6-2005

A Tool Supporting Validation of UML Models

Weng Liong Low

A TOOL SUPPORTING VALIDATION OF UML MODELS

by

Weng Liong Low

A Thesis
Submitted to the
Faculty of The Graduate College
in partial fulfillment of the
requirements for the
Degree of Master of Science
Department of Computer Science

Western Michigan University
Kalamazoo, Michigan
June 2005

ACKNOWLEDGMENTS

First I would like to thank my graduate advisor and thesis committee chair Dr Wuwei Shen for his strong support and help throughout the duration of this thesis. His enthusiasm for and support of software engineering, in particular tools that support software engineering, has led me to pursue the subject and therefore resulting in the work contained in this thesis. His constant support has helped me through difficult parts in the thesis-writing process and I truly appreciate that support.

Secondly, I would like to thank the other members of my thesis committee, Dr Dionysios Kountanis and Dr Li Yang, for being on the committee and spending the effort and time to review my work.

Lastly, I would like to thank both my parents and my sister for giving me moral and financial support halfway around the world back home.

Weng Liong Low

A TOOL SUPPORTING VALIDATION OF UML MODELS

Weng Liong Low, M.S.

Western Michigan University, 2005

Software design is an important phase in the computer software development life cycle. The Unified Modeling Language (UML) is a widely used notation to design software models. As a software system increases in size and complexity it is harder for developers to check that their UML models are correct with respect to the UML metamodel. Consistency among different diagrams in a UML model is also important to ensure that there are no design conflicts that will lead to problems later. Incorrect software design will result in loss of productivity, time and money as developers fix design errors. Therefore, to ensure consistency in a software design we propose a tool that will help developers validate UML models in a number of ways.

The tool is developed based on the concept of abstract state machines (ASMs), which have been used to perform verification of UML models. By translating UML models including OCL constraints into AsmL, an executable specification language based on ASMs, the tool is able to perform model instantiation checking, i.e., checking whether a diagram at one level of the metamodeling architecture is a valid instance of another diagram at a higher-level. The tool can also verify message ordering between sequence diagrams and state chart diagrams. The tool has been applied to various specific areas such as class diagram refinement and design pattern profile checking. With this tool software developers will be able to reduce the number of errors in their software designs.

TABLE OF CONTENTS

Table of Contents—continued

CHAPTER

Table of Contents—continued

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Software design is an important phase in the computer software development life cycle. It is especially important in the production of large and complex software. The Unified Modeling Language (UML) is a popular language for designing software programs. The UML specifies notations and constructs for software design and improvements to itself are constantly being made by researchers in the academic community as well as in the software industry.

The UML is based on a four-level modeling architecture developed by the Object Management Group (OMG) and has many different diagrams with each diagram allowing a software developer to specify different aspects of a software system. As a software system increases in size and complexity, it becomes increasingly difficult for software developers to check that the UML model of the software system is correct with respect to the UML metamodel. Furthermore, the task of verifying that the different diagrams in the UML model for a particular software are consistent with each other becomes difficult as well. Consistency among UML diagrams in a single UML model is important in order to avoid design conflicts that will lead to problems with implementation of the software. Incorrect software design will result in loss of productivity, time, and money as software developers go back to their design to find the problems, fix them, and change their code to meet the requirements of the corrected design. Therefore the importance of ensuring that a design is consistent and as error-free as possible during the design phase cannot be understated. Various tools have been written to assist software developers with this

problem and to reduce the amount of manual consistency checking that the developer must perform on the UML model in the hopes of decreasing the cost of software design.

With this in mind, we propose a software tool that validates UML models by performing a number of consistency checks between: (1) a UML model and the UML metamodel, and (2) different diagrams within a UML model. These checks are performed using two methods:

1. The UML model is represented using abstract state machines (ASMs) so that model constraints can be validated.
2. Specific algorithms written into the tool validate other aspects of the UML model that are not so easily validated using the first method.

The concept of abstract state machines (ASMs) [1] was first presented by Dr Yuri Gurevich more than ten years ago. An ASM is a state machine that computes a set of updates of its own variables by firing all possible updates based on the current state. The computation of a set of updates occurs at the same time and results in the generation of a new state. ASMs can be formally defined and can be used to define precise models of software. ASMs have been applied to UML in a variety of ways. Börger et al. have applied ASMs to provide semantics for UML activity diagrams and state machines [3, 4]. Cavarra et al have integrated UML static and dynamic views based on ASMs [5]. Ober has proposed translating UML class diagrams into ASMs by defining action semantics as well as an XML Metadata Interchange (XMI) to ASM translator with manually input Object Constraint Language (OCL) constraints [6]. Shen presented a static validation method for a UML model based on the XASM tool ([2]) [7]. Based on these previous works, we believe that abstract state machines can be used to support the validation of UML models.

AsmL [8] is an executable specification language based on the concept of abstract state machines that is developed by the Foundations of Software Engineering (FSE) group at Microsoft Research. It is a high-level specification language running on Microsoft's .NET framework and has language constructs such as sets and sequences and high-level operations that let the programmer specify what the program should do but not how it should be done. There are other languages based on abstract state machines, but we choose AsmL because it uses object-oriented programming, component-oriented programming, and functional programming. These features make AsmL easier to use because it is similar to other popular languages such as C++ or Java, and its object-oriented paradigm matches the object-oriented model employed by UML in its class diagrams so the translation from a UML model into an AsmL specification is simplified.

Architecture of the Tool

Figure 1 shows the architecture of our model validation (or checking) tool. It consists of four major modules: (1) a UML specification diagram parser, (2) a UML instance diagram parser, (3) an OCL parser, and (4) a library of OCL operations written in AsmL.

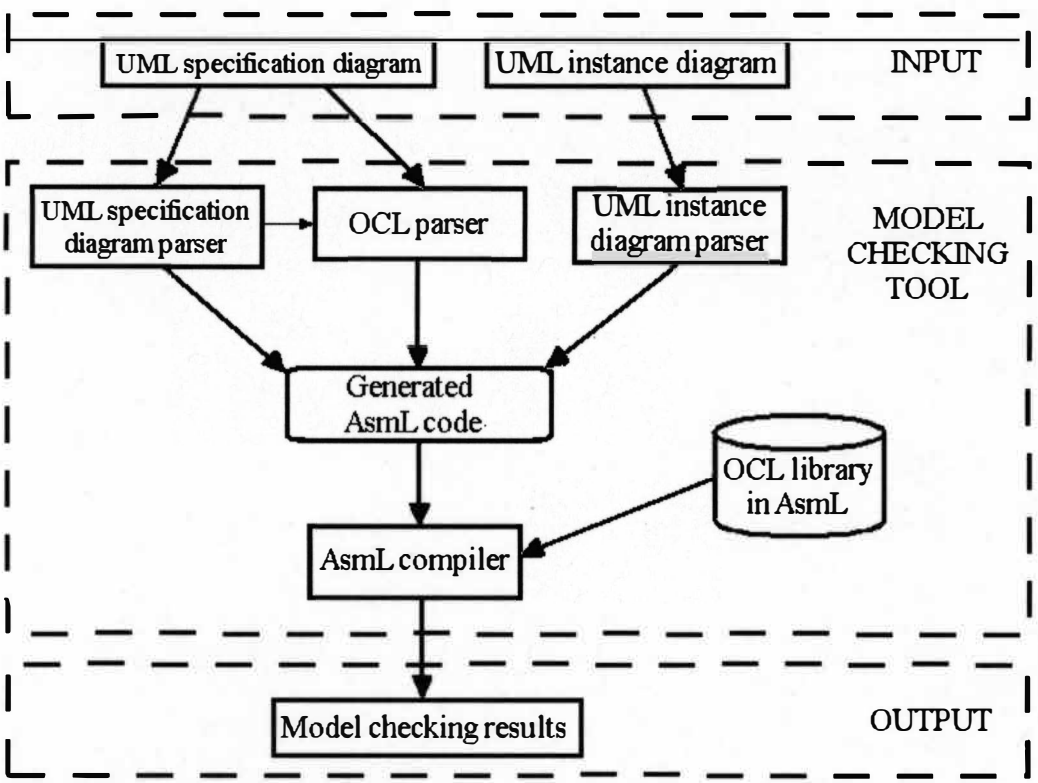

Figure 1. The UML model validation tool and its modules.

The three parser modules read a UML model given in an Extensible Markup Language (XML) file that is written in the XML Metadata Interchange (XMI) format. XMI [9] is a widely used interchange format for sharing objects using XML and is developed by the OMG. The XML file containing the UML model should be valid based on a Data Type Definition (DTD) for UML 1.3, which was released by the OMG. Once read in, each parser module will extract portions of the model that it is responsible for and converts that part into an AsmL specification. The combined AsmL specification from each of the three modules will then be passed to the external AsmL compiler - part of the AsmL installer package released by Microsoft's FSE group - together with the fourth module, which is the OCL library written in AsmL. These items are compiled and the resulting executable file can be run to perform

constraint checking on the UML model. The compiled AsmL program will display model checking results to the user. The model checking tool also performs a few other consistency checks after reading in the UML model but prior to generating the AsmL specification.

We divide the UML model into two parts: the specification diagram and the instance diagram. The specification diagram is the diagram or set of diagrams that specify the base model that the instance diagram is based upon. The instance diagram is the diagram or set of diagrams that instantiates the specification diagram. Instance diagrams are most commonly object diagrams, but in the case of checking a UML diagram against the UML metamodel, the instance diagram could be a sequence or state chart diagram which will have to be converted into an object diagram as an instance of the metamodel. This concept is the foundation for model instantiation checking performed by the tool, which is described later on in this work. With this division into specification and instance diagrams, we can check whether a UML model created by the software developer (designated as the instance diagram) conforms to the UML specification (designated as the specification diagram). Due to the four-level meta-modeling architecture of UML, we can also check if an object diagram (designated as the instance diagram in this case) in a UML model is a valid instance of a class diagram within the same model (designated as the specification diagram).

In order to make the separation between the specification and instance diagrams clear, the input UML model has two top-level packages - one for the specification diagram and one for the instance diagram. Packages are defined as containers in UML that partition UML model elements into separate logical groups. We make use of this construct to place all model elements for the specification

diagram and instance diagram into their respective packages and define a notation to indicate which package contains the specification or the instance diagram. In Figure 2 we see two top-level packages in a UML model. We use a dependency relation between the packages to denote the specification and the instance package. In this example 'Profile' contains the specification diagram and 'Instance' contains the instance diagram. In the rest of this work we will assume that all UML models given as input partition their diagrams and model elements this way.

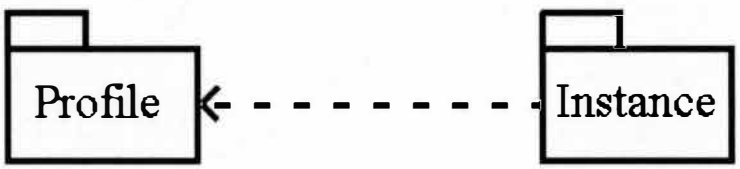

Figure 2. Top-level packages containing specification and instance diagrams.

Uses of the Tool

The major use of the tool is to perform model instantiation checking, which is to check that a UML instance diagram is a valid instance of a UML specification diagram. Besides this, the tool can also be applied to domain-specific model checking. For instance, it can be used in the validation of class diagram refinement. It can also be used to verify UML diagrams that have been generated by a third-party tool following a design pattern. These uses and applications of the tool will be described later in this thesis.

Development Environment

The development environment for the tool is limited to the Windows platform

mainly because the AsmL compiler uses the .NET framework that is currently only available on Microsoft Windows operating systems. The implementation language used is C++ because it supports object-oriented programming, which is required to implement data structures for the tool (described later in this paper), and because of the author's familiarity with the language. The compiler used is the Windows port of the popular open-source GNU compiler under the MinGW package [15]. This compiler is used instead of the more common Visual C++ suite used for Windows application development because it is free and portable, although with the use of AsmL portability is less of an issue because the tool is already limited to running on Windows platforms. However, the fact that the MinGW package is free means that anyone else can build this tool from its source code if she has the software, platform requirements, and the free compiler without paying for a commercial compiler suite. We strive to use libraries and tools that are free or open-source for this project so that more people can use this work without high costs and complex licensing issues.

This thesis will begin by spending the next few chapters discussing each of the modules of the tool in detail. This will be followed by a chapter on model instantiation checking as well as another one on message ordering verification, both of which are functions the tool can perform. Next some applications of the tool to refinement checking and design patterns are given. Finally we present limitations of the tool, future work and some conclusions.

CHAPTER II

SPECIFICATION DIAGRAM PARSER

The specification diagram parser module is responsible for reading in a specification diagram from the UML model provided in the XML file given as input to the tool. Usually the specification diagram will either be a UML class diagram that defines the static view of a software model or it could be the UML metamodel. The latter is used to validate a user-defined UML model with respect to a particular version of the UML specification released by the OMG. The specification diagram can also be a set of diagrams, including class diagrams, collaboration or sequence diagrams, and state chart diagrams, but in this case there is usually no corresponding instance diagram, since it does not make sense. In this case we usually perform validation on the individual diagrams within the specification diagram, such as checking for message ordering between sequence and state chart diagrams. The detailed discussion of this check is given in a later chapter. Figure 3 shows the functionality of the specification diagram parser module.

As mentioned in the introductory chapter, the input to the specification diagram parser is an XML file using the XMI format and the DTD for UML models. Because the format is a standard, our tool can read UML models in this format that are generated from other UML modeling software. An example of such software is Rational Rose by IBM. For our purposes we use XML files generated from UML models created by Rational Rose Enterprise Edition installed on the campus computers, but technically any valid XML files generated from other UML computer-aided software engineering (CASE) tools should work.

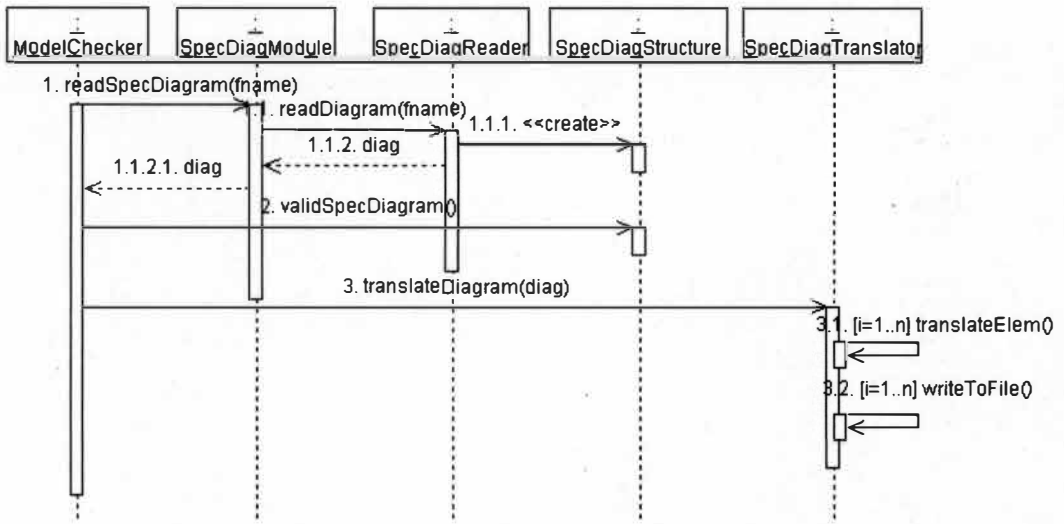

Figure 3. High-level view of translating a specification diagram.

Data Structure Design

Before going further into the discussion of the specification diagram parser, we will describe the data structure used to represent the UML model within the tool. This data structure is important because it is used by other modules of the tool in order to perform validation tests on the tool. The design of the data structure has gone through a number of revisions over the course of this work. At this point we note that the discussion on the data structure focuses mostly on UML class diagram elements, but can be applied to other diagrams and model elements that a UML model can have.

At the beginning the goal was to design a data structure generic enough to represent any UML class diagram regardless of the UML version. The problem that software developers face is the selection of a particular version of UML upon which their design is based. For example, most commercial software including Rational Rose use UML 1.3, while some tools support UML 2.0 (which is still a draft version). For our tool to receive widespread use, support of multiple UML

metamodels for different versions of UML would be good. However, due to the fact that new model elements can be introduced into later versions of UML, it is quite difficult to generate a minimal set of data structures and core classes that can fit all possible UML metamodels that can be developed in the future. One way of solving this problem is by designing and using separate data structures, one for each significant change in the UML version, but this solution is beyond the scope of this work.

As a result, we settle upon a data structure that can represent at least a UML 1.3 model, because it is the most widely supported UML version in commercial UML CASE tools including Rational Rose. The data structure also supports UML 1.5 diagrams because the changes between the two versions are not significant with the exception of the action semantics, which is not dealt with in this work. The data structure consists of core classes from the UML 1.5 metamodel [10] that are frequently used in UML diagrams, such as *Classifier, Generalization, Association, Attribute, Operation,* and so on. The data structure used in the tool preserves the complex class hierarchy and class relationships present in the UML metamodel and because the implementation language for the tool is C++, features like multiple inheritance in the UML metamodel can easily be implemented. For the purposes of this work, the data structure designed for the tool can support UML class diagrams, sequence or collaboration diagrams, state chart diagrams, and object diagrams.

Reading the UML Model

The specification diagram parser is actually an XML parser because the input file is an XML file. There are two common ways of parsing an XML file: (1) build a Document Object Model (DOM) [11] tree from the nested XML elements in the file,

and (2) use Simple API for XML (SAX) [12] to read XML elements sequentially from the file. We chose the latter method for our XML parser. The SAX interface was chosen over building a DOM tree because:

1. Building a DOM tree requires the entire XML file be read to build the tree. We are, however, ignoring most of the tags except what is needed for our purposes and therefore wasting a lot of time and memory. With SAX, however, we can choose to process only the relevant XML tags.
2. A DOM tree is appropriate for an application that makes changes to the elements within the tree, for example an editor that saves information into an XML file. However, our tool does not make changes to the XML file.
3. A DOM tree is not suitable for our purposes because we use a different representation for the information contained in the XML file, namely the data structure to store model elements from the UML 1.5 metamodel. We would be wasting memory and time copying from the DOM tree into our own data structure. On the other hand, with SAX the parser can read information from the XML tags directly into the data structures used by the tool.

One of the disadvantages of using SAX is that a lot more code needs to be written for the parser, especially for storing the state of the parser so that we know what tags to expect next after one has been read in before. SAX also uses callback functions which makes the design and implementation of the parser different than a normal file reader that reads a file linearly.

Building an XML parser from scratch is a lot of work, and why invent the wheel when there are good XML parser libraries out there? Therefore we choose to use the *libxml2* library, which is the "XML C parser and toolkit developed for the

Gnome project" [13]. There are other good XML parser libraries out there, both commercial and free, but we choose *libxml2* because it is open-source, very portable, implements a good number of standards related to XML, and its implementation passes a majority of compliance tests from the OASIS XML Tests Suite [14]. The Xerces-C XML parser [16] is also portable and has a large number of implemented standards, but an arbitrary choice was made in favor of *libxml2*. A small argument against Xerces-C is that it is licensed under version 2.0 of the Apache Software License, which is more restrictive and incompatible with the GNU General Public License for most open-source software.

The *libxml2* library provides both methods of reading XML files as described above, but we only use the SAX interface it provides. We use a Win32 binary version of the library provided on its web site in the form of dynamic link libraries (DLLs). The XML parser for the specification diagram parser and instance diagram parser modules are the same. A base class containing the XML parser routines to read the various XML tags from the input file is created. Ideally, the two parser modules should each have its own parser class - inheriting from this common base class - containing specific handling routines to process the XML tags that each module is responsible for. In practice, however, the specification diagram parser module has its own derived parser class but the instance diagram parser module does not. This is due to the fact that the instance diagram is represented as an object diagram in the internal data structure of the tool even if it originally was a class diagram. For example, if we are comparing a user-defined class diagram with the UML metamodel, the user-defined class diagram is represented as an object diagram, which is an instance of the UML metamodel's class diagram. Since the user-defined class diagram is considered a specification diagram and is handled by the specification diagram parser, we use this

parser to read in that diagram and have internal operations to convert this into an object diagram. More information on this process will be provided later.

Each model element in the XML file has an XMI identifier that is unique to a given XML file and can change when the file is regenerated by the modeling software like Rational Rose. It is used in some attributes of the UML model elements as a way of representing the associations between model elements in the UML metamodel. For example, a UML generalization has a child class and a parent class. The XML tag representing a generalization within a UML model will include an attribute that indicates the UML class that is the child class and another one that indicates the parent class. Each of these attributes will have a character string value - the XMI identifier - that indicates the child and parent class respectively. This means that we have to resolve these XMI identifiers to point to the actual model elements within the data structure representing the UML model. This is done after all the relevant model elements have been read into the data structures. Then for each model element in the UML model that contains attributes with XMI identifiers to be resolved, we search the data structure for the model element that has the particular XMI identifier. After that we link it to the originating model element by assigning it to a variable in the originating model element. In the example in Figure 4, the Generalization class in the data structure will have two string variables to store the XMI identifier for the parent class and child class respectively. The Generalization class will also contain two variables of type GeneralizableElement, which is one the parent classes of Classifier as specified by the UML metamodel. Once the XMI identifiers have been resolved, each of these two variables will point to the Classifier representing the parent class or child class respectively.

```
<UML:Class xmi.id = 'S.103.1714.33.92'
  name = 'A' visibility = 'public' isSpecification = 'false'
  isRoot = 'true' isLeaf = 'false' isAbstract = 'false'
  isActive = 'false'
  namespace = 'S.103.1714.33.91'
  specialization = 'G.90' />
<UML:Class xmi.id = 'S.103.1714.33.94'
  name = 'C' visibility = 'public' isSpecification = 'false'
  isRoot = 'false' isLeaf = 'true' isAbstract = 'false'
  isActive = 'false'
  namespace = 'S.103.1714.33.91'
  generalization = 'G.90' >
  <UML:Namespace.ownedElement>
    <UML:Generalization xmi.id = 'G.90'
      name = '' visibility = 'public' isSpecification = 'false'
      discriminator = ''
      child = 'S.103.1714.33.94' parent = 'S.103.1714.33.92' />
  </UML:Namespace.ownedElement>
</UML:Class>
```

Figure 4. Use of XMI IDs that are unique throughout a UML model.

AsmL Translation

Once the specification diagram has been read into the internal data structure and XMI identifiers have been resolved, we can now translate the specification diagram, which is usually a UML class diagram, into the target language AsmL. The architecture of the tool in Figure 1, already discussed in the first chapter, shows that the specification diagram parser, OCL parser, and instance diagram parser generates parts of an AsmL specification or code. We do this because we use abstract state machines to verify that an instance diagram is a valid instance of a specification diagram by checking constraints in the class diagram written in OCL. The resulting AsmL specification consists of three parts: (1) a set of classes generated from a UML class diagram, (2) a set of OCL constraints translated from OCL constraints given within the UML class diagram, and (3) a set of objects and their instantiations that are generated from an instance diagram (with respect to the UML class diagram). The specification diagram parser module generates the class diagram part of the AsmL specification. Translation of OCL constraints and instance diagram will be described

in their respective chapters.

When we discuss the translation of a specification diagram, we usually refer to a UML class diagram that defines the static structure of a UML model, which include classes and the structural relationships between those classes. A specification diagram, as defined earlier, can include sequence and state chart diagrams, but it does not make sense to translate them into the specification diagram part of the AsmL specification because these diagrams specify the behavior or dynamic view of a UML model, not the static structural view. The sequence and state chart diagrams within a specification diagram are used to perform message ordering verification, which is described in a later chapter. For now, it is sufficient to say that we use the resulting AsmL code to perform model instantiation checking, which will also be described later. Remember that this module is responsible for generating the first of three parts of an AsmL specification, so we ignore everything but the class diagram during translation.

Moving on to the translation schema to convert a UML class diagram into AsmL, one of the advantages of using AsmL as the target language based on the concept of abstract state machines is the fact that it supports object-oriented programming. This feature of AsmL makes translation much easier because UML is inherently object-oriented and a UML class diagram consists of classes, member variables and methods, and inheritance, all of which are supported by AsmL directly. Other specification languages based on ASMs such as XASM [17] do not have such a feature. Hence, we can translate UML class diagram elements in the following way:

- UML classes are directly translated into a class declaration in AsmL. For example, a class called *MyClass* with no parent classes would be translated into

public class MyClass extends libOcl.OclAnyImpl

where *libOcl.OclAnyImpl* is a class defined in the OCL library within the *libOcl* namespace. The reason for this will be given in the chapter describing the OCL library in AsmL.

- Features of a UML class such as attributes and operations will be translated into

  public var at_name as libOcl.OclString

  public var op_calcSalary() as libOcl.OClInteger

  where *OclString* and *OclInteger* are classes declared in the OCL library defining the OCL basic data types *String* and *Integer*. Each attribute and operation has a prefix attached to its name so that it can be easily identified as either an attribute or an operation within methods in the OCL library that return type information of the classes.

- UML associations are translated into sets or sequences of objects of the opposite association end type in the declaration of UML class. Sets are used when the association end is not ordered and sequences are used when it is ordered. For example, there is an unordered unidirectional association from *Customer* to *Order* in Figure 5. When translated into AsmL, the *Customer* class in the AsmL specification will have the following member variable:

  public var ae_order as libOcl.OclSet of Order

  where *libOcl.OclSet* is the OCL library class encapsulating the *Set* collection type in OCL. There is no similar member variable for the *Order* class because the association is unidirectional. We use a set even when the multiplicity of the association end is exactly one because an incorrect

object diagram can have an object that is connected by multiple links instantiating the association to objects at that association end. Graphical constraints including the multiplicity of an association end are validated by the tool within the instance diagram parser module and will be described in further detail in the chapter devoted to it.

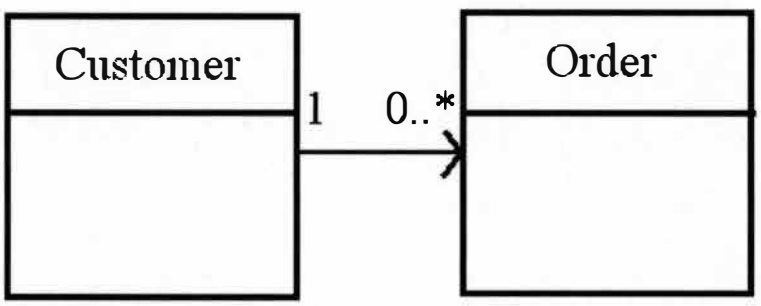

Figure 5. A unidirectional association between two classes.

- Generalizations in UML classes are translated into inheritance statements in AsmL. For example, if the class *Customer* has a parent class called *Person*, then the declaration of the *Customer* class would be

  public class Customer extends Person

  The only problem we encounter is the fact that like Java, AsmL does not support multiple inheritance. It supports the concept of interfaces but that is insufficient to convey the exact semantics of multiple inheritance in a UML class diagram. The first workaround was to have the class with multiple parents inherit from an arbitrary parent (for example, the first parent) and then copy the features and associations of the other parents into the declaration of the class itself while resolving any name conflicts arbitrarily. Recently it was discovered that this workaround would not work correctly in certain cases when the concepts of inheritance and

polymorphism are utilized.

For example, in the UML 1.5 metamodel, the class *Classifier* inherits from both *Namespace* and *GeneralizableElement*. If we make an arbitrary choice and have *Classifier* inherit from *Namespace* in the AsmL specification, everything works fine until we try to represent the association between *Generalization* and *GeneralizableElement* to indicate the child or parent classes in a UML generalization. There we have a problem because *Classifier* cannot be inserted into the association end set of *GeneralizableElement* (translated as described in the previous point) representing the child and parent classes respectively because to the AsmL compiler, *Classifier* is not a child class of *GeneralizableElement*. Because this discovery was recent, we were unable to implement another workaround to this problem and so we leave this as future work.

- A stereotype is an important lightweight extension mechanism of UML that allows a developer to attach new semantics to an existing metaclass in the UML metamodel. We can think of a stereotyped class as extending the functionality of an existing class, and so we use the same translation for stereotypes as we used for generalizations. For example, if the UML metaclass *Class* has a stereotype <<*Entity*>>, then the translation would be

  public class Entity extends Class

  and any tagged definitions for the stereotype would be translated into attributes of this class.

Using the translation schema described above, we are able to translate the frequently used core elements of a class diagram into AsmL. By using the object-

oriented capability of AsmL the translation schema has been greatly simplified. Besides the unfortunate feature of AsmL that does not support multiple inheritance, the translation schema allows most UML class diagrams to be converted successfully.

CHAPTER III

INSTANCE DIAGRAM PARSER

The instance diagram parser module is responsible for reading in an instance diagram from a UML model in the input XML file. The instance diagram parser module uses an XML parser to process the XML file. However, it does not have an XML parser of its own. Rather, as briefly mentioned in the previous chapter, the instance diagram is read into the internal data structure for a specification diagram by the XML parser in the specification diagram parser module. This is because some of the UML diagrams that can be instance diagrams with respect to the UML metamodel such as sequence and state chart diagrams can also be specification diagrams in certain modes of operation of the tool. Specifically, sequence and state chart diagrams are treated as specification diagrams instead of instance diagrams during message order verification, described in a later chapter. Furthermore, because the popular UML modeling application Rational Rose does not support object diagrams separately, we have to use collaboration diagrams to represent the objects, slots, and links in an object diagram. Sequence and collaboration diagrams are represented in the same way in the XML file and are handled as a single entity by the XML parser. Therefore there is no need for a separate XML parser in the instance diagram module to read in any instance diagrams. However, this brings up the need for the tool to convert the instance diagram from the specification diagram data structure into the data structure used to represent an object diagram. Figure 6 shows the functionality of the instance diagram parser module.

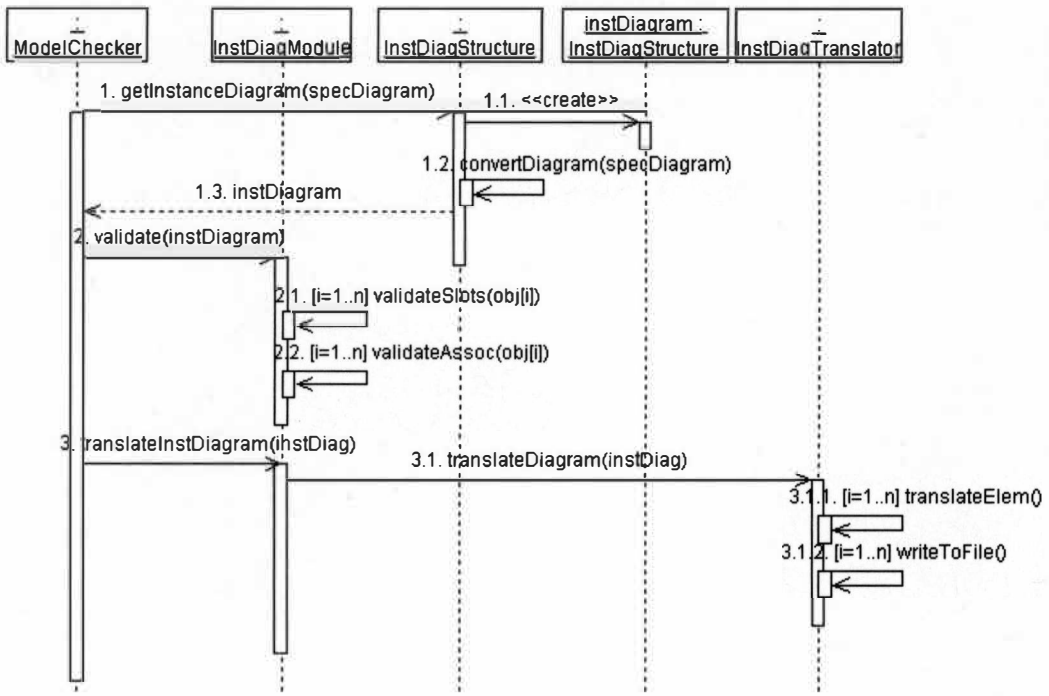

Figure 6. High-level view of translating an instance diagram.

Converting Instance Diagrams into Object Diagrams

In the first chapter we briefly mentioned that a UML model to be input to our tool is partitioned into two top-level packages, one of which contains the specification diagram and the other which contains the instance diagram. Because the XML parser from the specification diagram parser module reads both the specification and instance diagram packages, we have to convert the model elements in the instance diagram package into the object diagram representation. The internal data structure of the tool for the UML model is a tree-like structure where containers such as UML packages fully own the model elements inside them. The tool currently performs two types of conversions:

1. Conversion into an object diagram that is an instance of the UML

metamodel.

This is used mainly to validate a user-defined UML model against the UML metamodel. This validation allows the developer to see if the UML model follows the UML specification correctly. In this case, the specification diagram should be a partial or a complete UML metamodel, with or without OCL constraints given in the metamodel. The instance diagram in this case would either be a UML class diagram, sequence or collaboration diagram, or a state chart diagram and each model element would be converted into objects that instantiate meta-classes in the UML metamodel.

2. Conversion from a collaboration diagram representing an actual object diagram.

   This is the easier of the two, because each classifier role in the collaboration diagram will be directly converted into an object while each association role is converted into a link connecting objects at the association end roles. Slot values for each object are given in documentation tags that are tagged to each classifier role in the collaboration diagram. The use of documentation tags is necessary because we are improvising an object diagram with a collaboration diagram. When Rational Rose exports the UML model into an XML file, documentation tags are exported as well. To avoid confusing slot values for the objects and actual documentation, we enclose slot value declarations inside **BeginSlot** and **EndSlot** strings. We separate different slot values with semicolons, which are optional for the last value. For example, an object *John* of class *Person* that has the attributes *name* and *SSN* could have the

following slot values in the documentation tag for the classifier role representing the object *John*:

BeginSlot

name = 'John'; SSN = 123456789

EndSlot

When the internal data structure for the object diagram is finally created from the instance diagram package within the specification diagram structure, the memory allocated for the package and all its contents are freed, leaving the specification diagram structure with just the model elements for the specification diagram. After this step, the instance diagram structure has been separated from the specification diagram structure and the tool can now perform validation checks and translation into AsmL.

Validation Tests on the Instance Diagram

Some validation tests are performed on the instance diagram before it is translated into AsmL. Problems discovered from these tests are usually given as warnings, much like a compiler gives warnings, and the tool terminates with an error before the diagram is translated into AsmL. This means that the validation tests done here represent the first set of tests performed by the tool to validate parts of a UML model.

Checking Object Instantiation

The instance diagram parser module makes sure that each object in the object diagram that will eventually be translated into AsmL is an instance of a class that

exists in the specification class diagram. If it is not, then the AsmL translation would be an error because all objects must instantiate an existing class. However, it is entirely possible that some objects do not have a corresponding class in the class diagram. This usually occurs when the object diagram is generated from a class diagram, sequence or collaboration diagram, or a state chart diagram. Because this object diagram is an instance of the UML metamodel, the tool hard-codes the translation of some objects based on the UML metamodel version supported by the tool. However, we allow the developer to provide a partial metamodel that the developer wants to compare the UML model against, so some metaclasses may be missing. In this case, the tool just sets a flag on the object so that it will be ignored for translation. For example, the developer can provide a partial metamodel with only the metaclasses *Classifier* and *Attribute*. If there are objects instantiating the metaclasses *Generalization, Operation,* or *Association*, they will be ignored because the developer is not interested in their validation. No error or warning is reported to the user if any object fails this test.

Checking Slot Values for Class Attributes

Each object must have one slot value defined for each attribute that the class it instantiates has. Otherwise, the object diagram is not a valid instance of the specification class diagram. Besides checking for the existence of a slot for each attribute the instantiated class has, we also check to see if the type of the slot value matches the expected type for that attribute. The type of the slot value is evaluated based on the OCL representation for the type. Using the previous example of object *John* of class *Person*, if the *name* attribute is a *String* and the *SSN* attribute is an *Integer*, then the following slot values have the correct types:

BeginSlot

name = 'John'; SSN = 123456789

EndSlot

However, if the slot value for *SSN* was '123456789' instead (note the single quotes), the slot value would be interpreted as an OCL *String*, which does not match the *Integer* type of the *SSN* attribute. Therefore, this will be flagged as an error in the instance diagram in the form of a warning.

Checking Graphical Constraints

Graphical constraints refer to the multiplicities of association ends in the UML class diagram. These constraints can either be checked by algorithms within the tool itself or converted into OCL constraints that will be translated into the AsmL specification in addition to other OCL constraints a class could have. The latter method would have taken another step - namely the conversion into an OCL constraint - and then be treated like any other OCL constraint in the specification diagram. The former however can be done more efficiently within the tool itself by designing an algorithm that, for each association originating from the class an object instantiates, counting the number of links to objects that are exact types or subtypes of the opposite association end.

Furthermore, the role name of the association end, if it exists, must match the role name indicated by the link end for that link to be counted towards the instantiation of that particular association. This check is included to disambiguate multiple associations between two classes, whether or not the associations are inherited. Therefore it is extremely important for association ends in a UML metamodel or profile (i.e. a specification class diagram) to have role names and each

role name should be unique so that the tool can properly count instances of those associations in the instance diagram. If the final count is within the range of the multiplicity for that association end, no error is reported. Otherwise, the tool displays a warning indicating the expected multiplicity for that association end and the count. Stereotypes are treated as subclasses of the element it extends, so the algorithm takes care of that as well.

AsmL Translation

Once the instance diagram passes the checks described in the previous section, the instance diagram parser module proceeds to translate the instance diagram into AsmL. The translation is rather straightforward given the fact that AsmL supports object-oriented programming. Translated AsmL statements are placed within a *Main()* function in the AsmL specification. Objects are translated into object declarations, for example:

```
public var John as Person = new Person()
```

The above statement declares an object called *John* that instantiates the class *Person* using the example we have used in the previous sections. Each object in the instance diagram is declared consecutively. Following the object declarations are slot value assignments for each object. This is done in one step of the ASM. Recall that an ASM is a state machine, but updates to its member variables are done in parallel. The *Main()* function defines one such state machine, and so in one step of that ASM we can initialize any number of different object slot values at the same time using an update statement in AsmL. For example, we can initialize the *name* and *SSN* slots of the object *John* with the following block:

```
step
```

John.at_name.setVal ("John")

John.at_SSN.setVal (123456789)

We note that AsmL uses indentation to identify blocks, so all indentation is important. We also note that primitive types are encapsulated by classes in the OCL library module; hence the *setVal* methods used in the example to set values for primitive OCL types. Back to the above block, we can see that both slot values are initialized at the same time within one step of the ASM for the *Main()* function.

Associations are initialized the same way because they are translated into member variables that are sets or sequences of objects of the opposite association end type. This was described in the chapter on the specification diagram parser module. The only difference with slot initialization is that a single statement may assign an entire set or sequence of objects of the opposite association end. For example, if the object *John* had a set of *Bank* objects connected by links instantiating an association between the classes *Person* and *Bank*, we could have a statement like the following:

step

John.ae_bank.addItems({ BankOne, NatCityBank, BankOfUSA })

AsmL has support for set, bag, and sequence collection types, so the above block would add the set of three bank objects into the set representing the association between *Person* and *Bank* on the object *John*. Likewise, if the association end is ordered, a sequence notation would be used instead of a set.

After the statements initializing the objects in the instance diagram, the tool appends additional AsmL code to initialize the set of all the instances for each class in the class diagram. These sets are used in OCL's *allInstances()* operation, which evaluates to the set of all instances of a given class in the UML model. Finally, the tool adds AsmL statements that will loop through each object in the object diagram

and call its *verify()* method. The body of this method for each class in the specification class diagram contains the translated OCL constraints for the corresponding class and is generated by the OCL parser module. In effect, the AsmL code, when compiled and executed, will check the OCL constraints for each object and will report any constraint violations to the user.

CHAPTER IV

OCL PARSER AND LIBRARY

The OCL is used to represent constraints in the UML class diagram. Therefore, we parse OCL constraints and translate them into AsmL in order to enforce these constraints on the instance-level diagrams described in the previous section. There has been previous work [18] to write OCL compilers to translate OCL into languages such as Java. The OCL parser module, together with the OCL library written in AsmL, support the checking of OCL constraints within a UML model that is translated into AsmL. The OCL library will be discussed first, followed by the design and implementation of the OCL parser itself.

OCL Library in AsmL

The OCL library written in AsmL [22] implements operations of the basic and collection types for OCL as defined in the UML 1.5 specification. It also defines a class hierarchy consisting of all the OCL types including user-defined classes in the specification diagram so that AsmL code translated from an OCL constraint is able to use the specification diagram to check constraints on an instance diagram. Much of this will become clear in a moment when the details of this class hierarchy are provided.

OCL Metamodel

We use a class diagram to represent the structure of OCL types defined in the UML specification. Figure 7 shows the metamodel for OCL that is used for the OCL

library. The original design of the metamodel comes from [19] and has been modified to suit the needs and the limitations imposed by AsmL as a target language.

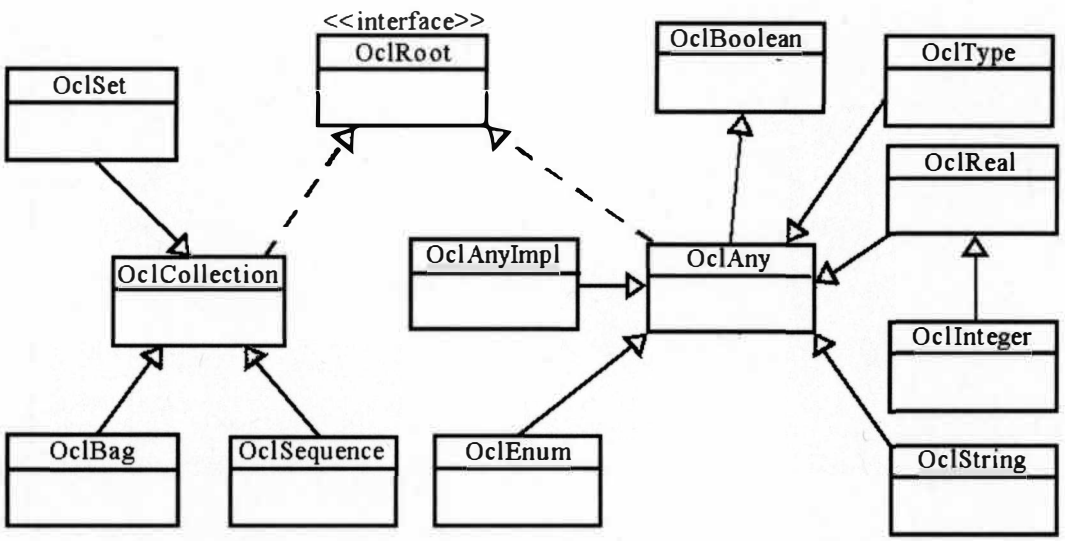

Figure 7. Metamodel showing class hierarchy of OCL types.

All basic OCL types (*Boolean*, *String*, *Integer*, and *Real*) inherit from the *OclAny* class, which is the supertype of all the basic and user-defined types. The *OclType* class has operations to retrieve type information from an OCL object and is implemented as a subtype of *OclAny*. The three OCL collection types, namely *Set*, *Bag*, and *Sequence*, inherit from an *OclCollection* class, which contains operations common to all the collection types. *OclAny* and *OclCollection* implement an *OclRoot* interface, which serves two functions. First, it connects all the OCL classes in the hierarchy together so that object-oriented features like polymorphism can be used. Second, it provides a general set of operations common to all the OCL types. *OclEnum* represents an enumeration in OCL, and we use constant values to define enumeration values. *OclAnyImpl* serves as the base class for all the classes defined by the user in the specification class diagram. This links the classes defined in the UML model with the rest of the OCL types so that we can perform OCL operations on them

during the course of checking OCL constraints.

The UML specification describes operations for basic and collection types in OCL. Since AsmL is a high-level specification language and has most of the basic and collection types that OCL has, most of these OCL operations are implemented as methods that encapsulate the existing operations of the corresponding types in AsmL. The OCL basic datatype *Real* corresponds to the AsmL basic datatype *Double* but is otherwise the same. Another difference is the *Bag* collection type in OCL. Earlier versions of AsmL do not have this collection type, so it was implemented using the *Sequence* collection type in AsmL. Even though the latest versions of AsmL now supports the *Bag* type, we encountered problems using it and there was insufficient documentation in AsmL to find out what was wrong. Therefore the current implementation of the OCL library still uses an AsmL *Sequence* to represent an OCL *Bag* since the functionality is almost the same.

Basic OCL Type Operations

As mentioned earlier, operations for basic OCL types are just encapsulations of the corresponding operations of the basic types in AsmL. For example, the following code fragment shows the implementation of some operations of the OCL *Real* datatype:

```
type Real = Double
public class OclReal extends OclAny
  public var val as Real
  public addition (b as OclReal) as OclReal
    return new OclReal (val + b.val)
  public getVal() as Real = val
```

```
public setVal (b as Real)
  val := b
```

The first statement declares a type alias, much like the *typedef* keyword in C++, to be consistent with the names of the OCL types. The next two lines define the class that encapsulates the AsmL data type for a real number. The addition operator for the *Real* datatype encapsulates the '+' operation of AsmL's Double datatype. Many of the other basic data type operations are defined in this way. The *getVal()* and *setVal()* methods are present in each class representing an OCL type so that we can retrieve and set the values of the internal data members that each of these classes encapsulate. There are also conversion functions converting between the primitive AsmL data types and the corresponding OCL class for that data type. These functions are used by some operations implemented in the OCL library.

Collection Type Operations

Collection types are encapsulated in a way similar to the basic data types, but this is done at the common base class for collection types, *OclCollection*. Part of the implementation for this class is shown below:

```
public abstract class OclCollection of T implements OclRoot
  public var collSet as Set of T
  public var collSequence as Seq of T
  public var usingSet as Boolean
```

AsmL has a feature called type classes that is similar to templates in C++ and allows a class to be instantiated to work on a particular type and this feature is used to define the classes for OCL collection types. The *OclCollection* class has a couple of data members, two of which are mutually exclusive and is used depending whether

the actual concrete derived class is a *Set* or either a *Sequence* or a *Bag*. Remember that the latter two collection types are implemented using an AsmL *Sequence* (whose AsmL keyword is actually just *Seq*). The *usingSet* member indicates to the common collection operations which of the two collection members to use depending on the actual concrete collection type. Previously, instead of two mutually exclusive data members, the implementation used AsmL's disjuctive type, where a variable can be one of a few defined types:

```
public var coll as Set of T or Seq of T
```

The statement above declares the variable *coll* as having either a *Set* or a *Sequence* as a type. The boolean flag is still needed to easily identify the actual type of the data member. However, problems encountered using the disjunctive data type in some operations eventually forced us to use the current implementation of two mutually exclusive data members.

Most of the OCL collection operations are implemented by translating their definitions given in the UML specification into AsmL. For example,

```
public size() as OclInteger
  if isSet() then
    return ToOclInteger (collSet.Size)
  else
    return ToOclInteger (collSequence.Size)
```

The above code fragment implements the *size()* operation in the OCL *Collection* data type. The proper data member in the class is used after determining whether the actual collection type is a *Set*, or a *Sequence* or *Bag*. One of the advantages of using a high-level language like AsmL is the presence of these collection types, which makes implementing the OCL operations a lot easier.

However, some collection operations cannot be implemented so easily. These include what we call iterate-based collection operations.

Iterate-based Collection Operations

There are a number of collection operations that have OCL expressions as parameters that work on all the elements of a collection. These are called iterate-based collection operations and are OCL operations that can be can be described in terms of an *iterate* operation. Operations such as *select*, *reject*, *collect*, *forAll*, and *exists* fall into this category. In the UML specification, *iterate* is defined as follows:

collection->iterate(elem : Type; acc : Type = <expression> |

expression-with-elem-and-acc)

It is a generic collection operation that evaluates the given OCL expression for each element in the collection and the result of each evaluation is accumulated until the final result is returned to the caller. In terms of an AsmL-like pseudocode, an *iterate* expression looks like the following:

step

initialize accumulator

step foreach elem in collection

evaluate OCL expression on elem

update accumulator with result of previous evaluation

step

return accumulator result

The Java-based OCL compiler [18] handles iterate-based collection operations differently using inner classes, which is a feature of Java. AsmL has no such feature so this method is used. Instead of implementing the iterate-based collection operations

in this way within the OCL library in AsmL, the OCL parser module will translate each occurrence of an iterate-based operation into the general form above in the generated AsmL code, replacing the relevant parts with the actual collection and the expression used. From experience this is the best way to handle expressions of this sort, and we believe this is the reason the *iterate* expression was provided in the UML specification.

OCL Type Information

In the UML specification, there are a number of operations that provide type information, such as the *OclType* operations *attributes()*, *associationEnds()*, *operations()*, *supertypes()*, and *allSupertypes()*. In Chapter 2 we mentioned that attributes, operations and association ends that were translated into AsmL had prefixes added to their names. These prefixes are used by some of these operations in order to quickly identify the data members in the AsmL class that are either attributes, operations, or an association ends. All these type operations are implemented using the reflection capabilities of the .NET framework that are defined in the *System.Reflection* namespace. For example, the following code fragment is the implementation of the *attributes()* operation of the *OclType* class:

```
public attributes() as OclSet of OclString
    var s as OclSet of OclString = new OclSet of OclString
    var a as Set of OclString = {}
    var f = objectType.GetFields ((System.Reflection.BindingFlags.Public
      + System.Reflection.BindingFlags.NonPublic +
      System.Reflection.BindingFlags.Instance) as
      System.Reflection.BindingFlags)
```

```
step foreach x in (f as PrimitiveArray of System.Reflection.FieldInfo)
  let temp = x.Name as String
  if temp.StartsWith ("at_") then
    p = new OclString((temp.Substring (3) as String))
    add p to a
step
  s.setVal(a)
step
  return s
```

OCL Parser

The OCL parser module consists of a lexical analyzer, a parser for OCL, a tree parser, and supporting classes for storing the symbol table, translated text, and for translating OCL constraints into AsmL. The first three components are generated from a parser generator called ANTLR [20]. The lexical analyzer retrieves tokens from a text file containing OCL constraints extracted from the specification class diagram and passes them to the parser, which will build a parse tree. The tree parser will then traverse this parse tree and then translate each OCL constraint into AsmL with the help of the support classes.

Writing a parser from scratch is definitely not a good idea because of the complexity of the grammar from OCL, which is available in the UML specification. Therefore a parser generator is used to automatically generate a parser. There are other commonly used parser generators available that output C or C++ parsers, such as yacc or bison for C parsers or bison++ for C++ parsers. The parsers generated from these programs, however, are bottom up parsers and only work well with LR(1)

grammars. Bottom up parsers also suffer from the problem of code readability because of the table-based method used in bottom-up parsing and there is also the problem of shift-reduce conflicts in grammars that are not easy to fix. On the other hand, ANTLR generates LL-based recursive descent parsers and supports semantic and syntactic predicates in a grammar specification, which means that it has more than one look-ahead token to parse grammars with less ambiguity. Recursive descent parsers are also easier to understand compared to bottom up parsers. Furthermore, ANTLR generates C++ parsers and is more suitable for the C++ implementation of the tool because yacc or bison generate C parsers. The bison++ tool produces C++ parsers, but still suffer from the problems of bottom-up parsing and LR(1) grammars that its predecessors had.

Another feature of ANTLR is the support of tree parsers, which are effectively parsers that will traverse a parse tree, such as one generated from a regular parser, and then perform actions on them. Tree parsers, or tree walkers as they are also called, are represented by a grammar specification, just as a language to be parsed is represented using a grammar. Therefore, for the tool we have two grammar specifications: one contains the lexical analyzer specification and the grammar for OCL, and the other one contains the grammar for the tree parser. Most of the actions are given in the grammar specification for the tree parser where the actual translation of OCL constraints are done.

Data Structures for the OCL Parser

The OCL parser uses a few data structures to store information about the OCL constraints and the translated code. First, it has a symbol table structure, which is common in parsers. Each symbol table entry contains at least the name of the symbol

and its type. If the symbol is a collection type, the type of the contained element is also stored. If the symbol is a method, then the return type is stored as well. Each symbol table is a linked list of symbol table entries. OCL supports user-defined operations, so the symbol table structure needs to support variable scoping so that operation parameters have local scope. Scoping is also needed to localize the scope of variable names used in different OCL constraints within the same class (referred to as the context of the constraint). Therefore, we use a tree data structure in which each node is a symbol table. The root of the tree is the symbol table for global symbols, and children nodes represent symbol tables for local scopes. The scope of each sibling node is mutually exclusive.

Besides the symbol table, there are data structures to store the translated OCL constraints. At the lowest level there is the *OclExpr* class, which stores information regarding an OCL expression within an OCL constraint or operation definition. This is the building block for translated OCL constraints. In addition to the text of the translated expression, each *OclExpr* object stores the evaluation type of the expression since OCL constraints consists of evaluation expressions, although most are boolean expressions. We also keep track of the index of the temporary variable name used to store the result of the expression as well as the indentation of this expression statement in AsmL. This is because in AsmL indentation is used to denote blocks rather than bracket pairs commonly used in other programming languages. There are a few other miscellaneous data members in *OclExpr*, but the most notable one is a linked list of *OclExpr*. This list represents translated sub-expressions that the current expression depends on. For instance, in the OCL expression "a and b", the sub-expressions *a* and *b* have to be evaluated first before the *and* expression can be evaluated. We could add the *OclExpr* objects containing the AsmL translation of the

sub-expressions into this list so that they will be written out to the AsmL specification first before the *and* expression. Even though this may not have been a good design decision, we can enforce the order of output of translated sub-expressions in this way.

The *ConstraintElement* class contains a linked list of *OclExpr* objects and represents one OCL constraint. It also has two string data members that contain header and footer AsmL code that are written to the AsmL specification before and after the translated statements that make up the OCL constraint. The *ContextElement* class represents a set of OCL constraints in one context, whether it is a class or an operation (for class or operation constraints respectively). Its data members include a linked list of *ConstraintElement*s, a *ConstraintElement* that stores translated OCL expressions defining support operations used by OCL constraints in a context, and a string data member containing AsmL statements declaring and initializing temporary variables within the scope of the context that are used by the containing OCL constraints.

Methodology for Translating OCL Constraints

We utilize a divide and conquer method to translate each OCL expression, whether it is a constraint, a pre-condition, a post-condition, or a "let" statement defining a local variable or method used by constraints. With the help of the parser, each OCL expression is subdivided into many sub-expressions and translation occurs from the leaves of the parse tree going upwards. Because OCL is an evaluative language, each sub-expression has a result type and therefore it can be evaluated and the result of that evaluation is used in the containing expression, and so on. This means that the translation of an OCL constraint results in many sequential AsmL statements that stores results of the evaluations of intermediate sub-expressions

before the larger containing expression is evaluated. As an example, let us say that the class *Person* has the following OCL constraint (an invariant):

inv: self.isMarried implies self.age > 18

This constraint consists of an *implies* statement, which contains two operands that are sub-expressions. The dot operator in the first operand further subdivides that sub-expression into two more sub-expressions, which match the property call rule in the OCL grammar. The second operand in the *implies* expression is a greater-than expression that can be further divided into two sub-expressions, the second of which is a literal (the smallest possible expression). Taking all of these into account, a translation in pseudo-code would be:

temp1 = self

temp2 = temp1.isMarried

temp3 = self

temp4 = temp3.age

temp5 = new libOcl.OclInteger (18)

temp6 = temp4.isGreaterThan (temp5)

temp7 = temp2.implies (temp6)

As you can see, each sub-expression is evaluated one by one so that its result can be used by the containing expression until the entire OCL constraint (or method) is evaluated. The actual translation may be slightly different but this pseudo-code captures the basic concept in our translation schema.

We translate iterate-based expressions in the same way but the tool adds additional AsmL statements following the pseudo-code for *iterate* expressions given in the section above on iterate-based collection operations. Indentation for the sub-expressions within these iterate-based expressions have to be computed correctly

because of AsmL's use of indentation to denote blocks. With indentation done correctly the tool can translate multiple nested iterate-based expressions without problems.

Property Calls in OCL Expressions

Property calls are one of the most important expressions in OCL because they provide the only access to members of classes in the UML model and operations defined in OCL types. Property calls can have three forms:

1. Property calls using the dot operator, such as **self.name**
2. Property calls using the arrow operator, such as **course->size()**
3. The property by itself, such as **self**

The first form is used to access members of a type where the type is not a collection and to navigate associations between classes. The second form is used to call operations of collection types. The third form is usually either **self**, which returns a reference to the current context, or a member or navigable association from the current context where **self** is implied.

When a property call is encountered during the translation of an OCL expression, the tool will try to search for the property depending on the context where it is encountered. In the first form, we assume that the caller expression (the expression before the dot operator) has been evaluated. We check to see if the property name matches any property (for example, class attribute) of the caller type. In the second form, the tool will check to make sure that the calling expression evaluates to a collection type. If it is, then we check a hard-coded list of possible collection type operations and if one is found, we try to translate it together with any parameters for the operation if they exist. Currently we perform limited type checking

on the parameters of these operations. In the third form, we first check for the "self" string. If found, we translate it to the "me" keyword in AsmL. Otherwise, we proceed to find a match in the current context by assuming that **self** is implied. In all three cases, an error is reported if any type check fails or if the property call is not found for the given caller type. Furthermore, the specification class diagram is only consulted in the first and third forms of the property call expression because the arrow operator in the second form is only used for operation calls from an OCL collection type, which do not involve any UML model.

There is a special case of property call expressions. These are operations of the *OclAny* type and apply to all the types in the model. These operations, which include *oclIsKindOf*, *oclIsTypeOf*, and *oclAsType,* are accessed using the dot operator, so the tool will check for these operation names and translate them into corresponding AsmL statements after doing some type checking. The first two operations translate into an **is** statement (for example, "self.oclIsKindOf(Class)" becomes "me is Class" where **me** is the translation of **self** in AsmL) while the latter is translated into an **as** statement (for example, "self.oclAsType(Class) becomes "me as Class"). To avoid run-time exceptions when the generated AsmL code is compiled and executed, developers writing OCL constraints are advised to use *oclIsKindOf* or *oclIsTypeOf* to validate an object's type before doing type casting with the *oclAsType* operation.

Short-Circuit Evaluation of Logical Operators

When translated into AsmL using the methodology explained earlier, logical expressions using operators such as *and*, *or*, and *implies* are translated so that their operands are evaluated first before the logical expression itself is evaluated. However, with the use of the casting expression using *oclAsType*, run-time casting exceptions

may occur if AsmL code tries to cast an object to an invalid type. An example is the following constraint:

context WebServer

inv: self.person.oclIsKindOf (Student) implies

self.person.oclAsType(Student).cgpa > 3.0

This OCL constraint means that if the navigated association is a *Student* (which is a child of the class *Person*) then we downcast the object to its actual type, *Student*, and evaluate the boolean expression of whether the student's CGPA is greater than 3. The *implies* operator means that if the *Person* object is not actually a *Student* object (but for example an instance of another child class of *Person* called *Worker*) then the constraint automatically evaluates to *true* according to the truth table for *implies*. The problem is that with the default AsmL translation we use, both sub-expressions are evaluated first and therefore the AsmL code may try to typecast a *Person* object that is not a *Student* object, resulting in a run-time exception.

The solution to this is short-circuit evaluation, in which the first operand in a logical expression is evaluated. If the result of the logical expression can be determined completely by the first operand, then the remaining operands are skipped. This concept can be extended to logical expressions with multiple operands, where the fewest amount of operands needed to determine the result of the expression is used. Short-circuit evaluation is a feature available in many programming languages, but involves some complexity in implementation in the tool. However, due to recent uses of our tool requiring OCL constraints that have type-checking followed by type-casting like the example above, it was implemented so that the problem can be solved.

CHAPTER V

UML MODEL INSTANTIATION CHECKING

One of the major purposes of the tool is to perform UML model instantiation checking. The UML is based on the four-level meta-modeling architecture. Each successive level is labeled from M_3 to M_0 and are usually named meta-metamodel, metamodel, class diagram, and object diagram respectively. A diagram at the M_i level is an instance of a diagram at the M_{i+1} level. Therefore, an object diagram (an M_0-level diagram) is an instance of some class diagram (an M_1-level diagram), and this class diagram is an instance of a metamodel (an M_2-level diagram). The M_3-level diagram is used to define the structure of a metamodel, and the Meta Object Facility [21] belongs to this level. The UML metamodel that we have been talking about belongs to the M_2-level. Model instantiation checking is therefore the process in which an M_i-level diagram is checked to see if it is a correct instance of the corresponding M_{i+1}-level diagram that we claim it is an instance of. The tool presented in this work can check a user-defined model against a UML metamodel and also an object diagram against a user-defined class diagram. The tool performs the former by converting the user-defined model into an object diagram and checking to see if it is a valid object diagram with respect to the metamodel. This is done by checking graphical and OCL constraints provided in the specification class diagram to ensure that the instance diagram is valid.

The previous chapters have already described the way in which the tool reads in the UML model containing both the specification and instance diagram provided by the user and extracting the specification diagram, OCL constraints, and the instance

diagram to produce a single AsmL specification that can be compiled and executed to perform model instantiation checking. As a result, this chapter will focus on the notation used in a UML model to be given to the tool for checking as well as other notes about what the tool needs from the standpoint of the user.

Notations Used in the UML Model

To perform model instantiation checking on the diagrams in a given UML model, we have come up with some notations to help the tool recognize parts of the model that it needs to read in. This is due to the fact that UML modeling software such as Rational Rose are designed to support a general UML model that is not specific to any particular subject domain. Another reason is the fact that no modeling software has complete support for all the features of the UML specification. For instance, Rational Rose does not support object diagrams, so we use collaboration diagrams to represent them instead.

To represent the specification and instance diagrams, we use two distinct top-level packages in the model, each of which will contain either the specification diagram or the instance diagram respectively. This notation was briefly introduced in the first chapter and shown in Figure 2. All the UML diagrams that belong to the specification diagram will be created inside the specification diagram package, which is the package at the supplier end of the dependency relation between the two packages at the top level. All the UML diagrams that belong to the instance diagram will be created inside the instance diagram package, which is the package at the client end of the dependency relation. For the purposes of checking multiple instance diagrams against a single specification diagram (which could be a UML metamodel), there can be more than one instance package at the top level package of the UML

model. However, there can only be one dependency relation. To check multiple instance diagrams against one specification diagram, the software developer needs to change the client of the dependency relation from one instance package to another while keeping the specification package as the supplier of the dependency relation. The tool can then be executed using this modified UML model. Conversely, the same technique can be used to check a single instance diagram against multiple specification diagrams, one at a time.

Notations Used in a Specification Diagram

The specification diagram usually consists of a class diagram. This class diagram could represent a complete or a partial UML metamodel or a UML profile (at the M_2 level). The tool supports a partial UML metamodel because the software developer may not want to validate a user-defined diagram against the entire metamodel. This is especially true when a UML profile is used. A UML profile is an extension of a UML metamodel using the lightweight extension mechanism of UML involving UML stereotypes to provide additional semantics, constraints and structural relationships between the original meta-classes in the metamodel. By using a UML profile, the software developer can tailor the UML metamodel to a specific application domain using stereotypes. More information on this can be found in the UML specification [10]. That said, the software developer may only want to validate a UML diagram that is based on the profile against meta-elements in the profile itself. Therefore, the developer can provide only the UML profile that includes the stereotypes and the meta-classes that these stereotypes extend. When the tool converts the instance diagram into an object diagram based on the profile, any objects created based on the UML metamodel that is not provided in the profile will be ignored, so

only objects of meta-elements in the profile will be checked.

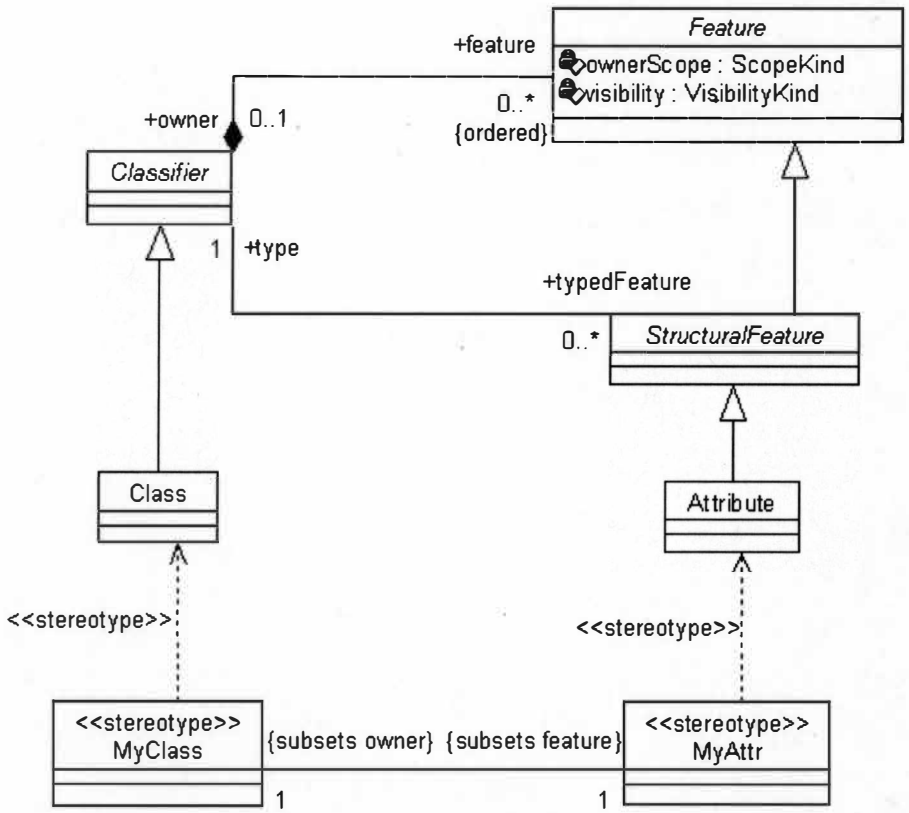

Figure 8. Some elements in an example UML profile.

Figure 8 is an example of a UML profile. In this simple profile, we have part of the UML metamodel elements, namely *Classifier*, *Class*, *Feature*, *StructuralFeature*, and *Attribute*. The profile extends the UML metamodel by adding two stereotypes, *MyClass* and *MyAttr*. These stereotypes extend the meta-classes *Class* and *Attribute* respectively. If a class diagram that uses these stereotypes is provided as an instance diagram, the tool will ignore *Generalization* and *Association* objects in the object diagram converted from the instance-level class diagram since the meta-classes for these elements are not provided in the UML profile.

Figure 8 also shows a number of notations used in a UML model that the tool

will read:

1. Stereotypes

   According to the UML specification, there are two ways to show stereotypes in a UML model. The first is to use a table form to represent the stereotype and listing its base class, descriptions, tag definitions, and constraints. The second is to use a graphical notation, which is the method we are using here. A stereotype is designated with a normal class notation with the stereotype "stereotype", denoted with <<*stereotype*>> in the figure. To indicate the base class that the stereotype extends, we use a dependency relation (also denoted with <<*stereotype*>>) connecting the stereotype (for example, *MyClass*) with its base class (for example, *Class*). Tag definitions for the stereotype are represented as attributes in the class notation (not shown in the figure).

2. Subset associations

   Subset associations are associations between two UML classes or stereotypes that are subsets of another association between the supertypes of these two model elements. The notation we have adopted here is widely used in the upcoming UML 2.0 specification and we use it to disambiguate between multiple associations between the same two classes or their supertypes. In the example in Figure 8, the stereotypes *MyClass* and *MyAttr* inherit two associations from their supertypes, namely the association between *Classifier* and *Feature* and the association between *Classifier* and *StructuralFeature*. Without any special notation, it is impossible to tell which of these two associations is the superset association for the one between the two stereotypes. By adding the

"subsets" string as a constraint at the association ends of the subset association, this problem is solved.

Another important notation used in a specification class diagram is the representation of a enumeration. Figure 9 shows the definition of two enumerations, *ScopeKind* and *VisibilityKind*, in the UML metamodel. As with stereotypes, we use a class notation to define the enumeration but this time the class has the stereotype <<*enumeration*>>. Enumeration literals are denoted with class attributes that do not have a corresponding attribute type. The visibility of the attributes do not matter in this case.

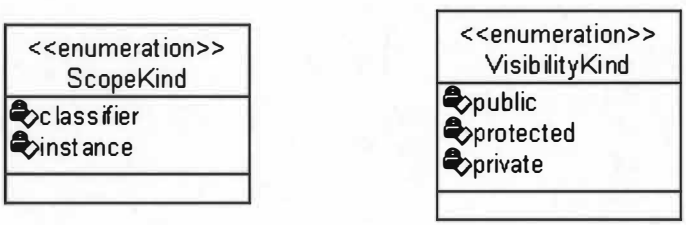

Figure 9. Notation for enumerations in a specification diagram.

OCL constraints for classes in a specification class diagram are given within the documentation part of each class. Rational Rose will export these documentation tags to the XML file containing the model so the tool can read them in. Constraints are placed within two keywords recognized by the tool so that other forms of documentation can be given and will be ignored by the tool. As an example, a class called *Person* may have the following OCL constraints in its documentation tag:

BeginOCL

inv: self.course->size() > 1;

inv: self.rank = #Senior implies self.age > 18

EndOCL

Notations Used in an Instance Diagram

An instance diagram can be an object diagram (at the M_0 level) if we are comparing it with a user-defined class diagram (at the M_1 level) or it can be a class diagram, sequence diagram, or state chart diagram if we are comparing it with the metamodel (at the M_2 level). Rational Rose, which is the most commonly used UML modeling software, does not have object diagrams. Therefore we use a collaboration diagram to represent an object diagram, like the one shown in Figure 10.

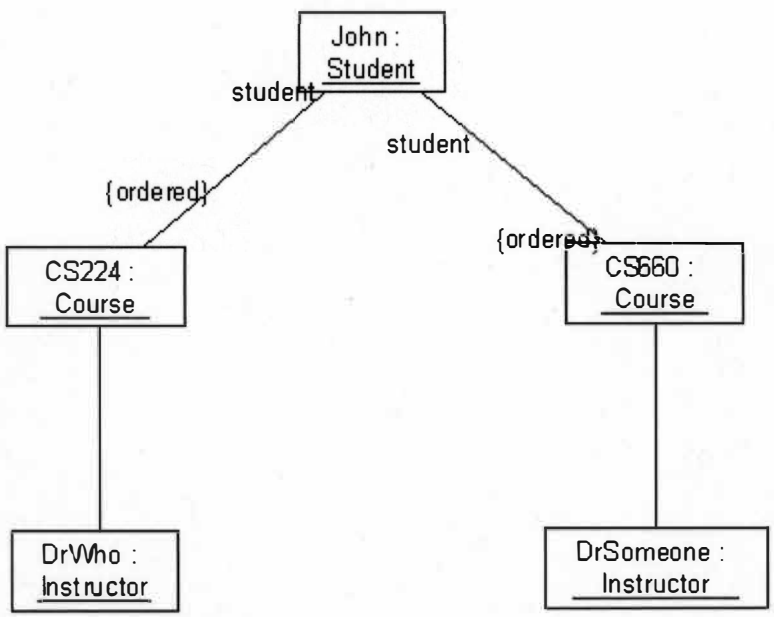

Figure 10. A collaboration diagram representing an object diagram.

Classifier roles are used to represent the objects in an object diagram, while association roles represent the links between the objects. Classifier roles have a name and the base class, which is mapped into the object name and the class that the object instantiates. Rational Rose allows the software developer to select an existing

association as the base for the association role and automatically adds role names to them, if they exist. Objects also have slot values, and they are given inside the documentation part of each classifier role representing the object and they follow the format described in the chapter about the instance diagram parser.

If we are comparing a M_1-level diagram against a UML profile, then the developer may need to provide values for tag definitions a stereotype in the profile might have. We still call them slot values rather than tag values because the M_1-level diagram will be converted to an object diagram that is an instance of the profile. Therefore the format and method to provide slot values for objects are used to provide tag values for instances of stereotypes. However, documentation tags for some elements, such as relations, are not exported to the XML file representing the UML model. Hence, if the UML profile contains stereotypes for these elements that also contain tag definitions, the workaround to provide tag values for instances of these stereotypes is to place them in comment tags, as shown in Figure 11.

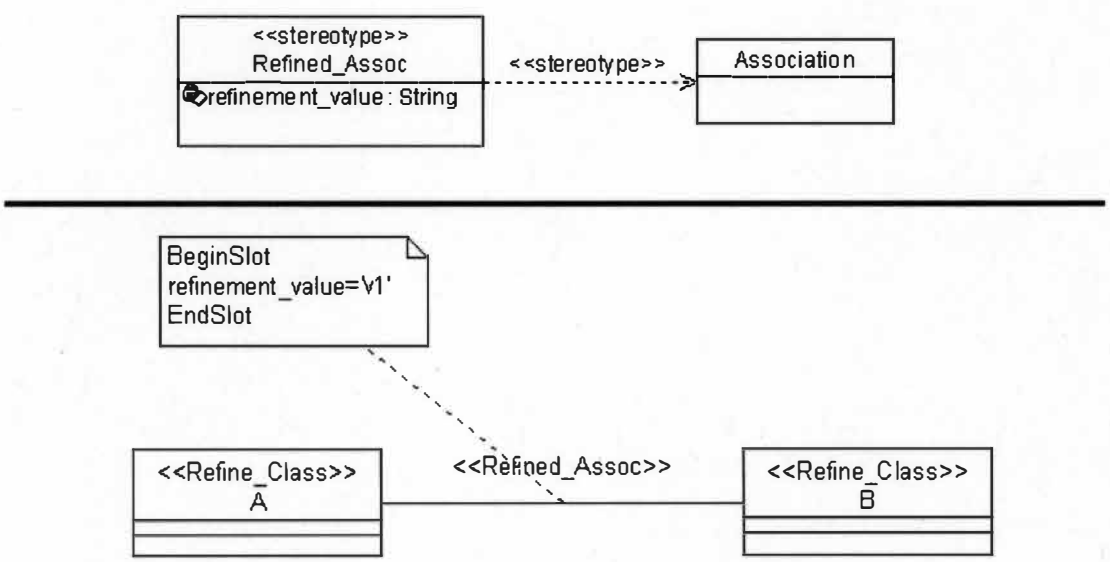

Figure 11. Using comment tags to provide tag values.

Miscellaneous Notes

Metamodel Version in Diagram Conversion

One thing the software developer should know is the fact that the tool hard-codes most of the UML 1.5 metamodel when converting a class diagram, sequence diagram, or a state chart diagram into an object diagram. For example, a class diagram with a class named 'A' with an attribute named 'age' of type *Integer* will be converted into an object diagram with an object of the meta-class *Class* named 'A', an object of the meta-class *Attribute* called 'age', and an object of the meta-class *Datatype* called 'Integer'. There will be a link between object 'A' and object 'age' that is instantiated from the association between meta-classes *Classifier* and *Feature* representing the relationship between a class and its attribute (see Figure 8). There will be another link between object 'age' and object 'Integer' which is an instantiation of the association between meta-classes *Classifier* and *StructuralFeature* representing the type of the attribute, where meta-class *Datatype* is a subtype of *Classifier*. The tool hard-codes the meta-classes and associations that will be used in the conversion when it needs to check an M_1-level model against the metamodel, and because supporting multiple metamodels at this point is beyond the current scope of our work, we have chosen UML 1.5 as the hard-coded metamodel.

What this means to the user of the tool is that the version of the UML metamodel used as the specification diagram for model instantiation checking should be 1.5 and this should also be true of the metamodel that a UML profile extends if a profile is used instead. If a different version of the metamodel is used, then the tool may report errors because it cannot find an association or an element that is present in the UML 1.5 metamodel but not in the version that was given to it.

Graphical Interface to Rational Rose

Currently the tool is written as a console-based Win32 application, which means it needs to run in a command prompt under Windows. This might be suitable for a Unix- or Linux-based application, but a Windows user would expect a graphical user interface. Therefore, we have integrated it with the Rational Rose modeling software as an add-in by writing a script using BasicScript in Rational Rose that provides a Windows dialog box prompting the user for input to be passed on to the tool via a batch file. Although there is no installer, instructions are provided in a text file that would allow another developer to add a menu option to the tool in Rational Rose. At this point we are emphasizing functionality over user-friendliness, but future work include providing a user-friendly graphical interface to the user. A screenshot of the user interface in Rational Rose is shown in Figure 12. The tool has four operating modes right now. The two modes at the top row are the model instantiation checking modes that have been discussed in this chapter. The ones in the second row will be discussed in more detail later. Note that the add-in runs independent of whatever model that has been loaded into Rational Rose because the only input file it requires is an XML file containing the UML model to be checked that has been generated prior to running the tool.

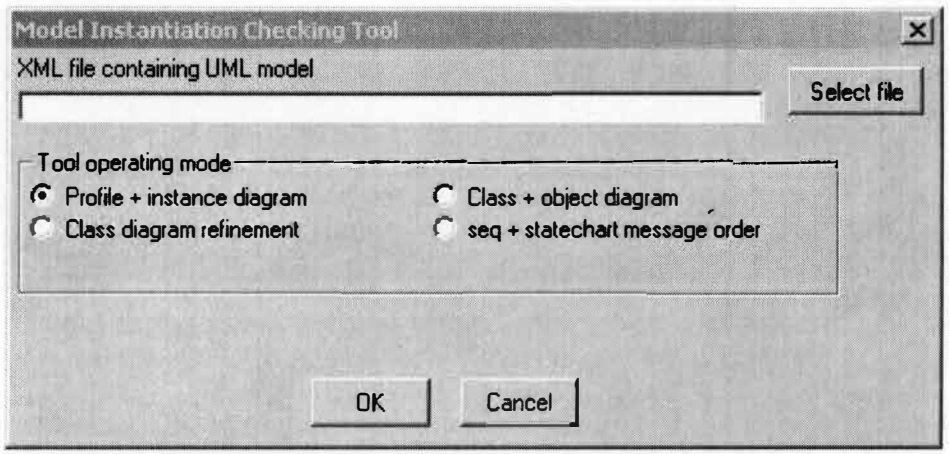

Figure 12. Graphical user interface for the tool in Rational Rose.

CHAPTER VI

UML MESSAGE ORDERING VERIFICATION

Besides model instantiation checking, our tool can also perform what we call message ordering verification, which involves two types of diagrams in a UML model: sequence diagrams and state chart diagrams. UML sequence diagrams are used to model the interactions between objects in a software system. A sequence diagram shows the sequence of messages exchanged between objects in a given scenario. For instance, Figure 13 shows a sequence diagram for a telephone system where the user uses the telephone exchange to make a successful phone call. UML state chart diagrams are used to model the behavior of individual objects by using states that show how the objects respond to various messages and events during the execuation of a software system. As an example, Figure 14 is a state chart diagram showing what a user in a telephone system can do in response to events.

In a valid UML model of a software system, the sequence of messages sent and received by an object in a sequence diagram should be consistent with the sequence of messages sent and received by that same object in a state chart diagram for the corresponding class. For example, if a sequence diagram shows a user picking up a phone, dialing a number and then getting a working phone connection, then the corresponding state chart diagram should show a possible scenario where the same sequence of events can occur. In this work, the process of ensuring that sequence and state chart diagram messages are consistent is called message ordering verification.

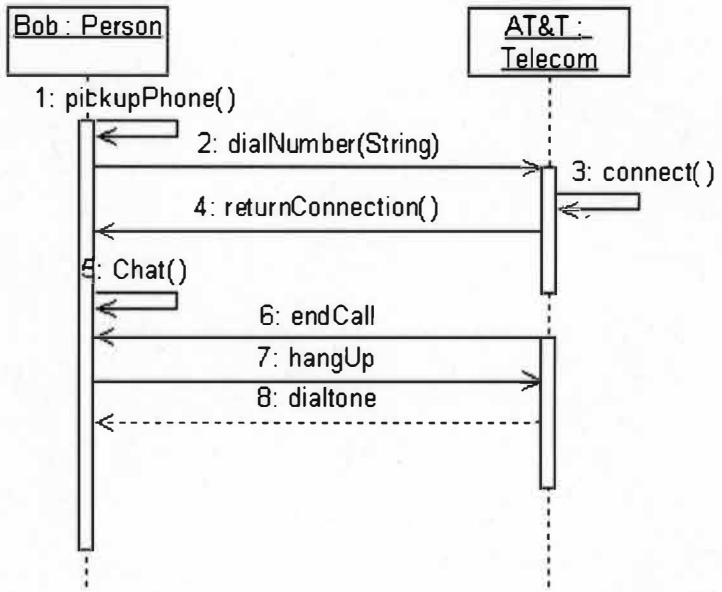

Figure 13. Sequence diagram for a scenario in a telephone network.

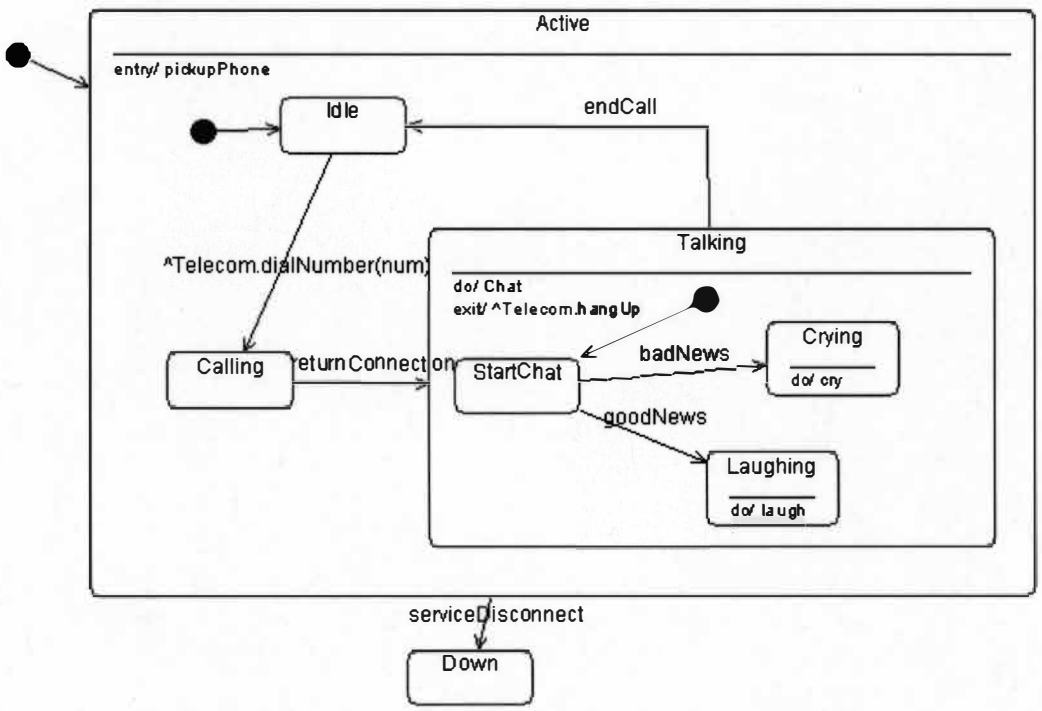

Figure 14. State chart diagram for a person in a telephone network.

To perform message ordering verification, the tool constructs a message graph for each state chart diagram and builds a list of messages sent and received by each object in a sequence diagram. After that the goal is to find an initialized trajectory in the message graph that matches the sequence of messages sent and received by the corresponding object. Basically this is just a graph search to find a path starting from the initial node in the message graph that containing the same sequence of messages in the message list for an object. The following pseudo-code illustrates the overall algorithm for message ordering verification. Details for the algorithm will be given later in this chapter.

```
***** MainLoop:
For each statechart diagram s in the model
        BuildMessageGraph(s)
For each sequence diagram x in the model
        n = number of collaboration roles in x
        create n empty lists, a[n]
        for each message m in the ordered message list for x
                add m to a[i] if i is a receiver or sender in m
        for each list y in a
                CheckMessageOrder(y, classifier for list y)


***** CheckMessageOrder(y,c):
G = message graph for c
p = initial node in G
for each message q in ordered list y
        find b in non-pseudo node successors of p where b = q
```

ifnone return false

p = b

return true

Data Structures for the Algorithm

The data structures used for the message ordering verification algorithm include a message graph, a list of messages for each object in a sequence diagram, and some additional data structures to store intermediate information.

Each state chart diagram will have one message graph, which is a graph where nodes represent messages in a state chart diagram and transitions represent the next possible messages. The message graph represents all possible sequences of messages that can be triggered and executed based on information in the corresponding state chart diagram. It is just a different representation of the state chart diagram, but in this representation each node in the message graph contains messages so searching for sequence of messages along a path in the graph becomes easier compared to doing the same search in a state chart diagram.

There are several node types in the message graph and they are: **StubEntry**, **Entry**, **DoActivity**, **Event**, **EventEffect**, **TransStart**, **Transition**, **TransEffect**, and **Exit** nodes. Most of them correspond to the events and actions in a state chart diagram. Although the message graph only contains message nodes, sets of nodes form a virtual state that has information about the corresponding state in the state chart diagram. As shown in Figure 15, we keep track of the message nodes that belong to a state in the state chart diagram while constructing the message graph. The transitions in the figure represent the possible messages that can occur after a particular message node. Most of the nodes are optional depending on what the state

has, but there will always be an **Entry** node (represented as a **StubEntry** node if there is no entry action) and a **TransStart** node (starting point for all outgoing transitions from the state) in each virtual state. **Event** nodes are the triggers for the internal transitions within a state. **Transition** nodes represent triggers for an outgoing transition. Exit actions occur after a transition event but before the effect of the transition so the **Exit** node for a state, if present, is inserted right after a **Transition** node.

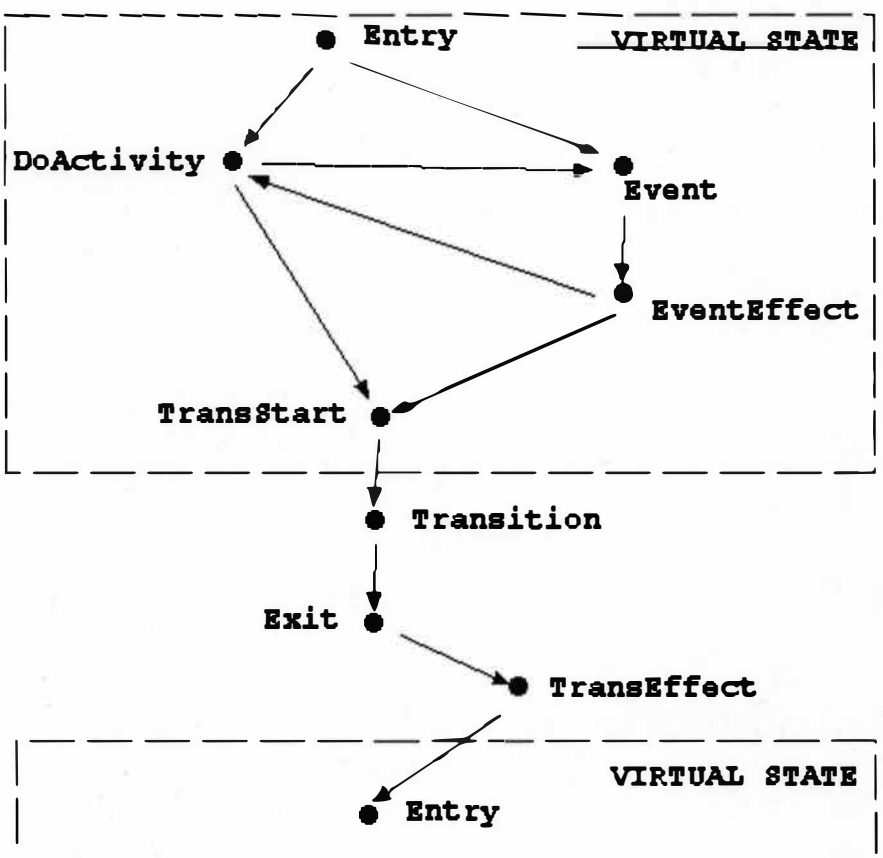

Figure 15. Virtual states in a message graph.

While constructing the message graph, the tool keeps track of the nodes in each virtual state in the graph. For each virtual state, pointers to the nodes

representing the entry action, do activity, and events are stored. Also stored is a pointer to the **TransStart** node for the state and a string containing the exit action, whose node will be inserted separately for each outgoing transition from the state. The state index of the parent state is also stored if the current state is a sub-state. Each state in the state chart diagram read from the UML model is given an integer index unique to a state chart diagram that allows states to be indexed in an array for easy access. The information for virtual states in the message graph is only stored during the construction of the message graph, where that information is used to help in connecting message nodes to each other in the graph. After construction, the message graph only stores the list of nodes and a pointer to the initial node.

Another important data structure is the list of messages for each object in the sequence diagram. Each object in the sequence diagram interacts with other objects by sending and receiving messages. For the purposes of message ordering verification, we want a list of messages sent and received by each object to be checked with the message graph that was constructed based on the state chart diagram of the corresponding class. Messages can have one of two types: sent or received. Sent messages are messages that are sent by the current object to another object in the sequence diagram. We treat messages sent by an object to itself (for instance, to an operation in the object's own class) as a sent message. Received messages are messages received by the current object from another object in the sequence diagram. In Figure 13 the list of messages for the object *Bob* will be all the messages in the sequence diagram except *connect* because that message is only seen by the object *AT&T*. From the context of object *Bob*, messages 1, 2, 5, and 7 are sent messages, while messages 4 and 6 are received messages. We currently ignore return values such as message 8.

Constructing the Message Graph

The first part of message order verification is to construct one message graph for each state chart diagram for a class in the specification diagram. The pseudo-code for the high-level *BuildMessageGraph()* function in the algorithm is shown below:

```
void BuildMessageGraph (StateMachine m)
    initMap(stateNodeMap)
    init = GetInitialState(m)
    buildSubGraphs(m, init)
    buildCompositeStates()
    buildTransitions()
```

The function *initMap()* initializes an array of structures that will store information about virtual states in the message graph. After finding the initial state in the diagram, *buildSubgraphs()* will create and connect message nodes for all the states so that they look like the nodes in Figure 15. Initial states and final states will only have the **StubEntry** and **TransStart** nodes because they do not have the other types of messages. The *buildSubgraphs()* function will also set the "parent" field in the virtual state structure if the current state is a substate within a composite state. Initial substates are also labeled at this point. At the end of *buildSubgraphs()*, the message graph contains many unconnected sub-graphs where each sub-graph represents a state or substate in the original state chart diagram.

The function *buildCompositeStates()* searches for all substates in the state chart diagram and connects each substate's message nodes to the message nodes of all

ancestor composite states:

void buildCompositeStates()

for each state x in the statechart diagram with a parent state

if x is an initial state

add x.entry as successor to parent.entry/doActivity

do

connectSubstate(x, parent)

connectSubstateTrans(x, parent)

parent = parent composite state of parent

until top-level composite state is reached

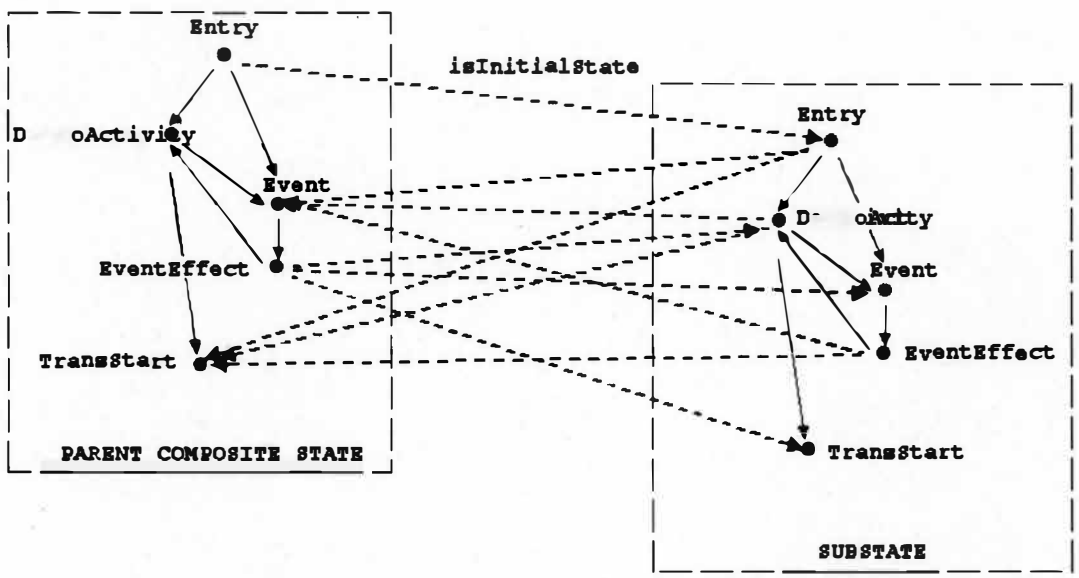

Figure 16. Connecting substates with parent composite states.

Figure 16 shows what *connectSubstate()* does to connect the message nodes of each substate with the message nodes of each parent or ancestor composite state. The messages are connected based on the possible sequences of messages that can occur. For instance an event in the parent state can be triggered during the do activity of a

substate, and after the event's effect or action has been executed, control might go back to the substate's do activity again. Remember that message order verification involves searching the message graph for any path that matches the sequence of messages an object sends and receives in a sequence diagram, so the message graph represents all possible scenarios and paths that can be taken depending on triggered events and guards in the state chart diagram.

The function *connectSubstateTrans()* connects a substate with outgoing transitions of its ancestor states, including a proper sequence of exit actions. This behavior is defined in the UML specification. For example, in Figure 17 there is a transition from state A to state D. If the transition is triggered while the current execution of the object instantiating this class is in substate C, the sequence of messages should be: transition trigger, exitC, exitB, exitA, transition effect, and the entry action of D. The transition trigger and effect as well as the entry action of D are optional. Assuming all the triggers and actions are present, the resulting portion of the message graph would look like Figure 18 after the *connectSubstateTrans()* function completes its task.

The final task in the high-level function B*uildMessageGraph()* is calling *buildTransitions()*, which connects each virtual state in the message graph to another virtual state based on transitions in the state chart diagram. This function links the sub-graphs in the message graph together the same way transitions link different states in the state chart diagram together. This procedure is similar to the one in Figure 18.

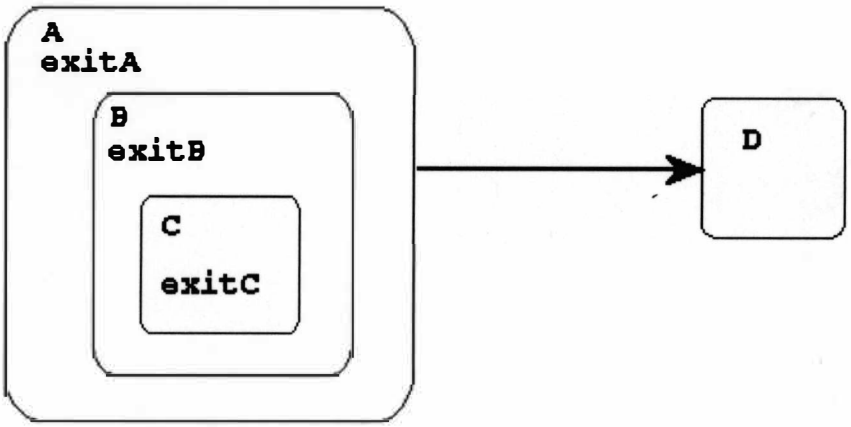

Figure 17. Cascading exit actions in a state chart diagram.

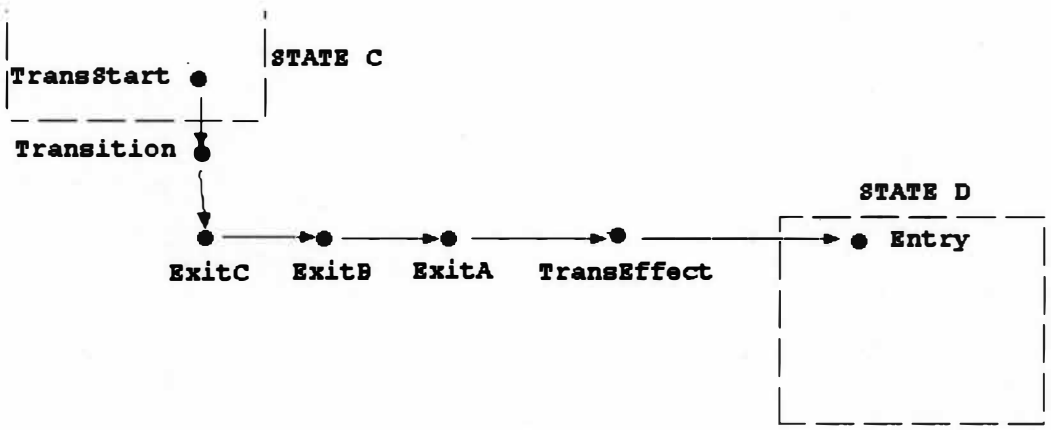

Figure 18. Connecting cascading exit actions.

If there are no transition triggers, effects, and exit action, then the **TransStart** node of the source state is directly linked to the **Entry** node of the target state. Figure 19 shows an example of different outgoing transitions to other states where some transitions may or may not have transition triggers and effects. Besides this, *buildTransitions()* also adds the proper sequence of cascading **Entry** nodes if the source state has a transition going into a nested substate, similar to what is done for cascading exit actions.

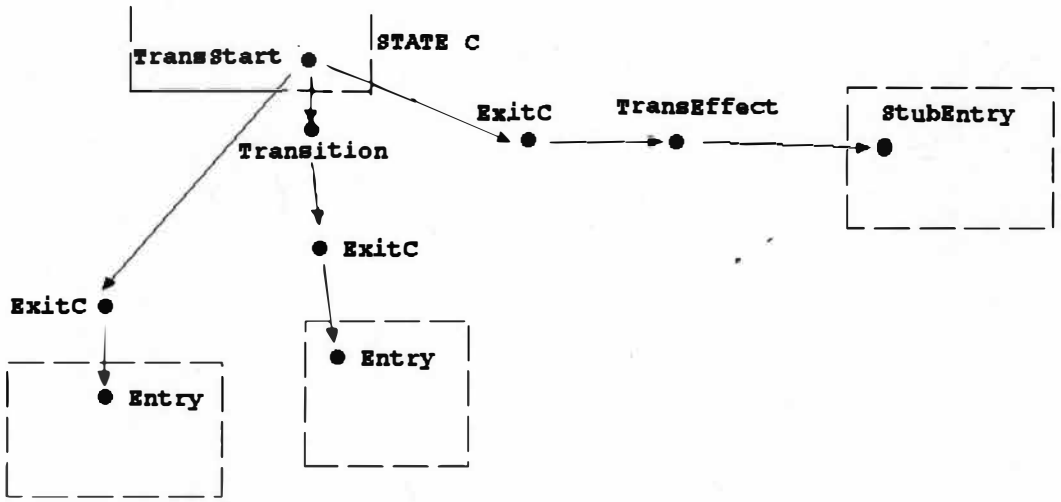

Figure 19. Connecting transitions between states.

Searching the Message Graph

After the message graphs have been constructed, the next step is to build the list of messages for each object in a sequence diagram. If there are multiple sequence diagrams in a UML model then the process is repeated. The pseudo-algorithm for this step was given near the beginning of the chapter. Once the message lists are created, searching the message graph is quite simple. Each message graph has a pointer to the initial message node (which is the **Entry** node for the initial state in the state chart diagram for the particular class). If we think of the message list for an object as a queue, then we first compare the front of the queue with the current node in the message graph. A match is found when the names of the two messages are the same and the message types match. Remember that there are two types of messages in the message list: sent and received messages. Received messages can be matched with event triggers or transition triggers in the state chart diagram because these triggers are messages that the current object receives from another object. Sent messages can be matched with entry actions, exit actions, do activities, event effects, and transition

effects because the messages are sent by the current object to another object or to itself.

When a match is found, then we move on to the next message in the queue. Then we try to find a successor node in the message graph that will match this new message. If no successors are found, then the recursive search algorithm will backtrack to the previously matched message and the previous message node and attempt to find another successor node with a match. The search returns an error if no path in the message graph can be found that matches the sequence of messages in the message list for the current object. The algorithm keeps track of the longest match and returns that in the case of an error so that the developer has more information on what caused the message ordering verification procedure to fail. The algorithm succeeds when a path with the sequence of messages in the message list exists in the message graph.

For example, checking the message list of the object *Bob* in Figure 13 with the message graph generated from the state chart diagram for class *Person* in Figure 14 results in success. If, however, we remove message 4 (*returnConnection()*) from the sequence diagram, the validation process would return an error because the algorithm would be unable to find a path in the message graph for class *Person* where the sent message *Chat* follows the sent message *dialNumber*. In terms of the original state chart diagram, the current state of object Bob after message 2 would be the substate Calling, which has no entry action, do activity, exit action, event effect, or transition effect named *Chat*. Therefore, the sequence diagram is incorrect with respect to message ordering.

Guards in Transitions and Events

Transitions and events (which are just internal transitions in a state) in a state chart diagram may have guards, which are boolean expressions that have to evaluate to a true value before a transition can be traversed. Guards provide some level of control over the behavior of an object depending on factors such as variable values and so on. Supporting guards would definitely make our message ordering verification algorithm better since they are part of the semantics of a state chart diagram. When the tool encounters a guard in a transition node in the message graph it is currently trying to match, the tool will display the guard expression and prompt the user to say whether the guard evaluates to true or false. If the guard is true, then the tool proceeds with the graph search along the current path for the next match. Otherwise, the tool treats the match as being false and the algorithm will backtrack to search for another matching successor node. The evaluation of the guard is therefore up to the developer, and he may let a guard be true or false on different runs to test different scenarios in the sequence diagram.

This method of handling guards in transitions and events may not be very efficient, but the alternative would be to implement a parser that can parse and evaluate guards, which also means that the tool has to keep track of all variables and values so that the guards can be evaluated automatically. It might be possible to perform message ordering verification in AsmL by generating AsmL code that can be compiled and executed, just like we did for model instantiation checking, but the entire design for this validation test will need to be changed.

Notation for Message Names

Message names in the sequence and state chart diagrams need to have a

particular notation for the messages to be compared for matches during the second phase of message order verification in which the message graph is searched. Depending on the UML model, there might be different ways of specifying messages. The simplest notation are the ones in our telephone network example where messages are just single names (for example, *hangUp*). In a more detailed and precise model, however, messages can have parameters since most messages between objects are actually operation calls (for example, *dialNumber(someNumber)* which has one parameter). In state chart diagrams, the name of the class that the message is sent to or the operation call is directed to can be prefixed to the message. For instance, *Telecom.dialNumber(someNumber)* is a message sent to an object of the Telecom class calling the operation of that class named *dialNumber* with a single parameter. These are three notations that can be used for messages in the two UML diagrams.

The tool handles message names by stripping class prefixes and parameters off, leaving only the actual message or operation call. Therefore in the two preceding examples we are left with the message *dialNumber*, which can be easily matched. A better but more complex way is to check the parameters as well, but that would require data structures and parser code so that parameter evaluation can be done. Just as with the complex checking of transition guards, this might be handled better by parsing and translating the sequence and state chart diagrams into executable form such as through the use of AsmL. This is beyond the scope of the our work because our main focus is on model instantiation checking.

CHAPTER VII

APPLICATIONS OF THE TOOL

Now that the features and uses of the tool have been described, we now look at what the tool can be applied to. One of the common uses of UML is to design UML profiles based on the metamodel and then creating diagrams based on the profile. Our tool allows software developers to check that the diagrams they create are correct instance diagrams with respect to a UML profile that is used in a specific domain. For this thesis two such applications of the tool are presented: (1) a UML profile assisting developers in class diagram refinement, and (2) verifying class diagrams that are based on UML profiles representing design patterns.

Class Diagram Refinement

Modern software development is a complicated process especially when the software system to be designed is large and complex. Software developers apply software refinement in order to proceed from a high-level design to a more detailed design by adding new diagrams, classes, and other elements to an existing UML model representing the software system. Class diagrams are important because they represent the static structure of a software system, and therefore we design a UML profile based on the UML metamodel to support class diagram refinement [23]. The main idea is to help developers check whether two consecutive levels of class diagrams, one of which is a refined version of the other, have any semantic discrepancies that could have been introduced during the refinement phase. This will help software developers find errors during software development and also lets them

know whether their development is on the right track since some discrepancies are caused by the refinement process from an imprecise problem to a precise solution.

We support class diagram refinement by designing a UML profile, which extends the original UML metamodel by using stereotypes, that software developers can use to tag classes and relations in the class diagrams at the different levels of refinement. Then with the help of OCL constraints written for some of the stereotypes, we can use the model instantiation checking feature of our tool to check whether the combined class diagram containing classes at the two levels of refinement is a correct instance of the profile, which would mean that they pass the rules of refinement that we provide in the OCL constraints.

Notation for Class Diagram Refinement

In order for the tool to support class diagram refinement, we have to specify a notation to represent class diagrams at two different levels of refinement within a UML model that will be given as input to the tool. The solution here is pretty simple. We use the same notation that we used to separate specification and instance diagrams in the UML model when using the tool for model instantiation checking, which is to use packages and connect them with dependency relationships. Therefore, the UML model used for this purpose would first have two top-level packages: one for the UML profile (as the specification diagram package) and one for the instance diagram. Instead of the normal instance diagrams, the instance diagram package would contain two or more packages that will each contain one class diagram at different levels of refinement. There has to be exactly one dependency relationship connecting two of those packages where the supplier is the higher-level class diagram and the client is the refined class diagram. This is shown in Figure 20, in which we

are designating the package labeled "Level 2" as the higher-level class diagram and the "Level 3" package as the refined class diagram. This way the software developer is able to compare any two levels of class diagram refinement, just as we can validate any instance diagram package with a corresponding specification diagram package by changing the client and supplier of the dependency relation.

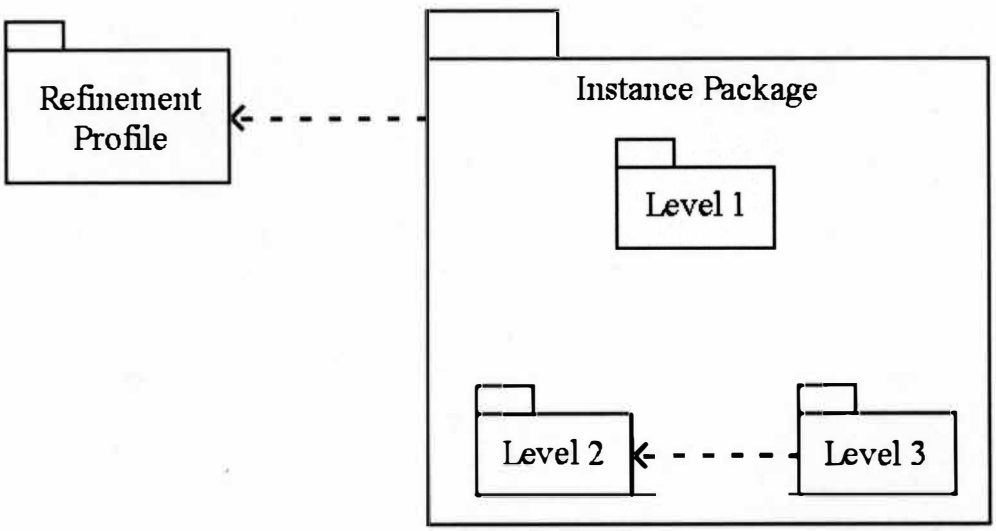

Figure 20. Package structure in a UML model supporting class diagram refinement.

We extend the UML metamodel by introducing stereotypes including *Refine_Class*, *Refined_Assoc*, *Refining_Assoc*, *Refined_Gen*, and *Refining_Gen*. A stereotype that has the prefix *Refined_* represents an element to be refined, i.e. its instance belongs to a higher-level model. A stereotype with the prefix *Refining_* represents an element that is used to refine an element at the higher-level model. The *Refine_Class* stereotype is used to represent classes at the ends of a refined or a refining relation. We define a tag named *mapping_name* that is used to represent a refinement relation between two classes during class diagram refinement. If two classes have the same value for *mapping_name*, then these two classes are involved in

some refinement relation. Each of the other stereotypes have a tag named *refinement_value*, which is used to represent elements that are involved in a refinement relation. All *Refined_* and *Refining_* stereotypes with the same value for *refinement_value* are involved in some refinement relation. In addition to this, each *Refining_* stereotype also has a tag named *mapping_order*, which is an integer value that represents the order in which the refining element appears in the relation that is refined in the refined class diagram. The purpose of this will be made clearer when the rules for refinement are discussed below. Figure 21 shows part of the UML profile supporting class diagram refinement.

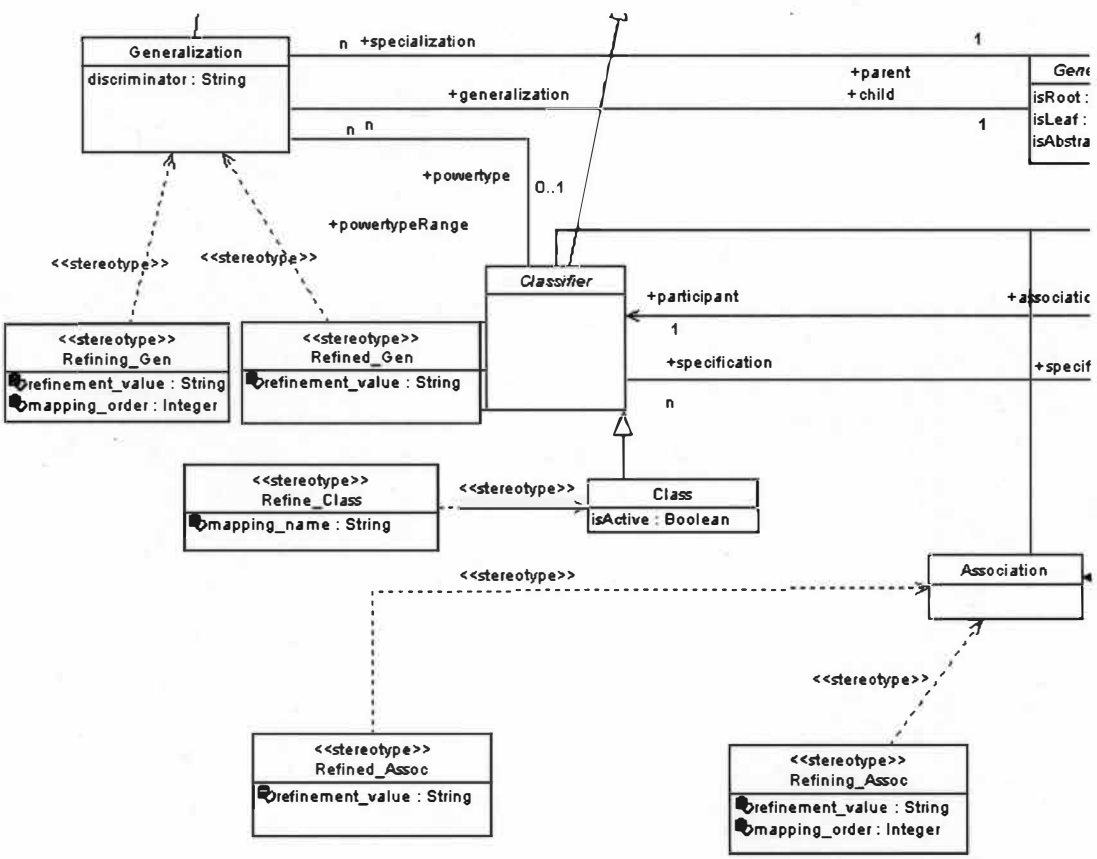

Figure 21. Part of UML profile supporting class diagram refinement.

Rules for Refinement

Rules for the refinement of relations from a higher-level model to a refined model are given in the form of OCL constraints attached to some of the stereotypes defined in the profile for class diagram refinement. We will present some refinement rules for generalization, bidirectional association, and unidirectional association.

Generalization

At level one, which is the higher-level class diagram, we assume that the class A is a subclass of B. Then at level two, which is the refined class diagram, the generalization can be refined into two or more generalizations by adding helper classes. We assume that class A maps to class X in the refined model while class B maps to class Y by using the same values for *mapping_name* respectively. We also assume that the order of refining generalizations go from the child to the parent end-classes. This rule is depicted in Figure 22.

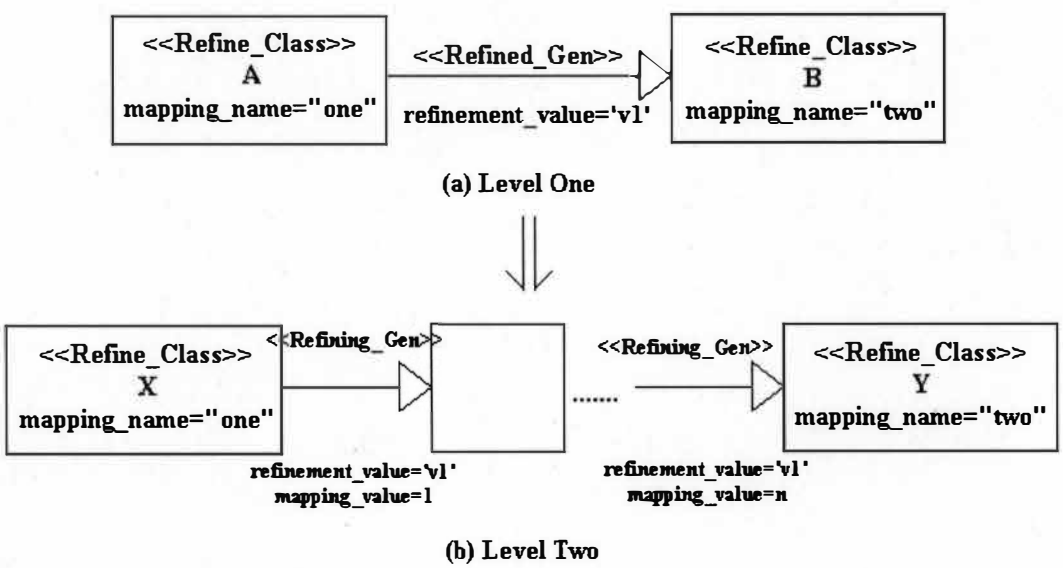

Figure 22. A rule for refinement of generalizations.

The stereotype *Refined_Gen* is used to represent a generalization at the higher-level class diagram that will be refined in the lower-level class diagram. It should satisfy restrictions between itself and its corresponding refining generalizations:

- The refined generalization should be refined to a set of generalizations
- The child of the refined generalization should correspond to the child of the first refining generalization
- The parent of the refined generalization should correspond to the parent of the last refining generalization
- Two consecutive generalizations should have the same class as their end class, and this class is the parent class of one generalization and the child class of the other generalization.

The OCL constraint for the stereotype *Refined_Gen* is shown in Figure 23.

```
context Refined_Gen
inv: let matchingGen : Set(Generalization) =
  Generalization.allInstances()->select(g| g.oclIsKindOf(Refining_Gen) and
    g.oclAsType(Refining_Gen).refinement_value = self.refinement_value)
  in
  self.parent.oclIsKindOf(Refine_Class) and
  self.child.oclIsKindOf(Refine_Class) and
  matchingGen->forAll (a,b | a.oclAsType(Refining_Gen).mapping_order =
    b.oclAsType(Refining_Gen).mapping_order implies a = b
  )
  and matchingGen->forAll (a|
    a.oclAsType(Refining_Gen).mapping_order >= 1 and
    a.oclAsType(Refining_Gen).mapping_order <= matchingGen->size()
  )
  and matchingGen->exists (a,b | a<>b and
    a.oclAsType(Refining_Gen).mapping_order = 1 and
    a.child.oclIsKindOf(Refine_Class) and
    a.child.oclAsType(Refine_Class).mapping_name =
    self.child.oclAsType(Refine_Class).mapping_name and
    b.oclAsType(Refining_Gen).mapping_order = matchingGen->size() and
    b.parent.oclIsKindOf(Refine_Class) and
    b.parent.oclAsType(Refine_Class).mapping_name =
    self.parent.oclAsType(Refine_Class).mapping_name
  )
  and matchingGen->forAll (c,d | (c<>d and
    c.oclAsType(Refining_Gen).mapping_order + 1 =
    d.oclAsType(Refining_Gen).mapping_order) implies
    (c.parent = d.child)
  )
```

Figure 23. Refinement rule for generalizations.

The constraint for the stereotype *Refining_Gen* is related to the stereotype *Refined_Gen*, which was already provided. Therefore there is no constraint for *Refining_Gen*.

Bidirectional Association

At level one, we assume that there is a bidirectional association between class A and class B. This association can be refined to a set of bidirectional associations using a set of helper classes. Figure 24 shows this refinement rule.

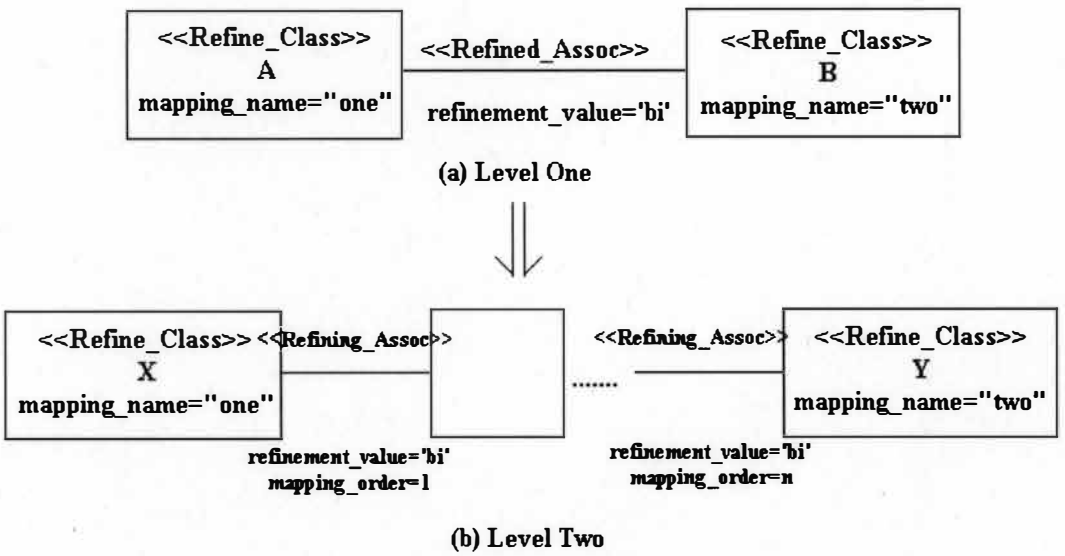

Figure 24. A rule for refinement of bidirectional associations.

The stereotype *Refined_Assoc* represents the association at the higher-level diagram that will be refined in the lower-level diagram. It should satisfy the following restrictions between itself and its corresponding refining associations:

- The refined association should be refined to a chain of associations
- The two end classes on the refined association correspond to the two classes at both ends of the chain of the refining associations

- Two consecutive bidirectional associations should have the same class as their end class.

The OCL constraint for the stereotype *Refined_Assoc* is shown in Figure 25.

```
inv: let matchingAssoc : Set(Association) =
  Association.allInstances()->select(g| g.oclIsKindOf(Refining_Assoc) and
    g.oclAsType(Refining_Assoc).refinement_value = self.refinement_value) in
  self.associationEnd->forAll (p,q | (p<>q and p.isNavigable and q.isNavigable)
  implies (p.participant.oclIsKindOf(Refine_Class) and
    q.participant.oclIsKindOf(Refine_Class) and
    matchingAssoc->forAll (a,b | a.oclAsType(Refining_Assoc).mapping_order =
      b.oclAsType(Refining_Assoc).mapping_order implies a = b)
      and matchingAssoc->forAll (a|
      a.oclAsType(Refining_Assoc).mapping_order >= 1 and
      a.oclAsType(Refining_Assoc).mapping_order <= matchingAssoc->size())
    and matchingAssoc->exists (a,b | a<>b and
      (a.oclAsType(Refining_Assoc).mapping_order = 1 and
        a.associationEnd->exists (e1| e1.isNavigable and
          e1.participant.oclIsKindOf(Refine_Class) and
          e1.participant.oclAsType(Refine_Class).mapping_name =
          p.participant.oclAsType(Refine_Class).mapping_name)
        and b.oclAsType(Refining_Assoc).mapping_order = matchingAssoc->size() and
        b.associationEnd->exists (e2| e2.isNavigable and
          e2.participant.oclIsKindOf(Refine_Class) and
          e2.participant.oclAsType(Refine_Class).mapping_name =
          q.participant.oclAsType(Refine_Class).mapping_name))
      or (a.oclAsType(Refining_Assoc).mapping_order = matchingAssoc->size() and
        a.associationEnd->exists (e1| e1.isNavigable and
          e1.participant.oclIsKindOf(Refine_Class) and
          e1.participant.oclAsType(Refine_Class).mapping_name =
          p.participant.oclAsType(Refine_Class).mapping_name)
        and b.oclAsType(Refining_Assoc).mapping_order = 1 and
        b.associationEnd->exists (e2| e2.isNavigable and
          e2.participant.oclIsKindOf(Refine_Class) and
          e2.participant.oclAsType(Refine_Class).mapping_name =
          q.participant.oclAsType(Refine_Class).mapping_name)))
    and matchingAssoc->forAll (c,d | (c<>d and
        c.oclAsType(Refining_Assoc).mapping_order + 1 =
        d.oclAsType(Refining_Assoc).mapping_order)
      implies (c.associationEnd->exists (x| d.associationEnd->exists (y|
          x.isNavigable and y.isNavigable and x.participant = y.participant)))))
)
```

Figure 25. Refinement rule for bidirectional associations.

The constraint for the stereotype *Refining_Assoc* is related to the one in *Refined_Assoc* and therefore no constraint is provided for *Refining_Assoc*.

Unidirectional Association

At level one, we assume that there is a unidirectional association from class A to class B, which is then refined to a set of unidirectional associations in the lower-level diagram. This rule is shown in Figure 26.

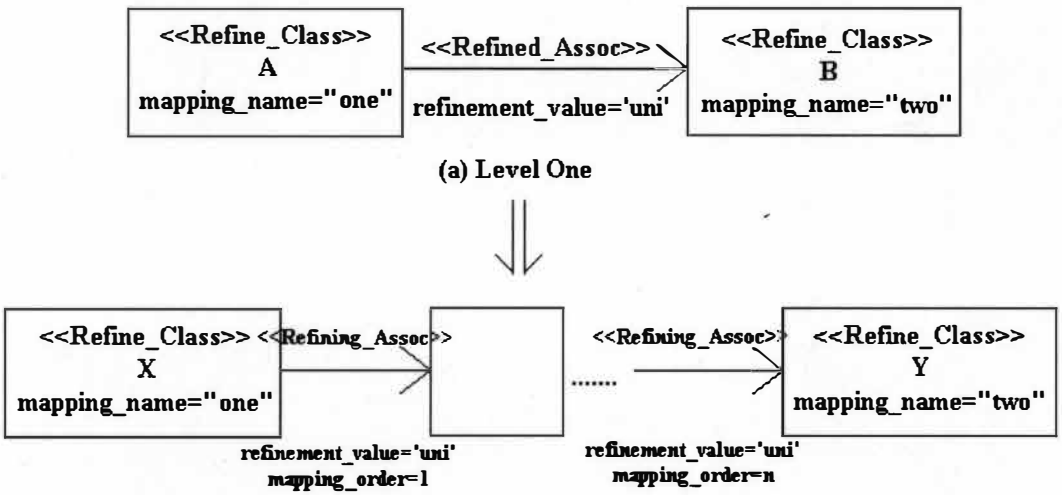

Figure 26. A rule for refinement of unidirectional associations.

The stereotype Refined_Assoc is also used here to represent a unidirectional association at the higher-level class diagram. It should satisfy the following restrictions between itself and the set of *Refining_Assoc* in the refined class diagram:

- The refined association should be refined to a chain of associations
- The class at the navigable end of the refined association should correspond to the class at the navigable end of the last association in the chain of the refining associations
- The class at the non-navigable end of the refined association should correspond to the class at the non-navigable end of the first association in the chain of the refining associations
- Two consecutive unidirectional associations should have the same class as their end class and this class is at the navigable end of one association and the non-navigable end of the other association.

The OCL constraint for the stereotype *Refined_Assoc* is shown in Figure 27. The refinement rules for bidirectional associations and unidirectional associations are

mutually-exclusive and can be seen in the first operand for the *implies* operator near the beginning of each OCL constraint, which ensures that the constraint is completely evaluated only if the required conditions are true. Remember that with short-circuit evaluation, the rest of the *implies* expression is ignored if the first operand evaluates to false.

```
inv: let matchingAssoc : Set(Association) =
  Association.allInstances()->select(g| g.oclIsKindOf(Refining_Assoc) and
    g.oclAsType(Refining_Assoc).refinement_value = self.refinement_value) in
  self.associationEnd->forAll (p,q | (p<>q and not p.isNavigable and q.isNavigable)
  implies (p.participant.oclIsKindOf(Refine_Class) and
    q.participant.oclIsKindOf(Refine_Class) and
    matchingAssoc->forAll (a,b | a.oclAsType(Refining_Assoc).mapping_order =
      b.oclAsType(Refining_Assoc).mapping_order implies a = b)
    and matchingAssoc->forAll (a|
      a.oclAsType(Refining_Assoc).mapping_order >= 1 and
      a.oclAsType(Refining_Assoc).mapping_order <= matchingAssoc->size())
    and matchingAssoc->exists (a,b | a<>b and
      a.oclAsType(Refining_Assoc).mapping_order = 1 and
      a.associationEnd->exists (e1 | e1.participant.oclIsKindOf(Refine_Class) and
        e1.participant.oclAsType(Refine_Class).mapping_name =
        p.participant.oclAsType(Refine_Class).mapping_name)
      and b.oclAsType(Refining_Assoc).mapping_order = matchingAssoc->size() and
      b.associationEnd->exists (e2 | e2.participant.oclIsKindOf(Refine_Class) and
        e2.participant.oclAsType(Refine_Class).mapping_name =
        q.participant.oclAsType(Refine_Class).mapping_name))
    and matchingAssoc->forAll (c,d | (c<>d and
        c.oclAsType(Refining_Assoc).mapping_order + 1 =
        d.oclAsType(Refining_Assoc).mapping_order)
      implies (c.associationEnd->exists (x| d.associationEnd->exists (y|
          x.isNavigable and x.participant = y.participant))))
    and matchingAssoc->exists (m | m.associationEnd->exists (n |
        not n.isNavigable)))
)
```

Figure 27. Refinement rule for unidirectional associations.

Advantage of Using Metamodel Methodology

The refinement rules presented here are conservative in the sense that they try to keep the transitive property for each relationship during refinement. The refinement rule for unidirectional associations actually allows some of the refining associations to be bidirectional as long as at least one of them is unidirectional and the navigation of the unidirectional associations is consistent with the corresponding unidirectional association at the higher-level model. One important advantage of the metamodel methodology is that developers can easily design their metamodel to reflect the

refinement rules they want.

Design Pattern Profile Checking

Another application of our tool is to the validation of UML class diagrams generated from a UML profile specifying a design pattern. France et al. [24] describe a technique to specify design patterns by specializing the UML metamodel to obtain a pattern metamodel that can be used in model driven architecture (MDA). The pattern specification is represented using a custom notation in a class or sequence diagram, which we can translate into a UML profile using stereotypes, graphical constraints, and OCL constraints. Class diagrams that are designed based on a particular design pattern can be regarded as an instance of the UML profile representing the pattern. This means that we can use our tool to verify that the generated diagrams are valid instances of the corresponding profile. This can be useful to the software developer, especially after details are added to the UML diagrams, to ensure that the diagram still follows the constraints and structure specified by a particular design pattern.

To illustrate the application of our tool to design pattern profile checking, we present an example of a user-defined class diagram that uses the State design pattern introduced by Gamma et al. [25] and check whether the class diagram is a valid instance of the UML profile representing the State design pattern.

A Profile for the State Design Pattern

The State design pattern is a behavioral design pattern that allows the behavior of an object to change when its internal state changes. This design pattern is used when an object's behavior depends on its state at run-time. The structure of the State design pattern is shown in Figure 28.

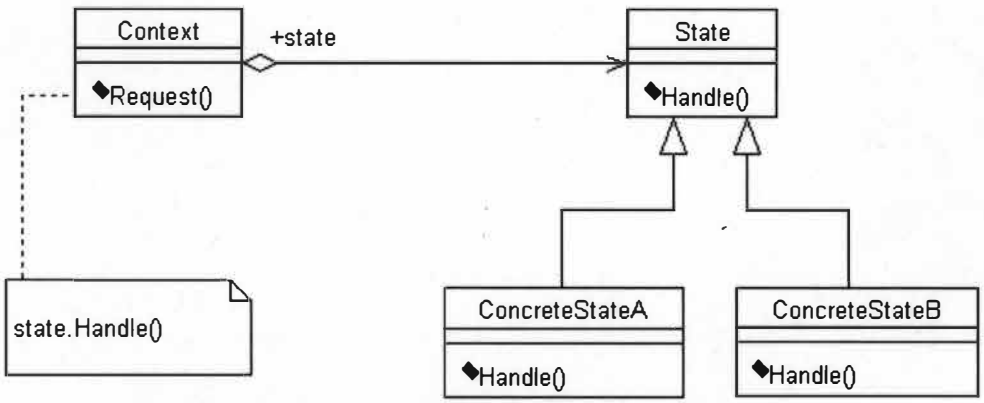

Figure 28. Structure of the State design pattern.

Based on the structure of the design pattern we can translate it into a UML profile. To do this we introduce a number of stereotypes:

1. Two stereotypes, extending the meta-class *Class,* named *StateContext* and *State.*
2. Two stereotypes, extending the meta-class *Operation,* named *StateRequestOp* and *StateHandleOp.*
3. A stereotype, extending the meta-class *Association,* named *aContextState* representing the association between *StateContext* and *State.*
4. Two stereotypes, extending the meta-class *AssociationEnd,* named *CSC* and *CSS* representing the two association ends for the association stereotype.

Besides the stereotypes, we also introduce associations between stereotypes that are subsets of the associations of their extended meta-elements. These associations are relevant to the State design pattern and allows us to specify graphical constraints through multiplicity values on the association ends. For instance, each class with the stereotype *StateContext* must have at least one operation that is stereotyped with *StateRequestOp*; hence we have a multiplicity value of *1..** at the

end of the association between the stereotypes *StateContext* and *StateRequestOp*. Figure 29 presents part of the UML profile representing the State design pattern. Notice that all the associations between the stereotypes use the subset constraint notation described in Chapter 5 to indicate that these associations are subsets of some association inherited by the base classes of the two end stereotypes respectively. To disambiguate the subset associations from their superset associations, arbitrary role names are given to association ends so that the set of ends at each stereotype have unique role names (where having no rolename is also considered one unique role name). For example, *aContextState* has two association ends connecting it with *CSS*, one of which comes from the association inherited from meta-class *Association* that has no role name and the other which is from the subset association with the role name *fromCSS*. Notice that the role name *fromCSS* disambiguates the ends of the two associations connecting these two stereotypes.

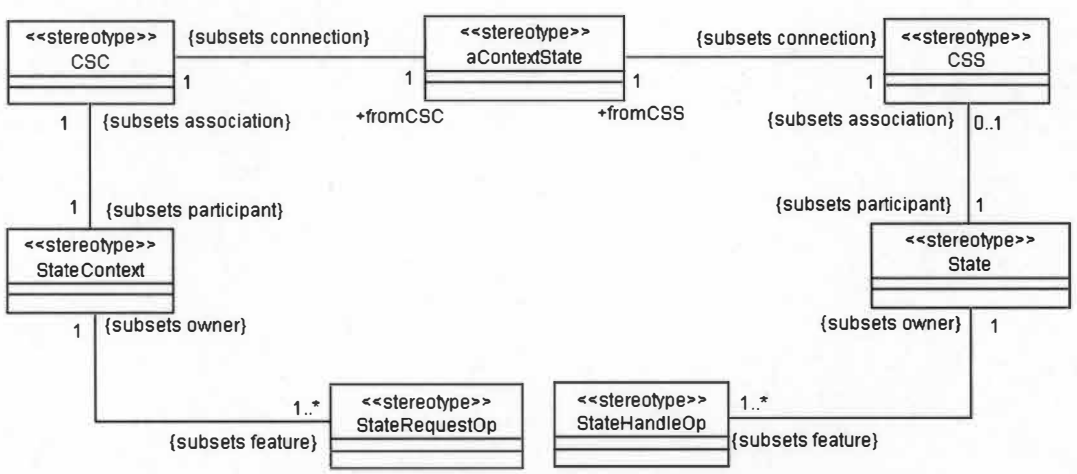

Figure 29. UML profile representing the State design pattern.

Graphical constraints are not enough to express all the restrictions of the design pattern, so OCL constraints are used as well. For instance, to ensure that there is a unidirectional aggregate association from a class stereotyped with *StateContext* to

a class stereotyped with *State*, we add the following OCL constraint to the *aContextState* stereotype:

inv: self.cSS.isNavigable and not self.cSC.isNavigable and

self.cSC.aggregation = AggregationKind::aggregate

Example Class Diagram Instantiating the Design Pattern

Now that a UML profile representing the State design pattern is available, a software developer can design a class diagram based upon this profile by using the stereotypes defined there. As an example, consider a class that represents a network connection whose behavior changes depending on the state of the connection [25]. From the description of the system it is obvious that the State design pattern can be applied to its design. Therefore, we can draw a class diagram using the structure of the State design pattern as a template and come up with a class diagram such as the one in Figure 30.

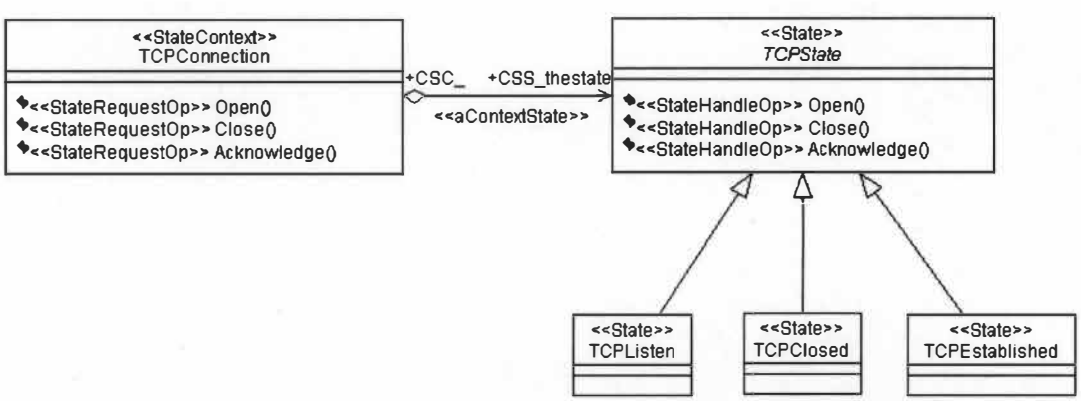

Figure 30. A class diagram based upon the State design pattern profile.

All the elements in this class diagram that are relevant to the UML profile representing the State design pattern are tagged with the appropriate stereotypes. This brings us to the notation used to denote the stereotype for association ends. They are

given as part of the association end's role name followed by an underscore character. The format of the role name is

Stereotypename_Rolename

Both the stereotype name and the role name are optional, but the underscore character must be present if the stereotype name is provided, even if there is no role name. The association connecting the classes *TCPConnection* and *TCPState* in Figure 30 shows how the stereotype name and role name will look like. Unlike association ends, stereotypes for the other common elements in a class diagram, such as classes, attributes, operations, associations, generalizations, and dependencies can be provided in the stereotype field for the respective element in a modeling tool such as Rational Rose. A limitation of the current prototype is that the tool cannot check the body of an operation in the class diagram, such as the one in Figure 28, so validation is currently limited to the structural aspect of the model as well as OCL constraints.

The above class diagram is a valid instance of the State design pattern profile, and so executing the tool on this UML model using model instantiation checking will yield no model checking errors. However, if none of the operations in the *TCPConnection* class are stereotyped, or if the association is bidirectional instead of unidirectional, then the errors will be caught by the tool at the graphical constraint or the OCL constraint level respectively. As with class diagram refinement, by using a UML profile to represent design pattern specifications software developers are able to modify the metamodel depending on their interpretations of the design patterns.

CHAPTER VIII

CONCLUSION

The work in this thesis attempts to cover a broad area, especially to support the entire UML specification, and given constraints on time and resources there is only so much that can be implemented into the current version of the tool. Therefore this chapter will provide a list of limitations of the tool as well as a list of things that can be done in the future to further improve the functionality and the use interface of the tool. Lastly we will summarize the conclusions obtained from this work.

Limitations

Here is a list of the limitations to the tool that are either beyond the scope of our work or limited due to time and resource constraints:

- The diagram parsers are written specifically to read XML files in the XMI format using the UML 1.3 DTD. If the DTD of a different UML version is used (for example, 2.0) our tool will not work. This is because the parsers are hand-written to follow the DTD that is currently used by the most popular commercial UML modeling software, Rational Rose. To solve this problem, one would have to write an XML parser that would read the DTD before reading the XML file, but then the software would need to map what it reads into the appropriate data structures or generate the data structures dynamically based on the DTD. Whatever the method used to solve this problem, the complexity of the solution and the time needed to implement it is beyond the scope of this work.

- One important assumption throughout the tool is that an association will have exactly two association ends, even though the UML metamodel specifies that associations can have two or more association ends. This assumption is valid for almost all UML class diagrams in practice and it makes the implementation of many algorithms much easier. The tool will usually abort execution or display an error message if the number of association ends for some association is not exactly two.
- Association classes are not supported in the tool mainly because of the lack of support for them in the translation schema from class diagram into AsmL. The XML parser in the tool can read association classes but they will be ignored.
- As mentioned before, the tool does not support multiple inheritance in class diagrams well because AsmL does not support multiple inheritance. This feature is one of the more important items in the list of future work since it is a common occurrence in UML models.
- Support for translating diagrams other than class diagrams, sequence diagrams, and state chart diagrams are not present due to lack of time and resources. The UML specification is large; thus it is not possible to support all the diagrams during the course of this work, but is left as future work.
- Some operations in OCL, such as *isUnique*, *sortedBy*, *any*, *one*, and so on are not supported in the current version of the tool. These operations were introduced in version 1.5 of the UML specification.
- The OCL parser currently does not support constraints or pre- and post-conditions for operations because to support them we would also need a

way to express the functionality of these operations in some way, such as through OCL or some other action semantic language. This task is beyond the scope of this work.

- Related to the previous limitation is the lack of proper type-checking of operation parameters.
- Other less frequently used class diagram and OCL features such as qualifiers and templates are not supported by the tool.
- In message order verification, concurrency in sequence and state chart diagrams is not handled because of the method used (graph search) to perform the validation test.
- There is no support for history states, submachine states, and other more advanced features of state chart diagrams at this point.
- Errors when parsing an OCL constraint are limited to the default error messages displayed by the code produced from the ANTLR parser generator. In order to implement the basic functionality of the OCL parser module, there was insufficient time to implement the complex error recovery grammar rules and actions within the ANTLR grammar specification for the OCL parser. This will be a high-priority item in the list of future work.

Future Work

Due to time and resource constraints, not everything could be accomplished during the design and implementation of the model validation tool. Here are some of the items that could be implemented in the future:

- Find a way to support multiple inheritance in UML class diagrams.

- Provide more helpful and detailed error messages throughout the tool, especially for parse errors in OCL constraints.
- Gradually add support for the remaining features or elements of the UML and OCL specification to the tool.
- Re-implement message ordering verification so that it is easier to perform the same task while at the same time being able to automatically evaluate guards and operation parameter values.

Conclusion

In this work we proposed a software tool that validates UML models in various ways, including model instantiation checking through the use of abstract state machines and message ordering verification through the use of message graphs and a graph searching algorithm. A translation schema from UML class diagrams into AsmL specifications as well as the support of OCL in AsmL via a library of OCL operations implemented in the language allows us to perform model instantiation checking on different levels. This resulted in applications of the tool to areas such as class diagram refinement and design pattern profile checking. Software developers can use this tool to aid them in the validation of their UML models and because of the tool's support for UML profiles, the tool can be applied to a specific domain depending on the profile given as input to the tool.

Appendix A

EBNF for Notations in UML Models

1) EBNF for syntax of OCL constraints in the UML model:

```
constraint := "BeginOCL" (constraintDef | constraintBody)+ "EndOCL"

constraintDef := "def" (NAME)? COLON (letExpression)*

constraintBody := stereotype (NAME)? COLON oclExpression

stereotype := "pre" | "post" | "inv"

oclExpression := ((letExpression)* "in")? expression

expression := logicalExpression

letExpression := "let" NAME (LPAREN formalParameterList RPAREN)?
   (COLON typeSpecifier)? EQUALS expression

formalParameterList := (NAME COLON typeSpecifier (COMMA NAME COLON
   typeSpecifier)*)?

ifExpression := "if" expression "then" expression "else" expression
   "endif"

logicalExpression := relationalExpression (("and" | "or" | "xor"
   | "implies") relationalExpression)*

relationalExpression :=  additiveExpression ((EQUALS | GREATERTHAN
   | LESSTHAN | GTE | LTE | NOTEQUALS) additiveExpression)?

additiveExpression :=  multiplicativeExpression ((PLUS | MINUS)
   multiplicativeExpression)*

multiplicativeExpression := unaryExpression ((STAR | DIV)
   unaryExpression)*

unaryExpression := ("not" | MINUS)? postfixExpression

postfixExpression := primaryExpression ((DOT | ARROW) propertyCall)*

primaryExpression := literalCollection | literal | propertyCall
   | LPAREN expression RPAREN | ifExpression

propertyCallParameters := LPAREN (declarator)?
   (actualParameterList)? RPAREN

literal :=  STRING | number | POUND NAME

typeSpecifier := simpleTypeSpecifier | collectionType;

collectionType := collectionKind LPAREN simpleTypeSpecifier RPAREN

simpleTypeSpecifier := pathName | oclType

literalCollection := collectionKind LBRACE (collectionList)? RBRACE

collectionList := collectionItem (COMMA collectionItem)*
```

```
collectionItem :=  expression (DOTDOT expression)?

propertyCall := pathName ("@pre")? (qualifiers)?
   (propertyCallParameters)?

qualifiers :=  LBRACK actualParameterList RBRACK

declarator := NAME (COMMA NAME)* (COLON simpleTypeSpecifier)?
      (SEMI NAME COLON typeSpecifier EQUALS expression)? BAR

actualParameterList := expression (COMMA expression)*

pathName := NAME (DBLCOLON NAME)*

collectionKind := "Set" | "Bag" | "Sequence" | "Collection"

oclType := "OclType" | "OclAny" | "Real" | "Integer" | "String"
   | "Boolean"

boolType := "true" | "false"

number := INT | REAL | boolType | oclType

INT := (DIGIT)+

REAL := (DIGIT)+ (('.' (DIGIT)+) |
   (('e' | 'E') ( '+' | '-' )? (DIGIT)+))+

NAME := ALPHA (ALPHA | DIGIT)*

STRING :  '\'' (ESC | ~('\\'|'\''))* '\''

DIGIT :=  '0' | .. | '9'

ALPHA :=  'a' | .. | 'z' | 'A' | .. | 'Z' | '_'

ESC := '\\' ('n' | 'r' | 't' | 'b' | 'f' | '"' | '\'' | '\\' |
   (('0' | .. | '3') ('0' | .. | '7') ('0' | .. | '7')? )?
   | ('4' | .. | '7') ('0' | .. | '7')?))
```

Note: The tilde (~) means the set of ASCII characters excluding the characters inside the parentheses.
Note2: Comment lines begin with two dashes ("--") and is not represented in the grammar.

2) EBNF for syntax of slot values in the UML model:

```
slotTag := "BeginSlot" slots "EndSlot"

slots := oneSlot (SEMI oneSlot)*

oneSlot := NAME EQUALS slotValue

slotValue := NAME | INT | REAL | STRING | boolType
```

BIBLIOGRAPHY

[1] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[2] M. Anlauff. XASM - An Extensible, Component-based Abstract State Machines Language. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines - Theory and Applications*, volume 1912 of LNCS, pages 69–90. Springer.

[3] E. Börger, A. Cavarra, and E. Riccobene. An ASM Semantics for UML Activity Diagrams. In *Algebraic Methodology and Software Technology, 8th International Conference, AMAST 2000, Iowa City, Iowa, USA, May 20-27, 2000*, volume 1816 of LNCS. Springer.

[4] E. Börger, A. Cavarra, and E. Riccobene. Modeling the Dynamic of UML State Machines. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines - Theory and Applications*, volume 1912 of LNCS, pages 223–241. Springer.

[5] A. Cavarra, E. Riccobene, and P. Scandurra. Integrating UML Static and Dynamic Views and Formalizing the Interaction Mechanism of UML State Machines. In E. Börger, A. Gargantini, and E. Riccobene, editors, *ASM 2003*, volume 2589 of LNCS, pages 229–243. Springer.

[6] I. Ober. An ASM Semantics of UML Derived From the Meta-model and Incorporating Actions. In *Proceedings of Abstract State Machines 2003. Advances in Theory and Practice: 10th International Workshop, ASM 2003*, volume 2589 of LNCS, pages 356–371. Springer, 2003.

[7] W. Shen. *The Application of Abstract State Machines in Software Engineering*. PhD thesis, Dept of EECS, the University of Michigan, September, 2001.

[8] AsmL home page. [cited 2005 Mar 2], Available at: http://www.research.microsoft.com/foundations/asml.

[9] The Object Management Group(OMG). XML Metadata Interchange (XMI) specification.

[10] The Object Management Group(OMG). Unified Modeling Language Specification, version 1.5, March 2003.

[11] World Wide Web Consortium (W3C). Document Object Model (DOM) Level 2 Core Specification, version 1.0, November, 2000.

[12] Simple API for XML (SAX). [cited 2005 Feb 14], Available at: http://www.saxproject.org.

[13] The XML C parser and toolkit of Gnome. [cited 2005 Feb 14], Available at: http://www.xmlsoft.org.

[14] OASIS XML Conformance Technical Committee. [cited 2005 Feb 14], Available at: http://www.oasis-open.org/committees/xml-conformance.

[15] MinGW - Minimalist GNU for Windows. [cited 2005 Feb 20], Available at: http://www.mingw.org.

[16] Xerces C++ Parser. [cited 2005 Feb 20], Available at: http://xml.apache.org/xerces-c/index.html.

[17] eXtensible Abstract State Machines (XASM). [cited 2005 Feb 20], Available at: http://www.xasm.org.

[18] Dresden OCL toolkit. [cited 2005 Feb 14], Available at: http://dresden-ocl.sourceforge.net/.

[19] F. Finger. *Design and Implementation of a Modular OCL Compiler*. PhD thesis, Dresden University of Technology, 2000.

[20] ANTLR Parser Generator. [cited 2005 Mar 1], Available at: http://www.antlr.org/.

[21] The Object Management Group(OMG). Meta-Object Facility Specification, version 1.4, April 2002.

[22] W. Shen and W. L. Low. Using Abstract State Machines to Support UML Model Instantiation Checking. In *Proceedings of The IASTED International Conference on Software Engineering*, 2005.

[23] W. Shen and W. L. Low. Using the Metamodel Mechanism To Support Class Refinement. Presented at *10th IEEE International Conference on Engineering of Complex Computer Systems*, 2005.

[24] R. France, D. Kim, S. Ghosh, and E. Song. A UML-Based Pattern Specification Technique. *IEEE Transactions on Software Engineering*, volume 30, number 3, pages 193-206. IEEE Computer Society. March 2004.

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