XMI-based Transformation of UML Interaction Diagrams to Activity Diagrams

by

Eric C. Wong

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

Master of Science
Information and Systems Science

School of Computer Science
Carleton University
Ottawa, Canada

January 2002

The undersigned recommend to the Faculty of
Graduate Studies and Research acceptance of the thesis

XMI-based Transformation of UML Interaction Diagrams to Activity Diagrams

Submitted by **Eric C. Wong**, B. Sc.

in partial fulfillment of the requirements for the degree of
Master of Science
Information and Systems Science

Director, School of Computer Science

Dr. Dorina Petriu, Thesis Supervisor

Carleton University
January 2002

ABSTRACT

The thesis proposes and implements a transformation method that takes UML interaction diagrams as input and generates equivalent activity diagrams as output. The transformation approach takes into account the concurrency characteristics of the interacting objects. The thesis describes the proposed transformation rules both at UML notation level, which is more intuitive, and at UML metamodel level, which corresponds to the actual implementation. A Java application was designed and built in the thesis for realizing the proposed transformation. The application takes as input XML files produced by an existing UML tool, which contain interaction diagrams in XMI format. The XMI standard defines how to represent UML models in XML in order to facilitate information interchange between different tools. The activity diagrams produced by our transformation are also represented in XMI format.

ACKNOWLEDGEMENT

First and foremost, I extend my deep gratitude and thanks to my supervisor, Dorina Petriu, for her encouragement and guidance through this research. Without her help, advice and trust, my studies in this area would not have been possible.

I would like to thank Carleton University, the School of Computer Science and the Department of Systems and Computer Engineering for giving me the opportunity to do my thesis work. I would also like to thank the members of RADS lab for creating a wonderful working environment and for their technical support.

I extend my appreciation to all my friends for their helpful comments and suggestions. Special thanks goes to Kathleen Hui Shen for her ideas and discussion. This work is reliant on those mentioned in the reference and upon the people mentioned above. I thank them for the solidity their shoulders have granted me.

The financial support from Communication and Information Technology Ontario (CITO) is greatly appreciated.

Finally, I'm very grateful to my parents for their moral support and constant encouragement during my studying at Carleton University.

TABLE OF CONTENTS

List of Figures

List of Tables

List of Acronyms

API	Application Program Interface
CASE	Computer Aided Software Engineering
CCM	CORBA Component Model
CORBA	Common Object Request Broker Architecture
DOM	Document Object Model
DTD	Document Type Declaration
FSM	Finite State Machine
LQN	Layered Queueing Network
MOF	Meta Object Facility
MSC	Message Sequence Chart
NSUML	Novosoft UML
OCL	Object Constraint Language
OMG	Object Management Group
PGML	Precision Graphics Markup Language
RFP	Request for Proposal
RPC	Remote Procedure Call
SAX	Simple API for XML
SVG	Scalable Vector Graphics
UML	Unified Modeling Language
W3C	World Wide Web Consortium
XMI	XML Metadata Interchange
XML	eXtensible Markup Language
XSL	XML Stylesheet Language

Chapter 1 Introduction

1.1 Motivation for the Thesis Research

In a relatively short time the Unified Modeling Language (UML) from OMG has emerged as the industry standard for designing and visualizing software systems. It provides several kinds of diagrams, which allow the description of different aspects and properties of systems, like static and behavioral aspects, interaction among system components and physical implementation details. Meanwhile, it has been recognized that performance analysis should be integrated in the software development life cycle from the early stages. Reasons that are in favor of using performance analysis during the software development include the end users' expectations, cost control and the fact that performance requirements are better met if attention to performance problems is paid earlier rather than later. Software Performance Engineering (SPE), introduced by Smith in her pioneering work [Smith90], has been the first comprehensive approach to the integration of performance analysis into the software development process, from the earliest stages to the end. More motivations for using performance engineering can be found in [Smith90], [Kahkipuro99] and [Wang99].

OMG recognized the importance of performance analysis by issuing a UML Performance Profile [Profile01]. The profile identifies the basic abstractions used in performance analysis, and describes how these abstractions are expressed in terms of lightweight extensions to the UML metamodel. By using UML models annotated with quantitative performance information, one can generate a performance model in order to conduct quantitative

performance analysis of the software represented by the UML models. The feedback to software designers gained from the performance evaluation, will give them insights in the crucial aspects of the system and allow them to refine the design at the UML model level.

There are various types of performance models, which include queueing networks and their extensions called Extended Queueing Networks (EQN) [Williams+98] and Layered Queueing Networks (LQN) [Woodside+95], Stochastic Timed Petri nets (STPN) [King+99], Stochastic Process Algebras (SPA) [Pooley99] and simulation models.

An open research problem and challenge is to completely automate the process of deriving performance models from software specification and to integrate the supporting tools in a unique environment, as shown in Figure 1-1 [Petriu+01a].

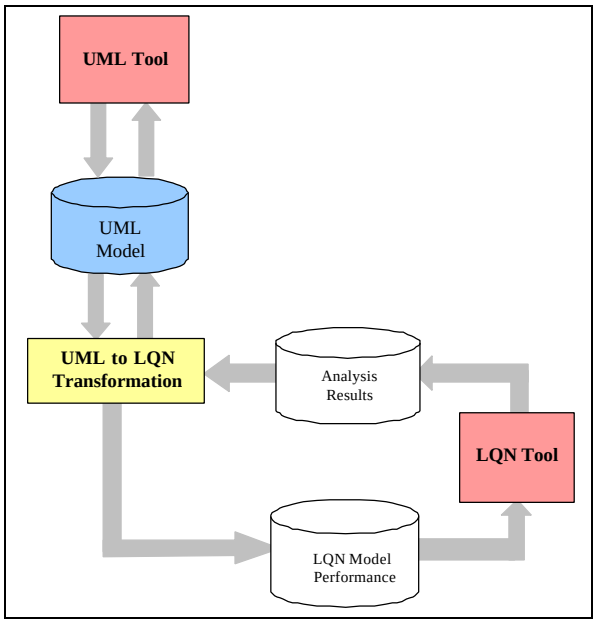

Figure 1-1 UML to LQN

Attempts ([Wang99], [Amer01]) have been made for the automatic transformation of a UML model into a LQN performance model using a graph rewriting tool PROGRES [Schurr90]. One of the limitations of using PROGRES is that it introduces an extra step necessary for translating UML models in XMI format into PROGRES input files. More research is under way to build a Java application that reads UML models in XMI format obtained from UML tools and transforms them into LQN performance models [Petriu+01a]. The work done in the current thesis is a part of this larger research effort.

Since performance is a dynamic property, scenarios play a key role in determining a system's performance characteristics from its UML models. [Profile01] decomposes a scenario in a sequence of one ore more scenario steps that are ordered conform to a general

predecessor/successor relationship. In UML a scenario is an instance of a use case [Quatrani98, pp. 65] and provides a means for the end user and the domain expert to state their expectations about the desired behavior of a system to its developers [Booch94].

Scenarios are usually modeled either using interaction diagrams or activity diagrams. Both interaction and activity diagrams describe the inter-object behavior of a system with emphasis on different aspects. They both provide the overall operations of a system. Interaction diagrams describe the detailed sequence of behavior from object to object and from method to method [Booch+99]. However, interaction diagrams usually represent a single scenario or a part of scenario. It is not always clear how different scenarios are pieced together in the system execution. UML offers another way to express the overall flow of control, in the form of activity diagrams. Although statechart diagrams also describe behavior, they provide a localized view of an object (intra-object behavior), which is not particularly useful for representing scenarios.

1.2 Scope of the Thesis Research

As mentioned before, the work done in this thesis is a part of a larger project to generate automatically performance models from UML models annotated with quantitative performance information.

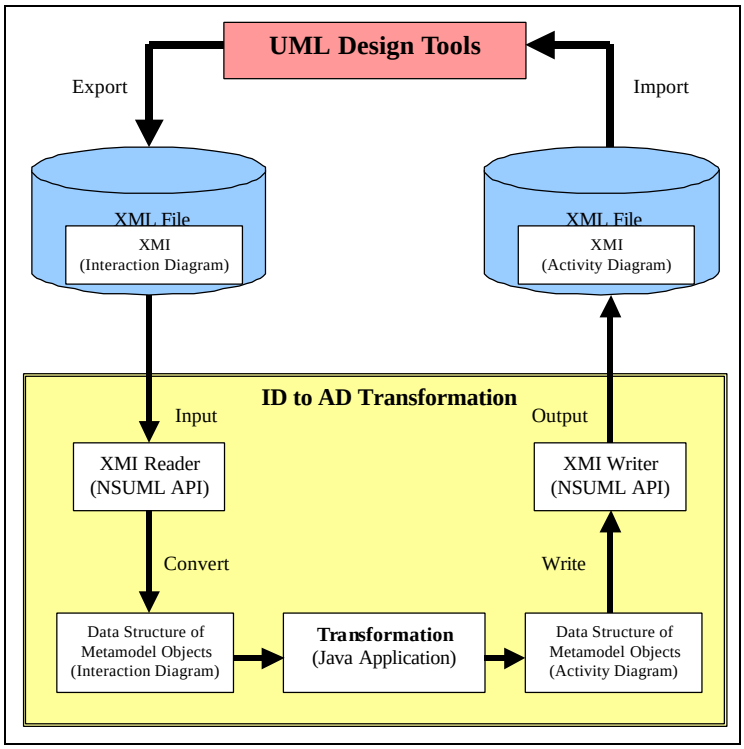

Figure 1-2 Scope of the Thesis

The subject of the thesis is to implement an automatic transformation of an interaction diagram (input) into an activity diagram (output). Both the input and output are represented in XMI format. The thesis approach is illustrated in Figure 1-2. A CASE tool, (e.g. Rose or ArgoUML), generates an XML file representing a UML model containing different diagrams. In the thesis, however, we are interested only in interaction diagrams (sequence or collaboration) and activity diagrams.

The Java application build in the thesis is represented by the gray rectangle named ID to AD Transformation. First, an XMI reader reads an XML file generated by a UML tool, converts its elements to UML metaobjects, and builds the internal data structure for the

model. Then the interaction diagrams from the model are converted to activity diagrams by applying the appropriate rules. The activity diagrams that are generated are contained in a new model. Finally, an XMI writer writes the new model to an XML file conforming to XMI format, which can be imported again by UML tools for further usage.

Our application uses the metamodel library NSUML and its API, which help us to read, process and create XMI files. XMI integrates three key industry standards: 1) XML – eXtensible Markup Language, a W3C standard; 2) UML – an OMG modeling standard; 3) MOF – Meta Object Facility, an OMG metadata repository standard. The integration allows developers to share object models and other metadata over the Internet in a standardized way, thus bringing consistency and compatibility to applications created in collaborative environments.

1.3 Contributions

The goal of the thesis is to define and implement a transformation process that accepts as input UML interaction diagrams (i.e. sequence or collaboration diagrams) and produces as output equivalent activity diagrams that represent the same behavior as the input diagrams. The contributions of the thesis are summarized as follows:

- Define transformation rules from interaction diagrams to activity diagrams at the UML notation level. The transformation rules take into account the concurrency characteristics of the interacting objects, and generate activity diagrams that contain a separate swimlane for every thread of control.

- Express the above transformation rules at the UML metamodel level. Identify the metamodel classes/objects used to represent the interaction and activity diagrams, and express each transformation rule in terms of metamodel objects, their attributes and relationships.
- Design, implement and test a Java application that realizes the above transformation. The application takes as input XML files produced by an existing UML tool, which contain interaction diagrams in XMI format encoded according to the XMI standard [XMI1.1]. The input interaction diagrams are transformed into equivalent activity diagrams, which are expressed also in XMI format according to the XMI standard.

1.4 Thesis Outline

The thesis includes 6 chapters, which are structured as follows:

Chapter 2 gives an overview of the background literature for the thesis. The UML and XMI are described at first, followed by a short introduction of two UML design tools. Then the UML Performance Profile is briefly presented, as the thesis transformation is part of a larger project aiming to transform UML models into performance models.

Chapter 3 explores the behavior aspects represented in interaction diagrams and activity diagrams, and gives a high-level view of the transformation approach, followed by the transformation rules expressed at UML notation level.

Chapter 4 investigates the UML metamodel and the Novosoft UML API that was used to implement the transformation [NSUML99]. The API implements the UML metamodel. The chapter continues by describing the transformation rules at the metamodel level. The algorithm for the transformation is also given.

Chapter 5 describes the Java implementation of the proposed transformation, focusing on the XMI input, output and transformation procedures. The verification of the Java application is discussed, and a case study is also investigated.

Chapter 6 summarizes the thesis research and opens some directions for future work.

Chapter 2 Literature Review

This chapter presents an overview of the background information related to the thesis, such as the Unified Modeling Language (UML), the eXtensible Markup Language (XML), the XML Metadata Interchange (XMI), UML design tools and the newly proposed UML Performance Profile.

2.1 Unified Modeling Language

The Unified Modeling Language (UML), adopted as a standard (UML 1.1) by OMG in 1997, has been rapidly and widely adopted and has almost completely superseded the earlier OO (Object-Oriented) methodologies, such as the Object Modeling Technique (OMT) [Rumbaugh+91], Booch's Methodology [Booch94], OORAM [Reenskaug96], Syntropy [Cook+94] and many others. The version 1.3 [UML1.3] is used throughout the thesis. The latest version 1.4 has been adopted as the standard in September 2001, which is described in [UML1.4]

Formally, UML is defined by an Object Management Group (OMG) document containing 9 sections. The following sections are particularly relevant to the thesis:

- UML Semantics: This section defines the UML "abstract syntax" in the form of a set of UML packages. Each package contains a set of UML class diagrams describing the UML metaclasses and their relationships. Each class in the metamodel and its attributes are described in English. Well-formedness rules for

UML models are expressed in the Object Constraint Language (OCL) [Warmer+99]. Each package within the meta-model is further described by additional English text that explains the intended interpretation of the elements in the package.

- UML Notation Guide: This section describes the graphical notation for the elements that compose eight kinds of UML diagrams, and how they work together. Examples of UML diagrams are given together with English description. Each notation element has a "Mapping" that describes in English how it is represented by the elements in the metamodel. The notation guide also contains brief summary of semantics.
- UML XMI DTD Specification: This section defines the XMI DTD for UML 1.3. The OMG XMI standard [XMI1.1] specifies a structure for interchanging models that uses XML (eXtensible Markup Language). One of the primary goals of providing this DTD is to enable OO modeling tool interoperability. As with the IDL (Interface Definition Language) definition [IDL99], the UML metamodel is subjected to minor modifications to create a "physical metamodel", which is then mapped into XML Data Type Declarations (DTDs) – schemas that define the structure of XML representations of UML models.

Although the UML provides a rich set of modeling concepts and notations to meet the needs of typical software modeling projects, users may sometimes require additional features and/or

notations beyond those defined in the UML standard. For this purpose, UML provides three built-in extension mechanisms:

- *Constraints* place semantic restrictions on particular design elements. UML uses the OCL to define constraints.
- *Tagged Values* allow arbitrary information to be attached to any model element.
- *Stereotypes* allow groups of constraints and tagged values to be given descriptive names and to be applied to other model elements.

Using the above mechanisms enables us to represent new concepts in UML. For instance, we could choose tagged value to add quantitative performance information to the UML diagrams. In fact, the extension mechanisms are used to define so-called UML profiles, which specialize UML for different application domains. One of the profiles being currently defined is described in section 2.4.

2.1.1 UML Metamodel

The UML metamodel is defined as one of the layers of a four-layer metamodel architecture, depicted in Figure 2-1. The four layers are:

- M_0: domain-specific information
- M_1: model of the domain-specific information, e.g. in UML
- M_2: meta-model, e.g. definition of UML
- M_3: meta-meta-model, e.g. definition of the way that UML is defined.

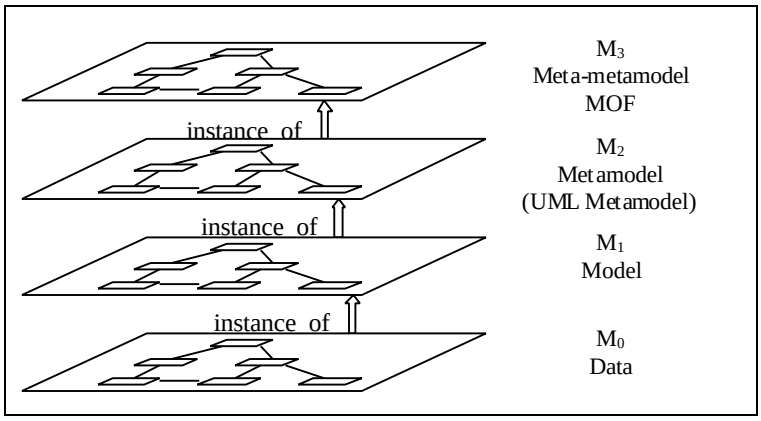

Figure 2-1 Four-layer Metamodel Architecture

The fundamental relationship between these layers is intended to be the instance-of relationship, which is clearly expressed in the UML specification [UML1.3]. The M_3 level, Meta Object Facility (MOF), defines the basic concepts from which specific metamdoels are created at the meta (M_2) level. This includes the UML metamodel, which is regarded as being an instance-of the MOF meta-metamodel. We will describe in more detail parts of the UML metamodel related to interaction and activity diagrams in chapter 4. Normal user models, created using the concepts of the UML, are regarded as residing at the M_1 level, and the ultimate run-time data is regarded as residing at the M_0 level.

It is always a difficult task for the UML specification to improve precision while maintaining readability. For this reason, the current UML semantics is informally specified. The definition of semantics, such as dynamic behavior, is expressed in English, which sometimes may lead ambiguity. Whether a UML model conforms to the semantics or not is purely a matter of human interpretation based on reading of the English. On the other hand,

UML defines a set of diagram types, but there is no metamodel representation of a diagram itself. In other words, diagram interoperability is not supported in general because there is no functional mapping between metamodel elements and diagrammatic elements. It is the responsibility of UML tools to map the UML notation to the metamodel elements. In fact, we found out that different UML tools use sometimes different mappings of the diagram elements to metamodel elements, which is against the spirit of the standard.

2.2 XML and XMI

UML is used extensively to model object systems. It can't, however, capture implementation details, interoperability semantics, information exchange format and so on. Over the past few years the OMG has created an architecture for managing metadata. This has resulted in several official metadata standards. The core standard is the Meta Object Facility (MOF). XMI (XML Metadata Interchange) is an extension of the MOF into the XML space. Thus, before focusing on the XML and XMI, it is important to grasp the basic concepts of the MOF.

MOF is a self-describing meta-metamodel used to describe UML, a set of technology metamodels (such as the CORBA Component Model (CCM), the Enterprise JavaBeans (EJB)), as well as any other user-defined metamodels. The MOF standard selects a subset of UML that is appropriate for modeling metadata. This subset is called the MOF core. The key point is that the MOF core is independent of CORBA, Java, XML or any other

middleware technology. This is due to the fact that UML (of which the MOF core is a subset) is technology-neutral.

MOF also contains a set of rules that define the interoperability semantics and information (metadata) exchange format for a given information model. The MOF to IDL (Interface Definition Language) transformation rules can be applied to any metamodel to produce a well-defined API. In addition to the API, the MOF rules also define the DTD corresponding to the metamodel. The current official version of MOF is 1.3 and described in [MOF1.3].

2.2.1 eXtensible Markup Language (XML)

XML is a new standard adopted by the World Wide Web Consortium (W3C) to complement HTML for data exchange on the Web. It is a way of working with information in a structured form. The description of XML in the thesis emphasizes its role as a data exchange format, not that of a document markup language. In this subsection, we outline the major features that make XML great for information storage and interchange. More information on XML can be found in [W3C], [Ducharme99], [Leventhal+98], [Abiteboul+00], [Maruyama99] and [Armstrong01].

XML is designed specifically to describe content, rather than presentation. The major features are summarized as follows:

1. New tags may be defines at will.
2. Tags identify the information and break up the data into parts.
3. XML documents are always constrained to be well-formed.
4. Structures can be nested to arbitrary depth.
5. An XML document can contain an optional description of its grammar.

XML allows users to define new tags to indicate the structure of their documents. It tells one what kind of data one has, not how to display it. Hierarchical structures make XML documents faster to access and easier to manipulate. Since XML is inherently style-free, a completely different stylesheet, such as XML Stylesheet Language (XSL) that lets you dictate how to portray the data, can be used to produce output in postscript, TEX, PDF, or some new format that hasn't even been invented yet.

XML consists of two parts: documents and DTDs (Document Type Declarations). DTD serves as grammar for the underlying XML document ([Martin99], [Maruyama99]). An XML document is valid if it conforms to his DTD. In other words, elements in a valid document may be nested only in the way described by the DTD and may have only the attributes allowed by the DTD. The use of XML introduces the need for extra tools such as parsers [XML] or APIs like Simple API for XML (SAX) [Armstrong01] and Document Object Model (DOM) [DOM]. More on the way XML is used for interchanging UML models is discussed in chapter 5.

XML is fast becoming the data representation of choice for the Web, especially when used in combination with network-centric programs that send and retrieve information. For example, a client/server application could transmit XML-encoded data back and forth between the client and the server [Armstrong01].

2.2.2 XML Metadata Interchange (XMI)

The “X” in XMI means both XML and eXtensible. XMI is designed to be compatible with upcoming XML technologies, which include Namespaces, XLinks, XPointers, and XML-Schema [Abiteboul+00]. In particular, XMI will use the future capabilities of XML-Schema, directly using new features such as XML data types and improved mechanisms for DTDs. The XMI generation rules described below provide that extensibility.

XMI defines two sets of rules that provide open interchange and leverage the capabilities of XML: DTD generation and document generation. The DTD generation is used to specify an interchange format, and document generation creates documents that use a given XMI DTD. The current official version of XMI is 1.1 and described in [XMI1.1].

XML DTDs alone do not have the ability to express the semantic meaning appropriate for the model. They require a whole sets of additional concepts that are only available through complete information architectures, such as UML, MOF, and others being developed by the OMG. For example, an UML-based DTD allows interchange of object-oriented UML

models. This results in the ability to interchange at both the data level (XML) and the semantic level (UML) [OMG].

2.2.2.1 XMI DTD Architecture

The XMI DTD architecture provides the necessary infrastructure for information transfer by defining a uniform treatment of object identity, internal and external references, document partitioning, tool-specific extensions, round-trip exchanges, incomplete models, and differences [XMI1.1].

Every XMI DTD contains the elements generated from an information model, e.g. a UML model, plus a fixed set of element declarations that may be used by all XMI documents. These fixed elements provide a default set of data types and document structure, starting with the top-level XMI element. Each XMI document contains one or more elements called *XMI* that serves as a top level container for the information to be transferred. XMI is a standard XML element and may stand alone in its own document or may be embedded in XML or HTML documents. Detail descriptions of DTD design and generation principles can be found in [XMI1.1].

2.2.2.2 UML DTD

UML DTD is the most widely used XMI DTD. It is a physical mechanism for interchanging UML models conforming to the UML metamodel. This metamodel, so-called physical metamodel, is fed into an XMI DTD generator to produce the UML DTD used by

tools to export and import UML models [UML1.3]. UML physical metamodel will be discussed in more later in chapter 4. Figure 2-2 shows the central part of the UML metamodel [UML1.3]. The metamodel is actually more extensive than this class diagram suggests, but most of the elements not shown here derive in some way from these backbone elements.

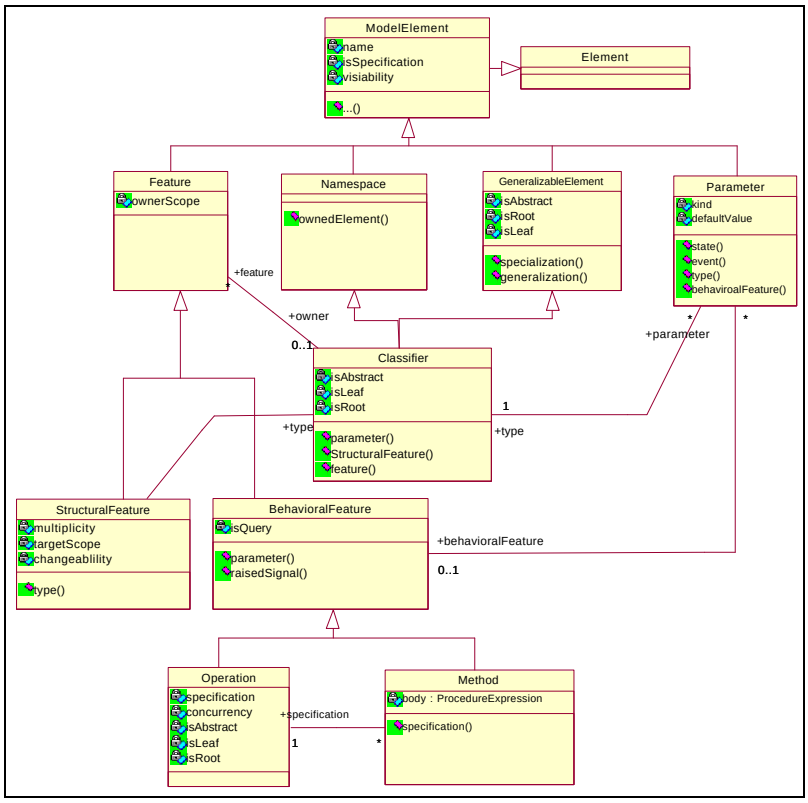

Figure 2-2 UML Metamodel: Backbone

2.3 UML Design Tools

UML design tools are UML-based CASE (Computer Aided Software Engineering) tools that support the use of design diagrams in the development of an object-oriented software

(also known as OOAD (Object-Oriented Analysis and Design)) tools. Assuming that XML will become a universal data exchange format, many software vendors are building tools for importing and exporting XML data. In this section we describe two UML design tools in brief: ArgoUML [ArgoUML] and Rational Rose [Rose]. Both tools use third-party products to support the UML DTD, and thus are XMI-compliant. We do not intend to give comparisons of the tools, instead, the description concentrates on their XMI aspects. Detailed user manuals for ArgoUML and Rational Rose are given in [ArgoUML] and [Rose], respectively.

One thing worth to point out is that when drawing diagrams with a tool, a notation on a computer screen may contain additional invisible information. Besides, not all modeling information is presented most usefully in graphical notation. Tools are responsible for keeping the consistence between the notation and the underlying model.

2.3.1 ArgoUML

ArgoUML is a Java-based cognitive CASE tool, and also an Open Source Development project where users are invited to contribute [ArgoUML]. Figure 2-3 shows ArgoUML's main window which has a menu bar and four main panes: *navigation*, *editing*, *to do* and *details*.

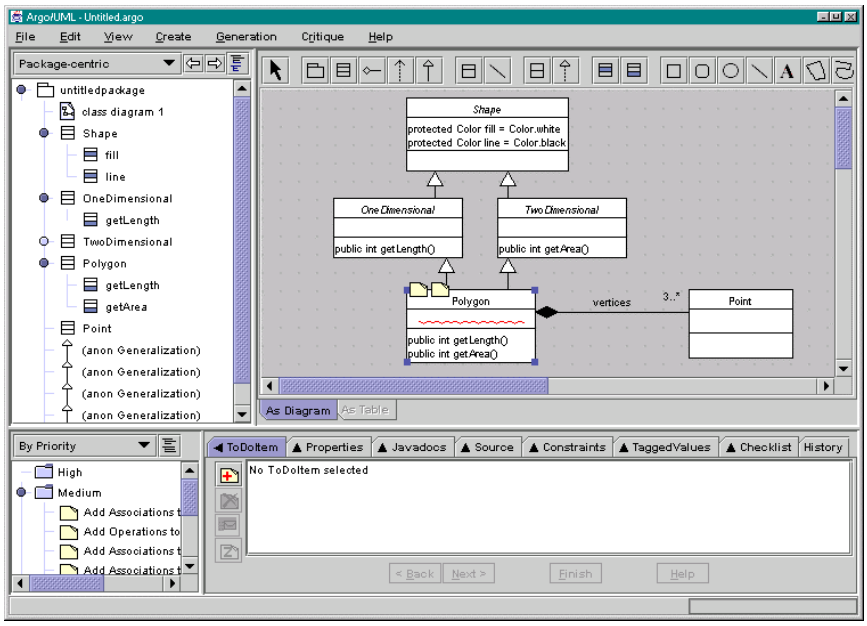

Figure 2-3 ArgoUML's Main Window

The major features related to UML and XMI are described briefly below:

- Runs on platform with Java 1.2: ArgoUML is coded entirely in Java and uses the Java Foundation Classes.
- Standard UML metamodel: ArgoUML is compliant with UML 1.3. The code for the internal representation of an UML model is genreated by a special metamodel library NSUML developed by Novosofts [NSUML99]. Some advanced features of UML are not yet available in the diagrams, but the foundation to completely fulfill all of UML is laid.
- XMI-Support: ArgoUML uses XMI (XMI version 1.0 for UML 1.3 is used) as standard saving mechanism so that easy interchange with other tools and compliance with open standards are secured. The NSUML not only implements the

UML metamodel, but also provides XMI reader and writer to enable importing and exporting for a UML model. Currently only the model information is saved in XMI, but no graphical information (like layout of diagrams).

- Diagram export formats: The standard saving format for diagrams is Precision Graphics Markup Language (PGML), but it will be changed to the upcoming standard for Scalable Vector Graphics (SVG) of the W3C consortium.

2.3.2 Rational Rose

Rational Software Corporation's well-known Rose modeling tool has led the object-oriented analysis and design market for years. With the three OOAD pioneers who started the creation of UML – Grady Booch, James Rumbaugh, and Ivar Jacobson – on Rational's staff, it is not surprising that Rose was one of the first tools to support the UML. Today, Rational is one of the OMG's most active participants in maintaining and enhancing the standard [Hess00].

Rose features include expanded round-trip engineering, support for UML 1.3, and built-in team development. It also includes Rose Extensibility for developing add-in functionality. Several third-party vendors already have used the extensibility features to integrate Rose with their tools or environments. One of them is Unisys' XMI add-in. Rose does not directly support generating XMI from UML models. Instead, Unisys' XMI add-in provides this support, which is available at Rational's Web site. It uses an XML tagging scheme that lets other modeling tools to work with Rose diagrams, so different organizations can

collaborate on software projects. In the thesis, we used the XMI add-in to generate XML files containing interaction diagrams, which are the input to our transformation process.

2.4 Performance Profile

As we mentioned in chapter 1, the thesis implements the transformation from interaction diagrams to activity diagrams, which, combined with information taken from other diagrams, will be used to build automatically performance models from UML models ([Amer01], [Petriu+00b]). Software Performance Engineering (SPE), initially introduced by [Smith90], integrates performance evaluation into the software development process from the early stages throughout the whole life cycle. Related works on building performance models for OO system from software specifications can be found in [Smith+97], [Kahkipuro99], and [Cortellessa+00].

OMG noticed that the lack of a quantifiable notion of time and resources in UML was an impediment to its broader use in the real-time and embedded systems. As a consequence, OMG issued a request for proposal (RFP) asking for a UML profile for “schedulability, performance and time”. A first draft of the profile was made public in August 2000, and an improved version in June 2001 [Profile01]. The profile focuses on properties that are related to modeling of time and time-related aspects such as timeliness, performance and schedulability. In particular, the profile does not intend to invent new analysis techniques. Rather, it is aimed to be able to annotate a UML model in such a lightweight way that

various existing and future analysis techniques will be able to take advantage of the provided features.

2.4.1 Performance Modeling Techniques

The profile on performance modeling (chapter 8 in [Profile01]) describes the following general performance analysis of UML models:

- associating performance-related *QoS characteristics* with selected elements of a UML model.
- specifying *execution parameters* which can be used by modeling tools to compute predicted performance characteristics.
- presenting performance results computed by modeling tools or found in testing.

Typical tools for this kind of analysis provide two important functions. The first is to estimate the performance using some kinds of modeling techniques. The second is to improve the system by identifying bottlenecks. There are three common techniques used in most modeling tools:

- Queueing Models: Define workloads that execute particular aspects in different scenarios. This may require the distribution of the demand, passive resources as well as devices, and the detailed scenario sequence. Different queueing models have been extended, such as Extended Queueing Networks (EQN) and Layered Queueing Network (LQN) developed in [Woodside+95].

- Simulation Models: Define multiple logical tokens which execute the software, following the detailed scenario structure and using execution time distributions for the operations of each step.
- Discrete-state models such as Petri Nets: Define tokens which execute the software, following the detailed scenario structure.

According to the SPE methodology [Smith90] and to the Performance Profile [Profile01], the building of a performance model starts from frequently executed scenarios annotated with performance information. These scenarios can be modeled in UML either by interaction diagrams or by activity diagrams. Even though we do not use directly the Performance Profile in the thesis, the automatic transformation from interaction to activity diagrams, which is the goal of the thesis, represents an important step in the process of building performance models from UML models.

Chapter 3 Consistent Behavior Representation in Interaction and Activity Diagrams

A picture can tell, what a thousand words can't. [Unknown]

This chapter investigates first the UML diagrams used to model behavior: interaction diagrams (including sequence and collaboration diagrams), activity diagrams, and statechart diagrams. The thesis will then focus on those diagrams that are most appropriate for describing scenarios: interaction and activity diagrams. The chapter continues by defining a transformation that takes an interaction diagram as input and generate the corresponding activity diagram as output.

The chapter is organized as follows. In the first section we describe briefly the elements contained in interaction and activity diagrams. Next we propose transformation rules represented by basic cases at the notation level. The metamodel representations behind the transformation rules will be described in next chapter.

3.1 Conceptual Description

UML describes five complementary views that are important in visualizing, specifying, constructing, and documenting a software architecture: the use case view, the design view, the process view, the implementation view and the deployment view. Each of these views involves structural modeling, as well as behavioral modeling [Booch+99].

The static parts of a system are typically described by one of the four following diagrams: class diagram, object diagram, component diagram, and deployment diagram. The UML provides other additional diagrams to view the dynamic parts: use case diagram, interaction diagram (sequence and collaboration diagram), activity diagram and statechart diagram.

Each kind of diagrams focuses on a certain perspective of the system. Particularly, interaction diagrams represent the behavior of a set of objects (inter-object behavior), while statechart diagrams look at each object individually and provides a narrow and deep view of its behavior (intra-object behavior). Activity diagrams, on the other hand, emphasize the flow of activities and can be used to represent both inter-object and intra-object behavior. In the UML metamodel, the activity diagrams (which are a later addition to UML) are considered as a kind of statecharts. However, this representation does not emphasize the big difference between statecharts and activity diagrams. The former are attached to individual objects and express only intra-object behavior, whereas the latter can be attached either to an object (to describe its behavior) or to a use case or an interaction diagram (to describe inter-object behavior).

The semantics and notation of statechart diagrams in the UML standard are substantially those of Harel's statecharts [Harel87]. A statechart diagram describes all the possible states a particular object can get into and how the object's state changes as a result of events that reach the object. It separates an object from the rest of the system and examines its

behavior in isolation. Statechart diagrams overcome the limitations of traditional Finite State Machines (FSMs) while maintaining the benefits of finite state modeling by introducing the concepts of both nested hierarchical states and orthogonality [Fowler97]. Further description on statechart diagrams is given in [UML1.3] and [Booch+99]. In the thesis we are concerned with modeling scenarios, therefore we will not use statecharts.

In next two sections, we describe in more detail the interaction and activity diagrams that are particularly relevant to the thesis.

3.1.1 UML Collaboration

A *collaboration* is "a society of classes, interfaces, and other elements that work together to provide some cooperative behavior that is bigger than the sum of all its parts" [Booch99, pp. 27-371]. It describes a collection of objects that interact to implement some behavior within a context. A collaboration has both a structural aspect and a behavioral aspect. The structural aspect defines the context by a set of roles and their relationships, which is typically rendered using class or object diagrams. The behavioral aspect specifies the dynamics of how those elements interact (i.e. how the set of messages exchanged by the objects are bound to the roles), which is typically rendered using an interaction diagram.

Figure 3-1 shows the relationship between Collaboration and Interaction in the UML metamodel (Conventionally the initial letter is capitalized to mean a metaclass in UML). A Collaboration specifies a set of roles played by Objects, and one or more corresponding

Interactions show how Objects cooperate with each other when playing these roles. Each role in a Collaboration is described with a Classifier Role, which specifies a projection of a Class. A Classifier Role expresses which features declared in a class (such as Attributes and Operations) are required in the Collaboration. An object playing a specific role must conform to the Classifier Role, i.e. the Object must offer the operations stated by the Classifier Role, and must contain Attribute Links corresponding to the Attributes of the Classifier Role. Moreover, a Collaboration defines an Association Role which specifies what associations are needed between the participating Classes.

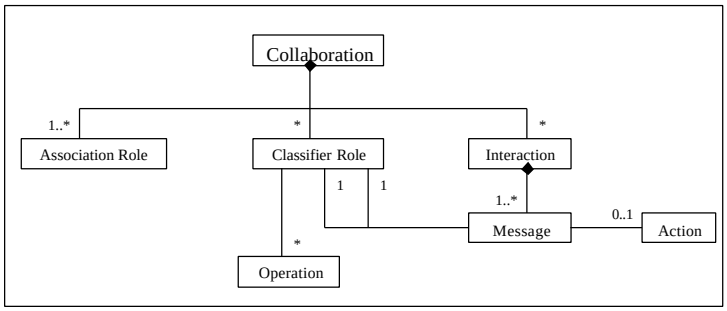

Figure 3-1 Collaboration and Interaction in the UML metamodel

3.1.2 Interaction

An Interaction is defined within the context of a Collaboration. More precisely, it specifies a collection of Messages between the various Classifier Roles of the Collaboration. Each message specifies one specific kind of communication. A certain realization of a Message is expressed with a Stimulus. A set of cooperating Objects playing the roles in the Collaboration interact according to the Messages of the Interaction by sending Stimuli to each other. These set of Stimuli are partially ordered based on the execution threads they

belong to. Within each thread, the Stimuli are sent in sequential order, while Stimuli of different threads may be sent in parallel or in an arbitrary order.

The order of Stimuli (Messages) reveals how the flow of control takes place among the objects (notice that although Stimuli and Messages are semantically different, they are treated in the same way in our implementation). This information is described by a *sequence-expression* used to label the Messages. The following are samples of Message labels:

```
2: message-name(argumet-list)
1.3.1: message-name(argument-list)
2.1a: message-name(argument-list)
[x<0] 4: message-name(argument-list)
```

The sequence–expression (such as 1.3.1) is a dot-separated list of sequence-terms followed by a colon (‘:’) [UML1.3, pp. 3-125]. Each term represents a level of procedural nesting within the overall interaction. If the control is concurrent, then nesting does not occur. Each sequence-term has the following syntax:

label recurrence

where *label* is

integer

or

name

The *integer* represents the sequential order of the Message within the next higher level of procedural calling. An example is: Message 2.1.4 follows message 2.1.3 within activation 2.1. The *name* represents a concurrent thread of control. An example is: Message 2.1a and message 2.1b are concurrent within activation 2.1. The *recurrence* represents conditional or iteration execution, in which the UML does not prescribe their formats. An example for a condition would be: [x<0]. An example for an iteration would be: *[i := 1..n]. Sequence number is a sequence-expression without any recurrence terms. It must match the sequence number of another Message.

For a procedural flow of control, the sequence numbers are nested. For a nonprocedural sequence among concurrent objects, the sequence numbers are not nested and are at the same level. Moreover, sequence numbers indicate a *predecessor*/*activator* association among Messages. The predecessors are the set of Messages that must be completed before the current Message may be executed. The activator is the message that invoked the procedure which in turn invokes the current message. The message corresponding to the numerically preceding sequence number is an implicit predecessor. An example is demonstrated in Figure 3-2, message 1.1 is the predecessor of message 1.2 within activation 1, whereas message 1.2.2a and message 1.2.2b are concurrent within activation 1.2.

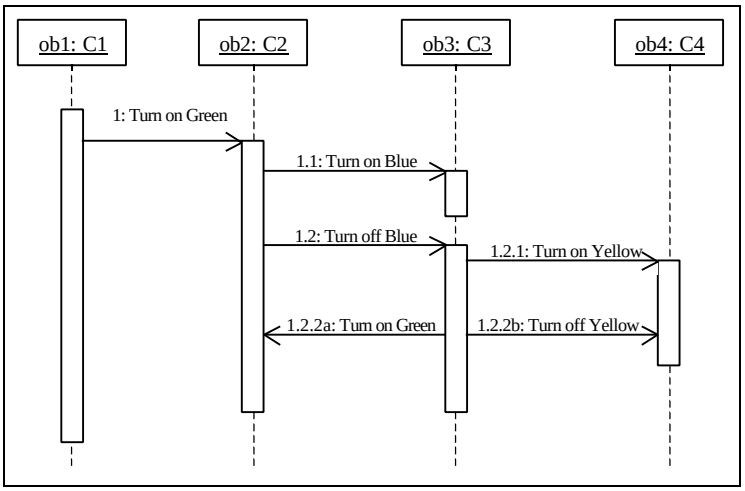

Figure 3-2 Sequence Numbering

The objects and messages involved in an interaction can be represented in two ways in the UML: one is using a sequence diagram that emphasizes the time ordering of the messages, the other is using a collaboration diagram that emphasizes the relationships among the objects that exchange the messages. Both sequence and collaboration diagrams are kinds of interaction diagrams based on the same underlying information, i.e. they are semantically equivalent. One can be transformed to the other in spite of their visual differences. In fact, Rational Rose provides a function to allow one to render a sequence diagram as a collaboration diagram, and vice versa. Figure 3-3 shows a sequence diagram. Its equivalent collaboration diagram is shown in Figure 3-4. This is due to the fact that Rose represents the sequence and collaboration diagrams by the same metamodel objects. This means that by looking at an XMI file produced by Rose that contains an interaction diagram, one cannot tell whether the corresponding graph was rendered as a sequence or as a

collaboration diagram. Another UML tool we have used, ArgoUML, does not support such a close equivalence. In ArgoUML, each kind of diagram has its own metamodel representation in XMI, which means that the interpretation of the UML metamodel by different UML tools is not unique. Chapter 4 will describe in more detail the metamodel representations for these diagrams.

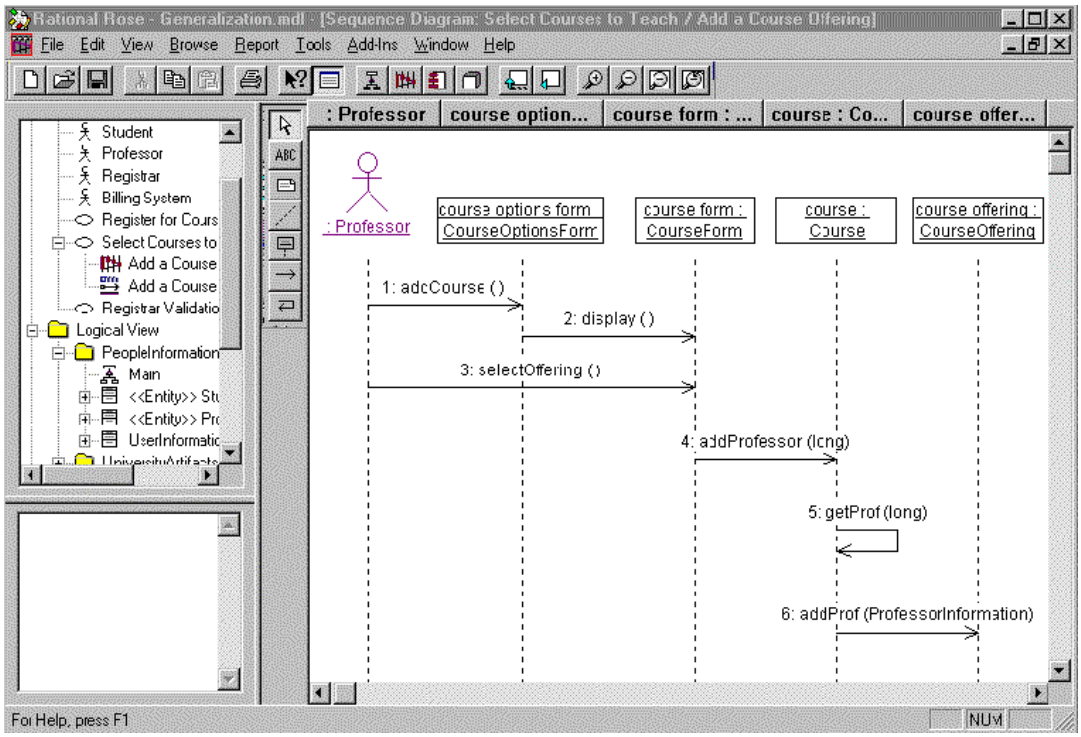

Figure 3-3 A Sequence Diagram in Rose

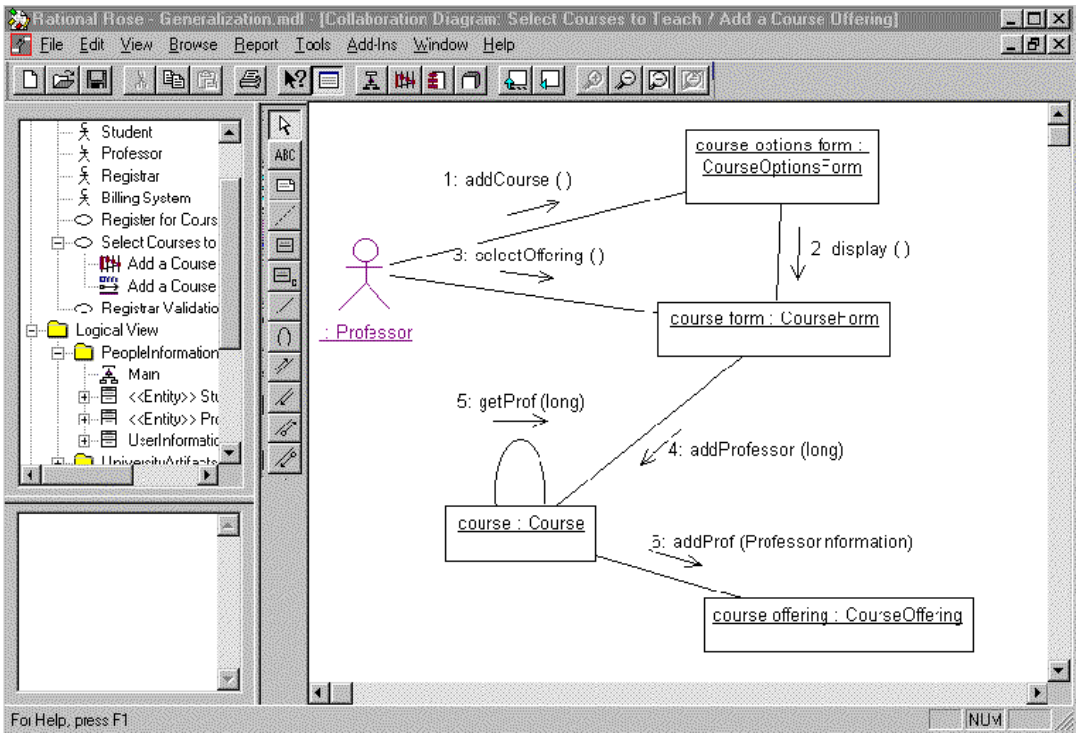

Figure 3-4 An Equivalent Collaboration Diagram in Rose

The UML standard defines the concrete and abstract syntax for Collaborations and Interactions and gives a description of the intended semantics. Idealy, the semantics of the language must be precise if tools are to perform intelligent operations on models expressed in the language, like consistency checks and transformations from one model to another. The abstract syntax in UML is specified with the graphical notation of class diagrams in UML itself, while the well-formedness rules of UML are given in an Object-oriented Constraint Language named OCL. This, as we mentioned in 2.1.1, makes the semantics of UML is still quite informal.

3.1.2.1 Sequence Diagram

Sequence diagrams are often most useful for showing scenarios, which are realizations of use cases. The graphical syntax of a sequence diagram has two dimensions: the vertical dimension represents the time and the horizontal dimension represents the different objects [UML1.3]. Time normally proceeds downwards, and an arrow between two vertical lines denotes a Stimulus sent between two objects (sender and receiver). Hence, the diagram gives a clear visual cue to the flow of control over time. Usually, sequence diagrams omit sequence numbers because the physical location of the arrow shows the relative sequences.

Sequence diagrams in the UML notation guide also provide presentation options for addition features, such as branch and iteration. A branch is shown in Figure 3-5 by multiple arrows leaving a single point, each labeled by a guard condition.

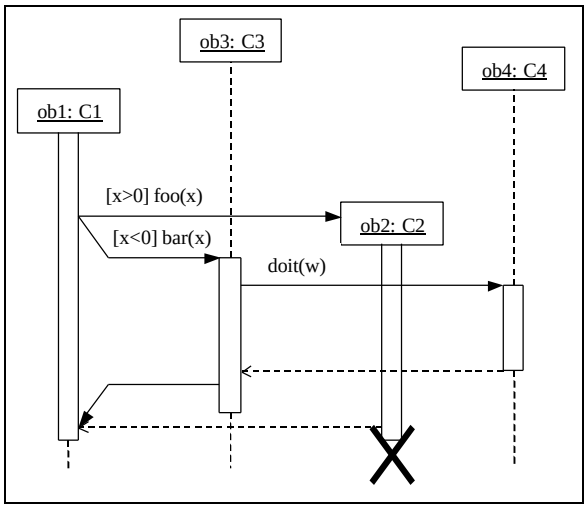

Figure 3-5 Sequence Diagram with a Branch

To present an iteration, a connected set of arrows may be enclosed and marked as an iteration [UML1.3, pp. 3-106].

Much of sequence diagram notation is derived from the Message Sequence Chart (MSC) notation, which is an older standard than UML [ITUT00]. Unfortunately, the sequence diagram notation from the current UML standard is still unsatisfactory. One of the problems is that the notation does not scale up well. For example, if a branch is long, the branch and merge may not be shown in the same sequence diagram. Another problem is that both branch and iteration are not supported yet by the current UML tools (Rose or ArgoUML). MSC standard described in [ITUT00] has a better solution. In fact, it is expected that the new version of UML (UML 2.0), on which OMG works right now, will improve sequence diagrams with respects to iterations, branch/merge and diagram decomposition.

The core elements in a sequence diagram are Object and Stimulus. In the UML metamodel, Object is a subclass of Instance and originates from a Class, which provides a full description of its objects. In our transformation a particular interest is given to the attribute *isActive* of Class, which specifies whether an object of the class maintains its own thread of control and runs concurrently with other active objects [UML1.3, pp. 2-27]. All instances of an active class are active objects. The notation for an active object is shown as a rectangle with a heavy border. Notice that in UML the thread of control represents an

abstract notion of control and not an operating system thread. In general, an active object is a composite that aggregates a number of passive objects executing within its thread. It has the general responsibility to coordinate the internal execution by dispatching messages to its constituent parts. Usually, active objects are implemented as threads or processes, even though UML does not specify how active objects should be realized.

3.1.2.2 Collaboration Diagram

A collaboration diagram represents a UML Collaboration, which contains a set of roles to be played by Objects, as well as their required relationships given in a particular context [UML1.3, pp. 3-111]. A collaboration diagram can be given in two different forms: at *instance* level or at *specification* level. A collaboration diagram given at instance level shows a collection of Objects and Links, whereas a collaboration diagram given at specification level shows Classifier Roles, Association Roles and Messages as well as their structures. The Objects and Links conform to the Classifier Roles and Association Roles of the Collaboration. A Classifier Role (Association Role) defines a usage of an Object (Link), while the *base* class (Association) specifies all properties of the Objects (Links). Compared to sequence diagrams, the collaboration diagrams do not show time as a separate dimension. Therefore, sequence numbers, described in section 3.1.2, are necessary to indicate the sequence of interactions and the concurrent threads.

3.1.2.3 Object and Classifier Role

As we mentioned before, an Object originates from a Class which is a Classifier. In a Collaboration, however, not all the features of the participating Classifiers are always required. Hence, a Collaboration is not actually defined in terms of Classifiers, but of Classifier Roles. The Classifier so represented is referred to as the base Classifier of that particular Classifier Role. Similarly, Association Roles, not Associations, between those Classifier Roles are considered in a Collaboration.

UML defines that an Object *conforms* to a Classifier Role if the Object has the properties specified by the Classifier Role, i.e. the Attribute Links and the Links of the object match all the Attributes and Association Roles specified by the Classifier Role, and all Operations specified by the role may be applied to the Object. The Object may, of course, include more Attribute Links than required by the respective Classifier Role [UML1.3, pp. 2-113].

3.1.2.4 Message and Stimulus

A Message is a specification of a communication between a sender and a receiver. The Message specifies the roles played by the sender object and the receiver object, and it indicates which Operation should be applied to the receiver by the sender. Moreover, the set of Messages in an Interaction is partially ordered. Recall that the interaction specifies the predecessors and activator of each message. Precisely to say, if a message has more than one predecessor, it represents the joining of two threads of control. If a message has

more than one successor, it indicates a fork of control into multiple threads. Thus, the predecessor relationship imposes a partial ordering on the Messages within a procedure, whereas the activator relationship imposes a tree on the activation of operations.

In UML there is a subtle difference between Message and Stimulus. A Stimulus reifies a communication between two Objects and uses a Link between the sender and the receiver for communication. A Message is a specification of Stimulus. The Message is connected to an Action, which, when executed, causes the communication specified by the Message to take place. There are different kinds of Actions in UML, such as Call Action resulting in an invocation of an operation on the receiver, Send Action resulting in the sending of a signal, Create Action resulting in the creation of a new object, and Destroy Action resulting in the destruction of an object. The properties of a Message are described in the next subsection.

UML indicates that a Stimulus *conforms* to a Message if the sender and receiver Objects of the Stimulus are in conformance with the sender and the receiver roles specified by the Message. Furthermore, the Action dispatching the Stimulus is the same as the Action Associated with the Message [UML1.3, pp. 2-115].

3.1.2.5 Message Properties

Messages play a key role in inter-object behavior. Most often, message passing will be realized with a simple direct call to a method in the target object, but that is not the only

realization of a message. Other realizations include remote procedure calls (RPC), sending messages via an OS message queue, IPC, and sending messages across a network.

[UML1.3] identifies two kinds of messages: sending a signal and invoking an operation. The major difference between them is that signal sending is always asynchronous, while operation call may be either synchronous or asynchronous. The essential properties of a message are:

- Sender
- Receiver
- Action
- Parameter list and return value
- Synchronization pattern an Arrival pattern

In most cases, a single receiver object is identified, but messages may be multicast to a list of objects. UML is not defining a notation for broadcast, where all objects receive a message without explicitly being part of a list. The arrival pattern, i.e. periodic or aperiodic, is useful in the analysis of real-time systems.

The UML standard defines several different kinds of actions, as shown in the action metamodel depicted in Figure 3-6. Actions contain Arguments and may be contained in an Action Sequence. Two kinds of actions are particularly relevant to our discussion: Call Action and Send Action. Call Action is associated with an Operation. The receipt of a Call

Action can raise a Call Event on the receiving object. Send Action is associated with a Signal – a specification of an asynchronous Stimulus sent from one Object to another Object. When a Signal is received by an Object, it can asynchronously raise an event called a Signal Event.

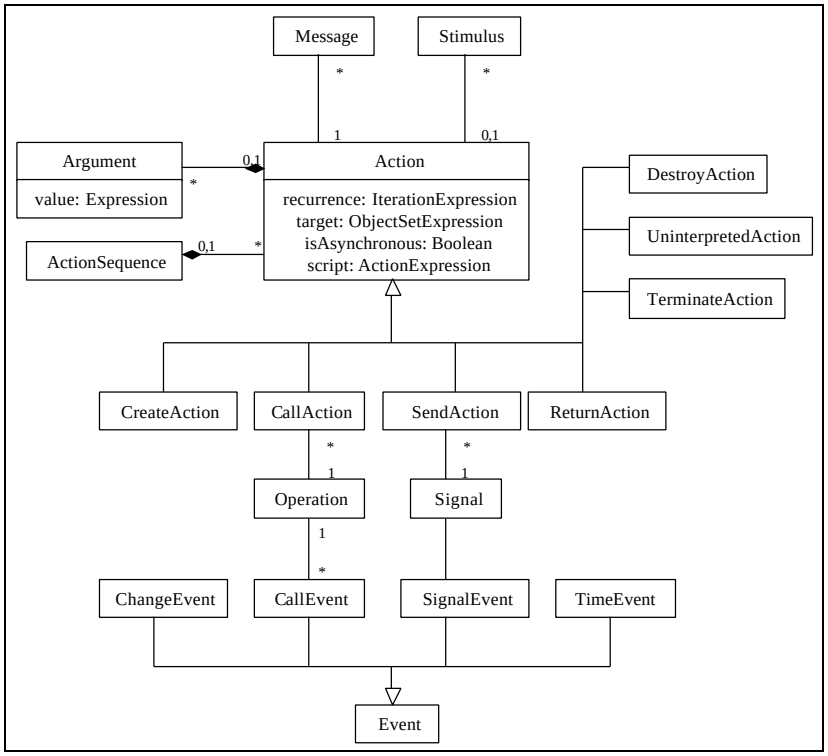

Figure 3-6 Action Metamodel

3.1.3 Activity Diagram

An activity diagram shows the flow of control from activity to activity. It is a special form of a state diagram in which most of the states are actions and in which most of the transitions are triggered by the completion of the actions. Normally, an activity diagram assumes that computations proceed without external event-based interruptions. Activity diagrams are particularly useful in connection with workflow and in describing behavior

that has a lot of parallel processing [Fowler97, pp. 129]. Most of the states in such a diagram are action states that represent atomic actions and do not permit transitions while they are active. An activity diagram is similar to a traditional flow chart that is normally limited to sequential control except it allows for concurrent control (forking/joining) in addition to sequential control [Rumbaugh+99]. Figure 3-7 (taken from [UML1.3, pp. 3-158]) presents an activity diagram and includes most common model elements. Next, the semantics of some key elements is described.

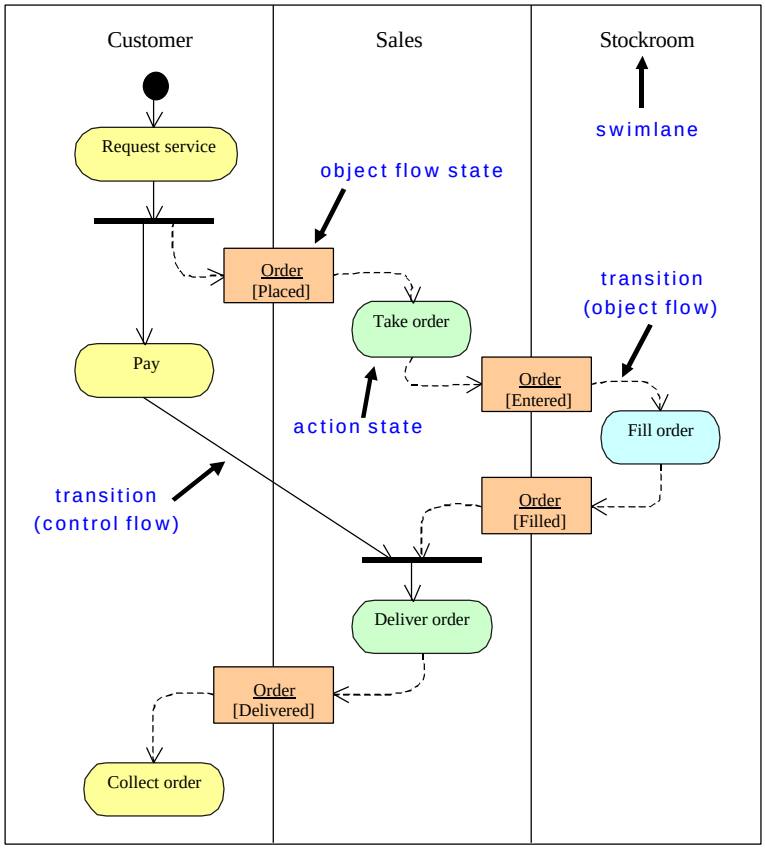

Figure 3-7 Activity Diagram with Swimlane and Object Flow State

3.1.3.1 Swimlane

Activity diagrams tell you what happens, but they do not tell you who does what. In the domain modeling, such as workflow modeling of business processes, this means that the diagram does not convey which business organization is responsible for each activity. Swimlanes are an attempt to solve this problem by labeling each activity with the responsible class or object. A swimlane is graphically separated from its neighbor by a vertical solid line, as shown in Figure 3-7. In the metamodel, a swimlane maps into a Partition of States in the ActivityGraph.

A swimlane specifies a locus of activities and represents a high-level responsibility for a group of activities. Each swimlane may eventually be implemented by one or more classes. There is a loose connection between swimlanes and concurrent flows of control. Independent and concurrent flows of control can, but do not necessarily, map to different swimlanes. For example, an activity diagram may represent the workflow in an enterprise, where different swimlanes represent different departments. Even though a department may have internal concurrent flows, this may not be shown in the activity diagram [Amer01].

Swimlanes are good in that they combine the activity diagram's depiction of logic with the interaction diagram's depiction of responsibility [Fowler97, pp. 138]. On the other hand, packaging objects involved in either interaction diagrams or activity diagrams appropriately into nodes and threads is vital for system performance [Douglass00, pp. 216]. One of the

key issues related to swimlanes is how to model the processing resources. This can be done in two ways. The most direct is to associate the appropriate stereotype with a partition (swimlane) that is linked to the appropriate object. However, this is only useful in cases where each object or classifier role is executing on its own host, e.g. each of them is active and has its own thread. Much more common is the situation where different partitions represent objects that are executing on different hosts and that some objects share hosts. In that case, neither the activity diagrams nor the interaction diagrams contain sufficient information to determine the allocation of objects to hosts. Under those circumstances, it is necessary to determine which processor resource is running which object with the information from deployment diagrams and/or component diagrams [Profile01, pp. 8-148].

In our case, however, we choose to build the activity diagrams at a granularity level where each swimlane corresponds to a single execution flow. In other words, a swimlane will contain the activities carried out by one active object and any number of associated passive objects. We choose to name the swimlane with name of the active object. An exception is made for a passive object shared by active objects, as in the case of the producer/consumer problem. More exactly, we consider that a passive object shared by several active objects has its own “pseudo” thread, and therefore its own swimlane in the activity diagram.

3.1.3.2 Action State

An action state represents the execution of an atomic action, typically the invocation of an operation. It is a simple state with an entry action whose only exit transition is implicitly triggered by the completion of the action in the state. An action state in graphical syntax is shown as shape with straight top and bottom and with convex arcs on the two sides.

In sequence diagrams, the object responsible for performing an action is shown by drawing a lifeline and placing actions on lifelines. Activity diagrams do not show the lifeline of the object, but contain swimlanes to indicate who is responsible for different actions. The actions within a swimlane can all be handled by the same object or by multiple objects [UML1.3, pp. 3-157].

3.1.3.3 Fork and Join

The concurrent control expressed in an activity diagram, as illustrated in Figure 3-7, is achieved by using a synchronization bar to specify the fork and join of the parallel flows. A synchronization bar is rendered as a thick horizontal or vertical line.

A fork represents the splitting of a single flow of control into two or more concurrent flows of control. Below the fork, the activities associated with each of these paths continue in parallel and are conceptually concurrent. A join represents the synchronization of two or more concurrent flows of control. Above the join, the activities associated with each of

these paths continue in parallel. At the join, the concurrent flows synchronize, meaning that each waits until all the incoming flows have reached the join, at which point one flow of control continues on below the join [Booch+99, pp. 264].

3.1.3.4 Branch and Merge

It is possible to express conditional branching (i.e. a selection between alternate branches) by having different possible transitions that depend on Guard conditions leaving from the decision point. UML provides a shorthand for showing decisions and for merging their separate paths back together. The notation for a decision is the traditional diamond shape, with one incoming arrow and with two or more outgoing arrows, each labeled by a distinct guard condition with no event trigger, as shown in Figure 3-8 (taken from [UML1.3], pp. 3-155). A merge symbol has the same diamond shape except that it has one outgoing arrow and two or more incoming arrows. Branching and merging are usually paired in a nested fashion. Both branch and merge symbols map into a Pseudostate of kind *junction*.

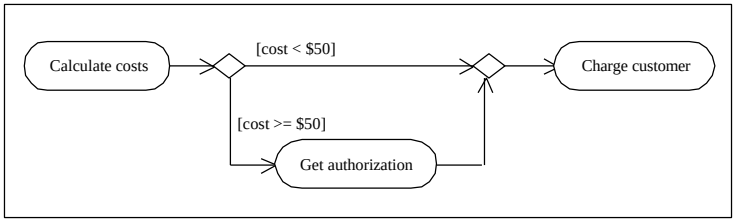

Figure 3-8 Branch and Merge

3.1.3.5 Object Flow State

An object flow between actions in an activity diagram (drawn as a rectangular shape) represents the data flow between activities. More exactly, the generation of an object by an action in an action state may be modeled by an object flow state that is triggered by the completion of the action state. The use of the object in a subsequent action state may be modeled by connecting the output transition of the object flow state as an input transition to the action state.

3.1.3.6 Transition

Transitions show the path from one action state to the next action state. Each transition is triggered upon the completion of its previous state and does not have a special trigger of its own. Transitions leaving an action state should not include an event signature (as do the transitions in a state machine). A transition may include a guard (Boolean expression) that is evaluated before the transition is triggered. Most of the transitions used in an activity diagram are normally very simple. More complex transitions like compound transitions are used in state machine diagram.

3.2 Transformation Rules at UML Diagram Level

In this section, transformation rules from a sequence to an activity diagram are illustrated at notation level. This provides a “bird’s-eye view” of the transformation approach. In the following chapter these transformations will be described in detail by making use of the UML metamodel which is more abstract and less readable.

The UML standard [UML1.3] defines a canonical UML notation that might be called the publication format for the models. Notation does not add meaning to a model and has no semantics, but it is more intuitive and it helps the user to understand the meaning of the model. Also, notation is more than pictures; it includes information in text-based forms and invisible hyperlinks among different presentation elements [Rumbaugh+99]. For simplicity, the transformation rules are described by using sequence diagram notation. These rules apply to collaboration diagrams as well, due to the semantic equivalence between sequence and collaboration diagrams.

The concept behind the transformation is to follow the flow of messages in a sequence diagram, considering the execution threads of all active objects involved in the collaboration. Sequential executions, conditional branching and action kinds are identified in the message flow, and are translated into appropriate states in the corresponding activity diagram. The activity diagram contains separate swimlanes for each active object to show the actions performed by the active object and its associated passive objects. Special treatments are given to messages exchanged between different threads of control that introduce fork/join connectors in the activity diagram. For example, creating an active object is equivalent to forking a new thread of control, sending an asynchronous signal also forks a thread, whereas the receipt of a message from another thread is equivalent to a synchronization point (*i.e.,* a join). Messages exchanged between execution threads carry

objects that can be represented as object flow states in the activity diagram according to the notation from [UML1.3].

The transformation from sequence diagrams to activity diagrams takes as input the following information:

- XMI file containing the sequence diagram
- additional user input regarding the grouping of the objects from the sequence diagram in execution threads (which become activity diagram swimlanes).

3.2.1 Basic cases

In this subsection 7 basic cases are given to illustrate how to convert a sequence diagram to an activity diagram. A more complex sequence diagram can be decomposed into simple fragments, each of them treated by applying these basic cases or a combination thereof.

In an activity diagram, the following notation is employed to express the names of the action states that are converted from sequence diagram messages.

`invoke(objectName.message)`	for an operation call on the sender side
`objectName.message()`	for an operation call on the receiver side
`send(message)`	for a signal send on the sender side
`receive(message)`	for a signal receipt on the receiver side
`new`	for a creation of a new object on the sender side

`init` for a creation of a new object on the receiver side

Generally, these names are given according to the kind of the action attached to the message. For example, if the action type is CallAction, then an operation call will be invoked by the sending object and executed by the receiving object.

All basic cases can be categorized roughly into two categories: cases a, b and c in which the messages are exchanged in the same thread of control, and cases d, e, f and g in which the messages are exchanged between different threads of control.

a) A set of consecutive sequential messages without any branching or iteration that pass between objects in the same execution thread, as shown in Figure 3-9, where r is an active object and m and n are passive objects executed in the same thread of control.

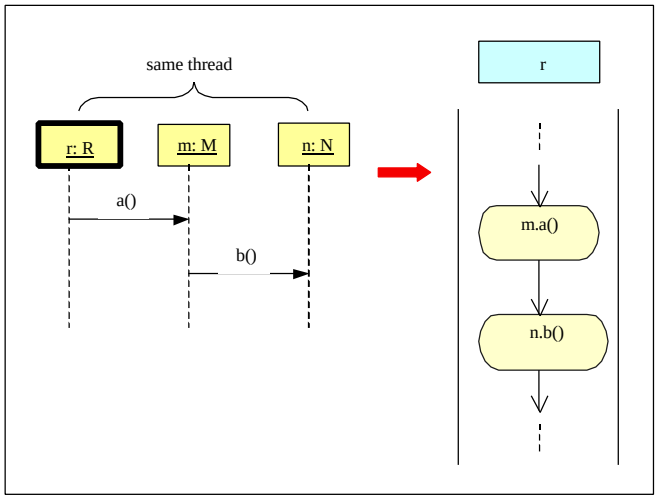

Figure 3-9 Sequential Execution

b) Messages with guard conditions that are alternatives of the same condition in a sequence diagram are mapped to a branch/merge structure in the corresponding activity diagram, as shown in Figure 3-10.

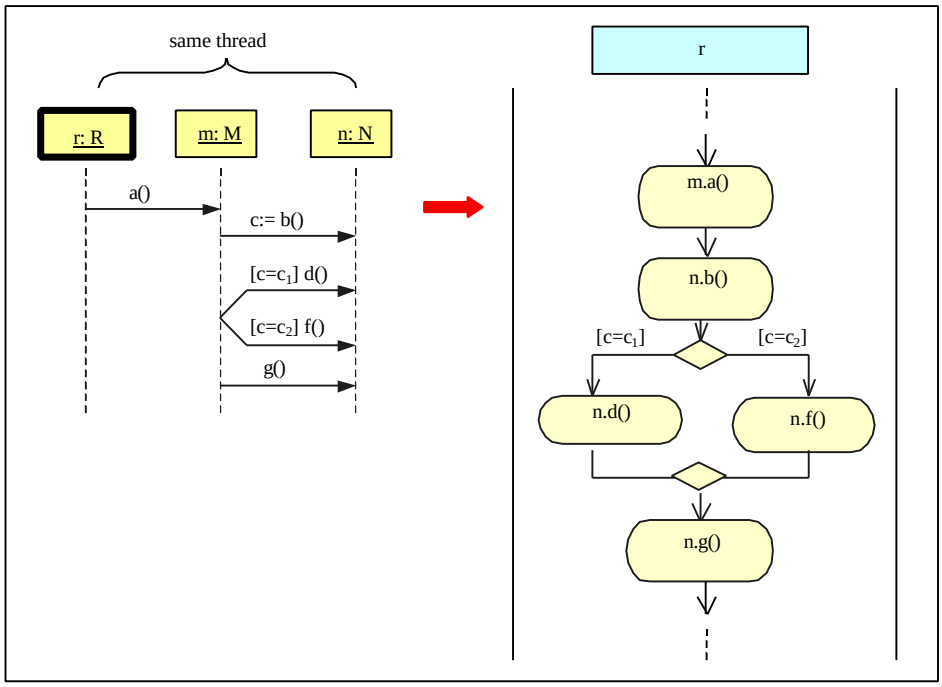

Figure 3-10 Branch and Merge

c) An iteration (loop) can be achieved by using one action state that sets the value of an iterator, another action state that increments the iterator, and a branch that evaluates if the iteration is finished [Booch+99, pp. 263], as shown in Figure 3-11. Notice that UML 1.3 does not prescribe the format of iteration or condition, it may be expressed in pseudocode or an actual programming language.

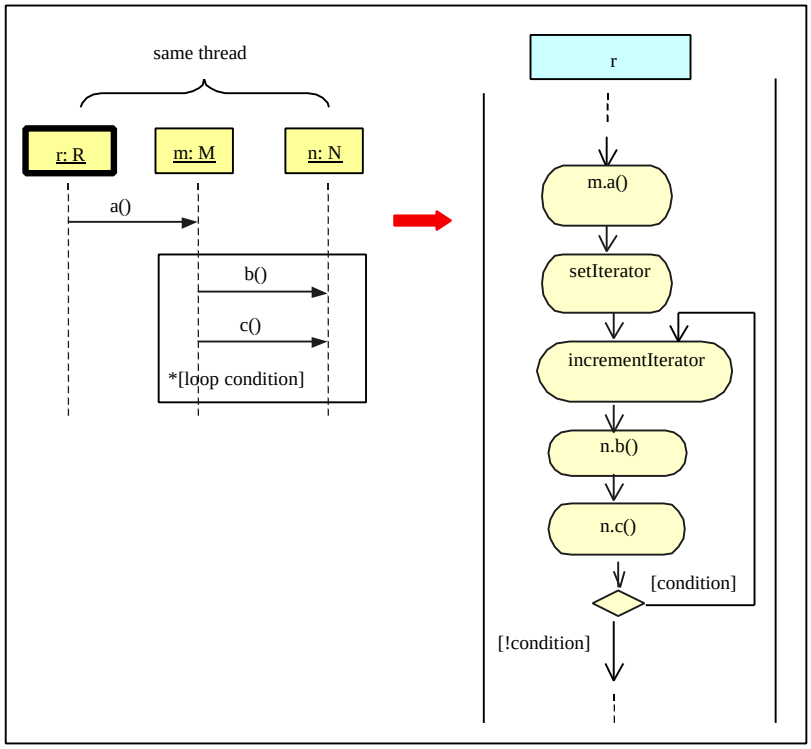

Figure 3-11 Iteration

d) A synchronous message between objects running in different threads of control is treated as a join operation on the receiving side in the corresponding activity diagram, and its reply marks the corresponding fork, as shown in Figure 3-12. The object flow is also shown. The sender's thread will be suspended from the moment it sends the message until the reply is received back. An "idle" action state plays the same role as an initial state to indicate concurrency among different execution threads.

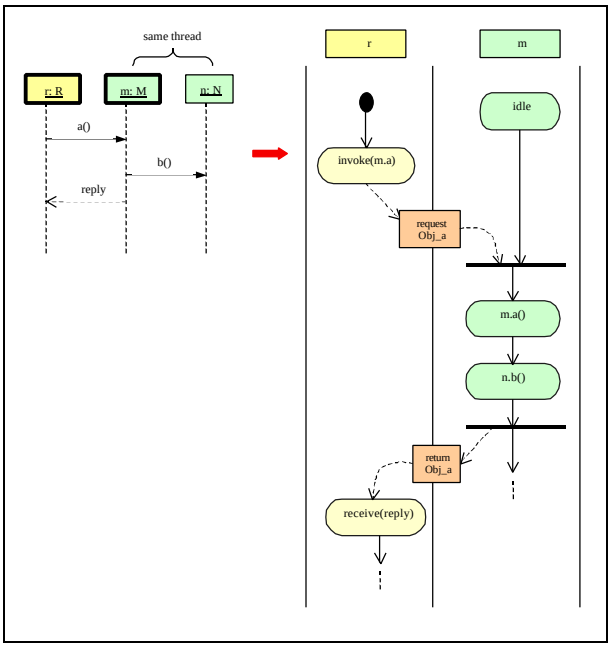

Figure 3-12 Synchronous Message Send and Reply

e) An asynchronous creation of an active object marks a fork operation in the corresponding activity diagram. Figure 3-13 shows also how to map self-call.

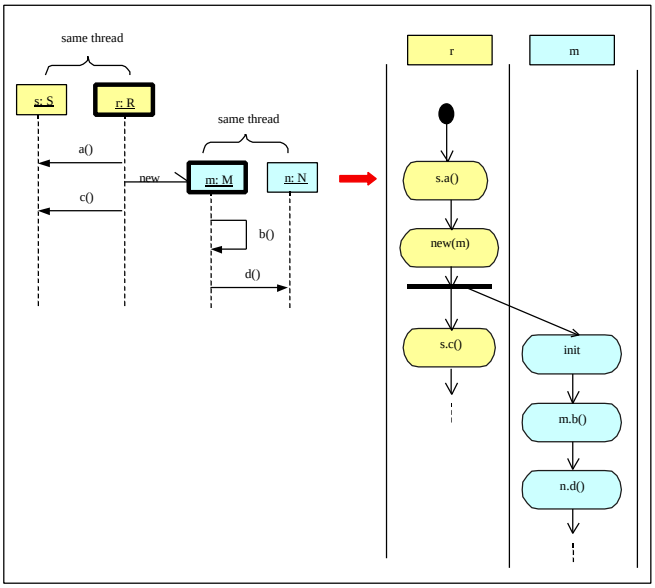

Figure 3-13 Asynchronous Creation of an Active Object

f) An asynchronous message sent to another thread of control indicates a join operation on the receiver side and a fork operation on the sender side in the corresponding activity diagram, as shown in Figure 3-14.

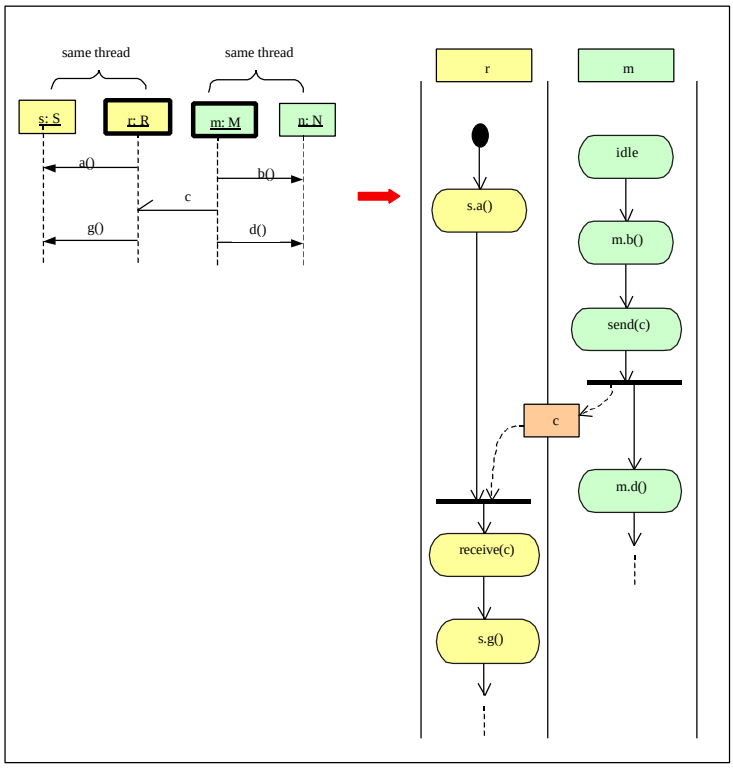

Figure 3-14 Asynchronous Message between Two Execution Thread

g) An asynchronous destroy action marks a fork operation. A terminate action indicates self-destruction of an object and maps to a “terminate” action state. If an active object terminates itself, its thread will stop to execute and become dead, as shown in Figure 3-15.

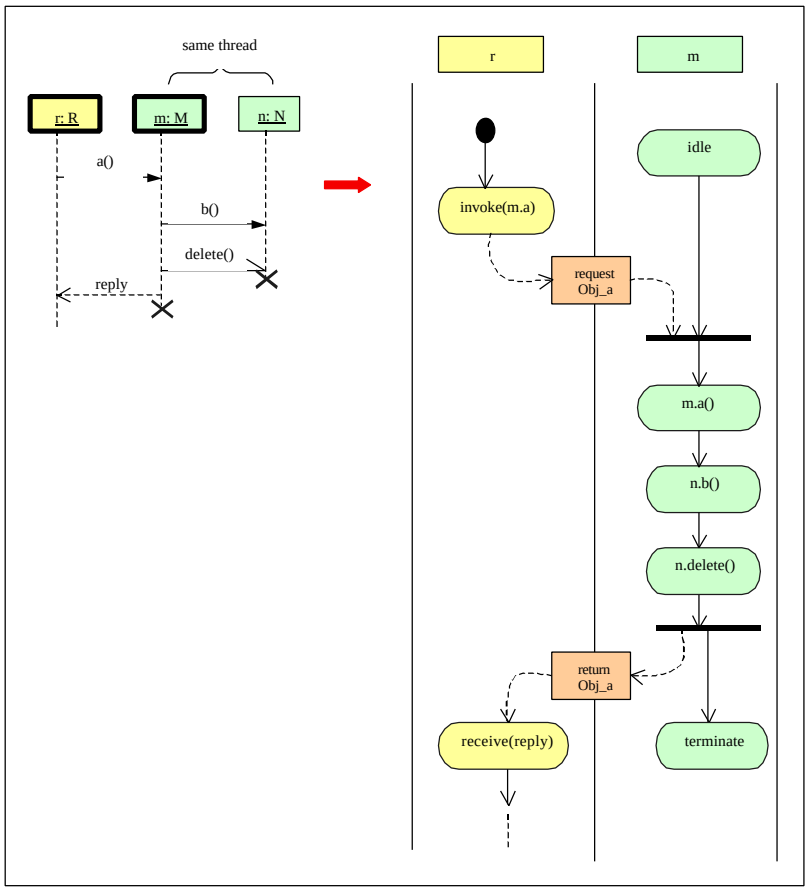

Figure 3-15 Destruction and Termination

3.2.2 Example

The transformation on a complex example, as shown in Figure 3-16 and Figure 3-17, is illustrated. The example is taken from [Petriu+98], which combines the different cases together.

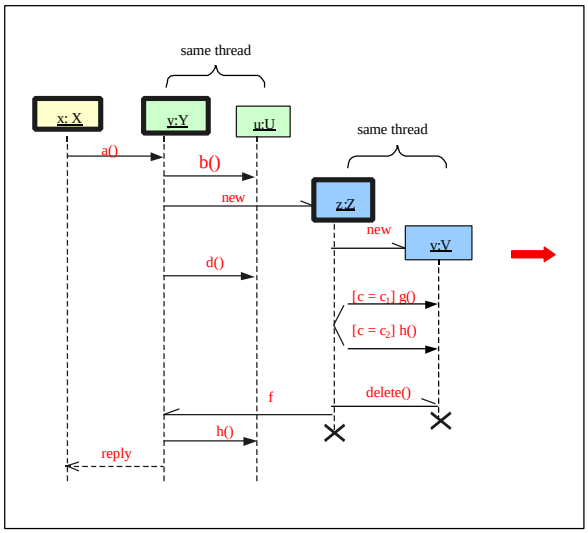

Figure 3-16 Example: Input Sequence Diagram

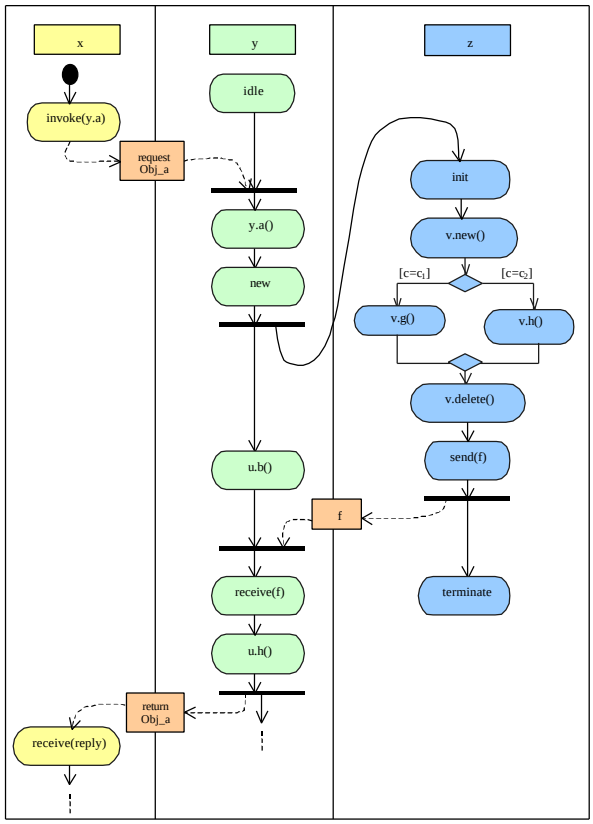

Figure 3-17 Example: Activity Diagram after Transformation

3.2.3 Discussion

There exist alternative representations for modeling the sending/receiving of an asynchronous message, as shown in Figure 3-18 [Petriu+01b].

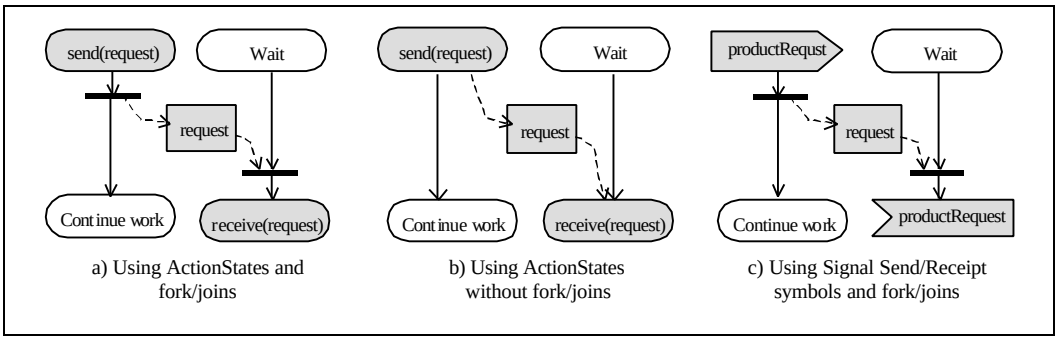

Figure 3-18 Alternative Representations for Modeling an Asynchronous Message

Figure 3-4 presents three notations to model an asynchronous message. In notation a), the action state that represents the sending of the message is followed by an explicit fork: one thread for the continuing the execution of the sender, and the other thread for the message just sent. This representation conforms to the UML notation guide and is very close to Petri net model. The disadvantage of the approach is that the object flow state is not directly connected to the sending and receiving action states. Another disadvantage is that it may introduce too many forks and joins that may cloud the understanding of the diagram.

Notation b) simplifies the previous one by making the fork and join implicit. Unfortunately, the approach assumes that the dotted transitions connected to the object flow behave differently than the other transitions, which is not supported by the UML standard. More exactly, it is assumed that after the sending action, the dotted transition is fired simultaneously with the normal transition leading to the next action state of the sender. Also, in order to enter the receiving action state, both its incoming transitions (dotted and normal) must be ready to fire.

Notation c) modifies the first one in yet another way: it uses the signal sending/receipt symbols from [UML 1.3, pp. 3-160]. The advantage is that the sending and receiving actions stand out, making the diagram easier to read. However, the mapping of the “signal sending” symbol given in [UML1.4 pp. 3-161] should be changed to an ActionState instead of a SendAction. This solution also inherits the disadvantages from the solution a: the object flow not directly connected with the action states that produce/take it as output/input, and too many forks and joins in the model.

It should be mention that the thesis implements solution a. Also, there are two alternatives in representing a synchronous message between threads of control, as shown in Figure 3-19. In the first solution (b), the flow of control of the senders is interrupted when the sender is blocked waiting for the reply, whereas in the second solution (c) the waiting state

of the sender is shown explicitly. We have chosen the first solution in the thesis, but the second can be also implemented with very little change.

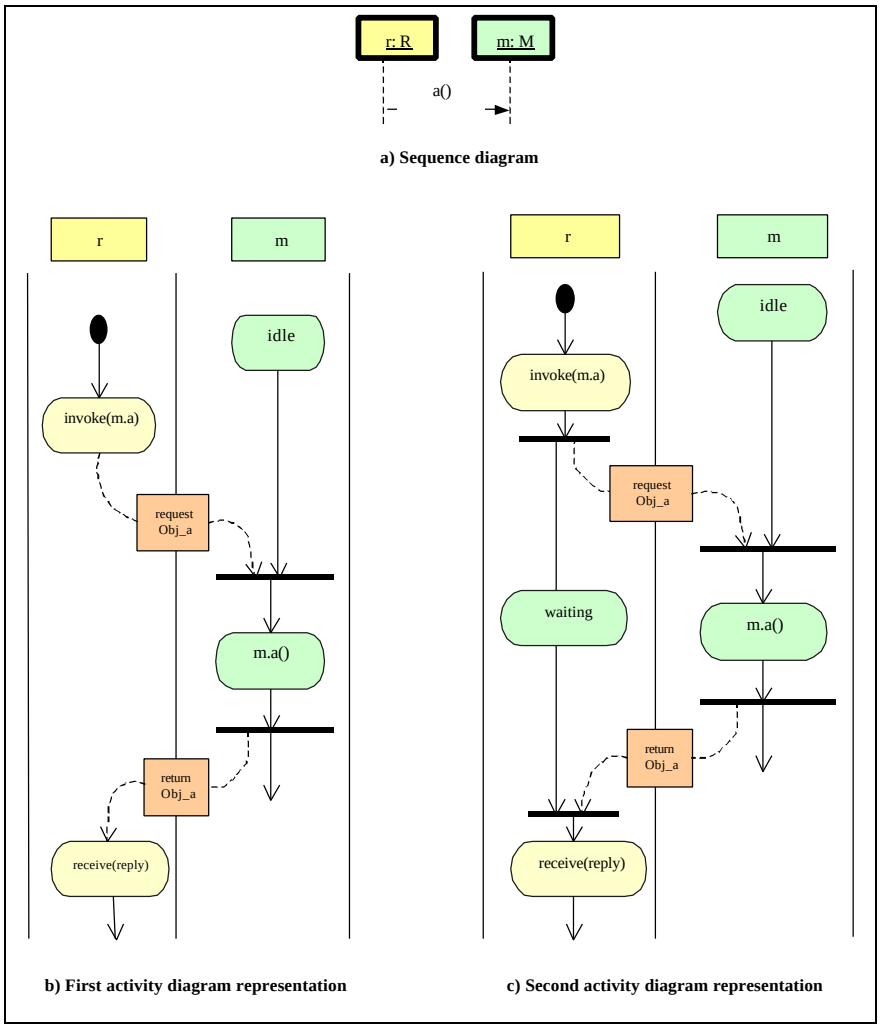

Figure 3-19 Alternate Ways of Representing a Synchronous Message

Chapter 4 Detailed Design of the ID to AD Transformation

The analysis and design of the ID to AD transformation concentrate on the manipulation of the metaobjects involved in the transformation rules.

Chapter 4 is structured as follows: the first section describes the UML metamodel and the Novosoft UML API as well as their relationship. The second section describes in more depth the API. The third section highlights the main points of the metamodel representations for interaction diagrams and activity diagrams. The fourth section gives the object diagrams that show the metaobjects, corresponding to the basic cases presented in the previous chapter. Finally, the last section describes the transformation algorithm.

4.1 Metamodel and API

As mentioned before, the UML standard consists of three main specifications: a notation guide that specifies the visual appearance of UML diagrams, a semantics specification that details the UML metamodel, and the OCL (Object Constraint Language) specification that adds a first-order predicate logic language for expressing constraints on UML models. The UML metamodel is itself a UML model that specifies how a UML design can be represented [Rumbaugh+99].

Novosoft UML (NSUML) API is an open-source Java library implementing the UML metamodel [NSUML99]. It consists of interfaces, classes, attributes and methods, which

supports the elements of the UML metamodels. There is a simple correspondence between metaobjects names and NSUML interfaces/classes names.

4.1.1 UML Metamodel

As described in section 2.1.1, the UML metamodel is defined as one of the layers of a four-layer metamodeling architecture. It is regarded as being an instance-of the MOF residing at the M2 level. The official version of the UML metamodel at the time of the thesis research was UML 1.3 (OMG released the latest version of UML 1.4 in September 2001). The metamodel concepts and semantic constructions are described in chapter 2 “UML Semantics” of [UML1.3]. The metamodel referred in the rest of thesis is version of 1.3 unless otherwise specified.

The metamodel is divided into three main packages, as shown in Figure 4-1.

- The foundation package defines the static structure of the UML.
- The behavioral elements package defines the dynamic structure of the UML.
- The model management package defines the organizational structure of UML models.

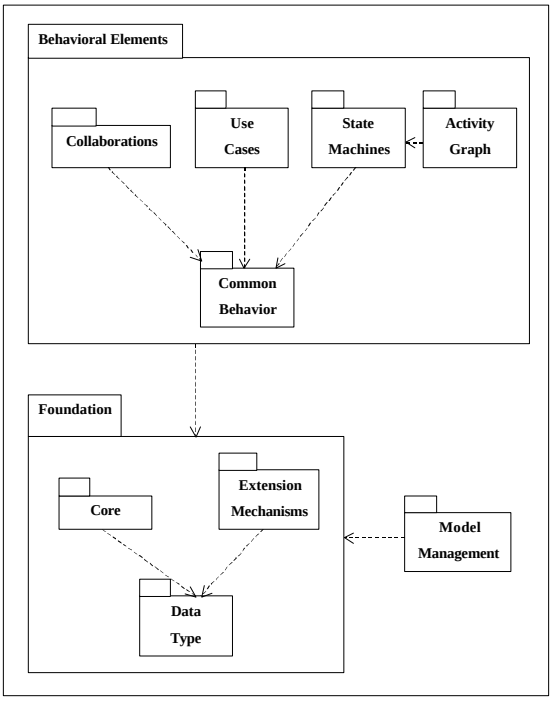

Figure 4-1 Package Structure of the UML Metamodel

4.1.2 UML Physical Metamodel

In addition to the UML metamodel, OMG also proposed the UML physical metamodel, which is more clear for realization and practical for implementation. The specifications of the physical metamodel are described in UML XMI DTD, which is a physical mechanism for interchanging UML models conforming to the UML metamodel. Chapter 6 in [UML1.3] contains a normative DTD that represents the UML 1.3 metamodel generated from the XMI 1.0 standard.

Some of the distinctions between the UML physical metamodel and the UML metamodel are as follows [UML1.3]:

Names

- Changed spaces in package names to '_'.
- Added names for association ends that did not have them. Convention: the name of the adjoining class with the first letter in lower case. If this resulted in a name duplication, then a numbered suffix was added.

Additions

- Added enumeration literals as attributes of the enumeration classes for enumeration data types.
- Added 'sorted' enumeration literal to OrderingKind.
- Added inheritance link from Message to ModelElement.

Association Classes

- Made ElementOwnership AssociationClass attributes by moving the visibility and isSpecification attributes to the ModelElement class.
- Removed the attribute "visiblity" from classes AssociationEnd and Feature.
- Made the AssociationClass ElementResidence a class by removing the association between Component and ModelElement and adding associations between ElementResidence and Component and between ElementResidence and ModelElement.

- Made the AssociationClass ElementImport a class by removing the association between ModelElement and Package and adding associations between ModelElement and ElementImport and between ElementImport and Package.
- Made the AssociationClass TemplateParameter a class by removing the association between ModelElement and ModelElement for template parameters and added associations between ModelElement and TemplateParameter and between TemplateParameter and ModelElement.

4.1.3 NOVOSOFT UML (NSUML) Metamodel and its API

Novosoft UML is an open-source Java software that implements the UML metamodel. The version of Novosoft UML API used in our transformation application is 0_4_19 that can be download from [NSUML99]. The API mentioned in the rest of thesis will refer to Novosoft UML API unless otherwise specified.

The major features provided by Novosoft UML API are described briefly as follows:

- The API allows various kinds of work with models, such as generating and serialization of UML models, organizing of access to model elements, modifying, adding and deleting of features (through attributes and opposite roles in associations).
- The API implements UML elements in: packages, datatypes, classes, their methods and associations. Besides, the API contains many other useful methods not specified by OMG.

- The API contains Reflective API. The main sense of reflective methods is the access to features by their names, instead of invocation of explicit Setter, Getter, Adder or Remover methods.
- The API supports the XMI standard. It can read and write UML models according to the XMI format.

Limited modifications were made to the metamodel to make it fit the Java language. Multiple inheritance used in the standard metamodel, for example, was replace with Java interfaces and single inheritance.

ArgoUML is an open-source UML tool that is using the Novosoft UML library. ArgoUML's implementation of the API uses JavaBeans-style method naming and changing notifications, which is supported by reflection in the API. For example, the attribute *target* of meta-class *Action* in the metamodel is accessed with methods *getTarget()* and *setTarget()* in the ArgoUML implementations. Also, whenever the concurrency of an action is changed, a standard JavaBeans property change event is fired with information about the name of the property that changed, its old value and its new value.

However, the fact that the API fulfills the metamodel does not mean that ArgoUML implements all the functions (features) supported by the API. For example, class *Partition* in the API provides methods to access *contents* and *activityGraph*. But ArgoUML does not support swimlanes (partitions) in an activity diagram. Therefore, no information of partition

can be obtained from an activity diagram generated in XMI format by ArgoUML. Similar problems exist with the XMI generated by Rational Rose as well. For example, Rose does not support object flow states in an activity diagram. Consequently, we were forced to modify by hand some XMI files produced by these two tools in case in which the tools do not support yet standard UML features. However, these modifications were in general minor, and most the XMI files used to test our implementation were produced directly by the UML tools.

4.2 Object Model in Novosoft UML API

This section will describe how the NSUML API constructs objects as well as their attributes and associations. The object model in the API contains four types of objects: *primitives*, *enumerations*, *datatypes* and *elements.* They correspond to UML types and metaobjects. The classification is based on UML stereotypes of the objects and the ways of how the API maps these objects to Java constructs.

Primitives have the stereotype *<<primitive>>*, *Enumerations* the stereotype *<<enumeration>>* and d*atatypes* and *elements* have no stereotypes. There are two major distinctions between *datatypes* and *elements.* First, any *datatype* is mapped only to one Java class in the API, whereas any *element* is mapped to one Java class and one Java interface. Second, *datatype* classes are created manually, whereas *element* classes and interfaces are created with the help of a generator program.

The API also contains auxiliary classes, which provide additional functions such as events and undo/redo. All auxiliary interfaces and classes are created manually.

4.2.1 Primitives

Primitives are the UML objects, which have the stereotype <<*primitive*>>. There are no special classes in the NSUML corresponding to them. The NSUML maps such objects to ordinary Java types, according to Table 4-1:

UML Primitives	***Java Types***
Boolean	boolean
Name	String
Integer	int
UnlimitedInteger	int
LocationReference	String
Geometry	String

Table 4-1 Primitives

4.2.2 Enumerations

The NSUML realizes *enumerations* as final Java classes with private constructors only. Names of enumerations begin with letter *M*, prefixed to original UML names. A NSUML enumeration classes are indicated in Table 4-2:

UML enumerations	***NSUML Java classes***
AggregationKind	MAggregationKind
CallConcurrencyKind	MCallConcurrencyKind
ChangeableKind	MChangeableKind
MessageDirectionKind	MMessageDirectionKind
OperationDirectionKind	MOperationDirectionKind
OrderingKind	MOrderingKind
ParameterDirectionKind	MParameterDirectionKind

PseudostateKind	MPseudostateKind
ScopeKind	MScopeKind
VisibilityKind	MVisibilityKind

Table 4-2 Enumerations

During the initialization of *enumeration* classes there are constructed several predefined final static public instances in accordance with the UML standards. For example, there are 3 available instances of the class *MVisibilityKind*: *MVisibilityKind.PRIVATE*, *MVisibilityKind.PROTECTED*, and *MVisibilityKind.PUBLIC.* In addition, the class contains 3 integer attributes that correspond to the above mentioned instances (see Table 4-3).

MVisibilityKind	
Predefined instances	*Corresponding class attributes*
MVisibilityKind.PRIVATE	MVisibilityKind._PRIVATE
MVisibilityKind.PROTECTED	MVisibilityKind._PROTECTED
MVisibilityKind.PUBLIC	MVisibilityKind._PUBLIC

Table 4-3 MVisibilityKind

4.2.3 Datatypes

Each *datatype* is mapped to exactly to one NSUML Java class. The creation of names for datatypes is the same as for enumerations, e.g. prefix *M* to original UML names. Below (Table 4-4) all *datatype* classes are presented.

NSUML Datatype classes	
MExpression	MActionExpression
MArgListsExpression	MBooleanExpression
MIterationExpression	MMappingExpression
MProcedrueExpression	MTimeExpression
MTypeExpression	
MMultiplicity	MMultiplicityRange

Table 4-4 Datatypes

Notice that Class *MExpression* is the superclass for all the group of *Expression* classes.

Class MMultiplicity is intended for description of role multiplicities. There are four predefined instances of this class, as shown in Table 4-5. They correspond to the most widespread types of UML multiplicities:

MMultiplicity	
Predefined Instances	*Corresponding UML Multiplicities*
MMultiplicity.M0_1	0..1
MMultiplicity.M1_1	1
MMultiplicity.M0_N	*
MMultiplicity.M1_N	1..N

Table 4-5 MMultiplicity

4.2.4 Elements

Elements form the biggest object class. UML elements are structured in packages, such as Foundation, Core, Behavior, and so on. The typical elements are *Package*, *Classifier*, *Attribute*, *Method*, *Operation*, etc. Each element is mapped exactly to one interface and one class in the NSUML. There exists a simple name correspondence between UML elements and NSUML Java interfaces and classes, as shown in Table 4-6:

Names' Correspondence		
UML Element	*NSUML Interface*	*NSUML Class*
Package	MPackage	MPackageImpl
Classifier	MClassifier	MClassifierImpl
…	…	…

Table 4-6 Names' Correspondence

The NSUML API contains one important class that is the superclass for all the element Java classes. This is the class *MBaseImpl*, which implements interface *MBase*. Many interesting additional functionalities of NSUML elements are realized due to the methods defined in the base class. It is supposed there is no need to create instances of this class, but to create instances of its subclasses. For example, NSUML Java class *MClassImpl* implementing UML metaclass *Class* is a subclass of the base class. There are two ways to create a new instance of the metaclass *Class*, as showed below:

```
MBase cls0 = new MClassImpl();
MClass cls1 = new MClassImpl();
```

This means users have to operate with interface references only.

```
MClassImpl cls = new MClassImpl(); //Error! Do not use similar references
```

Also, interface MBase contains overridden in all subclasses method *getUMLClassName()*, returning the real UML name of the metaclass, which is implemented in the NSUML. See the following segment:

```
public class MStereotypeImpl extends MGeneralizableElementImpl implements
Mstereotype {
   // ---------- code for class Stereotype ------------
   …
   public String getUMLClassName(){
      return “Stereotype”;
   }
   …
}
```

So, if there is a reference to interface *MBase*, it’s easy to recognize which UML metaobject corresponds to this reference.

4.2.5 Attributes and Associations

Element objects may contain attributes, whose types are *primitive*, *enumeration* or *datatype*. Two element metaclasses can be connected with the help of an association.

Each attribute and role (in association) is mapped to a set of public user method of element interfaces and classes. These methods support access to an attribute or role as well as modifications of their values. There is a simple correspondence between the names of attributes or roles in UML and the names of NSUML methods. This section contains the classification of user methods for accessing attributes and associations, the description of rules for naming of the methods and the role of these methods.

4.2.5.1 Access to Attributes

Attributes are divided into two types: *boolean* and *non-boolean*. A set of methods for access to *boolean* and *non-boolean* attributes is the same, but there exists a minor differences in naming the methods.

Attributes are stored as private objects in the NSUML element class. Access to object attributes is organized with the help of so-called *Getter* and *Setter* methods that are declared in the corresponding interfaces and implemented by each class. The roles of the methods are follows: *Getter* is a method that returns the value of an attribute; *Setter* sets the value of

the attribute. Figure 4-2 shows UML metaclass *Abstraction* that has an attribute *mapping* of type *MappingExpression*.

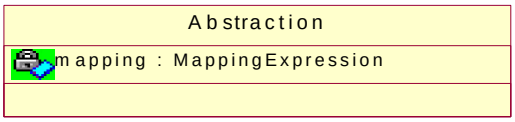

Figure 4-2 Metaclass Abstraction

In the NSUML API interface *MAbstraction* corresponding to the metaclass has the following form:

```
public interface MAbstraction extends MDependency {
   // generating attributes
  // attribute: mapping
  MMappingExpression getMapping();
  Void setMapping(MMappingExpression _arg);
  …
  //generating associations
  …
}
```

Methods *getMapping()* and *setMapping()* are implemented in the class *MAbstractionImpl*. Generally, the name of the *Getter* method is created by capitalizing the first letter of the attribute name and adding the prefix *get*, and the name of the *Setter* method is formed in the similar manner.

In a case of a *boolean* attributes, the interface looks like:

```
public interface MAssociationEnd extends MModelElement {
   // generating attributes
   …
   // attribute: isNavigable
   boolean isNavigable();
   Void setNavigable(boolean _arg);
   …
}
```

4.2.5.2 Access to Associations

Each association in the NSUML metamodel is an unnamed association between two elements. An association has two roles (or ends), and each role has its own name and multiplicity. The role attached to an element is the direct role, the other one is called the opposite role. The NSUML has no special objects for associations. It maps associations to fields and methods of *element* classes. Each element class contains and treats the information about the opposite role of the association to which it belongs.

Figure 4-3 represents a class diagram with two UML metaclasses: *Feature*, and *Classifier*, and an unnamed association between them.

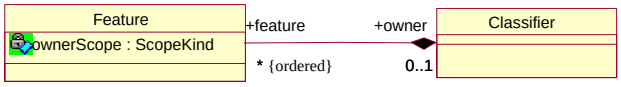

Figure 4-3 Access to Association

1. The role with the name *owner* is the opposite role for metaclass *Feature* and has the multiplicity *0..1*.
2. The role with the name *feature* is the opposite role for metaclass *Classifier* and has the ordered multiplicity *.
3. The association is read as " any *Classifier* contains an arbitrary number of ordered *Features.* A *Feature* can be attached to one *Classifier* only".

NSUML maps role *owner* and role *feature* to private fields in the corresponding class:

Java Class	**Private Field**
MFeatureImpl	MClassifier _owner

MClassifierImpl	List _feature

Table 4-7 Access to Association

Feature contains the information about the owner *Classifier* in the field *_owner*, and *Classifier* contains the list of *Feature* in the field *_feature*. These fields can be accessed through *Getter*, *Setter* and other methods defined in interfaces, as shown in the following segments:

```
pubic interface MFeature extends MModelElement {
   …
   // opposite role: owner  this role: feature
   MClassifier getOwner();
   void setOwner(MClassifier _arg);
   …
}

pubic interface MClassifier extends MNamespace, MGeneralizableElement {
   …
   // opposite role: feature  this role: owner
   List getFeatures();
   Void setFeatures(List _arg);
   Void addFeature(MFeature _arg);
   Void removeFeature(MFeature _arg);
   …
}
```

4.3 UML Metamodel Representations

This section summarizes the metamodel elements used to represent interaction and activity diagrams [UML1.3] by using the NSUML API. This corresponds to the code used in our implementation.

4.3.1 Interaction Diagram

The elements constituting a collaboration diagram and a sequence diagram, as described in [UML1.3], are presented in this subsection.

ModelElement:

A model element is an element that is an abstraction drawn from the system being modeled. It is the base for all modeling meta-classes in the UML. All other modeling meta-classes are either direct or indirect subclasses of Model Element.

Collaboration:

A description of a general arrangement of objects and links that interact within a context to implement a behavior, such as a use case or operation. In the metamodel, a Collaboration contains a set of ClassifierRoles and AssociationRoles, and may also contain a set of Interactions.

Interaction:

A specification of how messages are sent between objects and other instances to perform a task. The interaction is defined in the context of a collaboration. In the metamodel, an Interaction contains a set of Messages.

Classifier:

A classifier is an element that describes behavioral and structural features. It comes in several forms, including class, data type, interface, and component. In the metamodel, a Classifier declares a collection of Attributes, Methods, and Operations.

ClassifierRole:

A classifier role is a specific role played by a participant in a collaboration. It specifies a restricted view of a classifier, defined by what is required in the collaboration. A classifier

role has a reference to a classifier (the base) and a multiplicity. It can be connected to other classifier roles by association roles.

AssociationRole:

An association role is an association that is meaningful and defined only in the context described by a collaboration. In the metamodel, an AssociationRole is a composition of a set of AssciationEndRoles.

AssociationEndRole:

An association-end role specifies an endpoint of an association as used in a collaboration. In the metamodel, an AssociationEndRole is part of an AssociationRole and specifies the connection of an AssociationRole to a ClassifierRole.

Message:

In the metamodel, a Message defines one specific kind of communication between instances in an Interaction such as raising a Signal, invoking an Operation, creating or destroying an Instance.

Instance:

An instance is an individual entity with its own identity and value. In the metamodel, Instance is connected to at least one Classifier which declares its structure and behavior. Instance is an abstract metaclass.

Object:

An object is a discrete entity with a well-defined boundary and identity that encapsulates state and behavior. In the metamodel, an object is a subclass of Instance and it originates

from at least one Class. In this work, we represent two types of objects: active objects, which are instances of active classes, and passive objects, which are instances of passive classes.

Stimulus:

A stimulus is a communication between two objects that convey information. In the metamodel, a stimulus conforms to a Message. A stimulus will cause a Signal sent to an Instance, or an invocation of an Operation. It has a sender, a receiver, and may have a set of actual arguments, all being Instances.

AttributeLink:

An attribute link is a named slot in an instance, which holds the value of an attribute. In the metamodel AttributeLink is a piece of the state of an Instance and holds the value of an Attribute.

Link:

A link is a connection between instances. In the metamodel Link is an instance of an Association.

LinkEnd:

A link end is an end point of a link. In the metamodel LinkEnd is the part of a Link that connects to an Instance.

Signal:

A signal is a specification of an asynchronous stimulus communicated between instances. In this work, both signals and asynchronous stimuli are considered to be the same thing.

Action:

An action is an executable atomic computation that results in a change in the state of the model or the return of a value. An action has a target object, a reference to the signal to be sent or the operation to be performed, a list of argument values, and an optional recurrence expression specifying possible iteration.

ActionSequence:

An action sequence is a collection of actions. In the metamodel, an Action Sequence is an Action, which is an aggregation of other Actions.

CallAction:

A call action is an action resulting in an invocation of an operation on an instance. In the metamodel, CallAction is an Action. The designated Instance or set of Instances is specified via the target expression, and the actual arguments are designated via the argument association inherited from Action.

SendAction:

A send action is an action that results in the sending of a signal. In the metamodel, SendAction is an Action. It is associated with the Signal to be raised.

CreateAction:

A Create Action is an action resulting in the creation of an instance of some classifier.

DestroyAction:

A Destroy Action is an action results in the destruction of an object specified by the target association of the Action.

TerminateAction:

A Terminate Action results in self-destruction of an object. The target of a Terminate Action is implicitly the Instance executing the action.

Figure 4-4 and Figure 4-5 show the NSUML elements involving in collaboration diagrams and sequence diagrams, respectively. To simplify, only the attribute and operation names are shown, rather than their full description (arguments, return type, etc). Notice that the aggregations shown in two Figures actually mean compositions. Rational Rose does not provide a notation for a composition, which is a solid filled diamond.

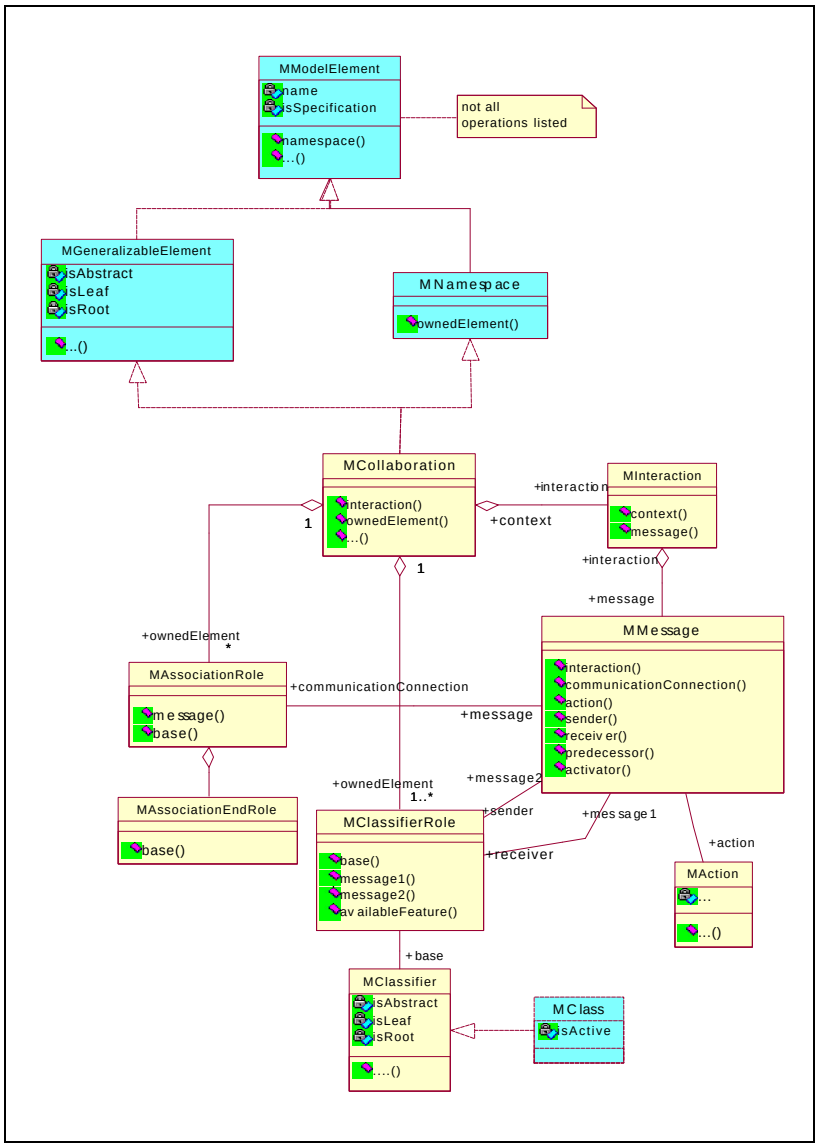

Figure 4-4 Metamodel Representation for a Collaboration Diagram

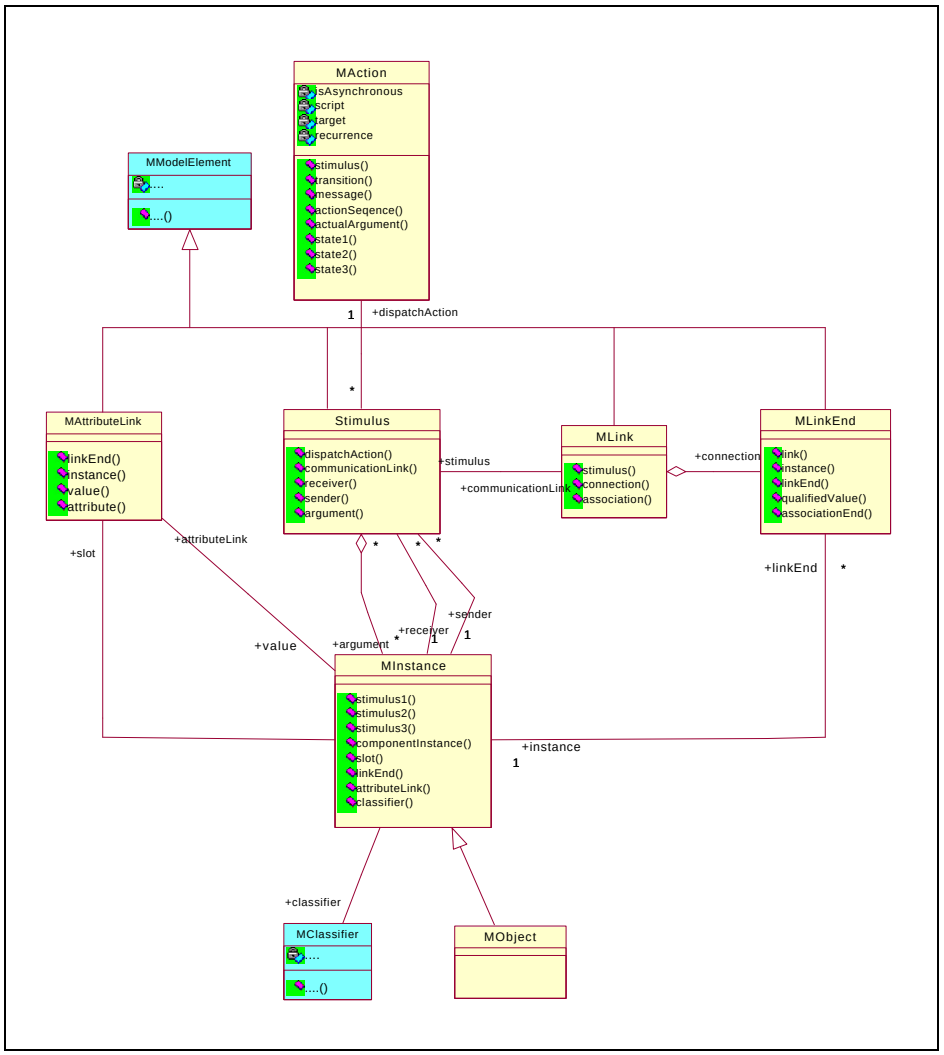

Figure 4-5 Metamodel Representation for a Sequence Diagram

4.3.2 Activity Diagram

Semantically, an activity diagram is a state machine that emphasizes the sequential and concurrent steps of a computational procedure. Therefore, an activity diagram shares many metamodel elements with a state machine.

The following elements are used in activity diagrams defined in [UML1.3]:

ActivityGraph:

An activity graph is a special case of a state machine that defines a computational process in terms of the control-flow and object-flow among its constituent actions. In the metamodel, ActivityGraph extends StateMachine.

Transition:

A transition is a directed relationship between a source state vertex and a target state vertex. Transition is a child of Model Element.

StateVertex:

A state vertex is an abstraction of a node in a state chart graph. In general, it can be the source or destination of any number of transitions. State Vertex is a child of Model Element.

State:

A state is an abstract meta-class that models a static situation, such as an object waiting for some external event to occur, or a dynamic situation, such as the process of performing some activity. The model element under consideration enters the state when the activity starts and leaves it as soon as the activity is completed. State is a child of State Vertex.

PseudoState:

A pseudo state is an abstraction that includes different types of transient vertices that are used to connect multiple transitions into more complex state transitions paths. Pseudo State is a child of State Vertex. Here are some of the pseudo states used in this work:

- An **initial** Pseudo state represents a default vertex that is the source for a single transition to the *default* state of a composite state. There can be at most one initial vertex in a composite state.
- A **join** pseudo state serves to merge several transitions coming from different source state vertices. The transitions entering a join vertex cannot have guards.
- A **fork** pseudo state serves to split an incoming transition into two or more transitions. The segments outgoing from a fork vertex must not have guards.
- A **branch** pseudo state splits the transition path into two or more segments, each with a separate guard condition. A merge converges multiple incoming transitions into a single outgoing transition. A merge is the inverse of a branch and uses the same notation (diamond symbol) as a branch except a merge has no conditions.

CompositeState:

A composite state is a state that contains other state vertices (states, pseudo states, etc.). A composite state is mainly used in state machine and can be decomposed into concurrent substates or into mutually exclusive disjoint substates. Composite State is a child of State.

ActionState:

An action state represents the execution of an atomic action, typically the invocation of an operation. It is a state whose purpose is to execute an entry action, after which it takes a completion transition to another state. An action state has no substructure, internal activities, or internal transitions.

ObjectFlowState:

An object flow state defines an object flow between actions in an activity graph. Operating on an object by an action in an action state may be modeled by an object flow state that is triggered by the completion of the action state. Generally each action places the object in a different state that is modeled as a distinct object flow state.

FinalState:

A final State is a special state signifying that the enclosing composite state is completed. A final state cannot have any outgoing transitions. Final State is a child of State.

Guard:

A guard is a Boolean expression that is attached to a transition as a control over its firing. If the guard is true at its evaluation time, the transition is enabled; otherwise, it is disabled. Guard is a child of Model Element.

Partition:

A partition (swimlane) is a mechanism for dividing the states of an activity graph into groups. Partitions often correspond to organizational units in a business model.

CallEvent:

A call event represents an event of receiving a call for an operation that is implemented by actions in state machine transitions.

SignalEvent:

A signal event represents the reception of an asynchronous signal.

Figure 4-6 shows the involved NSUML elements in activity diagrams:

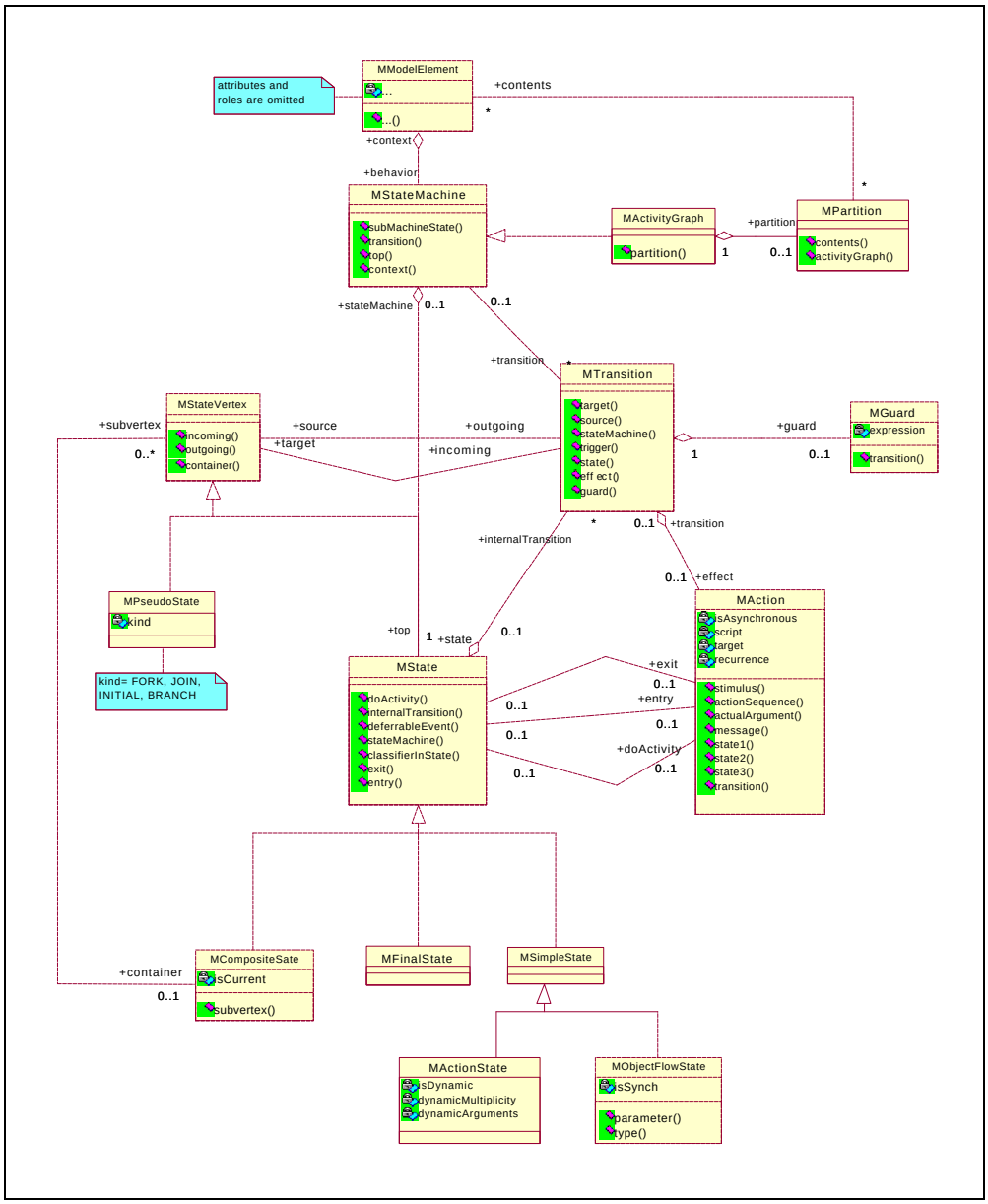

Figure 4-6 Metamodel Representation for an Activity Diagram

4.4 Metaobjects for Some Basic Cases

In the above section the metamodel representations for interaction diagram and activity diagram are depicted at metaclass-level. The transformation, however, will ultimately handle a specific diagram and deal with objects instead of classes. This section will use object diagrams, which are used to model object structures at a given moment in time, to illustrate objects' participation in several typical basic cases described in the previous chapter.

Each case described below contains two object diagrams: one represents the metaobjects in the sequence diagram, and the other represents the metaobjects generated for the equivalent activity diagram. For each case, the detail description is given in section 3.2.1.

4.4.1 Case 1: Sequential Execution in a Single Thread

Input Data Structure shown in Figure 4-7:

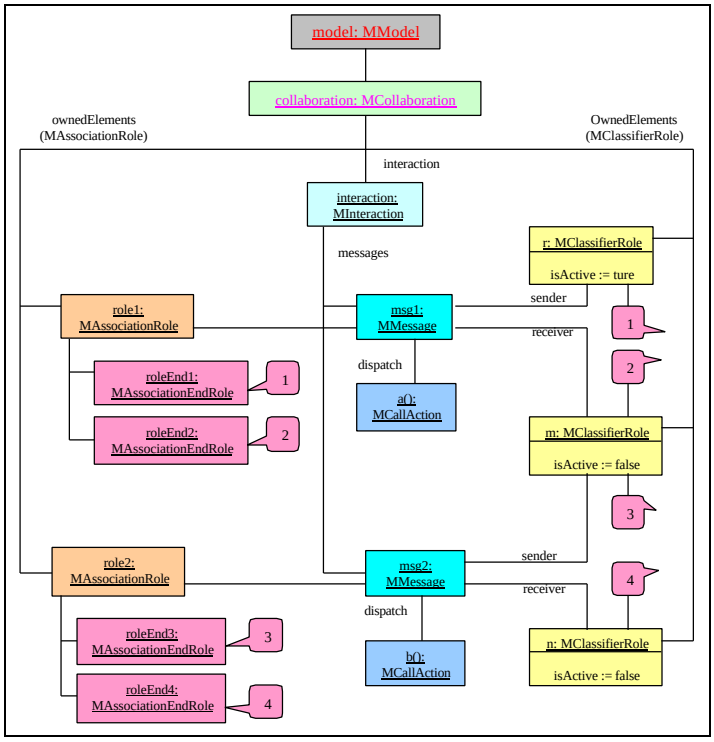

Figure 4-7 SD Metaobjects for Sequential Execution

Output Data Structure shown in Figure 4-8:

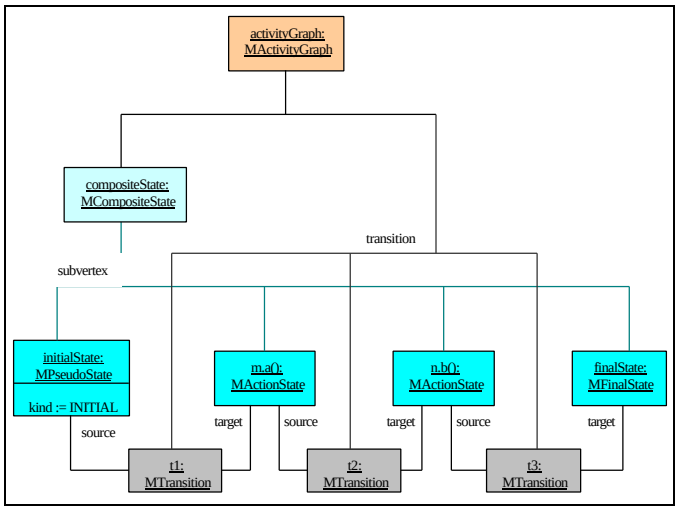

Figure 4-8 AD Metaobjects for Sequential Execution

4.4.2 Case 2: Synchronous Messages Send and Reply

Input Data Structure shown in Figure 4-9:

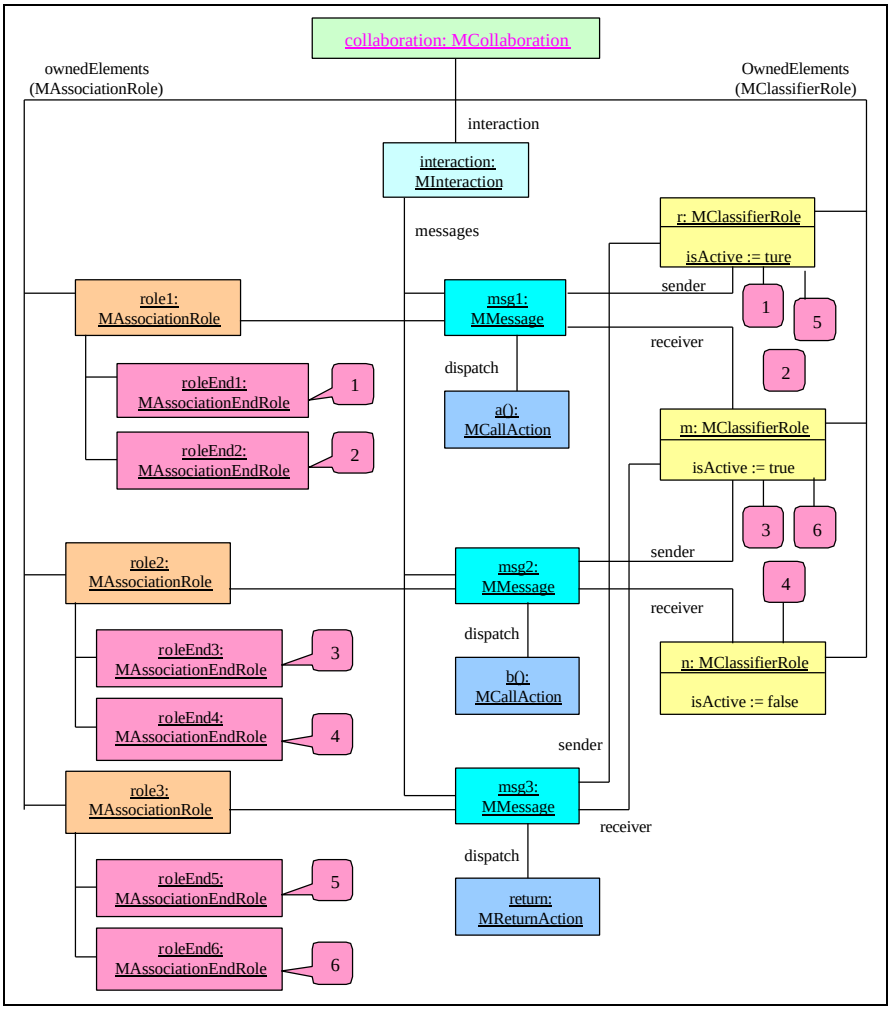

Figure 4-9 SD Metaobjects for Synchronous Message Send and Reply

Output Data Structure shown in Figure 4-10:

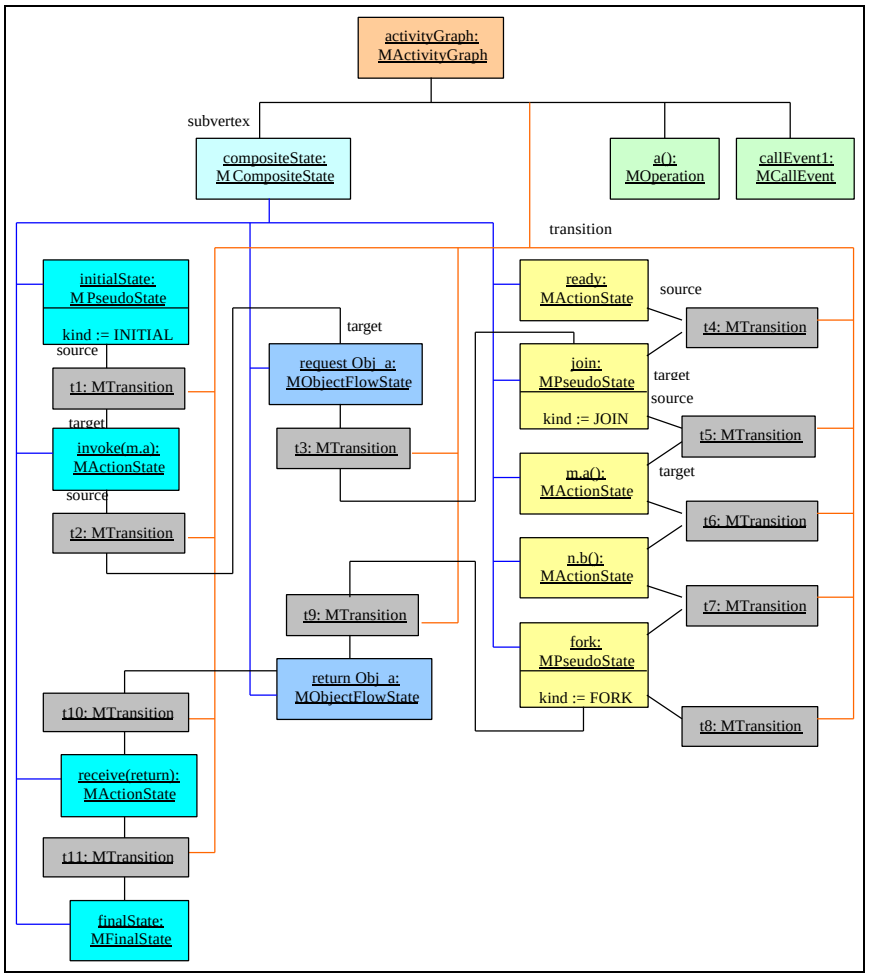

Figure 4-10 AD Metaobjects for Synchronous Message Send and Reply

4.4.3 Case 3: Asynchronous Creation of an Active Object

Input Data Structure shown in Figure 4-11:

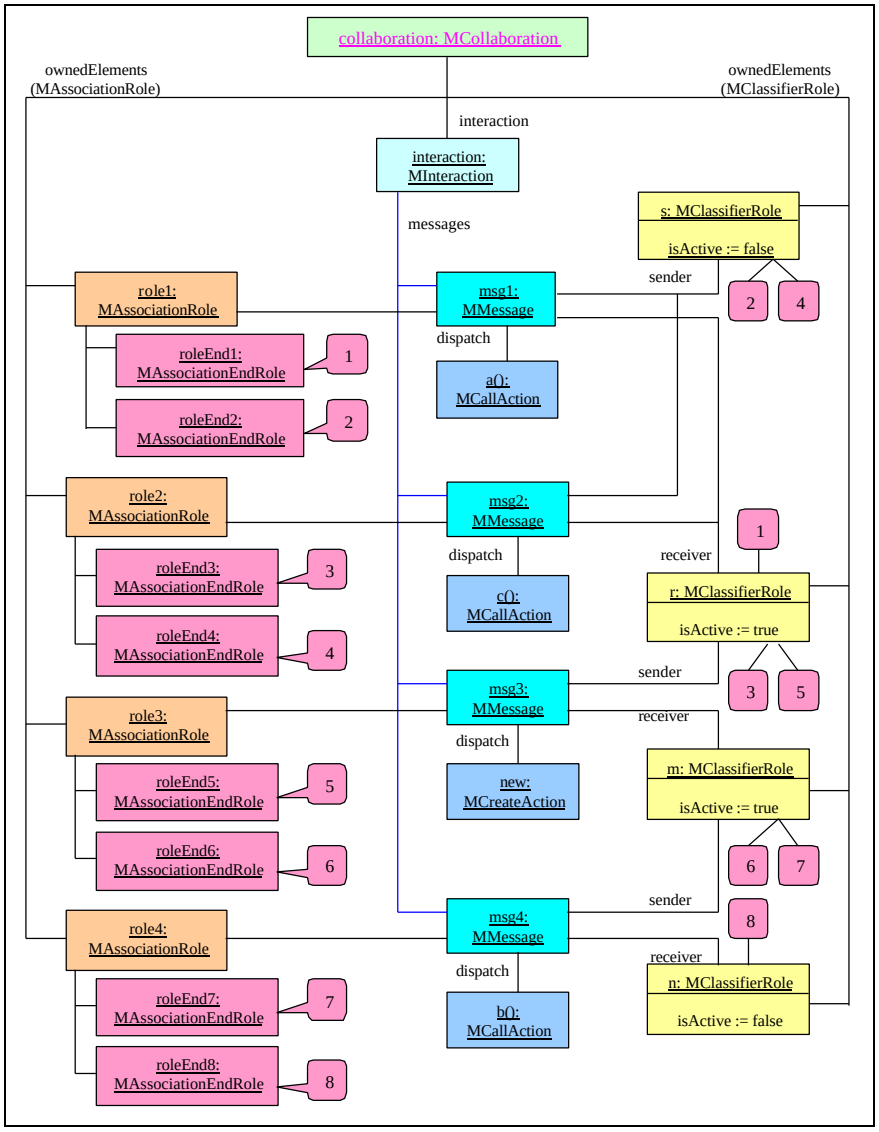

Figure 4-11 SD Metaobjects for Asynchronous Creation of an Active Object

Output Data Structure shown in Figure 4-12:

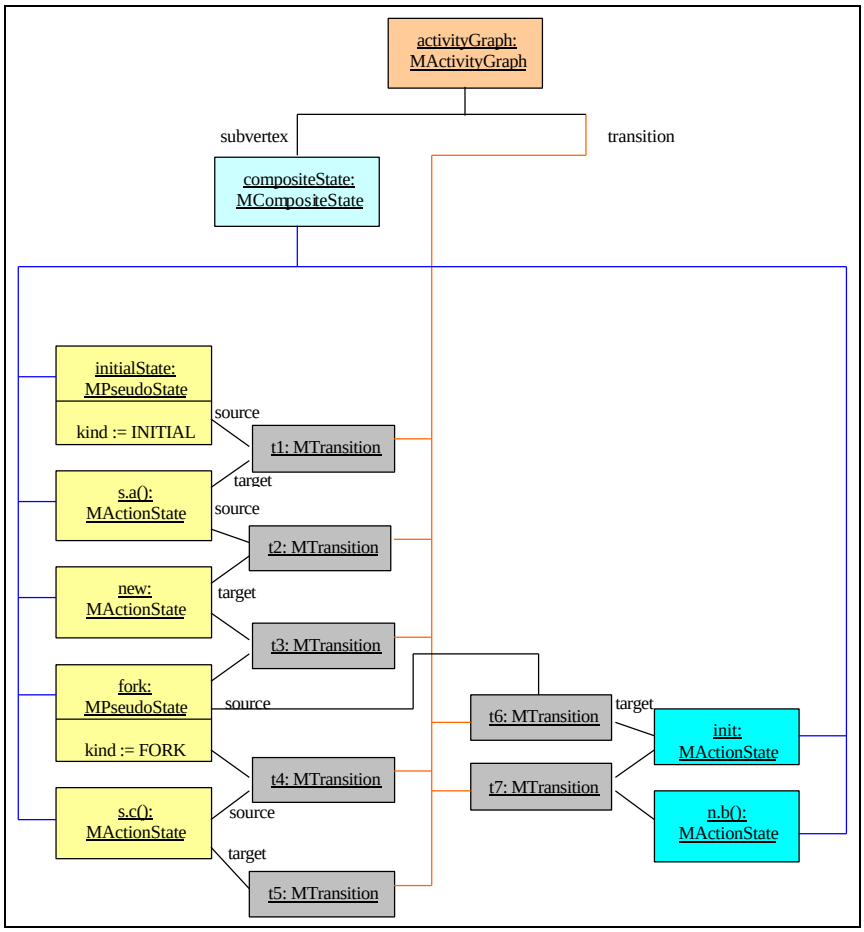

Figure 4-12 AD Metaobjects for Asynchronous Creation of an Active Object

4.4.4 Case 4: Asynchronous Messages between Two Threads

Input Data Structure shown in Figure 4-13:

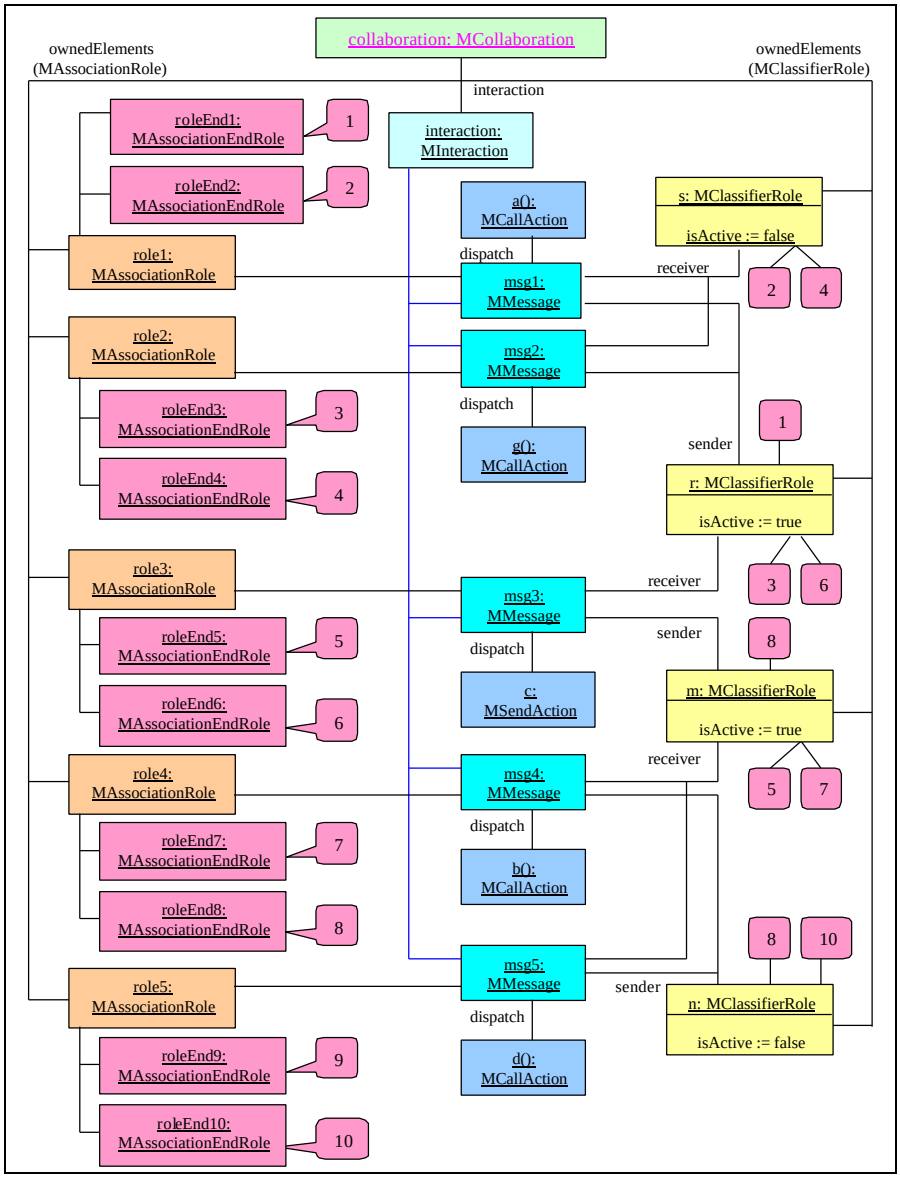

Figure 4-13 SD Metaobjects for Asynchronous Messages between Two Threads

Output Data Structure shown in Figure 4-14:

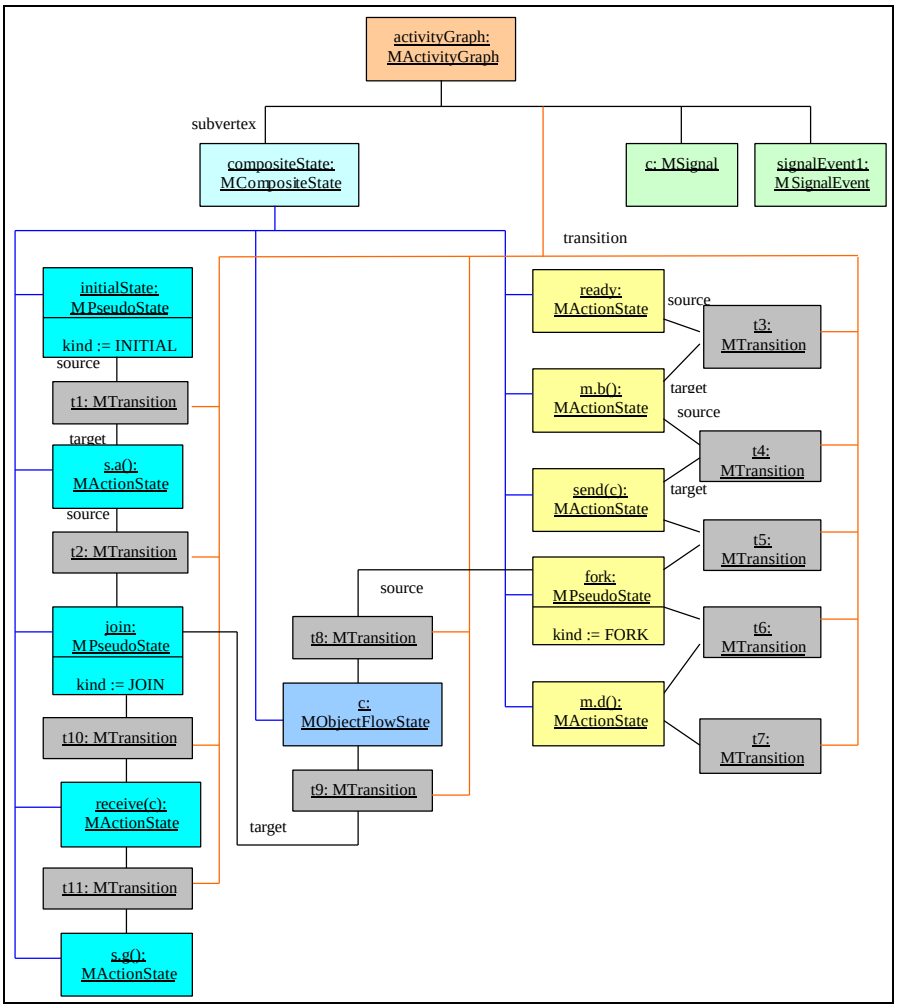

Figure 4-14 AD Metaobjects for Asynchronous Messages between Two Threads

4.5 Transformation Algorithm

The algorithm reads an XMI files containing the input interaction diagram, transforms it into an output activity diagram and finally writes the XMI file containing the output model. The most interesting part is the middle of the algorithm that does the actual transformation.

Primarily what the transformation does is to create an output model that represents the equivalent activity diagram from the input model, based on the transformation rules that were described in chapter 3. These rules can be roughly seen as a transformation algorithm expressed at the notation level. Each basic case reflects a facet of the algorithm in certain situations. In this section the transformation algorithm is described at the metamodel level, showing how it manipulates and creates metaobjects. The main steps of the transformation algorithm (at a high level of abstraction) are as follows:

```
1 initialize new model and activityGraph;
2 sort messages and put them into a list;
3 partition objects;
4 for(each message in the list) do
5   if(message is concurrent) then
6     create a fork;
7     handle concurrent messages at the same level;
8   endif
9   if(sender and receiver are in the same partition) then
10    create a StateVertex;
11    handle action with condition;
12    handle loop situation;
13    endif
14  else /* sender and receiver are in different partition */
15    handle CallAction;
16    handle SendAction;
17    handle CreateAction;
18    handle DestroyAction;
19    handle ReturnAction;
20    handle TerminateAction;
21  endif
22 endfor
23 finalize activityGraph and model;
```

The algorithm generally follows the flow of messages from the input interaction diagram. The messages passing between objects in an interaction diagram are partially ordered. This means that the messages may be sequential or concurrent. Within an interaction, the

messages are related by the predecessor and activator relationships, as discussed in chapter 3. We sort the messages based on their predecessor and activator relationships. The predecessor (sequencing) relationship organizes the messages into a linear sequence. If two messages have a common predecessor and are not otherwise sequenced, then they may be executed concurrently. The activator (caller-called) relationship defines nested procedure structure. Each call adds another level of nesting to the sequence. Within a call, messages have a predecessor relationship to establish their relative order (which may permit concurrency). The messages are traversed in order, according to the for loop found in the line 4 of the pseudocode. Each message will have a sender, receiver and an associated action. The algorithm first checks whether the sending object and receiving object are in the same execution thread (i.e. in the same partition), which falls into two situations. For each situation, it takes the appropriate way to handle different kinds of action. During the transformation procedure, different model elements, such as Transition, PseudoState, ActionState, ObjectFlowState, will be created and linked together to construct an ActivityGraph and thereafter, a new model.

The partitioning of objects by execution threads plays an important role in the way the activity diagram is generated. However, this information cannot be found in the input interaction diagram. Our algorithm needs to receive partitioning information either from the user, or from other UML diagrams, such as component or deployment diagrams. In our implementation, the user decides on object partitioning. However, when the algorithm will

be used in the larger content of transforming UML models into performance models, such partitioning information will be extracted from other UML diagrams.

The transformation algorithm was given at very high level. In next chapter, as we dig into the implementation, more details of the algorithm, such as handling different kinds of action and creating transitions to link state vertices, will be described.

Chapter 5 Implementation of Transformation Rules

The ID to AD transformation consists of three main parts: XMI input, XMI output and transformation, as illustrated in the thesis scope in Figure 1-2. The XMI input converts XMI elements into NSUML objects (`unmarshalling`). The XMI output can be seen as the inverse of XMI input, which converts NSUML objects into XMI elements (`marshalling`). The transformation part is the most important for the thesis, analyzing the objects of an interaction diagram and generating the objects of the corresponding activity diagram according to the transformation rules. The structure of chapter 5 is as follows. The first three sections correspond to those three parts, followed by a discussion of limitations. Then, a discussion of the implementation verification follows, which contains a description of DOM and of the testing evaluation. Finally an e-commerce system model is investigated as a case study.

The implementation is developed in Borland JBuilder 3© under the Windows NT© platform. Three kinds of APIs need to import: JDK 1.3 and JAXP 1.1 from Sun Microsystem©, and Novosoft© UML API 0.4.19. All of them can be download from the corresponding companies' Web sites. JAXP 1.1 API provides the necessary methods to handle XML and Document Object Model (DOM). Novosoft UML API implements the metamodel of UML 1.3 and provides XMI-support. The related environments to the implementation are summarized in Table 5-1:

UML Design Tools	XMI Support	UML	XMI
Rational Rose 2000 Enterprise Edition	Unisys Rose XML Tools 1.3.2	1.3	1.0
ArgoUML v.0.8.1a	NSUML 0.4.19	1.3	1.0

Table 5-1 Supporting Environments

5.1 XMI Input

ArgoUML and Rose use different mechanisms to generate an XMI file for a model. In ArgoUML, several files will be generated everytime when a model is saved. This includes an XMI file that stores model information and PGML files that store layout information. Rose saves a model in its mdl file. A separate XMI file (Rose uses .xml as its extension) is obtained through a function called “Export Model to UML” in the Tools menu.

This section is divided in two parts: the first one presents the XMI structure of an interaction diagram, which represents the XMI objects in a tree-structure. The second describes how the XMI reader translates XMI elements to Java objects.

5.1.1 XMI Structure of Interaction Diagram

Semantically, the XMI files generated from UML tools contain the metamodel objects that represent different diagrams, as discussed in the previous chapters. However, an XMI file is a special kind of XML file that follows the standard UML DTD. Consequently, an XMI file can be represented in a tree-look structure as any other XML file. Figure 5-1 and Figure 5-2 illustrate the tree-structure for collaboration and sequence diagrams.

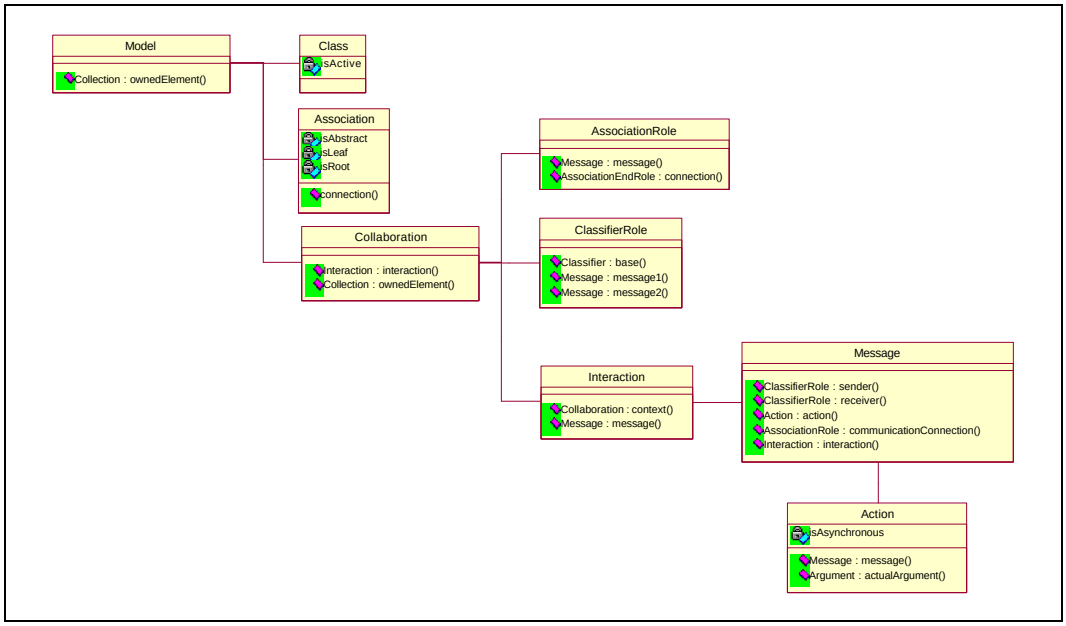

Figure 5-1 Collaboration Diagram XML Tree Structure

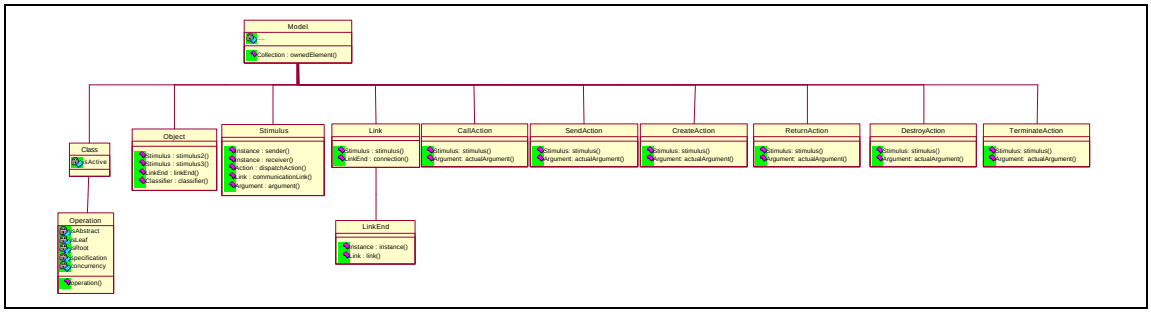

Figure 5-2 Sequence Diagram XML Tree Structure (only for ArgoUML tool)

If Rose is used to draw an interaction diagram (be it a collaboration or a sequence diagram), it maps to the XMI structure shown in Figure 5-1. This is because in Rose the two diagrams are semantically equivalent. Using ArgoUML to draw a collaboration or a sequence diagram results in a different XMI structure corresponding to Figure 5-1 and Figure 5-2, respectively.

5.1.2 XMI Reader

The XMI input is realized by the class `XMIReader` found in the NSUML. The class is a huge java file consisting of more than 20,000 lines of source code. Its responsibility is to create Java objects (NSUML objects) from an XMI file. In fact, XMIReader uses an existing parser to parse the input file, as described in the next section.

5.1.2.1 Simple API for XML (SAX)

`XMIReader` uses SAX (Simple API for XML) which is an event-based XML API. The following segment of code is taken from `XMIReader`:

```
public class XMIReader extends HandlerBase {
   …
   org.xml.sax.Parser parser = null;
   public XMIReader() throws SAXException,
                        ParserConfigurationException {
      …
      /* get SAX parser, which is event-driven */
      SAXParserFactory saxpf = SAXParserFactory.newInstance();
      parser = saxpf.newSAXParser().getParser();

      parser.setErrorHandler(this);
      parser.setDocumentHandler(this);
      parser.setEntityResolver(this);

      …
   }
   public MModel parse(InputSource p_is) {
      …
      parser.parse(p_is);
      …
   }
   …
}
```

The technique is to register a handler with the SAX parser, after which the parser invokes the appropriate callback methods whenever it sees a new XML tag or encounters an error.

Another major type of XML API is the tree-based DOM API. DOM API compiles an XML document into an internal tree structure and then allows an application to navigate that tree. DOM will be used to make the internal structure visible to facilitate the verification, which will be discussed in section 5.5. SAX and DOM APIs are defined by XML-DEV and by the W3C, respectively.

The basic structure of the SAX parser is shown below (Figure 5-3):

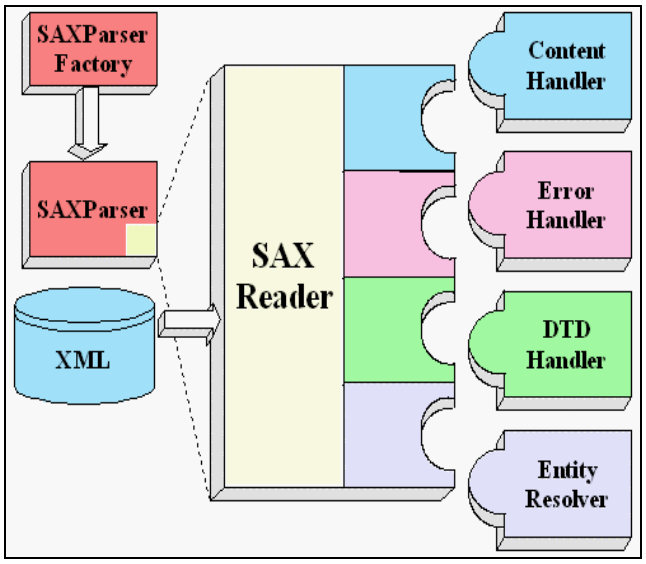

Figure 5-3 SAX Parser

Two packages are needed: `org.xml.sax` and `javax.xml.parsers`.

- Package `org.xml.sax` defines the SAX interfaces. This package also defines `HandlerBase` – a default implementation of a base class for the various "handlers" defined by the interfaces, as well as an `InputSource` class, which encapsulates information that tells where the XML data is coming from.

- Package `javax.xml.parsers` defines the `SAXParserFactory` class which returns the SAXParser. Also it defines the `ParserConfigurationException` class for reporting errors.

Here is a summary of the key SAX APIs:

- `SAXParserFactory`: generates an instance of the parser.
- `Parser`: The `org.xml.sax.Parser` interface defines methods like `setDocumentHandler` to set up event handlers and `parse(URL)` that, as the data in XML is parsed, invokes one of several callback methods defined by the interfaces `DocumentHandler`, `ErrorHandler`, `DTDHandler`, and `EntityResolver`.
- `DocumentHandler`: Methods like `startDocument`, `endDocument`, `startElement`, and `endElement` are invoked when an XML tag is recognized. This interface also defines methods `characters` and `processingInstruction`, which are invoked when the parser encounters the text in an XML element or an inline processing instruction, respectively.
- `ErrorHandler`: Methods `error`, `fatalError`, and `warning` are invoked in response to various parsing errors. The default error handler throws an exception for fatal errors and ignores other errors (including validation errors).
- `EntityResolver`: Method resolveEntity is invoked when the parser must identify data by a URI.

- `DTDHandler`: Methods defined in this interface are invoked when processing definitions in a DTD.

A typical application such as `XMIReader` needs to provide only a `DocumentHandler` at a minimum. It can override the methods for some events and ignore the methods for other events.

5.1.2.2 Elements Processing

To understand how an event-based `XMIReader` works, consider the following sample document:

```
<?xml version=”1.0”>
<doc>
<para>Hell, world!</para>
</doc>
```

An event-based interface will break the structure of this document down into a series of linear event:

```
start document
start element: doc
start element: para
characters: Hello, world!
end element: para
end element: doc
end document
```

`XMIReader` defines five methods to handle those events: `startDocument`, `endDocument`, `startElement`, `endElement`, and `characters`. When a start tag or end tag is encountered, the name of the tag is passed as a String to the `startElement` or `endElement` method, as appropriate. When a start tag is encountered, any attributes it defines are also passed in an `AttributeList`.

`XMIReader` has a method called `process(String, AttributeList)` which is mainly responsible for creating NSUML objects. Each object has a unique id (`xmi.id`) within a document. A HashMap will be used to keep the ids and objects. As described in chapter 4, every object may have attributes or associations (opposite roles). Thus the next step is to process an object's attributes and associations if they are available. Because the UML DTD determines the structure of an XMI file, `XMIReader` uses String comparison against the DTD to distinguish attributes and associations.

Processing an object's associations is more complicated than processing its attributes. This will involve how to build and keep object references. XMI provides `xmi.idref` that allows an XMI element to refer to another XMI element within the same document using the XML IDREF mechanism. Recall that a HashMap has been used to store all the objects and their ids. If an `xmi.id` that `xmi.idref` refers to can be found, then the referred object will be assigned as an association. But if a referred id cannot be found in the HashMap, this means the referred object is yet to be created. An internal private class `Link` in `XMIReader` is going to handle this situation. Class `Link` has instance variables `sourceObject` that represents the owner object of the association, and `parameterXMIID` that represents the id of referred object. After the parser completes parsing a document, a method `performLinking()` will be invoked to link a source object and its referred object together.

5.2 XMI Output

When a transformation from an interaction diagram to an activity diagram is done, it means that a new model that represents the newly generated activity diagram is created. This section illustrates what the XMI structure of an activity diagram looks like, and briefly explains how `XMIWriter` writes a model out in XMI format.

5.2.1 XMI Structure of Activity Diagram

The XMI structure of an activity diagram is shown in Figure 5-4.

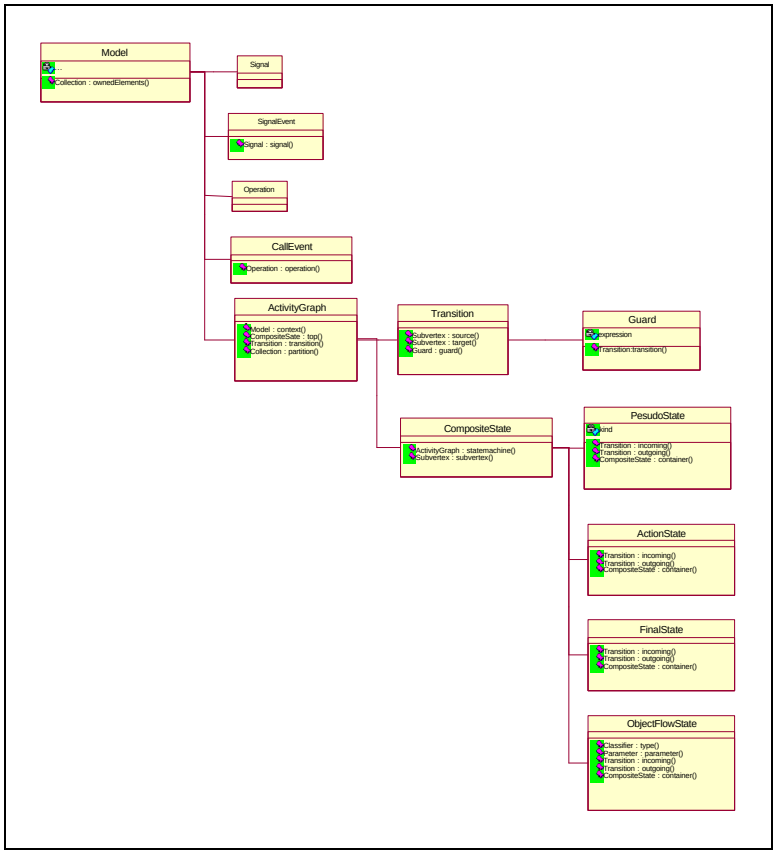

Figure 5-4 Activity Diagram XML Tree Structure

Several points should be mentioned about the figure. First, compared to its metamodel representation, the XMI structure is quite simplified. This is because relationships between classes are hidden in element associations, e.g. shown by `xmi.idref`. Second, The figure shows only the elements that have `xmi.id`. Not all classes converted to elements will be assigned `xmi.id`. For example, class `Partition` has no `xmi.id`. It is a collection that consists of different state vertices.

5.2.2 XMI Writer

`XMIWriter` consits of even more lines of code than `XMIReader`, but the structure of `XMIWriter` is actually simpler than `XMIReader` because object constructions are not needed anymore. Dismantling a machine is always easier than assembling it together again.

`XMIWriter` is the inverse of `XMIReader`, so they are similar in many respects. The following codes are segments of `XMIWriter`:

```
public class XMIWriter extends PrintWriter {
   …
   protected MModel mmodel = null;
   Public XMIWriter(MModel p_mmodel, java.io.Writer p_out) throws
          IOException {
     super(p_out);
     MModel = p_mmodel;
   }
   …
   protected AttributeListImpl al = new
      AttributeListImpl();
   protected org.xml.sax.DocumentHandler dh = null;

   public void gen(org.xml.sax.DocumentHandler p_dh) throws
         IncompleteXMIException{

      dh = p_dh;
      dh.startDocument();
```

```
        al.addAttribute("xmi.version", CDATA_TYPE, "1.0");
        dh.startElement("XMI", al); al.clear();

        dh.startElement("XMI.header", al);
        … /* declares some common XMI elements */
        dh.endElement("XMI.header");

        dh.startElement("XMI.content", al);

        printModelMain(getModel()); /* main process method */

        dh.endElement("XMI.content");
        dh.endElement("XMI");

        dh.endDocument();
        …
}
```

`XMIWrtier` extends `PrinterWriter` that is to print formatted representations of objects to a text-output stream. Its constructor takes two parameters: `MModel` and `Writer`. In order to generate an XMI, an instance of `XMIWriter` must indirectly call the method `gen(DocumentHandler)`. In this method a declaration comes first, which identifies the version and encoding scheme. After the declaration, a root element `XMI` is defined. Any other elements are contained within that element. Under the root element `XMI`, two nested elements are defined: `XMI.header` and `XMI.content`. Elements to be defined are not arbitrary, they must conform to the UML DTD so that the generated XMI is not only well-formed, but also valid.

`XMI.header` contains several common XMI elements, e.g. `XMI.exporter` and `XMI.metamodel`. Detail explanations can be found in [XMI 1.0]. `XMI.content` contains the actual model information being transferred. To output the model as XMI format is implemented by method `printModelMain()`.

The method `printModelMain()` bases on the UML DTD to write out a model and its contained objects. As we mentioned before, a DTD specifies the kinds of tags that can be included in a document, and the valid structure of elements. The object may contain attributes and associations, all those are converted to XMI elements. The element converted from an object will be given an `xmi.id` as an attribute, which is an integer starting from 1. The element converted from an association will be given an `xmi.idref` as its attribute, which refers to the xmi.id of the owner object that association belongs to.

5.3 Transformation

This section describes how we built the Java application to perform the transformation according to the algorithm that is given in chapter 4. The application contains several components, as shown in Figure 5-5.

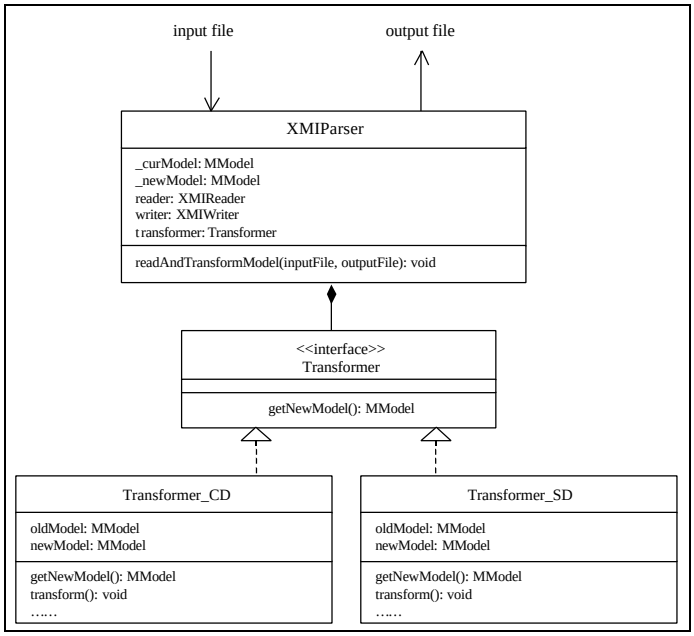

Figure 5-5 Transformation Components

In this figure, the class `XMIParser` has instance variables of `XMIReader` and `XMIWriter`. Besides, `XMIParser` needs to know whether the input is from a collaboration diagram or from a sequence diagram before it passes the input model information to an appropriate `Transformer`. `Transformer` is an interface that is realized by two classes: `Tranformer_CD` which handles the input from an collaboration diagram, and `Transformer_SD` which handles the input from a sequence diagram. These two classes follow exactly the same algorithm, but deal with different XMI input structures (as described in section 5.1.1).

5.3.1 Transform

The transformation algorithm is actually executed in the method `transform()`. The purpose of the method is to create a new model to represent the equivalent activity diagram. This is demonstrated by the segment of code given as follows:

```
public void transform() throws IOException {
  …
  sortMessages();

  partitionObjects();
  initializePartitions();
  …
  traverse();
  …
  finalizePartitions();
  addContentsToPartitions();

  setActivityGraph();
  newModel.addOwnedElement(actGraph);
  …
```

```
}
```

For sorting messages, UML defines the following syntax for a message label:

predecessor guard-condition sequence-expression return-value :=
message-name argument-list

Both *predecessor* and *sequence-expression* provide necessary information for sorting. In most cases, *predecessor* is implied by the numeric sequence numbers and need not be explicitly listed. The API provides two relevant methods: `getPredecessors()` and `getActivator()`, thus the relative order of the messages can be obtained. After the sorting, the messages will be put into an *ArrayList* like the following:

{msg_1, msg_2, … {msg_a, msg_b, …}, … , msg_n }

where the sublist {msg_a, msg_b, …} contains messages that are concurrent.

For partitioning the objects, the application asks the users how to partition by listing all active and passive object `xmi.ids` in a *COMMAND* window. We made the assumption that a partition (swimlane) contains the activities carried out by only one active object and any number of associated passive objects. In other words, each partition represents a concurrent component. HashMap will be used to hold each partition and its related objects for later references. The next step is to initialize each partition by assigning an initial state, which indicates the components are running concurrently. To be consistent with the UML standard, only the first partition, associated with an active object that sends the first message, will be given an Initial Pseudostate. Each of the remaining partitions will start

with an “idle” ActionState to indicate that is up waiting for a message. A HashMap `partition2StateVertices` is used to keep track of all the state vertices for each partition. Some utility methods, e.g. `add()` and `remove()`, are created to manipulate the HashMap.

5.3.2 Traverse

The for loop in the transformation algorithm is performed by the method `traverse()`. When traversing messages, two situations will be encountered: the sender and receiver of a message are in the same or in different execution thread. In each case, the transformation takes appropriate ways to handle according to the type of action associated with the message. The case with different execution thread is more complex than the case with one execution thread as more factors need to be taken into consideration. The description below will focus on the more complex case. For those messages that are running concurrently, e.g. msg_a, msg_b, a Fork Pesudostate will be created to indicate concurrency, then each message still falls into one of the two cases mentioned above. This may be done recursively.

When a message is being traversed, its associated action provides the most important information for the transformation. UML defines 7 kinds of Actions (see section Message Properties in chapter 3). The most two common kinds are CallAction and SendAction. Figure 5-6 shows how we transform these two kinds of Actions. The transformation

includes the creation of new state vertices, such as action states, fork/join pseudostates, and object flow states, and of new transitions to link the state vertices together.

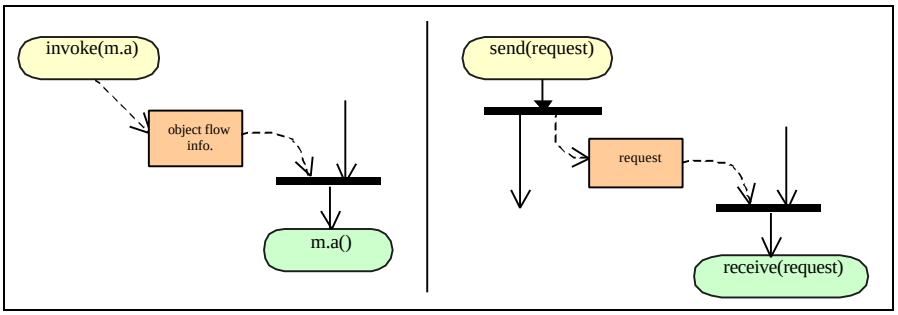

Figure 5-6 Transformation: CallAction (left), SendAction (right)

CallAction is synchronous. The caller is blocked and yields control to the called procedure until it returns. It is assumed in UML notation that every call has a paired return which may be omitted, i.e. implicit at the end of an activation. SendAction is asynchronous, resulting in an explicit fork. For both of them, ActionStates are created and given meaningful names. We use the notation of ObjectFlowState but do not adhere strictly to its semantics. It will be used to convey performance information between entries and tasks in the future when activity diagrams convert to LQN models.

When handling different kinds of Action, the transformation may create two particular kinds of PseudoState, fork and join. The value of Action's attribute `isAsychronous` indicates whether to create a fork PseudoState or not. A join PseudoState is needed when the receipt of a message is from another execution thread. There is an exception if there is an explicit return action in a procedure call since the receiver (caller) waits for the completion of the called procedure.

The transitions used in activity diagrams are simpler than in state machines. The method `connect()` is used to generate a transition that connects a source StateVertex and a target StateVertex, as follows:

```
public MTransition connect(MStateVertex from, MStateVertex to) {
   MTransition transition = new MTransitionImpl();
   …
   transition.setSource(from);
   transition.setTarget(to);
   …
   /* set guard if there is a branch */
   …
   from.addOutgoing(transition);
   to.addIncoming(transition);
   …
}
```

It is possible that StateVertex `from` or `to` is not finalized yet when the method is called. For example, if a transition is going to link from a `fork` PseudoState to an ActionState, the `fork` may have other outgoing transitions in a later. Therefore, the appropriate updates for both the transition and StateVertex will be necessary if one of them is changed. Similar treatments also apply to other model elements that have bilateral associations between them.

5.4 Limitations and Discussion

There is a compromise in the UML between the desire for precision and the need of developers to work with various design tools, which may have different interpretations of the UML semantics. On one side, the existing UML semantics documentation and the metamodeling approach already provide a good foundation for a precise semantics. But the

meaning of the UML, which is mainly described in English, is informal and unstructured, therefore does not provide a solid foundation for developing formal analysis and development techniques. The semantics of actions and argument lists, for instance, have therefore been left somewhat incomplete and ambiguous within UML itself. As a result, the API (NSUML API) we import is also incomplete in some aspects.

Another problem is that the current UML tools, such as Rational Rose or ArgoUML, do not support entirely the whole set of features provided by the UML metamodel. For example, both Rose and ArgoUML do not support some model elements or functions, such as object flow state, concurrent messages, iteration, and so on. This means that the XMI files obtained from the tools are either incomplete or imprecise. When we needed XMI files to test our implementation, we had to modify some of the files by hand in order to introduce features that are not yet supported by today's tools.

5.5 Verification

To facilitate the verification of the generated output in XMI format, it would be better to make the internal data structure in XMI visible. To do that a GUI application is built to display an XMI using Document Object Model (DOM).

5.5.1 Document Object Model (DOM) and JTree

In section 5.1.2.1, we surveyed the event-driven SAX API. An alternative to access an XML document structure is to use a tree-based DOM API.

DOM is a tree structure, where each node contains one of the components from an XML structure. It was developed by the W3C, primarily to specify how future Web browsers and embedded scripts should access HTML and XML documents. There is a core standard that applies to both HTML and XML (available from [DOM]).

DOM provides a set of APIs to access and manipulate nodes in the DOM tree. However, the DOM standard is silent on the subject of how to create a DOM from an existing XML file. This problem is solved by the JAXP `DocumentBuilder` interfaces, as shown in Figure 5-7.

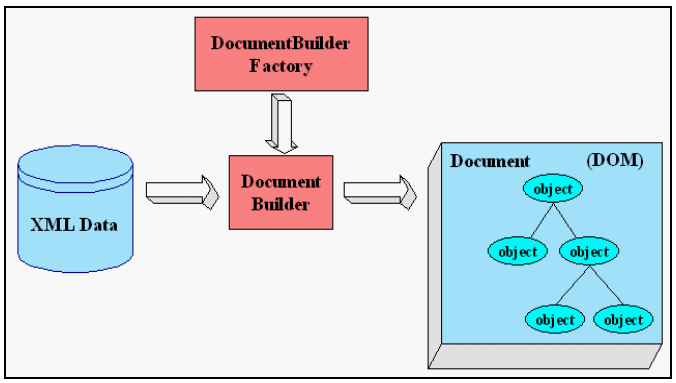

Figure 5-7 JAXP APIs

When the input source, either a File object, an input stream, a SAX InputSource object, or a URL, is parsed, the `DocumentBuilder` will return an `org.w3c.dom.Document` object:

```
DocumentBuilder builder =
      DocumentBuilderFactory.newInstance().newDocumentBuilder();
Document document = builder.parse(input source);
```

Details of JAXP API are given in [JAXP 1.1].

To help to have a clear idea of how nodes in a DOM are structured, it is better to display the internal structure in a GUI, as shown in Figure 5-8:

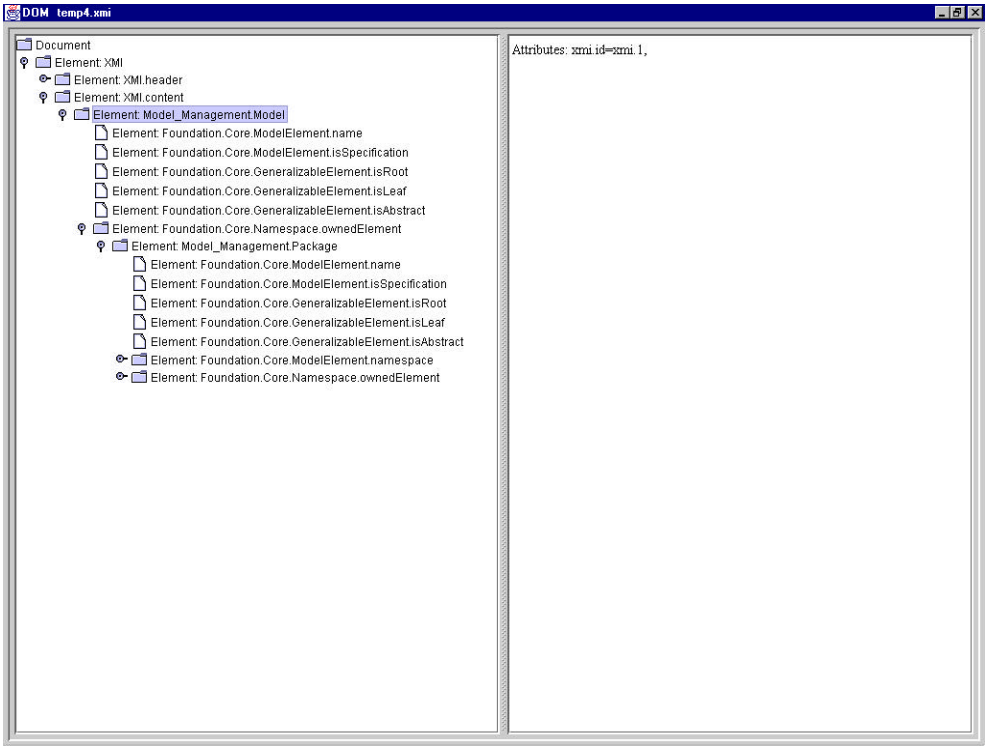

Figure 5-8 Tree View of a DOM

A class `Dom`, as shown below, is created to display a DOM tree. The class `Dom` converts a DOM into a JTreeModel and displays the full DOM in a JTree. It makes sense to stuff the DOM into a JTree, since the DOM is a tree, and the Swing JTree component is all about displaying trees. But a JTree displays a TreeModel and a DOM is not TreeModel. Therefore, an adapter class `DomToTreeModelAdapter` is created to make the DOM looks like a TreeModel to a JTree.

```
public class Dom  extends JPanel
{
  // Global value so it can be ref'd by the tree-adapter
  static Document document;
  …
  // Set up the tree
  JTree tree = new JTree(new DomToTreeModelAdapter());
  …
  public class DomToTreeModelAdapter implements
                    javax.swing.tree.TreeModel {…}

  public class AdapterNode {…}
  …
}
```

The inner class AdapterNode wraps a DOM node and returns the desired string to be displayed in the tree. What the TreeModel gives to the JTree will be in fact be AdapterNode objects that wrap DOM nodes. The class also includes a few additional utility methods.

One of the really nice things about the JTree model is the relative ease with which you convert an existing tree for display. Part of reasons for that is the clear separation between the displayable view, which JTree uses, and the modifiable view, which the application uses. For more on that separation, see [Armstrong+00]. For now, the important point is to satisfy the methods in the TreeModel interface we need and register the appropriate JTree listeners.

5.5.2 Testing Configuration

A new class `ApplicationDemo` is created to wrap the class `Dom` and the class `XMIParser` together in order to display the input and output in the same window. GUI configuration and testing window are shown in Figure 5-9 and Figure 5-10 respectively.

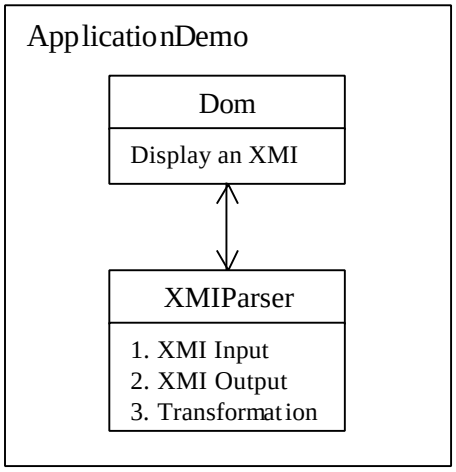

Figure 5-9 GUI Configuration

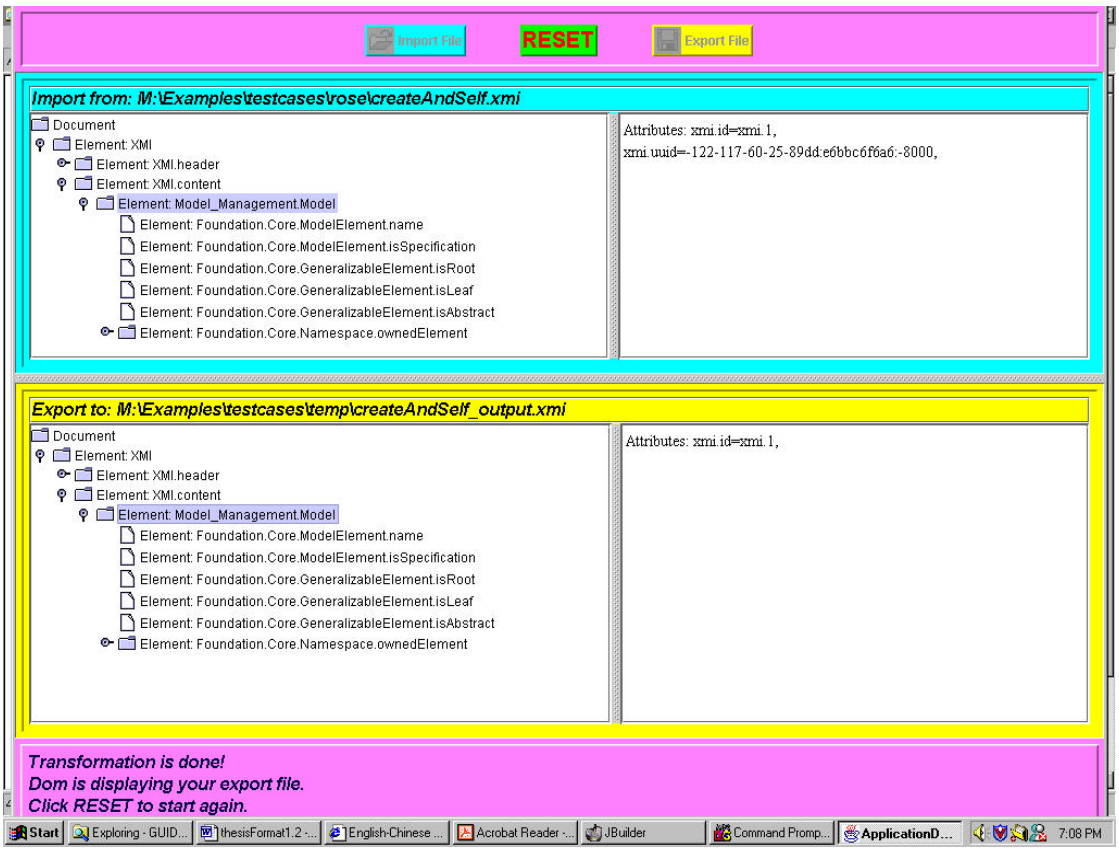

Figure 5-10 Testing Window

The main window is divided into half: the upper half is to display an XMI file from an interaction diagram; the lower half is to display an XMI file from the equivalent activity diagram. To simplify the operation of the application, three JButtons are created: *Import File*, *RESET*, and *Export File*. Functions of `XMIParser` are invoked whenever JButton *Export File* is clicked. This includes XMI input, transformation and XMI output that were described in previous sections.

5.5.3 Results Evaluation

Our testing results must satisfy two criteria: correctness and interoperability between UML tools. All basic cases were tested and inspected for these two criteria. One of the key features of XMI is that XMI eases the problem of tool interoperability by providing a flexible and easy to parse information interchange format. ArgoUML uses XMI as its standard saving mechanism. Rational Rose saves model information in a proprietary format (mdl file), but allows for importing an UML model in XMI format. We did try to use the tools to import our results to test the interoperability. In principle, ArgoUML displays metamodel information in its Navigation Panel, and Rational Rose in its Browser window. However, due to the limitations that we discussed in section 5.4, both tools lose information when importing our XMI files that contained features not yet supported by the tools. Rational Rose cannot display model elements that are not supported yet, e.g. object flow state. ArgoUML is even worse, unsupported model elements in an XMI must be discarded before the file is imported. Despite these problems, both tools were able to read our files and display the elements they understood. It should be mentioned that we could not use the diagrams in the usual UML notation because XMI, by definition, does not contain layout information.

It would be perfect to have both metamodel information and the equivalent diagram for the validation. The transformation we implemented, as mentioned before, is a metamodel-

transformation. Using the DOM to display the internal data structure makes an XMI more readable, which is helpful for checking the correctness of the model.

5.6 Case Study

In this section we show how our transformation algorithm was applied to an electronic-commerce system. The system is distributed as shown in the deployment diagram in Figure 5-11. There are two types of users: remote and local, who are using the system in the same way. However, they will experience quite different response times due to the different communication network delays. This is a three-tier server system: the users interact with a web server, which requests information from a e-commerce server, which in turn sends queries to two databases: a non-secure and a secure database. Each server component runs on its own node.

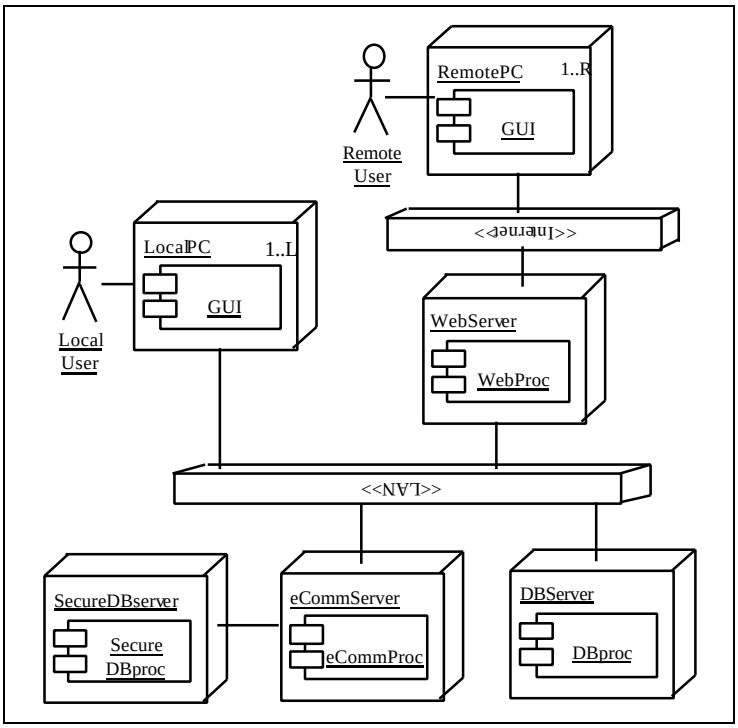

Figure 5-11 Deployment Diagram for E-commerce System

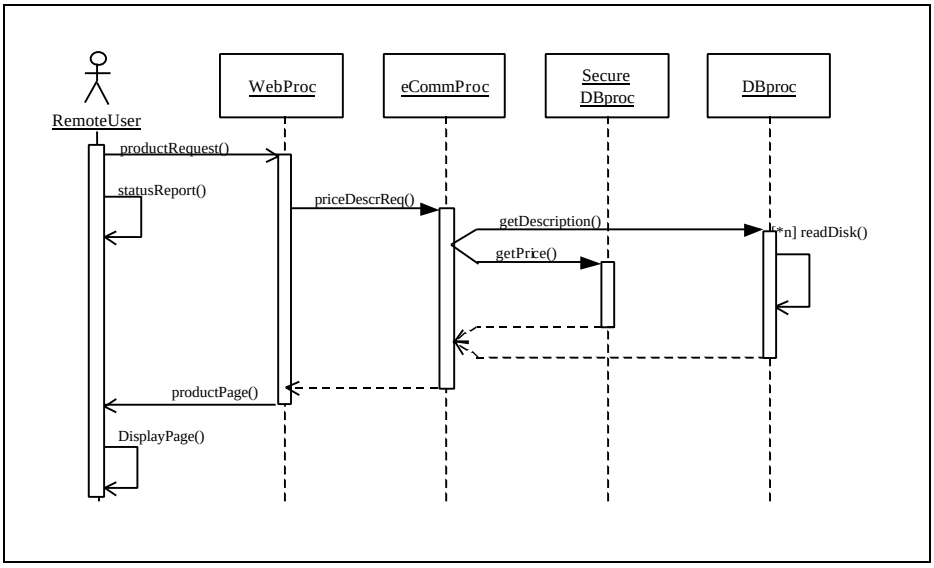

Figure 5-12 Sequence Diagram for “ Get product info.” Use Case

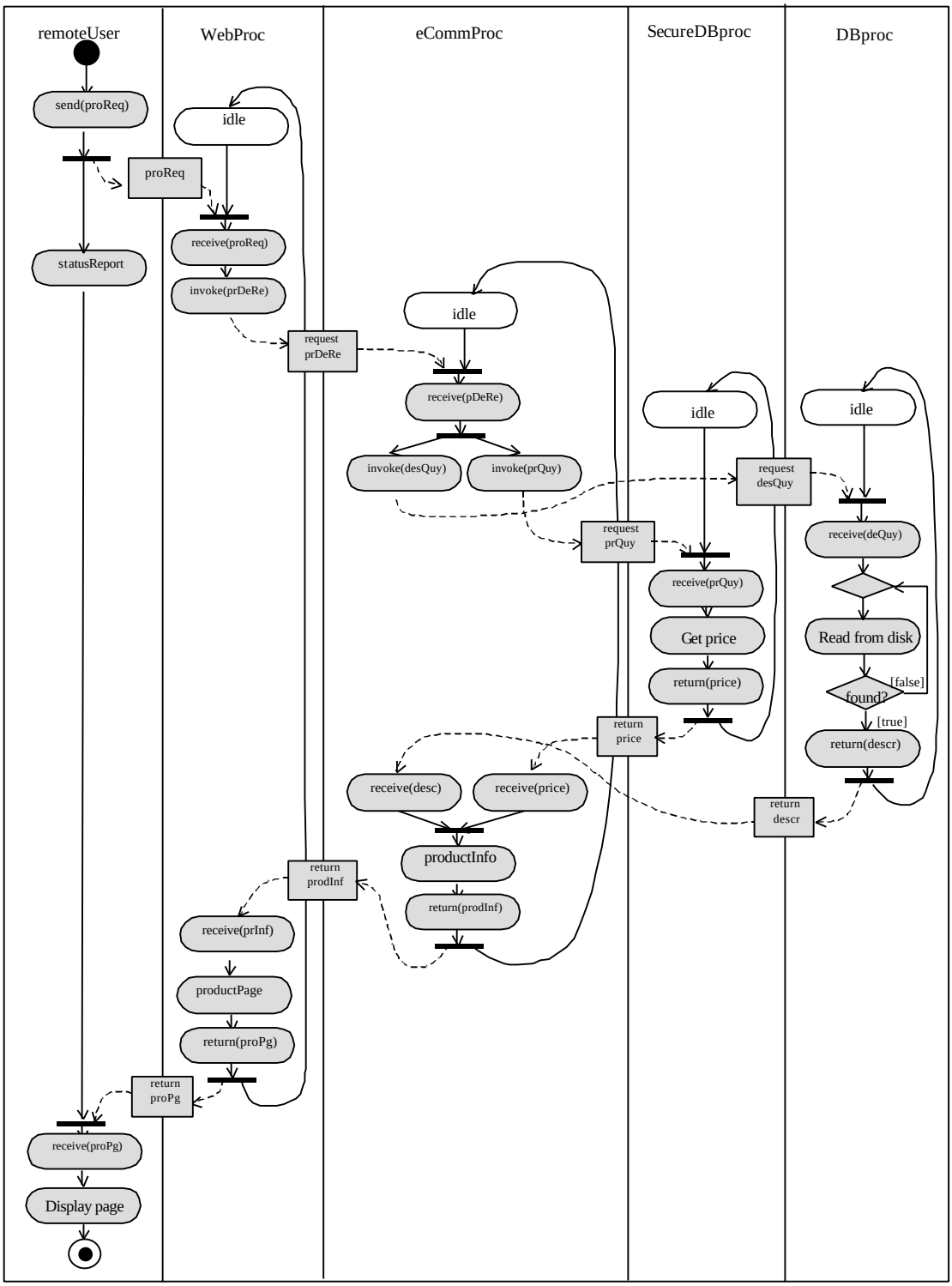
remoteUser
WebProc
eCommProc
SecureDBproc
DBproc
send(proReq)
proReq
statusReport
idle
receive(proReq)
invoke(prDeRe)
request prDeRe
idle
receive(pDeRe)
invoke(desQuy)
invoke(prQuy)
request desQuy
request prQuy
idle
receive(prQuy)
Get price
return(price)
return price
idle
receive(deQuy)
Read from disk
found?
[false]
[true]
return(descr)
return descr
receive(desc)
receive(price)
productInfo
return(prodInf)
return prodInf
receive(prInf)
productPage
return(proPg)
return proPg
receive(proPg)
Display page

Figure 5-13 Activity Diagram for "Get product info" Use Case

Figure 5-12 shows the sequence diagram for the "Get product info" scenario (the realization of the "happy path" of the use case with the same name), whereas Figure 5-13 shows the corresponding activity diagram. A user sends a request to the component WebProc, which gets product information from the eCommProc, then builds a HTML page that is sent back to the user. In turn, the component eCommProc gets in parallel, the product description from DBproc, a non-secure database server, and the price information from SecureDBProc, a secure database server. DBproc performs a sequential file access, hence the iteration for reading from disk until the desired information is found. The secure DB makes an indexed file access, so there is no iteration when reading from disk. Note that in order to show the intra-object behavior on the sequence diagram, one has to add some self-messages (such as those for the user and DBproc).

The example contains a number of client-server relationships, which are realized in two ways. The client-server relationship between the user and WebProc is realized through two separate asynchronous messages, one for the request and the other for the reply. It is impossible to use a synchronous message in this case because the user does not block immediately after sending the request; instead, it goes on to displays a status report to the user, and then it starts waiting for the reply. On the other hand, the client/server relationships between webProc and eCommProc on one side, and between eCommProc and each of the two databases on the other side are realized through synchronous messages.

Note that the component eCommProc has internal concurrency, as shown by the fork/join in both Figure 5-12 and Figure 5-13.

The Activity diagram from Figure 5-13 models the same behaviour as the sequence diagram, but with more explicit model elements that can be annotated with performance information. Some of the details from Figure 5-13 are only implied in Figure 5-12. The activity diagram contains a swimlane for every concurrent component. The asynchronous messages are represented as in Figure 5-6 described in subsection 5.3.2. A synchronous message is represented by two related asynchronous messages, one for the request and another for the reply. The sender of a synchronous message blocks immediately after sending the request, and waits for the expected reply. Before receiving any kind of message, a receiver should be ready for it.

Although there is some redundancy in the proposed activity diagram style, we have chosen it for two reasons: a) to be able to add performance annotations as mentioned before, and b) to create a visual clue that connects the sending and receiving of a message, and facilitates the reading of the diagram. The object flow state attached by two dotted transitions to the sending and receiving states represents the “handing over” of responsibility from one component instance to the next. It is easy to follow the execution flow for the scenario and the actions performed by each instance on its behalf (shown shaded in gray and in Figure 5-13).

The activity diagram shows also the execution thread(s) for each individual component on behalf of the respective scenario. We made the assumption that the instance that initiates the scenario starts at the initial state of the activity diagram and ends at its final state. All the other components are assumed to have a cyclic behavior, waiting in a state named “idle” to receive their first signal that triggers them into action. At the end of the scenario, these components will return to the idle state by default. Note that it is easy to represent a component with internal concurrency, as for example the eCommProc component. By collecting the partial behaviours for different scenarios, one can build the complete state machine for every component; however, this is beyond the scope of the thesis.

5.6.1 Testing Result

The sequence diagram shown in Figure 5-12 was drawn in Rose, which is shown in Figure 5-14. The differences between Figure 5-14 and Figure 5-12 are due to the fact that Rose does not support some features such as an branching or merging. The sequence diagram in XMI format that were obtained from Rose is shown in Figure 5-15. After the transformation was done, the equivalent activity diagram in XMI format is shown in Figure 5-16. The XMI file that represents the activity diagram was then imported back to Rose, which is shown in Figure 5-17. The left-hand-side Browser window in Rose shows the model elements that were described in the XMI file. As mentioned in section Verification, Rose cannot display the model elements that are not supported yet, e.g. object flow states.

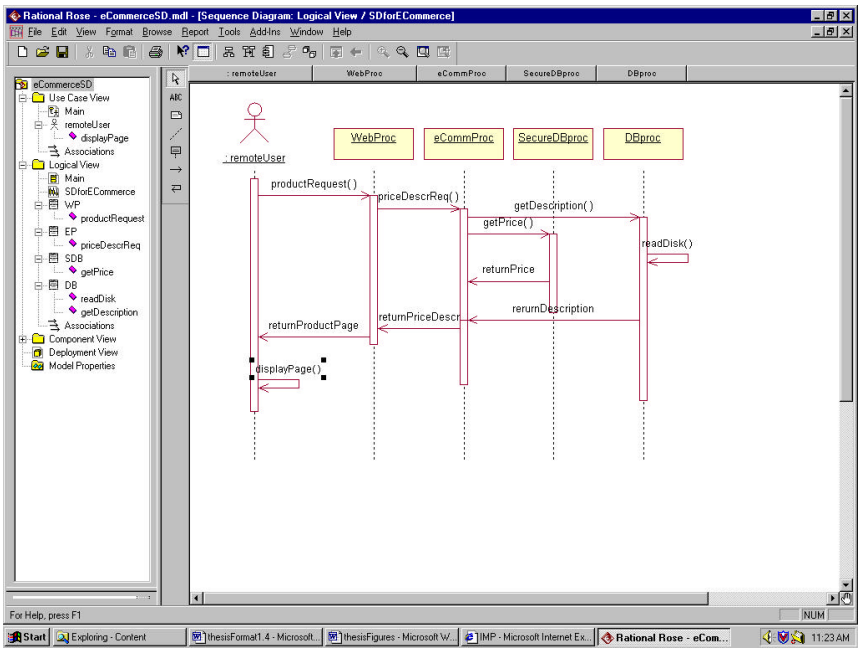

Figure 5-14 SD for "Get product info" use case in Rose

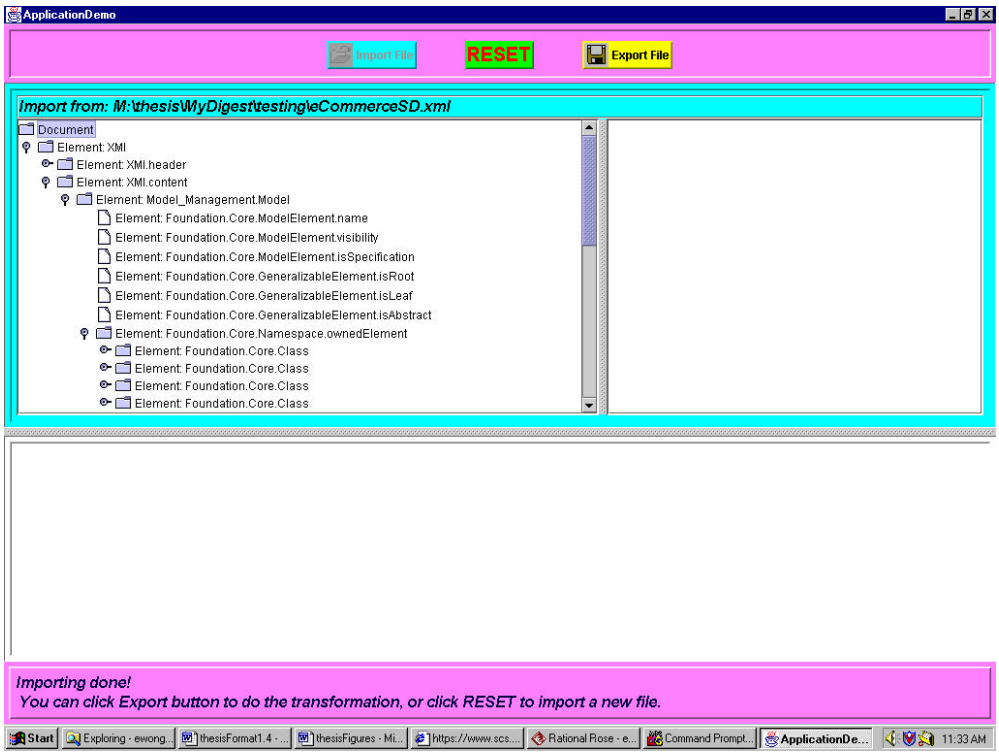

Figure 5-15 SD in XMI Format

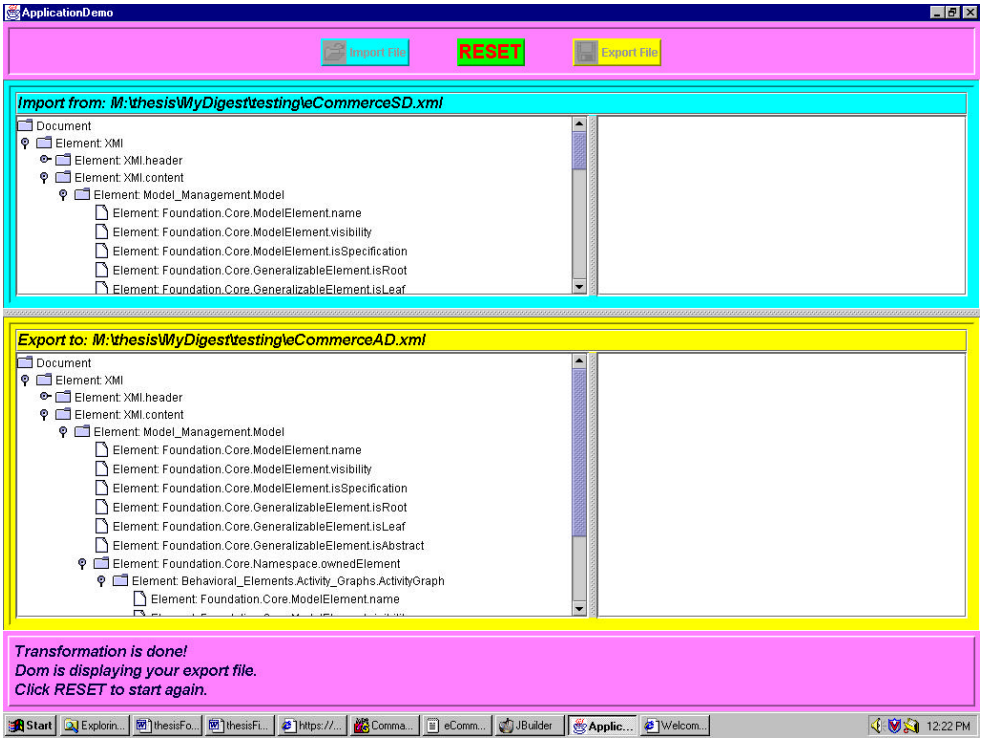

Figure 5-16 Display Both SD and Equivalent AD

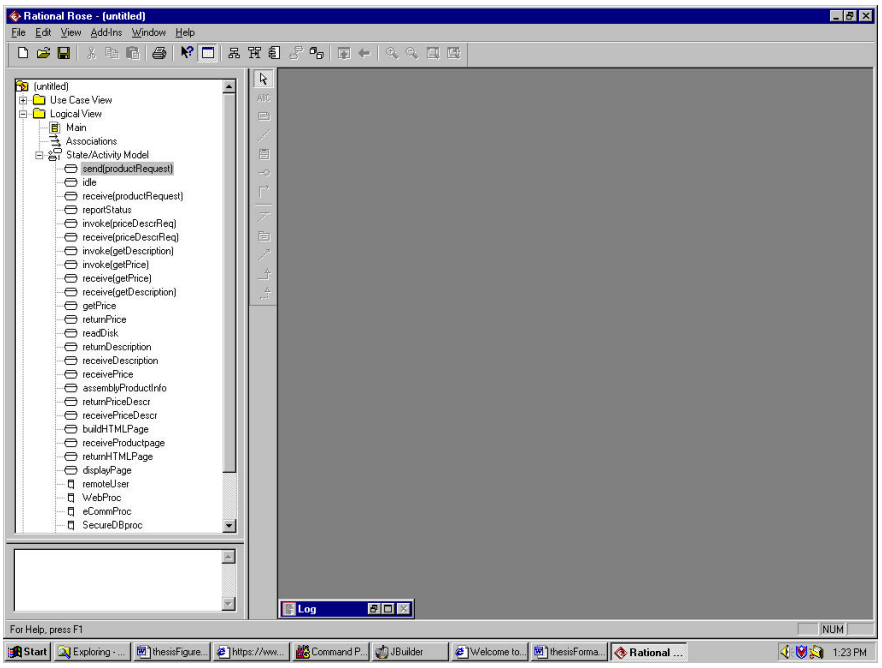

Figure 5-17 Rose Imports AD in XMI format

Chapter 6 Conclusion

6.1 Conclusion

The thesis proposed an approach to automate the XMI-based transformation from UML interaction diagrams to activity diagrams. The thesis introduced transformation rules at notation level and at UML metamodel level. The transformation pays attention to concurrency/distribution and parallelism issues, and captures the following behavioral asptects:

- The execution flow of the actions corresponding to a certain scenario, showing the potential parallelism such as fork/join and branch/merge.
- The concurrent instances (components) responsible for each action and the explicit "hand over" of responsibility between instances represented by the object flow carried by messages.
- The behavior of each concurrent component as it contributes to the respective scenario.
- The explicit sending/receiving actions executed by each concurrent component.

The thesis investigated in detail the UML metamodel representations for interaction and activity diagrams. It identified the participating metaclasses and their relationships in the diagrams. With the aid of a special metamodel library NSUML, the proposed transformation is conducted, in fact, at metaobject level. The metamodel information,

represented in XMI format, will facilitate information exchange and provide product interoperability among development teams in collaborative environments.

The thesis designed and implemented a Java GUI application, which takes an interaction diagram in XMI format that produced from a UML design tool as an input and automatically generates the equivalent activity diagram as the output by applying the transformation rules. The application used two different techniques to process an XMI: SAX API is used to build data structure of a model, and DOM API is used to visualize the internal structure of an XMI. The limitations of the implementation were also discussed.

The thesis made the first known attempt at UML diagrams transformation in terms of XMI. Our work is one step in a larger research project aiming at deriving performance models from UML models and integrating the results of performance analysis back to the UML models. Although the UML standard is still evolving and the XMI standard is evolving with it, we still believe that the conceptual approach proposed in this thesis will be applicable to the future versions.

6.2 Future work

A number of issues need to be addressed in the future work, some of which are currently under way. These issues are closely related to the challenge related to the automatic derivation of performance models from software specification and the integration of the feedback in the UML models:

- As mentioned before, other students are doing work on deriving LQN performance models from scenarios represented by activity diagrams annotated with performance information. Two immediate work items are: a) to introduce Performance Profile stereotypes and tagged values in the ID to AD transformation realized in this thesis, and b) to integrate everything in the UML to LQN transformation.
- The current transformation is actually implemented at metamodel level. It uses a special XMI reader and writer to import and export an XMI. One possible extension is to use XSLT (eXtensible Stylesheet Language Transformation) to directly transform an XMI into another XMI, eliminating the need of the metamodel library NSUML by a set of template rules. Each template rule contains a template and a matching pattern to specify how to transform the input file into an output file.
- An activity diagram generated in XMI format is more abstract and less readable. It would be very useful to display such activity diagrams in a UML tool, so that the user could see the actual UML graphical notation. An impediment to this is the fact that, by definition, XMI does not contain layout information.

References

[Abiteboul+00] Abiteboul, S., Buneman, P. & Suciu, D., *Data on the Web: From Relations to Semistructured Data and XML*, Morgan Kaufmann Publishers, San Francisco, California, 2000.

[Amer01] Amer, H., "*Automatic transformation of UML software specification into LQN performance models using graph grammar techniques*", Ottawa. Thesis (M.Eng.), Carleton University, 2001.

[ArgoUML] ArgoUML: An Open-Source UML-Tool, See http://argouml.tigris.org/.

[Armstrong01] Armstrong, E., "Working with XML: The Java API for XML Parsing (JAXP) Tutorial", See http://java.sun.com/xml/jaxp-1.1/docs/tutorial/.

[Armstrong+00] Armstrong, E., Santos, T. & Wilson S., "Understanding the TreeModel", See http://java.sun.com/products/jfc/tsc/articles/jtree/index.html.

[Balsamo01] Balsamo, S., & Simeoni, M., "Deriving Performance Models from Software Architecture Specifications", See http://www.dsi.unive.it/~balsamo/saladin/bal-sim.2.01.pdf

[Bradley00] Bradley, N., *The XML Companion*, 2nd Edition, Mass., Addison-Wesley, 2000.

[Booch94] Booch, G., *Object-Oriented Analysis and Design with Applications (2nd ed.)*, Benjamin/Cummings, Redwood City, 1994.

[Booch+99] Booch, G., Rumbaugh, J. & Jacobson, I., *The Unified modeling language user guide*, Reading, Mass., Addison-Wesley. 1999.

[Cook+94] Cook, S. & Daniels, J., *Designing Object Systems: Object-Oriented Modeling with Syntropy*, Prentice-Hall, Hemel Hempstead, 1994

[Cortellessa+00] Cortellessa, V. & Mirandola, R., "Deriving a Queueing Network based Performance Model from UML Diagrams", *Proceedings of the Second InternationalWorkshop on Software and Performanc*e, Ottawa, Canada, pp 58-70, Sept. 2000.

[DOM] Document Object Model, See http://www.w3c.org/DOM.

[Douglass00] Douglass, B.P., *Real-time UML; developing efficient objects for embedded systems*, 2nd ed. Reading, Mass., Addison-Wesley. 2000.

[DuCharme99] DuCharme, B., *XML:the annotated specification*, Upper Saddle River, N.J., Prentice Hall. 1999.

[Evans+99] Evans, A. & Kent, S., Core Meta-Modelling Semantics of UML: The pUML Approach, In Robert France and Bernhard Rumpe, editors, *Proceedings of the Unified Modeling Language: UML'99: Beyond the Standard, Lecture Notes in Computer Science 1723*. Springer-Verlag, 1999.

[Fowler97] Fowler, M., *UML Distilled: Applying the Standard Object Modeling Language*, Mass., Addison-Wesley, 1997.

[Gerard+01] Gerard, S. & Ober, I., "Parallelism/Concurrency specifications with UML", White Paper for the Workshop on Concurrency Issues in UML, International Conferences <<UML>> 2001, Toronto, Canada, Oct, 2001 See http://wooddes.intranet.gr/uml2001/WhitePaper /WhitePaperOnParallelism.pdf, October 2001.

[Harel87] Harel, D., Statecharts: A Visual Formalism for Complex Systems, in *Science of Computer Programming*, Vol. 8, 1997.

[Hess00] Hess, D. A., "Rational Rose Enterprise Edition", Tech update product info, See http://techupdate.cnet.com/enterprise/0-6119586-723-3121217.html

[Horstmann+99a] Horstmann, C.S. & Cornell, G., *Core Java 2 Volume II – Advanced Features*, UpperSaddle River, N.J., Sun Microsystems Press. 1999.

[Horstmann+99b] Horstmann, C.S. & Cornell, G., *Core Java 2 Volume II – Fundamentals*, UpperSaddle River, N.J., Sun Microsystems Press. 1999.

[IDL99] CORBA IDL definition, See http://cgi.omg.org/cgi-bin/doc?formal/99-10-01.pdf, 1999.

[ITUT00] International Telecommunication Union (ITU-T Z.120), Message Sequence Chart (MSC), 2000.

[JAXP 1.1] An XML API in Java, See http://java.sun.com/xml/jaxp/dist/1.1/docs/api/index.html.

[Kahkipuro99] Kahkipruo, P., UML Performance Modeling Framework for Object-Oriented Distributed Systems, In Robert France and Bernhard Rumpe, editors, *Proceedings of the Unified Modeling Language: UML'99: Beyond the Standard, Lecture Notes in Computer Science 1723*. Springer-Verlag, 1999.

[King+99] King, P., & Pooley, R., "Derivation of Petri Net Performance Models from UML Specifications of Communication Software", Proc. of XV UK Performance Engineering Workshop, 1999.

[Leventhal+98] Leventhal, M., Lewis, D. & Fuchs, M., *Designing XML Internet applications.* Upper Saddle River, N.J., Prentice Hall PTR, 1998.

[Marshall00] Marshall, C., *Enterprise modeling with UML; designing successfull software through business analysis*, Reading, Mass., Addison-Wesley. 2000.

[Martin99] Martin, T.A., *Project cool guide to XML for Web designers*. New York, John Wiley, 1999.

[Maruyama99] Maruyama, H., *XML and Java; developing Web applications*. Reading, Mass., Addison Wesley, 1999.

[MOF1.3] OMG: Meta Object Facility (MOF) Specification, Version 1.3, See http://www.omg.org/cgi-bin/doc?formal/00-04-03, April, 2000.

[NSUML99] Novosoft UML API, See http://www.novosoft-us.com/.

[Jacobson00] Jacobson, I., *The road to the unified software development process*. Rev. and updated by Stefan Bylund. Cambridge, UK, Cambridge University Press, 2000.

[Jacobson98] Jacobson, I., Booch, G. & Rumbaugh, J., *The Unified Software Development Process*, MA: Addison Wesley Longman Inc., 1998.

[Oestereich99] Oestereich, B., *Developing software with UML; object-oriented analysis and design in practice*, Harlow, England, Addison-Wesley, 1999.

[Overgaard99] Overgaard, G., A Formal Approach to Collaborations in the Unified Modeling Language, In Robert France and Bernhard Rumpe, editors, *Proceedings of the Unified Modeling Language: UML'99: Beyond the Standard, Lecture Notes in Computer Science 1723*. Springer-Verlag, 1999.

[Petriu+01a] Petriu, D.C., & Shen, H., "*Applying the UML Performance Profile: Graph-Grammar-based Derivation of LQN Models from UML Specifications*", submitted to the 12th International Conference on Modeling Tools and Techniques for Computer and Communication Systems Performance Evaluation Tools'2002, to be held in London, April 2002.

[Petriu+01b] Petriu, D.C., & Wong, E., "*Using Activity Diagram for Representing Concurrent Behavior*", White Paper for the Workshop on Concurrency

Issues in UML, International Conferences <<UML>> 2001, to be held in Toronto, Canada, See http://wooddes.intranet.gr/uml2001/WhitePaper/WhitePaperOnParallelism.pdf, October 2001.

[Petriu+00a] Petriu, D.C. & Sun, Y., "*Consistent Behavior Representation in Activity and Sequence Diagrams*", to appear in Proc. of UML'2000, York, GB, Oct 2000.

[Petriu+00b] Petriu, D.C. & Wang, X., "From UML Description of High-level Software Architecture to LQN Performance Models", In Nagl, M., Schurr, A., Munch, M. (eds): Applications of Graph Transformation with Industrial Relevance, AGTIVE'99, Lecture Nodes in Computer Science, Vol. 1779, p.47-62, Springer, 2000.

[Petriu+98] Petriu, D.C. & Wang, X., "*Deriving Software Performance Models from Architectural Pattern by Graph Transformations*", Proceedings of the Sixth International Workshop on Theory and Application of Graph Transformations TAGT'98, Paderborn, Germany, Nov. 1998.

[Pooley99] Pooley, R., "Using UML to Derive Stochastic Process Algebra Models" Proc. of XV UK Performance Engineering Workshop, 1999.

[Profile01] OMG: "Response to the OMG RFP for Schedulability, Performance, and Time", OMG document number ad/2001-06-14, http://www.omg.org/cgi-bin/doc?ad/2001-06-14, June 2001.

[Quatrani98] Quatrani, T., *Visual modeling with rational rose and UML*, Reading Mass., Addison-Wesley. 1998.

[Rose] Rational Software Cooperation, See http://www.rational.com/products/rose/index.jsp.

[Reenskaug96] Reenskaug, T., *Working with Objects*, Manning, Greenwich, 1996.

[Rumbaugh+91] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., & Lorensen, W., *Object-Oriented Modeling and Design.* Prentice-Hall, Englewood Cliffs. 1991.

[Rumbaugh+99] Rumbaugh, J., Jacobson, I. & Booch, G., *The Unified modeling language reference manual*, Reading Mass., Addison-Wesley. 1999.

[Schurr90] Schurr, A., "Introduction to PROGRES, an Attributed Graph Grammar-based Specification Language", In: Nagl, M., (ed): Graph-Theoretic Concepts in Computer Science, Lecture Notes in Computer Science, Vol. 411, pp. 151-165, 1990.

[Smith90] Smith, C.U., "*Performance Engineering of Software Systems*", Reading Mass., Addison Wesley, 1990.

[Smith+97] Smith, C.U. & Williams, L.G., "Performance Engineering Evaluation of OO Systems with SPE.ED", in Marie, R., et al. (eds), *Computer Performance Evaluation – Modeling Techniques and Tools*, Springer LNCS, 1997.

[UML1.3] OMG: *Unified Modeling Language Specification*, version 1.3, See http://www.omg.org/cgi-bin/doc?formal/00-03-02, March 2000.

[UML1.4] OMG: Unified Modeling Language Specification, version 1.4, See http://www.omg.org/cgi-bin/doc?formal/01-09-67, Sep 2001.

[Wang99] Wang, X., *Deriving Software Performance Models From Architectural Patterns By Graph Transformation*, M.Eng. thesis, Department of Systems and Computer Engineering, Carleton University, 1999.

[Warmer+99] Warmer, J.B. & Kleppe, A.G., *The Object Constraint Language: Precise Modeling with UML*, Addison-Wesley, Reading, Mass. 1999.

[Williams+98] Williams, L.G., & Smith, C.U., "Performance Evaluation of Software Architectures" in Proc. of WOSP'98, Santa Fe, New Mexico, USA, 1998

[Woodside+95] Woodside, C.M., Neilson, J.E., Petriu, D.C., & Majumdar, S., "The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-Like Distributed Software", IEEE Transactions on Computers, Vol. 44, Nb. 1, January 1995.

[W3C] World Wide Web Consortium, See http://www.w3c.org.

[XMI1.0] OMG: *XML Metadata Interchange specification*, version 1.0, See http://cgi.omg.org/cgi-bin/doc?formal/00-06-01, 2000.

[XMI1.1] OMG: *XML Metadata Interchange specification*, version 1.1, See http://www.omg.org/cgi-bin/doc?formal/2000-11-02, 2000.

[XML1.0] eXtensible Markup Language (XML) version 1.0, Tim Bray, et al, W3C, 10 Feb 1998, See http://www.w3c.org/TR/REC-xml.