    }
}
```

We can first convert the object to the EObject and then test to determine if it is an UML model. If it is an UML model, we keep the reference of the model. Once we can get the reference of the model, which exists in the memory, we then can utilize the UML2 API to
access the profiles applied to the model and the stereotype applied to the model elements. We can also do other functions such as manipulate the model elements to change their properties, rebuild a new model, etc. The process of extracting those in-memory models from RSA is encapsulated in the ComposeAction class.

4.4.4 The CreateModelEngine Class

The CreateModelEngine class can be used to create the composed activity diagram, which was performed by the createComposedActivity() method. This is shown in the following illustrated pseudocode snippets:

```java
public void createComposedActivity(){
    ...
    validateModel(aspectActivity);
    bindingNames(aspectActivity);
    findNodesByType(aspectActivity, type)
    findEdgesByType(aspectActivity, type)
    addPartitionsToActivity(primaryActivity);
    addNewNodesToPartitions(List newNodeList);
    addNewEdgesToPartitions(List newEdgeList);
    connectEdgesToNodes(List edgeList);
    ...
}
```

For the `validateModel()` method, we first analyze the aspect activity diagram, finding out all the matching nodes, and verify if they exists in the primary activity diagram. If yes, the validation succeeds and the transformation continues; otherwise, we prompt the model designer about the mistakes.
We instantiate the aspect activity diagram by finding the applied stereotype and the tagged value. If the tagged value exists, then the name is bound according to the binding rules given by the user. Otherwise, the name remains unchanged.

After instantiation, we need to find the new parts that will be inserted into the primary model. The `findNodesByType()` and `findEdgesByType()` return a list that contains those nodes and edges according to their type; for example, type "new" will return a list that contains the nodes and edges modeled as "new" that will be inserted into the primary model. After we find those nodes and edges, we still need to examine the new partition and to add the new partitions into the primary model.

We insert the new nodes and new edges into the primary model. The nodes and edges are still in the same partitions as they were in the aspect model.

The last step is to reconnect the nodes that were just inserted into the primary model with the nodes in the primary model. In this process, there two scenarios may occur:

1. User indicates the insertion points
   
   In this case, all we need to do is follow the user directives to insert the new part to the indicated point. We find the name of the "insertionpoint" edges and its source node and target node, connect the source node with the "in" edge that have the same name of the "insertionpoint" edge, and the target node with the "out" edge that has the same name of the "insertionpoint" edge. Finally, we need to remove the "insertionpoint" edges.

2. User does not indicate the insertion points.
This case is more complicated since we need to perform the default composition. We first find the source node (S) of the “in” edge (I) and the target node (T) of the “out” edge (O) in the context-specific model. Then we traverse the nodes in the primary model to find the node that matches the node S and node T; we call it S’ and T’ (please note we use the name based rule to find the nodes, which means S’ must have the same name with S). If S’ and T’ are connected directly by edge E’, then we can insert the new part (the maximum connected part, which is between the “in” edge (I) and “out” edge (O)) to the position of E’, connect edge I with node S’, edge O with node T’, then remove E’. The process will repeat until all the new parts are inserted into the primary. If any part that cannot be inserted into the primary model exists, the default composition cannot be performed. In this case, the model designer has the responsibility to indicate the position of the insertion point.

Normally, scenario 1 and 2 will exist concurrently in the transformation and our transformation engine will perform both of the transformation rules to generate the composed model, although, the transformation will always perform the default transformation rule if no composition directives are available or the composition directives are insufficient to compose all the new parts, we suggest that the model designer indicate the insertion points to reduce or avoid the mistakes.
Chapter 5: Verification and Case Study

In the previous chapters, we have discussed the concepts and the algorithm that we used to compose the primary activity diagram with the aspect activity diagram and we illustrated the AOM composition by a DES system. In this chapter, we will discuss the test cases used for the verification of the implementation and will complete the composition of the DES system by using our transformation engine and will compose another aspect with the DES system.

5.1 Test Cases

Table 6 gives a summary of the main test cases used for verifying the implementation. The test cases #1 - #4, tests the simple transformations described in Chapter 3 which covers all the features of the composition algorithm. The more complex test case, Test Cases # 5, verifies a mixed combination of some of these simple transformations using a case study, as described in section 5.2.

<table>
<thead>
<tr>
<th>TC#1</th>
<th>One edge in, one edge out</th>
<th>One edge in, more than one out</th>
<th>More than one graph rewriting rule (no new partition)</th>
<th>More than one graph rewriting rule (with new partitions)</th>
<th>Composition with multiple Aspect Models</th>
</tr>
</thead>
<tbody>
<tr>
<td>TC#2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TC#3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TC#4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TC#5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6: Summary of Test Cases
5.1.1 Result Evaluation

The model generated by the transformation engine is in XMI format, and it must satisfy two criteria: correctness and interoperability between UML tools. One of the key features of XMI is that XMI eases the problem of tool interoperability by providing a standard way for model interchange.

The XMI document is generated using the transformation engine and the API built in RSA. RSA has its own built-in XML tools that can validate XMI format documents that are fed back into RSA. RSA can discover any errors based on the given XMI schema as well as confirm that the documents produced are well formed. If the XMI schema is not compatible or the version is different than the one in RSA, RSA will not show those model elements properly. For example, the model generated by RSA6 cannot be shown correctly in RSA7, since the XMI schema that is used to serialize model is different.

We imported our generated model back into RSA7.0 to test the interoperability and correctness. The model elements that are supported by RSA7.0 will be recognized correctly and shown in the Project Explorer of RSA, as illustrated in Figure 55. It should be mentioned that the tool cannot show the diagrams as the usual UML notations because XMI does not contain any layout information.
5.2. Case Study

5.2.1 CheckAuthorization aspect model composition

By using the binding rules gave in the Chapter 3 for the DES system, we can get the instantiated context-specific aspect activity model which is shown in Figure 56. The input models for our transformation engine are Figure 9 from page 23, the primary activity model, and Figure 56, the context-specific aspect model. The primary model and the context-specific aspect model are stereotyped with the Aspect Profile. The model designer also has to indicate the insertion point to guide the composition rules.
The transformation engine will take care of inserting the new behavior defined by the context-specific aspect model into the primary model and serialize the composed model in XMI format. The activity partitions of the composed model are the union of those from the aspect and the primary model and the new actions remain in their original partitions.

5.2.1.1 Importing the Composed Model Back to RSA

After the composition, we can import our generated model into RSA to verify the correctness, as shown in Figure 57. All the activity nodes and edges and their applied stereotypes are kept and are shown in the Project Explorer.
5.2.2 CheckAuthentication Aspect Model Composition

By composing with the Authorization aspect model, the security of original Document Exchange System is enhanced, however, we should know that authorization sometime is tightly coupled with the authentication.

Authorization checks the user's ability to access a resource based on the user account
rights and privileges, while authentication is used to provide assurance that users are who they say they are. An authentication may be as simple as a password challenge system. In order to verify the identity of a user, the authentication system typically asks the user to provide his unique information such as password, fingerprint, etc, if the authenticating system can verify the presence of such information, the user is considered authenticated.

Figure 58: Generic CheckAuthentication Activity Aspect Model
A secured Document Exchange System can be generated by composing the primary model with the checkAuthentication aspect model and the checkAuthorization aspect model. As the checkAuthorization aspect model is already composed with the primary model, the next step is composing the checkAuthentication aspect model. Since the focus of the thesis is activity composition, we will not show the generic checkAuthentication deployment model. The generic checkAuthentication activity aspect model is shown in the above Figure 58.

The binding rules used to instantiate the checkAuthentication is given in the following pairs:

- **Operation bindings:**
  
  (call |operation, call getDocument), (|doOperation, doGetDocument)

- **Component bindings:**
  
  (|Client, User); (|Server, DocServer);

By applying the binding rule, we would get the instantiated context-specific model, shown in Figure 59.

The input model for our transformation engine is the primary model, the context-specific checkAuthorization aspect model and the context-specific checkAuthentication aspect model. The transformation engine will search the input model to find out the primary model and the aspect models, perform the validation; instantiate the aspect model by renaming the nodes if given the binding parameters and then compose the aspect model one by one with the primary model.
Figure 59: Context-Specific checkAuthentication Aspect Model

The composed model obtained by weaving the two aspect models with the primary model is given in Figure 60. Please note the Figure 60 is for illustration only; the composed model is generated in XMI format and cannot be displayed in RSA as a visualized activity model because there is no layout information contained in the generated model.
The activity nodes in dark grey are the new parts that inserted into the primary model.

The nested checkAuthorization activity diagram is shown in Figure 14 in Chapter 3.

The generated XMI file for the composed model is given in the Appendix.
Chapter 6: Conclusions and Future Works

6.1 Conclusions

The thesis proposed and implemented a Java-based algorithm to compose aspects with the primary model, all represented as activity diagrams. The thesis introduced the algorithm at the UML graphical notation level as well as at the metamodel level.

The input of the transformation algorithm is the Java data structure of a UML 2.0 annotated model, which contains the primary model and aspect model. It can be created either by an Eclipse-based tool with graphical UML editor, such as RSA7.0, or programmatically with UML2, an Eclipse plug-in tool. The output is the composed model containing the composed activity diagram in standard XMI format. The standard XMI format facilitates information exchange and provides interoperability among different tools and/or development teams in collaborative environments. The serialized model in XMI format can be read by other tools for further analysis or transformation.

An important part of the thesis is the application of the Aspect Profile for our composition. The composition process is heavily reliant on the Aspect Profile profile, which gives the user a mechanism to include the binding rules and composition directives as stereotype and tagged values. Once the Aspect Profile has been applied to the model and the stereotype applied to model elements, the model then contains user directives that will be fed into our transformation engine. The correct profile application is very important to the success of the composition. We assume that the user will apply the
Aspect Profile to the input diagrams, following the patterns presented in Chapter 3. If the input model does not follow these restrictions, there is no guarantee that the composition will be correctly performed. The implementation demonstrates the feasibility of the approach that composes two activity diagrams at the UML metaobject level.

6.2 Limitations

As we mentioned, the model generated by the transformation is in standard XMI format. The benefit of the XMI format is that the model can be fed back into other UML tools that use the same XMI schema. However, the XMI standard does not define a standard layout format for the diagrams, so the model we generate in the XMI format does not contain any layout information. Thus, the UML tool cannot display the composed diagrams graphically; just list the model elements in the model browser (see section 5.1.1 or 5.2.1.1). Thus the readability of the composed model is considerably reduced.

The second limitation restricts the way in which the aspect models are built. Our algorithm works only if the new functionality introduced by an aspect models is contained in one or more connected subgraphs of new activities, so that only one edge (marked with “in”) flows into the new part, and at least one edge marked “out” (but possibly additional edges marked “exit”) are flowing out of the new part, as described in section 3.3.2. We found that many security aspects can be described this way: with one input point, a normal output point and potentially any number of output points returning exceptions. However, the composition algorithm needs to be developed in order to accept more a general aspect behaviour. (For instance, models with more than one input)
Another limitation we should mention is the sequence of composing multiple aspects into the same primary model. The thesis deals only with the case where the aspects are independent and non-interfering. In this case, the order of composition should not matter.

6.3 Future Work

The following are some of the many possibilities for future research:

- The first extension would be to implement the composition of the structural views of the aspect model and primary model.

- The second extension is to develop the composition algorithm to accept more general aspect models (with several inputs and outputs).

- Another extension would be to add performance annotations to the input UML models, and extend the composition algorithm to deal with the performance annotations. The composed model annotated with performance information could be transformed into performance models by using the PUMA technique developed in the Carleton RADS laboratory.

- The composition in this thesis is actually implemented in Java by manipulating UML meta-objects. It utilizes the UML2 API to parse the UML input models, to generate the composed model, and finally to serialize the generated model in XMI format. One possible extension would be to use the new standard MOF-QVT model transformation languages for performing the composition. Another possible extension is to use eXtensible Stylesheet Language Transformation (XSLT) to directly transform the XMI files into other XMI file by a set of composition rules that could be used to substitute the Aspect Profile. The composition rules may also be in the XML format, which contains the user composition directives to indicate how to
transform the input models.

- An activity diagram generated from the transformation engine is in XMI format; thus, it is more abstract and less readable. It would be very useful to display such activity diagrams in a UML graphical tool, so that the user could see the actual UML graphical notation. An impediment to this is the fact that XMI does not contain any standard layout information and, thus, the model generated using XMI schema also does not contain any standard layout information.

- Extend the ability to compose multiple aspects that may interfere with each other. The order of composition may matter and recursive composition of an aspect inside another one may be also considered.
References


[AOSD] Http://www.webopedia.com/TERM/A/aspect_oriented_software_development.html, last accessed 11/12/06


[EclipseProject] www.eclipse.org


[Erickson01] Mar Erickson, "Working with Eclipse Platform"


volume 3273 of LNCS, pages 84-97. Springer, 2004


[Mahoney+05] Mark Mahoney , Tzilla Elrad “Weaving Crosscutting Concerns into Live SequenceCharts Using the Play Engine” 7th International Workshop on Aspect-Oriented Modeling


[RationalRSA] Rational Software Architect,
/index.html, last accessed on July 12, 2007


[Reddy+05] Raghu Reddy, Robert France, Sudipto Ghosh, Franck Fleurey,
Benoit Baudry. “Model Composition- A Signature based approach”

Aspect Oriented Modeling (AOM) workshop held in conjunction with MODELS/UML 2005 conference, Montego Bay, October, 2005


[UML2Super] UML2 Super Structure Specification V2.1.1 formal/07-02-06

[UML2Infra] UML2 Infrastructure Specification V2.1.1 formal 07-02-05


[Whittle+06] Jon Whittle, João Araújo Ana Moreira “Composing Aspect Models with Graph Transformations” Workshop on Early Aspects at ICSE:
Aspect-oriented requirements engineering and architectural design,
28th International Conference on Software Engineering (ICSE 2006)


- 105 -
Appendix. Composed Model: Primary Model composed with checkAuthentication and checkAuthorization Aspects

<?xml version="1.0" encoding="UTF-8"?>
<xmi:XMI xmi:version="2.1" xmlns:xmi="http://schema.omg.org/spec/XMI/2.1"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmni:AspectProfile="http://schemas/AspectProfile/_uEYFsDrEEdyUA5pziQ6JRA/8"
xmni:uml="http://www.eclipse.org/uml2/2.0.0/UML"
xmni:schemaLocation="http://schemas/AspectProfile/_uEYFsDrEEdyUA5pziQ6JRA/8 ...
.../Profiles/Profile.epx#_uEYFsTrEEdyUA5pziQ6JRA">
  <uml:Model xmi:id="_U6McM0LEdy8SqBNG-U1Q" name="CaseStudy">
    <packageImport xmi:id="_U6MccOLEdy8SqBNG-U1Q"
      href="pathmap://UML_LIBRARIES/UMLPrimitiveTypes.library.uml#_07 />
    <importedPackage xmi:type="uml:Model"
      href="pathmap://UML_LIBRARIES/UMLPrimitiveTypes.library.uml#_07"/>
    <packagedElement xmi:type="uml:Activity"
      xmi:id="_U6Mcs0LEdy8SqBNG-U1Q" name="Activity1">
      <node xmi:type="uml:InitialNode" xmi:id="_U6Mc80LEdy8SqBNG-U1Q"
        name="start"
        outgoing="_U6MsM0LEdy8SqBNG-U1Q"/>
      <node xmi:type="uml:OpaqueAction" xmi:id="_U6MdM0LEdy8SqBNG-U1Q"
        name="call getdocument" outgoing="_U6Mks0LEdy8SqBNG-U1Q"
        incoming="_U6Mts0LEdy8SqBNG-U1Q _U6MvM0LEdy8SqBNG-U1Q"
        inPartition="_U6M6c0LEdy8SqBNG-U1Q"/>
      <node xmi:type="uml:OpaqueAction" xmi:id="_U6MdcOLEdy8SqBNG-U1Q"
        name="idle" outgoing="_U6Mj80LEdy8SqBNG-U1Q"
        incoming="_U6Mps0LEdy8SqBNG-U1Q"
        inPartition="_U6M7M0LEdy8SqBNG-U1Q"/>
      <node xmi:type="uml:JoinNode" xmi:id="_U6Mds0LEdy8SqBNG-U1Q"
        name="join1"
        outgoing="_U6MmM0LEdy8SqBNG-U1Q"
        incoming="_U6Mj80LEdy8SqBNG-U1Q _U6Mks0LEdy8SqBNG-U1Q"/>
      <joinSpec xmi:type="uml:OpaqueExpression" xmi:id="_U6Md80LEdy8SqBNG-
        U1Q"/></node>
    </packagedElement>
  </uml:Model>
</xmi:XMI>