Vrije Universiteit Brussel – Belgium

Faculty of Science

In Collaboration with Ecole des Mines de Nantes – France

EMOOSE 2001

FORMALIZATION OF UML USING ALGEBRAIC SPECIFICATIONS

A thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science

By: Liang Peng

**Advisor: Annya Romanczuk-Réquilé (Ecole des Mines de Nantes)
& Jean-Claude Royer (IRIN – Université de Nantes)**

Acknowledgements

First of all, I would like to thank all the people who helped me in the development of this thesis. Thank you! ;-)

I would like to thank my advisors Annya Romanczuk-Réquilé and Jean-Claude Royer for the great support devoted to me during the whole process of this thesis work. It is very important and helpful for me that we can have a technical meeting once a week or two weeks, I will remember all the memorable time we had together.

Thanks to all the members of the Information Department of EMN, especially Object Group, for their help and support during the six months. In particular, thanks to Christine Violeau, Andres Farias Riquelme for setting up the necessary things when I began my thesis in EMN, and also later.

Thanks to Hong Zheng and Janick le Hetet for their encouragement and kind help when I got into trouble.

I would like also to thank Zhiqiang Lu, Wen Song, Xiangke Wang, Li Zhuang, Yi Chen, Xiaomei Li, Fuming Liu, Huixue Zhao and Zhiyu Qian, all my Chinese friends I met in EMN and France, and the great support give by them, the great Chinese dishes we have had every weekend. They make me not feel so lonely living here.

Finally, very special thanks to my dear parents and my sister who, even so far, gave me great support, encouragement and love during my whole study period in France.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS III

TABLE OF CONTENTS V

LIST OF FIGURES IX
LIST OF TABLES XI

ABSTRACT XIII

CONTEXT OF MY WORK 15

INTRODUCTION 17

CHAPTER ONE 21

MOTIVATIONS AND DIFFICULTIES TOWARDS UML FORMALIZATION 21

1.1 WHAT DO "*SEMANTICS*" REALLY MEAN? 21
1.1.1 What is special towards UML semantics? 22
1.2 UML SEMANTICS: CURRENT STATUS 23
1.3 DIFFICULTIES TO UML FORMALIZATION 24
1.4 MOTIVATIONS TO UML FORMALIZATION 25
1.5 WHAT IS A FORMAL UML SEMANTICS GOOD FOR? 27
1.6 SEMANTICS VARIETY 28

CHAPTER TWO 29

SEMANTIC FORMALIZATION BACKGROUND 29

2.1 UML PRECISE GROUP 29
2.2 OO FORMALIZATION METHODS CLASSIFICATION 30
2.3 FORMAL LANGUAGES CLASSIFICATIONS 30
2.4 SOME DEFINITIONS 32
2.4.1 Definitions in the Context of Formal Languages 32
2.4.2 Definitions in the Context of UML Dynamic Models 32
2.5 OBJECT ORIENTED ANALYSIS AND DESIGN FORMALIZATION APPROACHES 33
2.5.1 The Maude System 33
2.5.1 The Coloured Petri Nets 35
2.6 CONCLUSIONS AND SUMMARY 41

CHAPTER THREE 43

UML AND ADT: A SEMANTIC FRAMEWORK PROPOSITION 43

WHAT IS MIXED SYSTEM? 43

WHY "FORMAL SPECIFICATIONS OF THE MIXED SYSTEMS"? 44
ALTERNATIVE SEMANTICS PROPOSITIONS 44
3.1 MAIN POINTS CONSIDERED IN THE FRAMEWORK COMPOSITION 45
3.1.1 The Formalization Method Chosen........ 46
3.1.2 The Formal Language Chosen........ 46
3.1.3 ADT Structure........ 47
3.2 A CONCRETE EXAMPLE AND ASSUMPTIONS 49
3.2.1 Terms Specification in Translation Rules........ 49
3.2.2 Example Description........ 50
3.2.3 Some Assumptions........ 56
3.2.4 The Description of Work process........ 58
3.3 THE TRANSLATION RULES FROM UML TO ADT 59
3.3.1 Static Class Translation Rules 60
3.3.2 Dynamic Class Translation Rules........ 62
3.3.3 Composite System Translation Rules........ 66
3.4 CONCLUSION AND SUMMARY 69
3.4.1 Summary table........ 70

CHAPTER FOUR 73

TECHNOLOGIES SUPPORTING THE SEMANTIC FRAMEWORK 73

4.1 THE PRACTICAL CONTEXT TO APPLY THE FRAMEWORK 73
4.2 RELATED TOOLS 74
4.2.1 Larch Prover........ 74
4.2.2 The Rational Rose UML CASE Tool 78
4.2.3 The Unisys Rose XML Toolkit........ 79
4.2.4 XML4J parser........ 80
4.2.5 XMI2LP translations Tool........ 80
4.3 XMI (XML METADATA INTERCHANGE) STANDARD 81
4.3.1 What is XML and Its Capabilities?........ 81
4.3.2 What is XMI?........ 85
4.3.3 Why XMI?........ 86
4.3.4 XMI file framework........ 88
4.4 CONCLUSION 90

CHAPTER FIVE 93

CONCRETE IMPLEMENTATION OF THE SEMANTIC FRAMEWORK........ 93

5.1 ACCESS XMI INFORMATION 93
5.2 FOR SEQUENTIAL COMPONENT 95
5.2.1 Tools Architecture........ 95
5.3 FOR COMPOSITE SYSTEM 97
5.3.1 Driving technologies........ 98

5.3.2 Tools Architecture ***100***
5.4 EXTENSIONS AND RELATED WORKS 102

CHAPTER SIX **105**

CONCLUSION AND FUTURE WORK **105**

6.1 CONTRIBUTION........ 105
6.2 CONCLUSION 106
6.3 FUTURE WORK........ 106
6.3.1 About Tools ***106***
6.3.2 About Semantics Coverage ***107***

REFERENCES **109**

List of Figures

Figure 1: Maude Formalization of Statechart Diagram in Figure 2 35
Figure 2: The matching Statechart Diagram.. 35
Figure 3: Example UML Collaboration Diagram.. 37
Figure 4: CPN Segment for Asynchronous Active Objects 38
Figure 5: CPN Segment for Periodic Active Objects... 39
Figure 6: CPN Segment for Entity Objects 40
Figure 7: Synchronization Statechart Diagram of the Ticket-Purchase System...... 50
Figure 8: The Class Diagram of Bank Component.. 51
Figure 9: The Statechart Diagram of Bank Component 52
Figure 10: The Class Diagram of Reservation Component....................................... 53
Figure 11: The Statechart Diagram of Reservation Component 53
Figure 12: Collaboration Diagram of the Composite System 54
Figure 13: Statechart Diagram of Composite System.. 55
Figure 14: Synchronization Representation 58
Figure 15: Merging of two Sequential Component Operations................................ 68
Figure 16: The Tool's Work Process.. 74
Figure 17: The menu option to run the translation from UML to XMI.................... 79
Figure 18: XML Element structure.. 82
Figure 19: Document Object Model... 85
Figure 20: XMI Simplified schema.. 86
Figure 21: Open Interchange with XMI 88
Figure 22: XMI2LP Tool Architecture 95
Figure 23: Concurrent Product Generation Process 97
Figure 24: Algorithm for the Concurrent System Generation.................................. 99
Figure 25: CS Tool Architecture 101

List of Tables

Table 1: Larch Prover Syntax Example 48
Table 2: Larch Prover Syntax Description 48
Table 3: Formal Specification for Static Class 70
Table 4: Formal Specification for Dynamic Class 71
Table 5: Formal Specification for Composite System 71
Table 6: A Larch Prover Sample Proof 76
Table 7: XML Document for Electronic Catalog 83
Table 8: Document Type Definition 84
Table 9: UML DTD Fragment 88
Table 10: Simple XMI framework 90
Table 11: Java code for accessing XMI information 94

Abstract

UML is a standard graphical modeling language; it presents a set of notation for visualizing, specifying, constructing, and documenting a software-intensive system. And it demonstrated that it is a well-suited approach to analyze and design complex systems such as industrial ones. But UML is not a formal language; it is composed by graphical notation, natural language, mostly means English, and formal language, OCL, for example. It gives a syntactic description of the language but not a complete and precise specification of it semantics. This can always lead to confusion and different understanding when we analyze a model. To improve software reliability and reusability, the use of formal specification is necessary but it may be difficult. My thesis work is trying to present an algebraic semantics framework for UML, focus on the dynamic part, and defining a set of translation rules between UML model and algebraic specification. At last, we use Larch Prover – a theorem prover, to validate and verify the formal specification of UML models, which can lead to early detection of errors and inconsistencies in the software development process. Meanwhile, tools are needed to assist the formal specification process, besides to prove or to verify some parts. We present the implementation of the translation tool using XMI standard and the XML4J parser based on the Rational Rose UML CASE tool. And In my thesis, I propose a method to formally specify concurrent and communicating components with data in UML.

KEY WORDS

UML, Formalization, Algebraic Specification, Semantics, XML, XMI, Larch Prover, Abstract Data Type, Dynamic Models.

Context of My Work

Before navigating readers' reading along my thesis report, we had better take a general view about the context of my thesis work. Such as, what stuff should you know and understand before you go ahead with my thesis report? What technologies dose it use? What aspects do it focus on? What benefits and advantages do they really make? etc. I will try to present a general description of these knowledge as simply and concisely as I can.

What is UML?

The Unified Modeling Language (UML) is a graphical language for visualizing, specifying, constructing and documenting the artifacts of software systems. Such as Enterprise information system, banking and financial services, telecommunications, distributed web-based services etc. The UML was originally derived from the object modeling languages of three leading object-oriented methods: Booch, Object Modeling Technique (OMT) and Object-Oriented Software Engineering (OOSE). It was first added to the list of OMG adopted technologies in 1997, and has since become the industry standard for modeling objects and components.

UML demonstrated that it is a well-suited approach to analyze and design complex systems, It gives us a way to write a system's blueprint, covering conceptual things, such as business process and system functions into concrete and visual model, which is more understandable and easy to control and communicate between different development teams.

What are Static and Dynamic Part in UML?

UML is used to model a software system, and normally we can look at a software system from two perspectives, the Static and Dynamic point of view. Static parts focus on system structure information, while Dynamic parts focus on the dynamic behavior information. And in UML, we describe the Static Part of a system by using Class Diagram, Object Diagram, Component Diagram, Deployment Diagram, and illustrate the Dynamic information by using Sequence Diagram, Collaboration Diagram, Statechart Diagram, and Activity Diagram. In my thesis research field, we focus on the dynamic part formalization work, especially dynamic components composite system.

What is UML Formalization?

In my work, Formalization means to give a precise algebraic specification to UML modeling language, and my subject focus on the Dynamic Part formalization. Static Part formalization is a thesis subject of previous EMOOSE student Aline, who has done a perfect work in this field, and meanwhile, what she has completed provides a good basis for my work. I mean I will reuse some static part formalization paradigm, concept and idea in my work. For example, for the translation of UML class diagrams.

What is Algebraic Specification?

The core issue of my thesis is defining a set of mapping rules between UML diagram and Algebraic Specification syntax (Abstract Data Type mostly). So what is that? Algebraic Specification means using the algebraic notations, and formulas to formally define a software system, some Algebraic Specification Languages (e.g. Larch Prover Language and ACT-ONE), there are capable to specify software systems. And Algebraic Specification Languages are assumed to be more compatible for Abstract Data Type specifications.

Introduction

Object Oriented analysis and design languages are very popular in the industrial developer community. In this field, the Unified Modeling Language (UML) [RJB99] is becoming a *de facto* standard. Using UML, one may represent most of the current applications for software systems. To improve software reliability and reusability the use of formal specification is necessary. It is currently recognized that the UML language suffers from a lack of formal semantics [FEL97, Eva98]. Many Object Oriented (OO) methods, including those from which UML is derived, suffer from a lack of a precise semantics. This can lead to confusions and different interpretations when analyzing a model. Consequently, this decreases the ability to develop tools and guidelines to help the specifier. Tools are helpful to assist the formal specification process, but also to prove or to verify software. Designers require a more formal semantics to be able to check, to validate and to refine UML models.

This thesis presents an approach to translate some UML models into formal specifications, and then verification tools may be used to early validate the models. We deal only static classes, active classes, statecharts and collaboration diagrams. And there are several important difficulties to provide a formal approach for UML. The first is the need to construct UML diagrams with formal notations to get formal UML components. The second is to give a semantic framework to the UML diagrams and last is to verify or prove properties. And our approach reuses previous experiences with the LOTOS language [PCR99], symbolic transition systems and algebraic specifications [Roy01b] and formal software component [CPR01a] in order to complete UML diagrams. Our aim is to suggest method to propose algorithms and tools to help the specifier. Because of the thesis time limit, we do not address all these problems in my dissertation, and we will restrict our study to the translation of UML sequential and concurrent components with data into formal specification, and reuse the static part formalization paradigm, which has been done by Aline, a previous EMOOSE student, following the academic year 1998-1999.

From the technical point of view, the main purpose of this thesis is to propose a semantic framework in order to support the formalization of the main dynamic model elements of UML using Algebraic Specifications. Algebraic Specifications are used to describe abstract data types (ADT) as well as concurrent system.

The semantic framework is based on a set of mapping rules defined to the translation from UML elements to algebraic specifications. These translation rules are written in accordance to the syntax and semantics of each UML model element considered. The semantics of the model elements was evaluated considering the UML metamodel [UML99] and UML model [BRJ99a]. Therefore the result of the translation process is an ADT specified to each model element through the corresponding mapping rules established to it.

To establish these translation rules for generating the formal specifications, some other approaches on Object Oriented Analysis and Design formalization, focusing on UML, were evaluated and taken into account. In [LB98] a semantic framework for part of UML, named RAL is presented. Another algebraic approach using Larch Shared Language (LSL for short) was also analyzed. It is described in [HHK98] being a formal, modular approach to specify the semantics of object-oriented models expressed in UML. LSL is an algebraic language, which in conjunction to LP and other technologies composes the Larch family of languages and tools.

In both approaches a great importance is given to model theory composition in order to describe models and sub-models. This allows the establishment of constraints among model elements. The level of granularity considered to the formal specifications is also an important aspect outlined in both approaches.

In the semantic framework presented here it is adopted an intermediate degree of granularity. Formal descriptions are used to describe classes, class interface, state machine and associations as well as some other constructs. It is also considered the idea of constraints at the model level what is achieved through general descriptions grouping some individual model elements.

The implementation of the semantic framework is undertaken considering the integration of different technologies: Rational Rose UML CASE tool, Unisys Rose XMI tools adding-on for Rational Rose, XML4J java package with Java programming language and Larch Prover theorem prover. Rational Rose UML CASE tool is instantiated with the UML grammar to build class diagrams, statechart diagram and collaboration diagram, etc. The Unisys XMI tool is used in Rational Rose allowing the automation of the XMI file generation from a UML informal model to well-formed XMI file format. The XML4J is a validating XML parser written in 100% pure Java, which contains a set of API for parsing, generating, manipulating, and validating XML documents, the tool XMI2LP for generating LP algebraic specification from XMI file is developed based on this Java package. Larch Prover reads then these formal algebraic specifications in the form of abstract data types, and checks and proofs can be performed on them resulting in error detection on the design phase.

As the work of this dissertation considers just part of dynamic models of UML, and the formal method presented here can be also extended in future in order to cover other dynamic UML constructs.

Structure of the Dissertation

Chapter one presents an overview of the current state in UML semantics and gives some motivations and difficulties towards UML semantics formalization.

Chapter two shows the State of the Art in UML semantics formalization domain. It presents formal methods and formal languages, which can be used to the formalization of Object Oriented analysis and design languages. In the core of the chapter is the presentation of two formalization approaches existent, one of them focusing specifically in UML, and the other one, The Maude System, showing a more general formalization method that can be applied to any OO design and analysis language.

In chapter three the core of the Thesis is described. This chapter shows the main points considered to compose the semantic framework, as the formal syntax followed, the process to determine the translation rules, the structure of an ADT, going then deep in the description of the set of mapping rules for each UML model element considered in the formalization. The translation rules are described based on the semantics aspect that leads to their definition.

As explained in this introduction, the implementation of this semantic framework takes into account different technologies that need to be well integrated in order to allow the framework working. Each of these technologies employed and the way taken to their integration is explained in chapter four.

Chapter five gives then the link between the theoretical parts presented in chapter three and the practical aspects detailed in chapter four. This chapter takes a concrete example drawn in the CASE tool developed as part of this work and shows the results of the translation process performed to it. Therefore the formal specifications in the form of LP syntax resultant from the implementation of the translation rules are referenced. After the translations are done, this chapter goes on presenting some inconsistencies and diagram errors, which can be detected in UML models through the use of the Larch Prover theorem prover.

Chapter six ends up by giving some conclusions, which are taken from the development of this work and presenting contributions and future work that can be used to improve the semantic framework. And also the practical domain this application can be used and some lack of the translation tools will be addressed here.

Chapter One

Motivations and Difficulties towards UML Formalization

The Unified Modeling Language (UML) is a graphical language for visualizing, specifying, constructing and documenting the artifacts of software systems. The UML was originally derived from the object modeling languages of three leading object-oriented methods: Booch, Object Modeling Technique (OMT) and Object-Oriented Software Engineering (OOSE). It is more compatible to be used to model object-oriented software systems. The Object Management Group (OMG) has approved UML in November 1997 as the standard notation for object-oriented analysis and design, and has since become the industry standard for modeling objects and components. At the end of 2000, the OMG has issued a Request For Information (RFI) with regard to UML 2.0.

The main motivation towards UML formalization is that its semantics is not precisely described through UML official documents and books. In this chapter other motivations and some difficulties encountered in order to achieve UML semantics formalization are presented.

1.1 What do "*Semantics*" really mean?

Before we talk about UML semantics, we had better make it clear the semantic of term "semantics". What does it mean? Today, a lot of confusion arises from the fact that the word "semantics" itself has many different semantics! Developers tend to use the word "semantics" when they talk about the behavior of a system they develop. This kind of usage is almost contradictory to the semantics in scientific areas like Mathematics or Logic. There, "semantics" is a synonym for "meaning" of a notation - this is regardless of whether this notation deals with structure or behavior information.

Basically, a semantics is needed if a notation (syntax) is given or newly developed, and its meaning needs to be defined. Almost all approaches define the semantics of its elements by relating it to another already well understood language.

This is comparable to natural languages. For example Chinese can be (roughly) understood if a Chinese-English dictionary is available. Of course grammar, or the definition of how elements of a language are modified and grouped together, also need to be mapped. In computer science, the pattern is similar. A new language is given a meaning in three steps:

- Defining precisely the syntax of the new language, which characterizes all the possible

expressions of that language
- Identifying a well understood language, herein called the semantics language, and
- Defining a mapping rules or explanation from expressions in the syntax of the new language to the semantics language.

1.1.1 What is special towards UML semantics?

UML does have some specific characteristics, which makes the task of semantics definition interesting:

- A substantial part of UML is visual and diagrammatic.
- UML is not for execution, but for modeling, thus incorporating abstraction and under-specification techniques.
- UML is combined of a set of partially overlapping sub-notations.
- UML is of widespread interest.

Whereas the last issue leads to the sociologically interesting question, how to reach agreement for a semantics definition, the other three topics lead to problems of a technical nature. The fact that a large part of UML is diagrammatic makes it somewhat more difficult to deal with its semantics, but it is not a problem in principle. Currently, its semantics is explained in English: UML semantics is ambiguous and imprecise. We speak of a formal or precise semantics for UML if the semantics domain of his translation is a formal language and, very important, the translation itself is precisely defined. This goal can be achieved, as several graphic formalisms, like Statecharts, Petri-Nets, or dataflow-diagrams have shown. The first step for UML is to precisely define its syntax. In the UML standard, this has been done by using the meta-model approach, which in the UML documents is mainly used to describe the abstract syntax of the UML [UML99] itself. Thus a meta-model for diagrams replaces the abstract syntax tree of textual notations.

The usage of UML as a modeling language and not as a programming language has an important impact that is often poorly recognized. A UML model is a visual abstraction of the real system to be developed. The model is used to capture important properties, but to disregard unimportant ones. As an effect, a UML model typically has a set of more than one possible implementation. A semantics definition must reflect this by making the under-specification of the model explicit.

Third, the UML is composed of a set of notations that partially overlap. For example [DW98] shows how (a subset of) the state diagram notation can be used to express the same information that could be expressed in terms of pre/post conditions on operations in a class diagram; but there are other aspects of state diagrams which cannot. This introduces another problem that semantics definitions for each of the UML notations need to be consistent with each other. Only then will an integrated use of these notations be feasible. To check the consistency of semantics definitions, it is necessary either to have a common semantics domain for all of them, or to establish precise mappings rules between different semantic domains.

1.2 UML Semantics: Current Status

The current status of the UML semantics is that it is described in an informal manner. The 'UML Notation Guide' document gives an overview on the concepts, and the 'UML Semantics' document presents the abstract syntax together with context-sensitive conditions in form of class diagrams and OCL expressions. Both documents as well as the semi-official books by Booch, Rumbaugh, and Jacobson do not use formal techniques for explaining the semantics.

Concerning the reference documents, when studying the UML and especially the UML semantics, one has to take into account the official OMG UML definition, especially the 'UML Notation Guide' and 'UML Semantics' documents. But the problem is that the UML is an evolving language. Therefore many versions of these documents exist. The current recognized version is version 1.3 [UML99] but there is already a call for contributions for version 2.0. In addition, there are also many books and papers on the subject, including the semi-official ones by Booch, Rumbaugh, and Jacobson, especially the 'UML Reference Manual'. Because of publication lead times, laziness of researchers and so on, one has to be very careful when reading a paper or book to identify on exactly which version of the UML it is based.

For example, one is likely to come up with a very different semantics for signals specification, when you read the UML standard or the UML reference guide:
- Signals ... have no operations.
UML Notation Guide, Version 1.3, page 3-138, line -4..-2. [UML99]
- A signal ... may have operations.
UML Reference Manual, page 428, line 3. [BRJ99b]

UML encompasses structural and behavioral aspects in order to describe OO software systems. Even being a de facto standard, its semantics are semi-formal described. In [UML99], the *UML semantics* document, version 1.3 (last RTF version), the semantics of the language is described using the metamodel. The metamodel stands a combination of graphical notation, natural language and formal language. It gives a syntactic description of the language but not a complete and precise specification of its semantics.

The graphic part is reflexive using a subset of the own UML notation. The formal language is the OCL (Object Constraint Language) that has been a first approach in order to get a precise description for the UML. It is an assertion language used to describe navigation and constraints for specifying invariants, preconditions and postconditions in UML Diagrams. Although OCL helps in the semantics description being used to the specification of well-formedness rules, it does not provide a basis for controls and validations. Moreover it does not solve some ambiguities in UML interpretations.

UML carries a complex set of notations that as explained do not gain a clear meaning through the metamodel. The UML official documents contain a paper called the "Semantics of UML". However, this paper does not focus much on semantics, but mainly on syntactic issues. The meta-model of UML gives a precise notion of what the abstract syntax is. However, it currently does not cope with semantics. Analogously, the semantics of Java language can not be understood from the context-free grammar (without knowledge of similarly structured languages). Furthermore, context conditions are by no means semantic conditions, but purely constrain the syntax. They give well-formedness rules, e.g. each variable must be defined before use, and without telling you what a variable is. In the UML case, context conditions are usually explained using OCL. A context condition tells us what is constrained, not why it is constrained. The latter is a task of the semantics definition.

1.3 Difficulties to UML Formalization

The lack of a precise formal semantics for the UML is justified in many ways:

- The architects of the language claim: *“the state of the practice in formal specifications does not yet address some of the more difficult language issues that UML introduces” [UML99].*
- Formal specifications are hard to deal with for non-expert users. Developers, users of UML, are not familiar with formal mathematical specifications and because of it they tend to resist to their use.
- To be of industrial use, formal specifications need to be integrated to CASE tools, supporting graphical modeling constructs, in such a way that developers can directly manipulate the OO models they have created to analyze, transform and enhance them.

In contrast to the difficulties showed above, the own authors of the modeling language also recognize the importance of formality. According to [CE97] the authors of the language agree in the sense that it lacks from a precise semantics description, and that its formalization could lead to unambiguous interpretations of the models and could permit extensibility allowing future changes in object-oriented analysis and design.

1.4 Motivations to UML Formalization

UML is not a formal language (even if it is a industry standard); it is composed by graphical notation, natural language (mostly means English) and Formal language (for example, OCL Object Constraint Language). It gives a syntactic description of the language (for example, you can create aggregation relationship between two different classes, but you can not create aggregation relationship from a class to itself, the loop diagram is not allowed.) but UML doesn't present a complete and precise specification of its semantics.

Many motivations are given to justify the importance of formalization. They can be grouped according to some primitives, as: clarity, consistency, correctness and enhancement it can bring to the models. Because of these benefits, formalization is really helpful in forward and reverse reengineering efforts as well as in the restructuring of systems. On the other hand, a really understandable and consistent system is more suitable for reuse. Follows some motivations towards formalization according to the primitives stated.

- **Clarity**

UML is a complex language that holds a really great number of modeling elements. Because of its complexity and lack of precise description, its constructs are not clear defined and the language can lead users to ambiguous interpretations of the models. Formalization can help in clarifying the meaning of UML model elements. In [CE97] it is stated:
“Clarity acts as a reference – if at any point, there is confusion over the exact meaning of a particular UML component, reference can be made to the formal description to verify its semantics.”
A deeper understanding of OO concepts is also gained, allowing the development of more rigorous semantic analysis tools and better use of OO techniques.

- **Consistency**

UML presents nine different diagrams to express different system perspectives. The consistency among these diagrams representing a model can be ensured since all of them are formalized and hence precisely described. This leads to a more complete and unambiguous interpretation of a model, allowing development teams to have a better communication and understanding among them.
Consistency can also be achieved between code and specifications. Having a precise description of the models, implementations can be validated against the design checking if it fulfills the specifications. On the other hand, formalization can also be a bridge from implementation to design in a reverse engineering process.

- **Correctness**

Correctness of the models can be achieved through the application of proofs over the formal specifications. Therefore inconsistencies can be detected. A mapping between the model elements of UML to formal specifications can help in adapting proofs and validations to

CASE tools what leads to early detection of errors in the systems.
The establishment of proofs can be done upon the properties of a system described in UML, forming a basis for future automatic proof techniques.
Moreover with a mapping allowing the generation of formal specifications from informal models it is possible to identify ambiguous and inconsistent structures in the models.

- **Enhancement:**

Enhancement of models is expressed through design refinements. In [EBFLR98] refinement is defined as:
"It is the process by which an abstract model of a system (containing relatively little implementation detail) can be incrementally transformed into a model that can be readily implemented in a specific programming language. At each stage the correctness of the more detailed model must be verified against the abstract model."

As UML is a diagrammatical modeling language, refinement of a UML model implies a process of diagrammatical transformations. In this context, the definition of a set of semantically-based transformation rules is important to provide a set of correct transformations that are equivalencies or enhancements of models. Some properties of models can be deduced and proved through transformations. Proving that one form of the model is equivalent to another can make correct properties arise.

Refinements of models based on transformations are useful not only to support forward engineering as well as reengineering efforts. Model refinements can be helpful in the restructuring of designs.

Design Patterns can be applied in refinement steps being checked for correctness. Once checked, a pattern can be used again and again without having to be re-checked.

Basing in the primitives previously stated and going into detailed explanations, more justifications for formalization can arise. In [FELR97] they say:

- Developers can waste time making considerations over correct usage and interpretation of notations. Because of the informal descriptions provided in reference books, it is not easy to achieve an interpretation that can be considered precise.

- It is difficult to ensure model reviews, rigorous semantic analysis based on informal techniques. In [FELR97] it is stated:

"Review meetings can be further enhanced if the notations used have a precise semantics. The results of model validations and verifications can be presented in reviews as evidence of the quality of the models. Rigorous semantic analysis techniques also facilitate the early detection of modeling errors which considerably reduces the cost of error removal."

- Tool support for OO modeling notations is limited because of the lack of a precise semantics for the constructions of the language. Hence tools stay limited to cover just syntactic concerns.

In [EBFLR98] it is stated that:

“The desire to formalize UML was originally motivated by the overall wish to develop practical, industrial strength, formal methods. The advent of the UML as a likely de-facto industry standard, and its recognition that as a standard it needs to be precisely described, made UML a natural choice for a combined investigation.”

As it can be realized the motivation to formalize OO methods was not originally motivated by UML emergence. Formalization had already been recognized as useful and necessary not only for academic purposes but also for industrial use before UML has appeared. Formalization aims to support reliable and precise modeling language to be used in any context. The advent of UML as a standard OO modeling language made the efforts turned to it.

1.5 What is a formal UML semantics good for?

Free Communication
Semantics of UML is a means to understand how UML should be used, and to ensure that when UML models are communicated amongst different developers, there is a common shared understanding of what they mean. On the other hand, the actual practice of applying UML is necessary to get a feeling for it. A semantics definition is a necessary prerequisite, but certainly not sufficient. Furthermore, it is not necessary to understand the complete language to start using it.

Semantics is a bridge for people who speak the same semantic language (formal or informal) to discuss certain UML properties and improve the notation and use of UML in terms of semantics definition.

Machine Processing
Formal Semantics can be performed for certain automatic tasks by machine. For a machine to process a language, that language must be defined precisely. If it is to perform semantics-oriented tasks, then its semantics must be defined precisely. Examples of semantics-oriented tasks are: model simulation or (partial) execution; checking that different views on a model (class diagrams, invariants, state diagrams, sequence diagrams, etc.) are consistent with one another; checking that the behavior of a superclass is preserved in a subclass, and detecting errors; and so on.

A Benchmark
A precise semantics provides an unambiguous benchmark against which a developer's understanding or a tool's performance can be measured: Does the developer use the notation

in ways, which are consistent with the semantics? Does a tool generate code as the semantics would predict, or does it check the consistency of a model in accordance with the semantics? All those puzzles can be resolved by a precisely semantics framework.

1.6 Semantics Variety

The advantage of having a single, standard semantics for UML is that it is easier for one person to understand another person's UML models, while the advantage of having a variety of semantics is that you can choose what works best in your current project. We believe it is possible to support both standardization and variation. Therefore, 'should UML have a single semantics towards UML model?' is not always positive answer.

Meanwhile, It is important to clarify the purpose of a semantics definition. There may be different semantics definitions to suit different purposes: the definition for explaining semantics to users of the notation may be different to that required to perform sophisticated automatic processing tasks, and both may be different to a semantics definition whose purpose is to demonstrate properties about the language, such as a measure of how expressive it is compared to other languages.

In practice, there are many individuals and groups who can contribute to discussions on the semantics of UML, and many ways for them to disseminate their proposals. No single group or individual has control of the semantics of UML; there will be a variety of semantics for UML. Some will be more popular than others, meaning that more people understand them and use them.

Chapter Two
Semantic Formalization Background

In the previous chapter many motivations were sketched to justify the efforts invested in UML formalization domain. This chapter starts showing some formal methods and languages to support formalization. Afterwards, the main OO analysis and design formalization approaches will be studied, some of them focusing on UML dynamic part are presented.

2.1 UML Precise Group

It is a very important workgroup towards UML formalization work. And Before presenting the formal methods and formalization approaches, it is necessary to point out the importance and contributions of the UML Precise Group in the context of UML formalization.

The UML Precise Group (PUML) was created for two main purposes:

- Investigate the completeness of the UML semantics.
- And develop novel approaches to use UML more precisely.

This group was formed in late 1997. By giving precise semantics to UML, the group intends to develop a formal *reference manual* for this language. In [FELR97] they say:

"A major objective of the project is to develop a formal reference manual for the UML. This will give a precise description of core components of the language and provide inference rules for analyzing their properties. In developing the reference manual we will build upon the semantics given in the UML semantics document by using formal techniques to explore the described semantic base."

In this formal reference manual, the intention is to re-express the formal semantics in terms of a suitably expressive language, which could be a mixture of notations such as an enhanced version of the UML metamodel, the OCL (Object Constraint Language), and precise natural language statements.

2.2 OO Formalization Methods Classification

The classification presented in this section is also a contribution work from some members of the UML Precise Group. In [FELR97] it is presented three general categories for OO formalization methods: *supplemental, OO-extended formal language, and methods integration.*

Supplemental method: In the *supplemental* method, formal statements substitute annotations in the models that are expressed in natural language. This clarifies the meaning of the models, but the semantics of graphical constructs are not necessarily precisely defined.

OO-extended formal language: In the *OO-extended formal language* method, an existing formal notation is extended with OO features. This is the case of Z++ and VDM++, for example. In this case the formal languages are really enriched and, on the other hand, OO concepts need to be formalized in order to be able to be adapted to formal languages. The problem with this method is the considerable gap between model elements representing real world concepts and the mathematical representations in the formal notations.

Methods Integration: *Methods Integration* approach defines the generation of formal specifications from informal OO models. It is stated:

"…the generation of formal specifications from informal models is only possible if there is a mapping from syntactic structures in the informal modeling domain to artifacts in the formally defined semantic domain."

In this case a formal description of the mapping rules becomes essential in order to check if the formal specifications indeed capture the intended interpretations of the informal models.

Mathematical Language is not mandatory
A precise definition of the model semantics domain is usually given either by explicitly defining the notion of "System" using mathematical terms, or by using a formal language, like Z or Object Z, as the semantics language. However, precision does not require the language to be mathematical in the traditional sense.

2.3 Formal Languages Classifications

In [CHS$^+$97] four major underlying models upon which the formal specification languages can be based are described. Follows the identification of these models and examples of formal languages classified in each one of them.

First-order logic and set-theory

According to [CHS$^+$97], this approach can be defined as:

"The first-order logic and set-theory approaches are also often called model oriented because they support the specification of a system by constructing a mathematical model for it."

In this group there are:

- Z language;
- Object-Z (OO extension of the Z notation);
- VDM++ (OO extension of the Vienna Development Method);
- Z++ (OO extension of the Z notation).

Algebraic approach

This approach uses algebraic equations in order to establish the semantics of the operations in a specification. Examples of languages are:

- TROLL;
- Maude;
- AS-IS (Algebraic Specification with Implicit State);
- CASL (Common Algebraic Specification Language);
- Larch;

Petri nets/algebraic nets

This approach is described in [CHS$^+$97] in the following way:

"Petri nets and high-level nets are two representative of the model-based class in the sense that they describe the state of a system by means of places which contain "black tokens" for the conventional Petri nets and structured tokens for high-level nets. A set of transitions which consist of a pre- and a post-condition, describes how the system state changes by consuming and producing tokens in the various places of the net."

Examples of languages in this family are:

- CLOWN (Class Orientation with Nets);
- CO (Cooperative Objects);
- OPN (Object Petri Nets);
- COOPN/2 (Concurrent Object-Oriented Petri Nets).
- CPN (Colored Petri Nets)

Temporal logic

In [CHS$^+$97] it is described as:

"Temporal logics are axiomatic formalisms that are well suited for describing concurrent and reactive systems. A common aspect associated with temporal logics is a notion of time and state."

Examples of languages are:

- TRIO+;
- OO-LTL.

Follows the description of two UML formalization approaches that deal with set-theory (Z) and algebraic formal languages.

2.4 Some Definitions

2.4.1 Definitions in the Context of Formal Languages

Some definitions are necessary in order to understand the following OO analysis and design formalization approaches and the remaining stuff of this document. They are:

What are Terms?
By terms it can be understood as an expression that refers to an object, such as: `sizeof(Array)`.

What is first-order logic?
By first-order logic it is understood that equations can be written using variables that represent all the values that can be extracted from a specific Universe. The equation can then be proved valid by exemplification.

2.4.2 Definitions in the Context of UML Dynamic Models

My thesis work focuses on the Dynamic aspect of UML models. And in order to explain the UML dynamic diagrams, we shortly repeat the explanations of the notations given in the UML semantics for the meta-classes under considerations. The notions considered are: **Action**, **Event**, **Exception**, **Message**, **Method**, **Signal**, **Stimulus**, **Operation**, and **Reception**.

Action: An *action* is a specification of an executable statement that forms an abstraction of a computational procedure that results in a change in the state of the model, and can be realized by sending a *message* to an object or modifying a link or a value of an attribute.

Event: An *event* is a specification of a type of observable occurrence. The occurrence that generates an *event* instance is assumed to take place at an instant in time with no duration.

Exception: An exception is a signal raised by behavioral features typically in case of execution faults.

Message: A *message* defines a particular communication between instances that is specified in an interaction.

Method: A *method* is the implementation of an *operation*. It specifies the algorithm or procedure that affects the results of an operation.

Operation: An *operation* is a service that can be required from an object to affect behavior. An operation has a signature, which describes the actual parameters that are possible (including possible return values).

Signal: A *signal* is a specification of an asynchronous *stimulus* communicated between instances. The receiving instance handles the signal by a state machine. *Signal* is a generalization element and is defined independently of the classes handling the signal. A *reception* is a declaration that a class handles a signal, but the actual handling is specified by a state machine.

Stimulus: A *stimulus* reifies a communication between two instances.

Reception: A *reception* is a declaration stating that a classifier is prepared to react to the receipt of a *signal*. The reception designates a *signal* and specifies the expected behavioral response. A *reception* is a summary of expected behavior. The details of handling a *signal* are specified by a state machine.

2.5 Object Oriented Analysis and Design Formalization Approaches

In this part, some formalization approaches will be sketched, which cover the most aspects in UML dynamic models, such as Statechart Diagram, Collaboration Diagram, Sequence Diagram and Activity Diagram etc. We address not only the formal representation method for these diagrams, but also how to use this formal representation to prove the consistency and correctness among them. Different concept and ideas is our interest for pursuitting an appreciate way in our thesis work later.

2.5.1 The Maude System

Maude is a high-performance reflective language and system supporting both equational and rewriting logic specification and programming for a wide range of applications. Maude has been influenced in important ways by the OBJ3 language, which can be regarded as an equational logic sublanguage. Besides supporting equational specification and programming, Maude also supports rewriting logic computation.

Rewriting logic is a logic of concurrent change that can naturally deal with state and with concurrent computations. It has good properties as a general semantic framework for giving executable semantics to a wide range of languages and models of concurrency. In particular, it supports very well concurrent object-oriented computation. The same reasons making rewriting logic a good semantic framework make it also a good logical framework, that is, a meta-logic in which many other logics can be naturally represented and executed.

Maude supports in a systematic and efficient way logical reflection. This makes Maude remarkably extensible and powerful, supports an extensible algebra of module composition operations, and allows many advanced meta-programming and meta-language applications. Indeed, some of the most interesting applications of Maude are meta-language applications, in which Maude is used to create executable environments for different logics, theorem provers, languages, and models of computation.

Rewriting logic: is a very flexible reflective logic that has very good properties as a logical and semantic framework. It can be interpreted logically or computationally, the latter interpretation-giving rise to an executable specification language implemented as the Maude system. It is a logic of concurrent change that can deal naturally with state and with highly non-deterministic concurrent computations. In particular, it supports concurrent object-oriented computation. These properties of rewriting logic make it an ideal framework in which to formalize UML. Moreover, since Maude is based on conditional rewrite rules, it is very natural to express transformations of UML models.

A rewrite theory is a pair `(T,R)` where `T` is an equational theory and `R` is a collection of labeled and possibly conditional rewrite rules involving terms in the signature of `T`. Rewrite rules are of the form `r: t → t'` and can be applied modulo associativity, commutativity, identity and idempotency axioms. This leads to a large number of possible rewriting paths, which can be controlled by strategies implemented using Maude's reflective capabilities.

Alemán & Toval [AA00] shows how Maude can be used to formalize UML class diagrams. A similar approach can be used for UML behavior models. As an example, the UML statechart shown in Figure 3 is formally specified by a pair `(transitions, hierarchy)` where `transitions` denotes a list of transitions between states, and `hierarchy` represents a state hierarchy (e.g. see formalization in Figure 2). These formal terms are expressed according to the existing UML Statechart formal specification [AA00] at the UML metamodel layer. Once the statechart is formally represented, it can be mathematically manipulated and prototyped. Likewise, rigorous transformations can also be applied. So far, class diagrams, statecharts, and a subset of OCL have been formalized as Maude models.

```
transitions1=
transition (initialState, s1, empty) transition (s2, xTrue, m1)
transition (initialState, s5 empty) transition (s3, s5, m2)
transition (initialState, xFalse, empty) transition (s3, s4, m3)
transition (s1, s2, m2) transition (s4, s5, m2)
transition (s1, s3, m1)
hierarchy1 =
OrState (ST, empty)
OrState (xFalse, empty, simpleState (s1, empty) simpleState (s2, empty))
OrState(xTrue,    empty,    simpleState    (s3,    empty)    simpleState    (s4,    empty)
simpleState(s5,empty))
```

Figure 1: Maude Formalization of Statechart Diagram in Figure 2

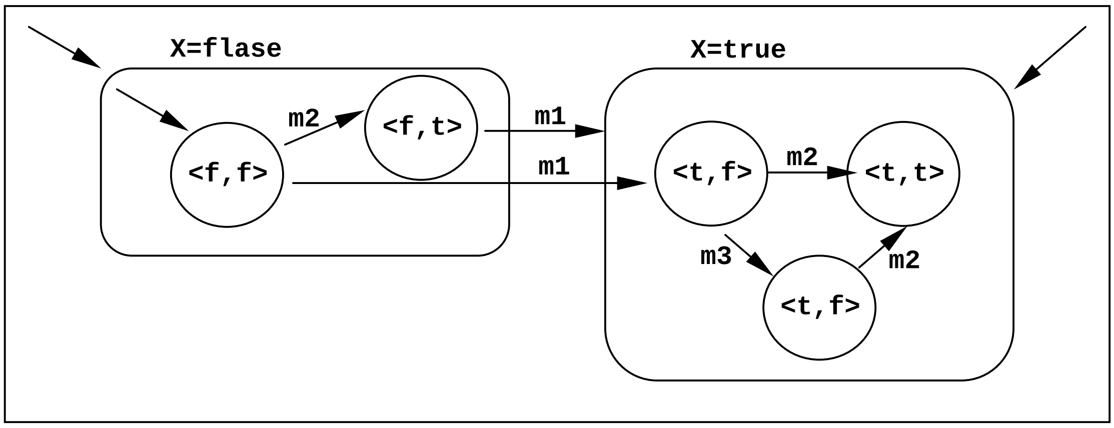


Figure 2: The matching Statechart Diagram

The novelty in this approach is the use of Maude and rewriting logic. The use of Maude allows proofs to be developed automatically at design time, which gives much greater flexibility than if all proofs had to be done by hand off-line.

2.5.1 The Coloured Petri Nets

2.5.1.1 What is Coloured Petri Nets

Coloured Petri Nets is a graphical oriented language for design, specification, simulation and verification of systems. It is in particular well-suited for systems that consists of a number of processes which communicate and synchronize. Typical examples of application areas are communication protocols, distributed systems, automated production systems, workflow analysis etc.

And Coloured Petri nets (CPN) is a special case of Petri net in which the tokens have identifying attributes; in this case the color of the token [Jen97]. At first, colored Petri nets seem less intuitive than the basic Petri net. However, by allowing the tokens to have an associated attribute, Coloured Petri nets scale to large problems much better than the basic Petri nets.

There are three general characteristics of Petri nets that make them interesting in capturing concurrent, object-oriented behavioral specifications.

- First, Petri nets allow the modeling of concurrency, synchronization, and resource sharing behavior of a system.
- Secondly, there are many theoretical results associated with Petri nets for the analysis of such issues as deadlock detection and performance analysis.
- Finally, the integration of Petri nets with an object-oriented software design architecture could provide a means for automating behavioral analysis.

And in Coloured Petri nets, circles represent places, whereas bars or boxes represent transitions. Tokens are used to mark places, and under certain enabling conditions, transitions are allowed to fire, thus causing a change in the placement of tokens.

2.5.1.2 Modeling UML Dynamic Behavior Using Colored Petri Nets

The general idea of this approach is to use a CPN model to augment the behavioral specifications of concurrent Object-Oriented design architectures created with the COMET method. COMET is a Concurrent Object Modeling and Architectural Design Method for the development of concurrent applications, in particular distributed and real-time applications. As the UML is now the standardized notation for describing Object-Oriented models, COMET uses the UML notation throughout.

The COMET Object-Oriented Software Life Cycle is highly iterative. In the Requirements Modeling phase, a use case model is developed in which the functional requirements of the system are defined in terms of actors and use cases. In the Analysis Modeling phase, static and dynamic models of the system are developed. The static model defines the structural relationships among problem domain classes. Object structuring criteria are used to determine the objects to be considered for the analysis model. A dynamic model is then developed in which the use cases from the requirements model are refined to show the objects that participate in each use case and how they interact with each other. In the Design Modeling phase, an Architectural Design Model is developed. Subsystem structuring criteria are provided to design the overall software architecture. Each concurrent subsystem is then designed in terms of active objects and passive objects. Inter-object communication and synchronization interfaces are also defined at this point. This architectural design model (captured with UML collaboration diagrams) serves as the focal point for the UML to CPN mapping. Specifically, for capturing (and subsequently validating) the dynamic behavior of concurrent and real-time systems we are interested in modeling such architectural design features as the asynchronous or periodic behavior of concurrent objects, message communication between objects, and mutually exclusive access to shared data objects. Figure 4 provides an example UML collaboration diagram illustrating these architectural design features.

In the example below, an actor initiates some event on the system. The first active (asynchronous) object performs some processing on the input event and sends an asynchronous message to the active periodic object and a synchronous message to the second active asynchronous object. There is also an entity object that encapsulates data and provides operations to access the data. Since the entity object is being read by and written to by two active objects, it must also provide mutually exclusive access controls, which must in turn be represented by the corresponding CPN model. The following sections further discuss the roles of these elements in terms of the COMET method and discuss the mapping between these UML elements and the corresponding CPN segments.

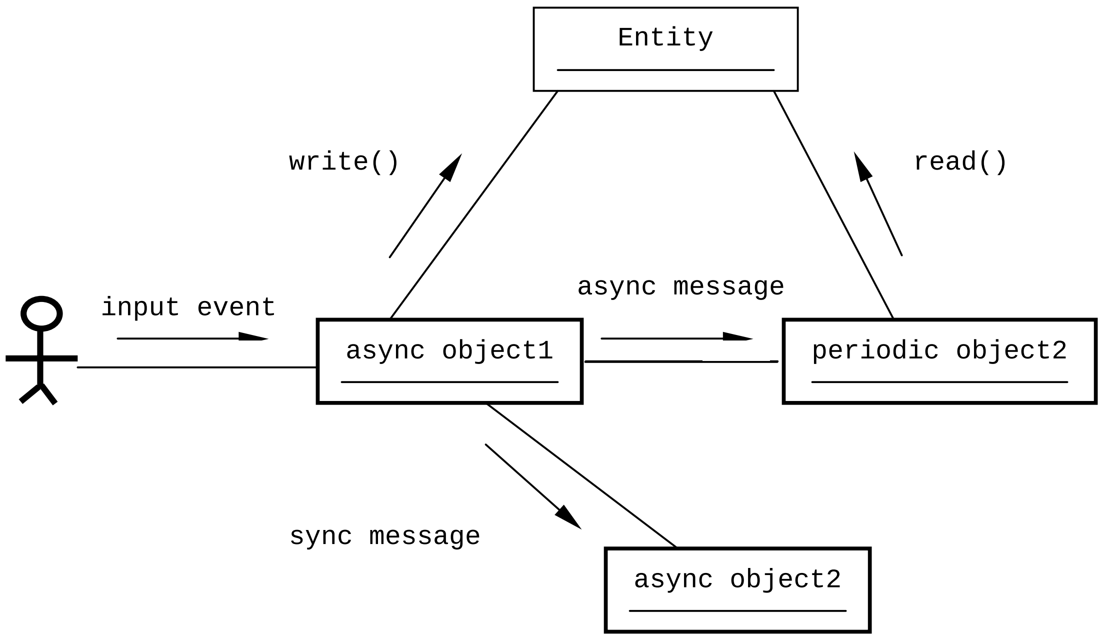


Figure 3: Example UML Collaboration Diagram

Mapping Active Objects to Colored Petri Nets

Active objects form the basis of concurrency within the UML. Active objects may be found in interfaces or processing objects and may operate in either asynchronous or periodic modes. And in this approach, interface objects and processing objects will be treated the same in terms of creating Petri net templates. The only difference is that interface objects receive ***events*** from external sources, whereas processing objects receive ***messages*** from other objects within the system.

An **asynchronous** active object is activated by an asynchronous stimulus (e.g. message or interrupt) rather than a timer event. The CPN representation of an asynchronous active object consists of a series of places and transitions that use a control token to represent the flow of control within the object. Figure 5 shows an example of modeling an asynchronous object with a CPN. In this CPN template, event tokens enter from an external source (external device, system, or application object, depending on whether we are dealing with an interface or a processing object). When the CPN segment is ready to process events as indicated by the presence of a control token in the ***Async Control*** place, the transition is fired and the event and control tokens are given to the ***Event Received*** place. Notice the "**@**" notation on the ***Process Event*** transition, this indicates that all timed tokens are incremented by some arbitrary processing time in order to simulate the real-time nature of execution. In the process described by this paper, all control tokens are timed.

Next, the internal event and control tokens are passed to the ***SendMsg*** transition to be translated (using special guard or code segments) to the appropriate CPN segment, representing the receiving object in the UML model. This message may be sent immediately in the case of asynchronous communication or may be blocked until the receiver is ready in the case of synchronous communication. Communication mechanisms are discussed in more detail in Section 0. Once the message has been sent, the control token is returned to the ***Async Control*** Place and the CPN segment is ready to process the next event or message.

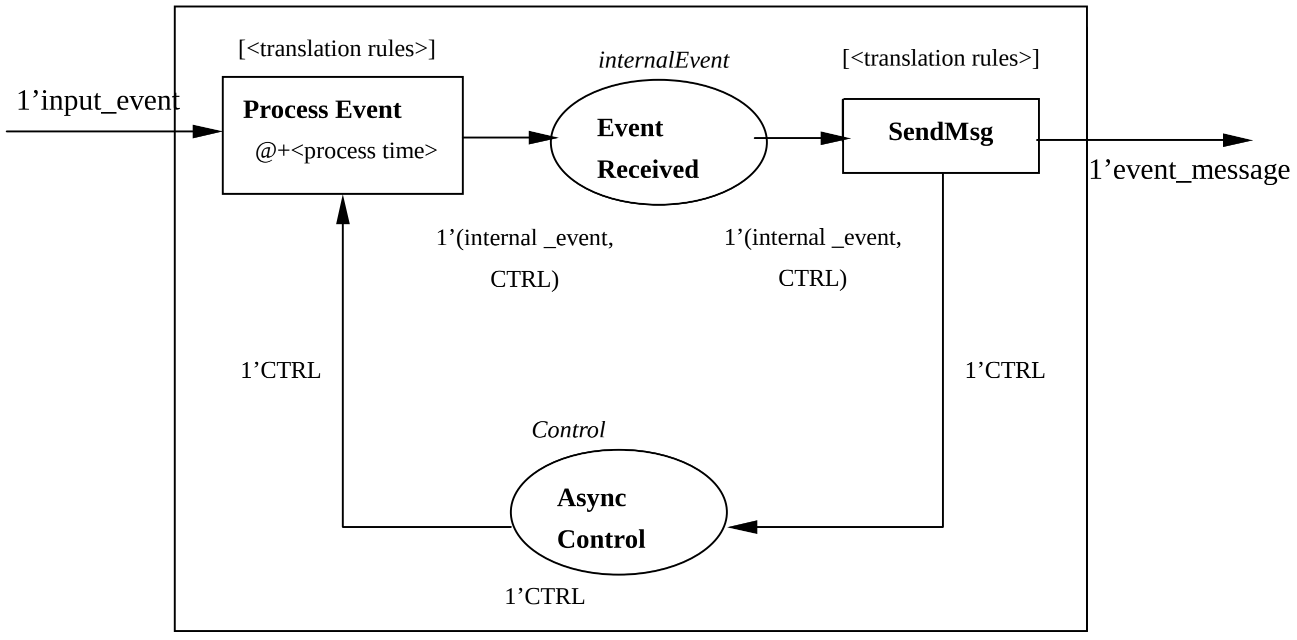


Figure 4: CPN Segment for Asynchronous Active Objects

A **periodic** active object is activated at regular time intervals rather than on demand. To represent periodic objects with CPNs, a ***Wakeup*** transition is introduced that will delay control tokens from returning to the ***Periodic Control*** place by the desired period of activation. Figure 5 provides the CPN segment that represents periodic objects. In this example, the object being modeled starts execution with a control token in the ***Sleep*** place. The ***Wakeup*** transition to move from the ***Sleep*** place has a duration of ***<sleep time>*** associated with it. After the specified duration has been reached, the object essentially "***wakes up***" and acts on any waiting events or messages in the same manner as the asynchronous CPN segment.

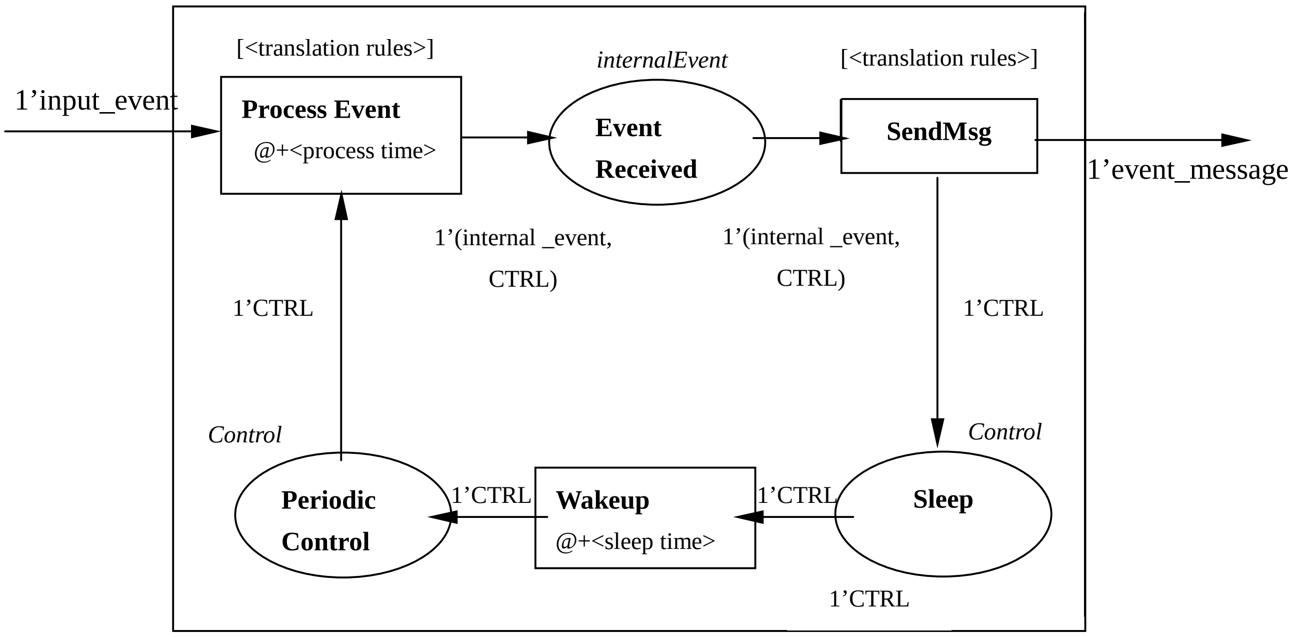


Figure 5: CPN Segment for Periodic Active Objects

Mapping Passive Objects to Colored Petri Nets

In COMET, ***entity*** objects are passive objects that provide mechanisms to encapsulate or store data that needs to be accessed by other objects within the system. These entity objects must also provide the protection mechanisms to enforce mutual exclusion rules necessitated by the passive objects being accessed by multiple active objects. The general CPN segment for entity objects with mutually exclusive access protection is illustrated in Figure 6.

When this generic segment is instantiated, there will be one ***ReadOp*** and ***WriteOp*** transition for each read and write operation for each attribute accessed from the object interface. To use the read and write operations, two places per read or write operations are needed - one place for the request and one for the response/return. In addition to the read and write places, there is one ***Free*** place per unit of protection (e.g. attribute) that is used to enforce the mutual exclusion rules. This Free place contains one token indicating if the attribute is in use. If the Free token is available (i.e. the attribute is not in use), the corresponding read or write transitions are allowed to fire, thus allowing the attribute token to be retrieved or modified.

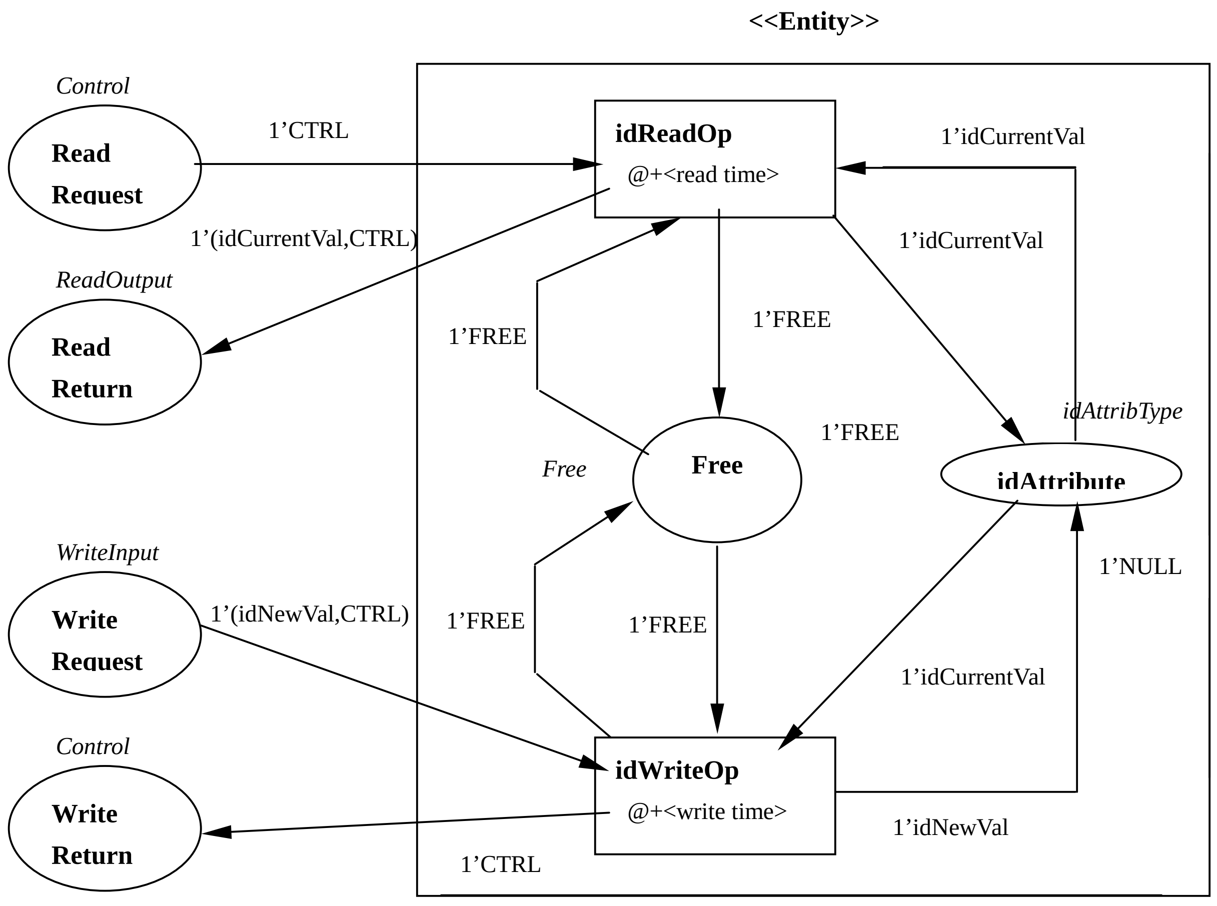


Figure 6: CPN Segment for Entity Objects

Mapping Message Communication to Colored Petri Nets

Normally, There are two general forms of message communication that can occur between concurrent active objects: ***asynchronous*** and ***synchronous*** message. With **asynchronous communication**, a producer object places a message on a queue and then continues its processing. A consumer object would then retrieve the first message from the queue, do some processing based on the message, and then retrieve the next message from the queue (if any).

Modeling FIFO queuing behavior using CPNs can be complex for large buffer sizes case. There are currently no CPN formalisms to enforce ordered placement or retrieval of tokens to and from a given place. In the absence of ordered token placement and retrieval capabilities to and from a single place, ***2n*** places and ***n+1*** transitions are needed to model a queue of ***n*** elements (one place for storage and one place to indicate whether a place contains a token). Given that a queue has at least one free space, a producer would first place a token on the end place of the CPN queue. Through the series of ***n+1*** transitions and ***n*** free place indicators, the enqueued token will advance to the furthest available slot in the queue. To dequeue an element (i.e. retrieve a message from the queue), a consumer would remove a token from the place representing the head of the queue. Tokens are then shifted to the right as one is removed.

In the case of **synchronous communication**, a producer object sends a message to the consumer object but instead of continuing with its processing, it will wait for the message to be received by the consumer. This form of communication is handled simply by passing a token to a CPN segment (e.g. the ***input_event*** of Figure 5) and then having a ***Control*** token returned to the sender either after the token has been received (at the first transition of the CPN segment) or after a return token (message) has been generated.

2.5.1.3 Validating Dynamic Behavior

The first step to validating the dynamic behavior of a UML architecture using CPNs is to translate the concurrent object architecture model (represented by a UML collaboration diagram) into a corresponding CPN network. This is accomplished by replacing each object and message communication element by the appropriate CPN segment as partially illustrated in the previous section 2.5.1.2. (More specific CPN segments were used in the actual research effort.)

Once the UML architectural model has been translated to a CPN, an occurrence graph [Jen97] is generated to construct a graph of all reachable markings for the CPN. These graphs can be extremely large and complex, but are still capable of being automated using tools such as DesignCPN, which was applied for this research. Based on these graphs, Petri net theory may be applied to validate the absence of deadlock or starvation conditions as well as providing statistical analysis of the architectural usage. Furthermore, DesignCPN also provides a timed simulation capability that allows architectural timing constraints to be evaluated.

2.6 Conclusions and Summary

From the approaches presented in this chapter some meaningful ideas can be reused in algebraic specifications defining ADTs, and be taken into account in the work of this thesis.

From the **Maude System**, it is mainly considered:
The idea of coming up with UML behavior model transformations is not new. However, the particular kinds of transformations considered in this method are novel. First, they are not necessarily correctness-preserving. Second, they include large grain transformations, which do much more than make minor modifications. It is mostly small grain transformations that have been studied previously. Small grain transformations alone are not sufficient. To make any useful change would require a long sequence of such transformations. Moreover, most

previous transformations work only on static UML models (e.g. class diagrams), while Maude system presents a new idea over the transformations of UML behavior models.

And from the study of **Coloured Petri Nets (CPN)** approach, it makes use of another graphical modeling system; lots of benefits are taken, including the proof ability.

It is an approach for using CPN segments to model the dynamic behavior of concurrent object architectures expressed in the UML. Given a concurrent architecture and the CPN segments, an engineer may proceed with behavioral analysis by first mapping the UML architectural elements into a CPN representation. The resulting CPN is then used to validate such dynamic properties as the absence of deadlock and starvation conditions as well as providing a timing analysis of the architecture through simulation. This analysis through CPNs reduces the overall risk of software implementation by allowing behavioral characteristics to be validated from an architectural model rather than waiting for the system to be coded. And the novelty is that it integrates Colored Petri Nets with Object-Oriented software design methods (UML) for concurrent and real-time systems. It is the goal of this continuing research to arrive at a set of CPN translation rules that can be effectively integrated with software design methods to provide increased reliability and analytical capabilities at multiple levels of abstraction.

An alternative way to write down the mapping rules is algorithmically - a recipe for converting expressions in the (abstract) syntax to expressions in the semantics language. This would be useful where it is intended that the mapping is going to be automated by tools. For example where the semantics domain is an OOPL such as Java, and the mapping corresponds to code generation. Unfortunately, using a programming language as a semantics domain leads to a severe problem, which needs to be considered: any model defined in an executable language can only describe one implementation and therefore can not exhibit any form of under-specification. As discussed earlier, modeling languages like UML need to allow under-specification. Thus code generation necessarily involves a selection of one of the possible implementations - possibly a wrong one. And that’s why we choose a formal algebraic specifications language to describe Abstract Data Types, which will be discussed detailedly in next chapter.

Chapter Three

UML and ADT: a Semantic Framework Proposition

It is frequently made assumption that Object-Orientation is based on the principles of data encapsulation and data abstraction. In this section, we closely examine the relationship between Object-Oriented modeling and the classical algebraic approach to data abstraction. Data abstraction is the principle of specifying a data type together with its characteristic operations in such a way that the internal structure of the data in kept hidden and that the specification has clear meaning independent of the context in which it is used. While on one hand, there is conclusion that Object-Oriented specifications of software do not follow the classical data abstraction principles in an obvious way. There is a concept mismatch, which leads to problems in translating semi-formal Object-Oriented specification into formal algebraic specifications. And on the other hand, the idea of data abstraction was a very successful contribution from the more theoretically oriented point of view. Modern class libraries like Java libraries or STL are organized according to the data abstraction principle. But less successful were achieved for the formal language, which were developed for abstract data type specification. There is more to data abstraction principle than just a mechanism for describing standard data types. Abstract data types provide a useful paradigm for the construction of stable software modules [Par72]. The usage of this principle turned out as practically successful for construction of very large software systems.

In this chapter, we will propose an appropriate semantics framework according to the work we have to do – building Formal specifications of mixed systems, and describe our approach based on a concrete example. It includes a certain specification language, and a set of mapping rules defined in this language between UML model and algebraic specification.

What is Mixed System?

Mixed system is a kind of system, which is composed of static, dynamic and functional aspects. These are complex and realistic systems where the use of several formalisms is required. Static aspects deal with the signatures, the types and the relations between types. Functional aspects describe the semantics for operations or explicit some conditions and invariants. Dynamic aspects focus on the so-called dynamic behavior of systems, it is related to concurrency and communications. The two main issues with mixed systems are first to ensure the consistency between the different aspects and second to provide specification and proof guidelines. And in my example, the composite system is composed of several sequential components, which represents sequential process inside the system respectively. Detail information about the workflow of the whole process will be mentioned in section 3.2.3.

Why “Formal Specifications of The Mixed Systems”?

Why do we choose “Formal Specifications of Mixed Systems” to show the semantics framework Proposition?

- First, Formal specifications of mixed systems are one of the main issues in software engineering, however several difficulties remain, the ability to produce a coherent mixed specification and to provide a fully integrated semantic is difficult.
- Second, this research project contains most of concerns in UML dynamic part, including class diagram, statechart diagram, collaboration diagram and sequence diagram. We can use this example to cover the majority of UML Dynamic part formalization issue, and show the benefit and advantage to give a formal specification to UML through the research in this field.

Alternative Semantics Propositions

It is common belief that there is a substantial difference between model-oriented (e.g. Z and VDM) and Algebraic Specification languages (e.g. LSL and ACT-ONE) with respect to their applicability to the specification of software systems. While model-oriented specification languages are assumed to be suited better for the description of state based systems (abstract machines), Algebraic Specification languages are assumed to be better for Abstract Data Type specifications. First, the concept of algebraic specification is linked to Class. Earlier B.Meyer felt that Abstract Data Types (ADT) is important in the context of Object-Oriented programming [Mey97], however some difficulties remain about inheritance and concurrency. Second, for more than twenty years, there has been a great amount of work about Algebraic specifications and the definition of abstract data types has been done. There are several efficient tools to handle them like Larch Prover, PVS, or Isabelle. One last interest is that this approach is able to catch both data types and concurrent systems in an homogeneous framework, see [EAE99] for a good survey.

In my approach we shall demonstrate how an algebraic specification language (the Larch Prover tool Language) can be used to write specifications of abstract state machines, and how to express communication and concurrency information between sequential components. And how support tools for algebraic specification languages, e.g. type checker and theorem provers, can be used to reason about abstract state machines. Precisely define a software system by using Algebraic specification.

Algebraic specification languages (e.g. LSL and ACT-ONE), their applicability to the specification of software systems, algebraic specification languages is assumed to be better for abstract data type specifications. The semantic framework proposed in this work is based on algebraic specifications describing Abstract Data Types (ADT). In the previous chapters the importance of UML formalization and some approaches in this direction have been presented. From these approaches some important outlined points are taken into account. The goal of this chapter is to explain the algebraic formal semantic framework through the translation rules, which support the automatic translation from UML model elements to algebraic ADTs.

3.1 Main Points Considered in the Framework Composition

In order to compose the formal framework, the semantics of the main UML dynamic model elements was evaluated, such as Statechart diagram, collaboration diagram, etc, and also some static part constructs are concerned, for example Class Diagram etc. The main motivation towards UML formalization is the fact that the semantics of the UML model elements is not precisely described in the official UML semantics document [UML99]. Consequently, in some ambiguous points it was necessary to get help from other sources of information to achieve a good interpretation. Long times of discussion were also necessary to achieve final conclusions.

According to the final interpretation of the semantics, the translation rules were defined having as a result the algebraic formal specifications for some UML dynamic constructs. Here we used a proper approach based on Graphic Abstract Data Type [Roy01b]. A component is described by a Symbolic Transition System (a STS is a simple form of statechart) and from that an algebraic specification may be built. The semantics of concurrency and synchronization are obtained from the synchronous product of STSs in a similar way than for the synchronous product of automata [Arn94]. This choice is justified in the next section.

To start with the formalization, in this work it is considered the UML core concepts respecting to the behavior aspects of the UML, which are:

- ***Statechart Diagram*** – a diagram that shows a state machine; statechart diagrams address the dynamic view of a system;
- ***Collaboration Diagram*** – an interaction diagram that emphasizes the structural organization of the objects that send and receive message; a diagram that shows interactions organized around instances and their links to each other.
- ***Class Diagram*** – a diagram that shows a set of classes, interfaces, and collaborations and their relationships; class diagrams address the static design view of a system; a diagram that shows a collection of declarative (static) elements.
- ***Values*** - a type defines the values of its instances and the value of an instance consists of the values of its attributes at a point in time;
- ***Operations*** – the implementation of a service that can be requested from any object of the class in order to affect behavior;
- ***Associations*** – a structural relationship that describes a set of links, in which a link is a connection among objects; the semantic relationship between two or more classifiers that

involves the connections among their instances.

As in [CE97], the core concepts are extracted from the Core Object Model specification presented by Houston and Josephs [HJ95] written in Z that captures a precise description of the Object Management Group's emerging standard for objects.

Starting from the core concepts it makes feasible that future extensions to the semantic framework can easily proceed.

Another important aspect to point out is that the semantics framework presented is typed. However it is assumed that once translations to algebraic ADTs are proceeded, type-checking problems are not carried to the specifications. The ADTs are written in Larch Prover as will be shown in section 3.1.3.

3.1.1 The Formalization Method Chosen

The approach chosen for the formalization is the integrated one, inspired from pervious work on *Graphic Abstract Data Type* [Roy01b]. This approach is justified in many ways:

- A mapping rules between graphical and formal constructs can uncover problems with the modeling notations;
- It can help identifying ambiguous and inconsistent structures;
- It can help defining semantically well-formed informal models;
- The mapping rules can be adapted to a CASE tool in such a way that formal specifications can be automatic generated from informal models (to express the whole or at least part of the models). This can help in proving properties of the models and in generating code from them.

The integration of the translation process built with Rational Rose UML CASE tool is explained in chapter 4 with a concrete example, and how to use these technologies to implement the translations work given in chapter 5.

The translation rules making the bridge from UML models to formal models are explained in section 3.2.

3.1.2 The Formal Language Chosen

The language used to write the formal specifications is Larch more specifically with the syntax of Larch Prover. It is an algebraic method not yet extended with OO concepts. However Larch is really suitable to the description of Abstract Data Types because it allows the semantics of the operations to be described in an abstract way, i.e. just as equations notation stating equal relations between them. In addition Larch Prover allows verifications and proofs to be applied over the formal specifications. This is really helpful in order to ensure the correctness of the models described. More information on Larch Prover is found in chapter 4, section 4.2.

3.1.3 ADT Structure

We refer here to [Wir90, EAE99] which are rather comprehensive documents about these formal specification techniques. In brief an algebraic specification of a data type is composed of three main parts:

- A heading part containing information about the module, mainly they are: the name (or sort) of defined the data type, the imported modules (or types), and the generator names (or constructors).
- The signatures part, which describe the operators' syntax.
- The axioms part, which describe the semantics of operations.

The ADT example presented here use the following notations, where ~ is logical not, /\ is and, \/ is or, => is implication and = is syntactic equality. It is followed Larch Prover syntax. The key words of Larch Prover are in thick font. Related notes are between slashes.

```
%-----------------------
%FileName: StaticBank.lp
%ClassName: StaticBank
%-----------------------

set name StaticBank
declare sort StaticBank,List,Real,Natural,Account
declare variable sb:StaticBank,r1:Real,n1:Natural,l1:ListofAccount
declare operator
     newStaticBank : List,Natural,Real,StaticBank -> StaticBank
     enough : StaticBank -> Bool
     theAcount : StaticBank -> Account
     exists : StaticBank -> Bool
     price : StaticBank -> Real
     number : StaticBank -> Natural
     accounts : StaticBank -> List
```

```
..

assert
sort StaticBank generated by newStaticBank;

%selectors
price(newStaticBank(r1,n1,l1))=r1;
number(newStaticBank(r1,n1,l1))=n1;
accounts(newStaticBank(r1,n1,l1))=l1;

%operation axioms
exists(newStaticBank(r1,n1,l1))=has(l1,n1);
exists(newStaticBank(r1,n1,l1))=>enough(newStaticBank(r1,n1,l1))=amount(theAcount)>=r1
;
exists(newStaticBank(r1,n1,l1))=>theAcount(newStaticBank(r1,n1,l1))=find(l1,n1);
exists(newStaticBank(r1,n1,l1))=>decrease()=newStaticBank(substract(l1,r1,n1),0,0);
..
```

Table 1: Larch Prover Syntax Example

%---------	Annotation
set name	define the name of the sort
declare sort	declare the data types used in this sort specification
declare variable	declare the variables with the corresponding data types that will be used in the axioms
declare operator	define the operators that apply to the values of the data types being defined
`accounts : StaticBank -> List`	operation signature, which is composed of operator’s name, data type of parameters (after “:”), and data type of return value (after “->”) respectively.
Assert	semantics of the operators are described through the axioms written in the assert section
sort ... **generated by**	constructor of the defined sort

Table 2: Larch Prover Syntax Description

The meaning of an axiom like:

```
exists(newStaticBank(lc, p, n)) => theAccount(newStaticBank(lc, p, n)) = find(lc, n)
```

is: IF the account number exists THEN the account object will be the result of a find operation in the list of accounts. To make reading easier for non-specialist, such axioms may be rewritten in a more familiar and object-oriented way, but we ignore this here.

From the Larch Prover file example showed above, it is obvious to see that the axioms are compound from equations that are equalities or equivalencies between terms with variables. Variables represent a valid value inside a Universe of its data type.

3.2 A Concrete Example and Assumptions

3.2.1 Terms Specification in Translation Rules

Before we start presenting the translation rules, we'd better give a specification to the terms, which appear in our translation rules specification.

- ***observer operation*** **-** is an operation whose resulting type is not the defined sort. Usually it is used in guard condition expression of the state transition in the statechart diagram.
- ***internal operation*** – is an operation which has the defined sort as resulting type.
- ***generator operation*** - are internal operations, which are sufficient to generate all the values of the data type. All the actions, which label the state transitions in the statechart diagram, are generator operations.
- ***selector operation*** – are operations, which use defined sort as only parameter and each attributes of defined sort as return value. Every attribute in the defined sort has its matching selector operation.
- ***operation axioms*** – axioms used to defined the semantics of the sort's operations .
- ***definedness axioms*** **–** are axioms used to defined whether a return value from a generator operation is a valid defined sort or not.
- ***precondition axioms*** – are axioms used to define the precondition, which should be satisfied before the invocation of certain operation.
- ***state predicates axioms*** – are axioms, which define the condition conjunction, with which certain state can be reached.
- ***composite system*** – the system, which is composed of sequential components.
- ***synchronization rules*** – giving those operation pairs, which should be synchronized in the system execution.

- ***synchronous product*** – the statechart diagram of composite system, which is generated through the information of sequential component and constraints of synchronization rules.

3.2.2 Example Description

Now we take a look at a concrete example. And we will describe the whole process of our translation approach by this example.

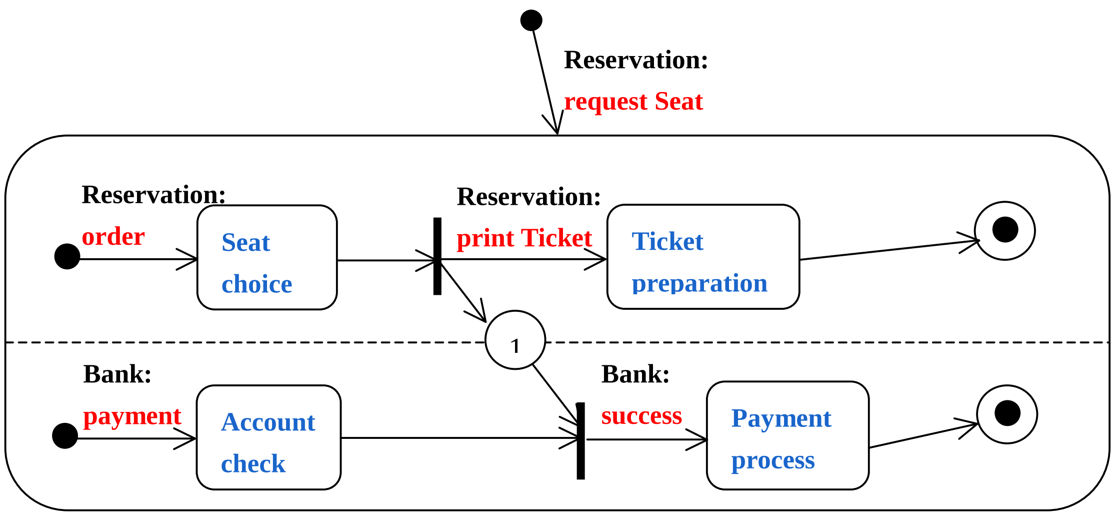


Figure 7: Synchronization Statechart Diagram of the Ticket-Purchase System

It is a part of a ticket-purchase system for illustrating the translation principles, and we try to model this system with two concurrent sequential components in UML: **Reservation** and **Bank** component, **Reservation** component for the seat reservation and **Bank** component for simulating the bank. The client gives its account number when he requests a seat, and if there is a seat, the seat reservation process invoices the price to the bank, and if all it is ok the client gets a ticket otherwise the reservation fails.

In the diagram above, black font represents component in the system, read font represents the related operation with this component, and blue font represents states. Meanwhile, the two sequential components represent two independent processes indeed. And they communicate and synchronized by some defined rules.

Here we present the Synchronization Statechart Diagram (Figure 8) of the system in order to get a general view about system functions. First of all, **Reservation** component launch *request Seat* operation asking for a plane ticket, and then **Reservation** component let the user select a proper seat, meanwhile, **Bank** component launch *payment* operation asking for the user's bank account to pay the ticket. Only when the proper seat is selected by user and user's bank account is satisfied with account checking operation, the two independent sequential processes can move on (this is one pair of synchronous operation), and then **Bank** component executes *success* operation to draw the money from user's account, and **Reservation** component executes *print Ticket* operation to prepare the plane ticket. These two operations are also necessary to be synchronized, because the ticket can only be printed to user after the money has been charged from the correct bank account, but we don't show them explicitly in the reason of simplicity. In this diagram, operations **Reservation**:*order* and **Bank**:*payment* are synchronized, which is illustrated by inserting a synchronization state between seat choice and account check state.

3.2.2.1 The two Sequential Components

Now we describe these two sequential components **Bank** and **Reservation**, which occur in the system, in UML diagram.

Bank Component

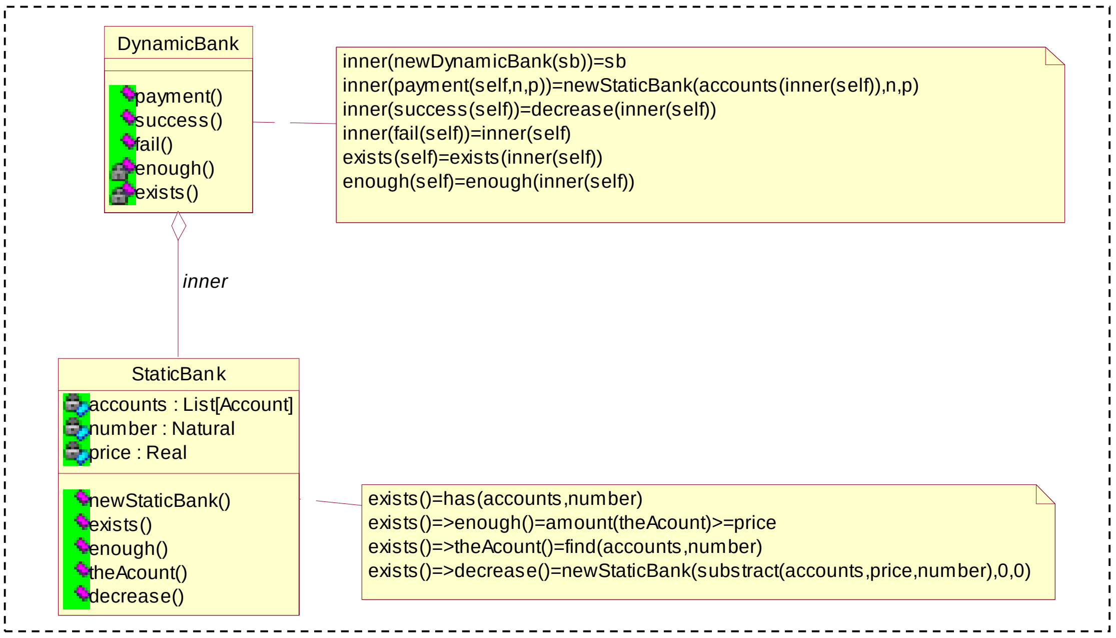

DynamicBank
payment()
success()
fail()
enough()
exists()
inner(newDynamicBank(sb))=sb
inner(payment(self,n,p))=newStaticBank(accounts(inner(self)),n,p)
inner(success(self))=decrease(inner(self))
inner(fail(self))=inner(self)
exists(self)=exists(inner(self))
enough(self)=enough(inner(self))
inner
StaticBank
accounts : List[Account]
number : Natural
price : Real
newStaticBank()
exists()
enough()
theAcount()
decrease()
exists()=has(accounts,number)
exists()=>enough()=amount(theAcount)>=price
exists()=>theAcount()=find(accounts,number)
exists()=>decrease()=newStaticBank(substract(accounts,price,number),0,0)

Figure 8: The Class Diagram of Bank Component

The class diagram of **Bank** Component represents the business functions of a Bank in Ticket-Purchase system, or to say, what it can do. It has ***payment(n,p)***, ***success()***, ***fail()*** operations.

- ***payment()***: operation for asking for a valid user's bank account to pay for the ticket.
- ***success()***: operation for drawing the money from the user's bank account.
- ***fail()***: if the user's account is invalid or dosen't have enough money, Bank component will execute *fail* operation, print the error message and return to middle state, and ask for

the user to input another correct account No. Maybe after three times attempt, the machine will be locked automatically, but it is the concerns of implementation issue, we don't care about it, we only concentrate on the interface operation in each component.

Below is the matching statechart diagram of Bank component, which describe the dynamic behavior information of it, in other words, how these operations can take place and interact with each other? And those actions take place along the state transition correspond to the operations in the component operation interface.

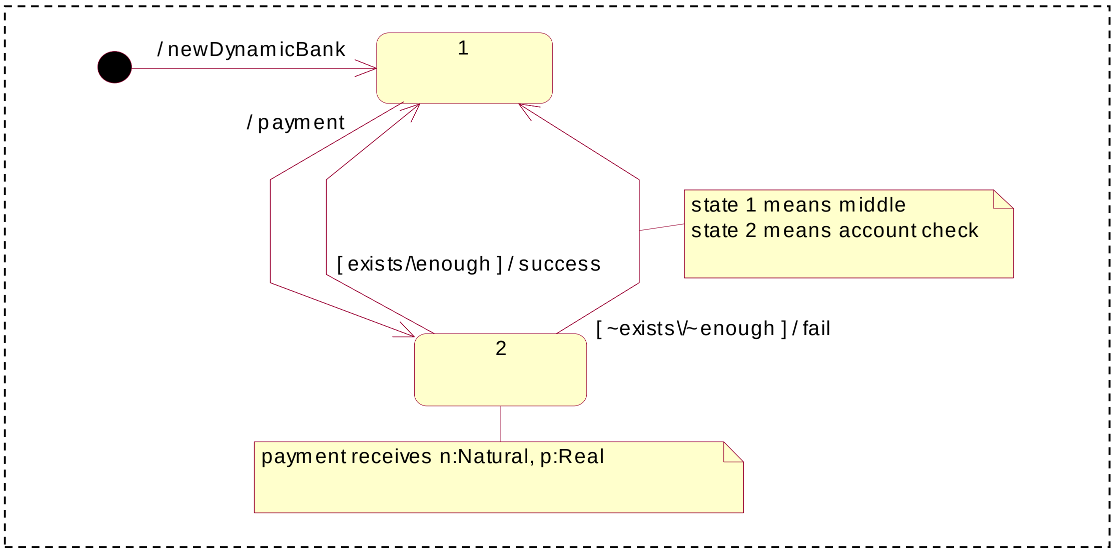
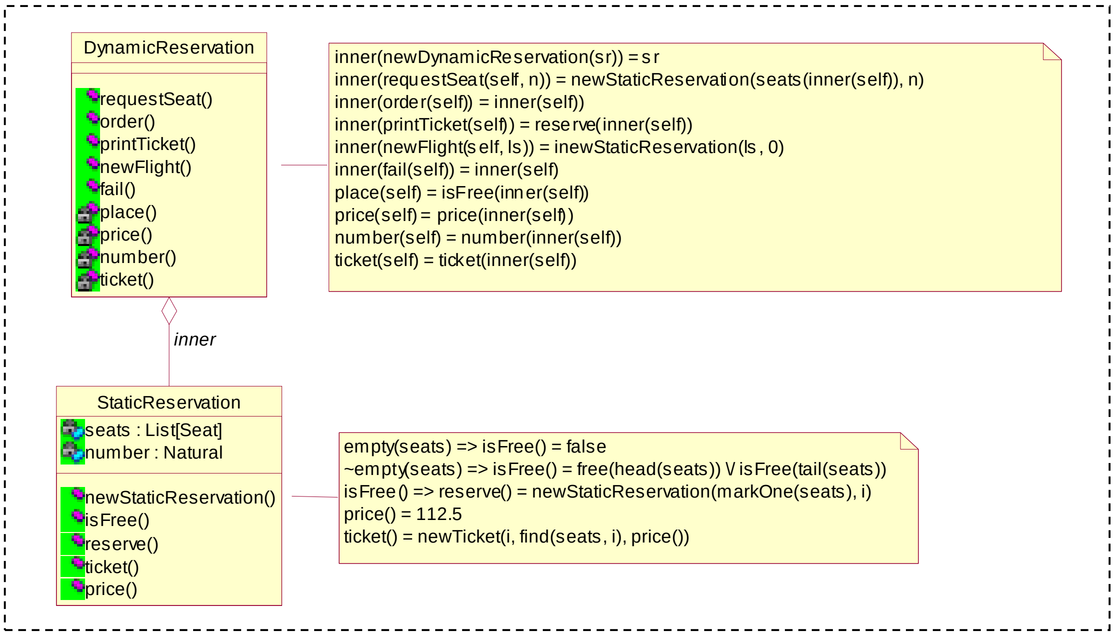

/ newDynamicBank
1
/ payment
[exists/\enough] / success
state 1 means middle
state 2 means account check
[~exists\/~enough] / fail
2
payment receives n:Natural, p:Real

Figure 9: The Statechart Diagram of Bank Component

Reservation Component



```
inner(newDynamicReservation(sr)) = sr
inner(requestSeat(self, n)) = newStaticReservation(seats(inner(self)), n)
inner(order(self)) = inner(self))
inner(printTicket(self)) = reserve(inner(self))
inner(newFlight(self, ls)) = inewStaticReservation(ls, 0)
inner(fail(self)) = inner(self)
place(self) = isFree(inner(self))
price(self) = price(inner(self))
number(self) = number(inner(self))
ticket(self) = ticket(inner(self))
```

```
empty(seats) => isFree() = false
~empty(seats) => isFree() = free(head(seats)) V isFree(tail(seats))
isFree() => reserve() = newStaticReservation(markOne(seats), i)
price() = 112.5
ticket() = newTicket(i, find(seats, i), price())
```

Figure 10: The Class Diagram of Reservation Component

The class diagram of **Reservation** Component represents the business functions of a Reservation process in Ticket-Purchase system, it has ***requestSeat()***, ***order()***, ***printTicket()***, ***newFlight()***, ***fail()*** operations.

- ***requestSeat(n)***: operation asking for a plane ticket in certain flight, and argument ***n*** represents the bank account number for paying the ticket.
- ***order()***: order this plane ticket.
- ***printTicket()***: print this ticket to user.
- ***newFlight()***: if this flight doesn't have any empty seat avaiable, ***newFlight()*** will move to a new flight, and return to middle state waiting for new seat request.
- ***fail()***: execute after the ***order()*** operation failed, and print error message.

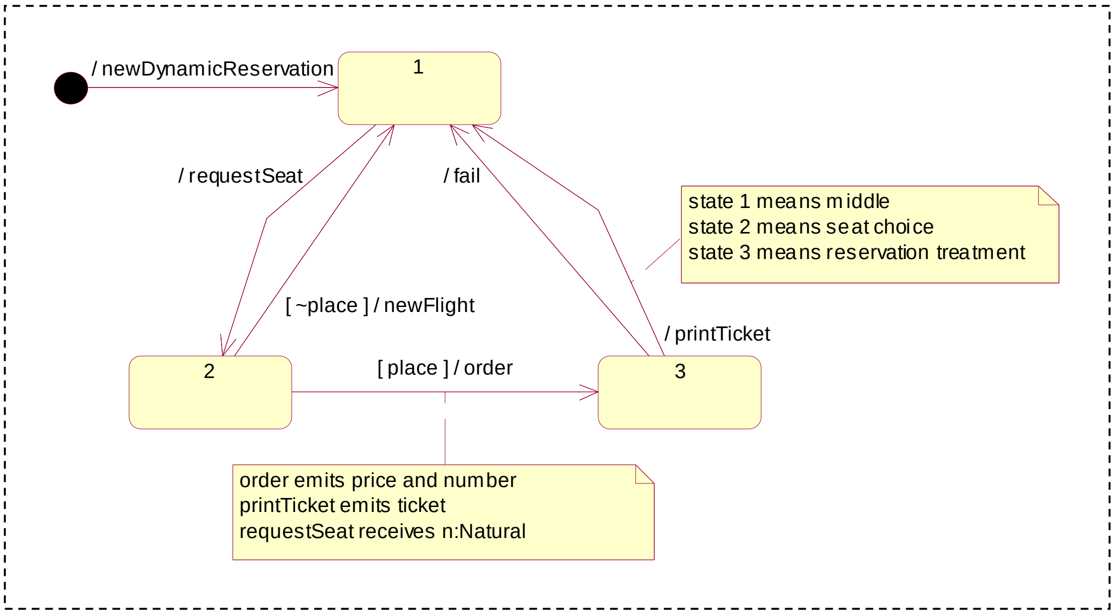


Figure 11: The Statechart Diagram of Reservation Component

3.2.2.2 Composite System

The Collaboration Diagram of the Composite System

The System Collaboration diagram is also a source diagram, which provides necessary information for the specification work. It presents the communication and concurrent information of the system, and a set of synchronization rules we should take into account when building synchronous product the composite system.

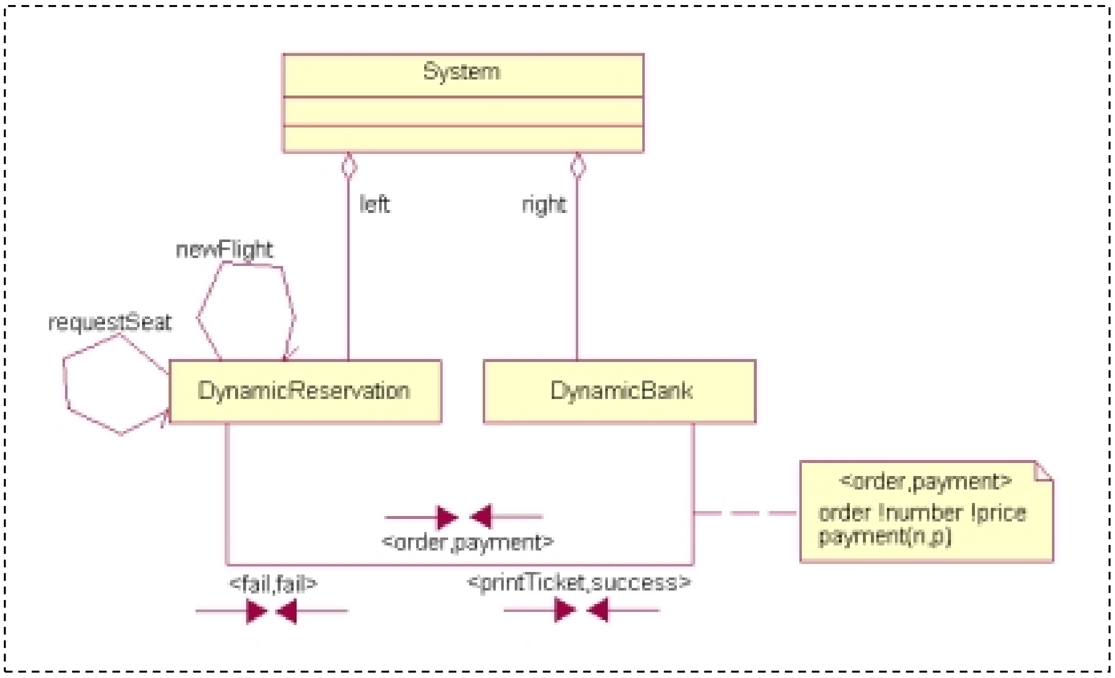


Figure 12: Collaboration Diagram of the Composite System

Asynchronous messages are illustrated by single half arrowhead. And synchronous messages are illustrated by solid arrowhead. And the notes element describe the internal data communication between two synchronized operations, here it means that ***order()*** operation emits ***number*** and ***price*** value, and meanwhile ***payment()*** operation need ***number*** and ***price*** value as the parameter. So when we generate the composite system, those internal data communication will not be taken into account in the composite system operation signature.

The Statechart Diagram of Composite System

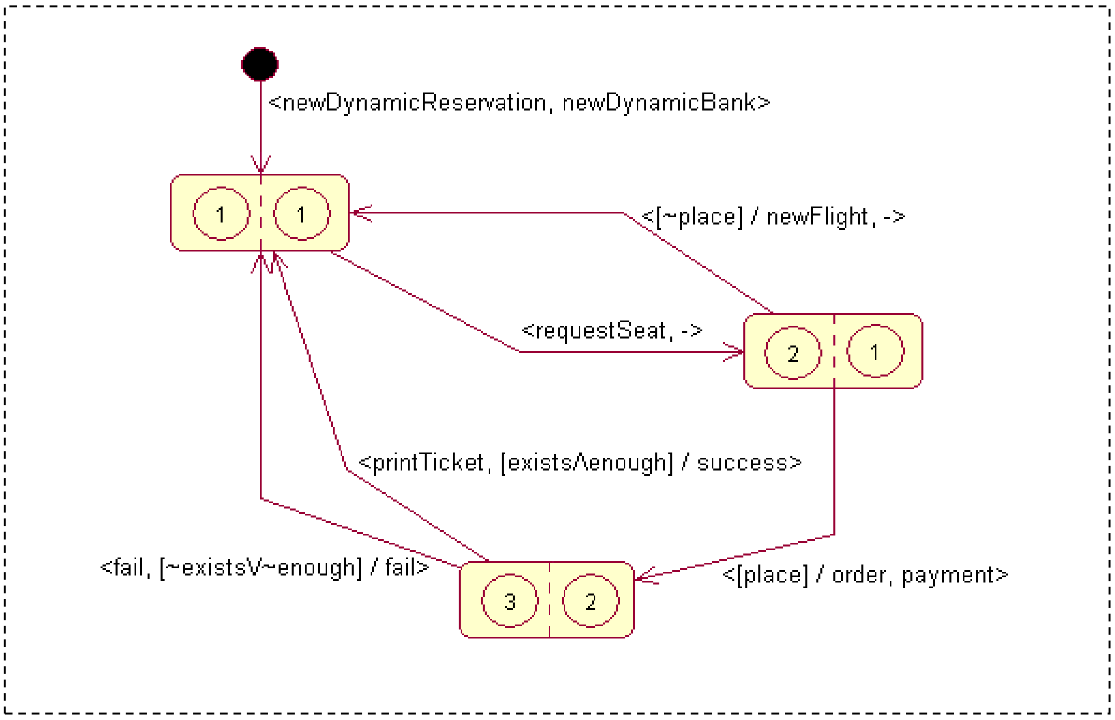

<newDynamicReservation, newDynamicBank>
1
1
<[~place] / newFlight, ->
<requestSeat, ->
2
1
<printTicket, [exists∧enough] / success>
<fail, [~exists∨~enough] / fail>
3
2
<[place] / order, payment>

Figure 13: Statechart Diagram of Composite System

Note that this Composite System diagram does not exist actually, I mean, it is not provided by user, but is generated by CS.java (Composite System), a tool, which I developed, with the synchronization rules information retrieved from Composite System Collaboration Diagrams. In this statechart diagram, each composite state encompasses two sequential component states, and each composite transition is composed by merging of two sequential component transitions as well.

3.2.2.3 Technical Justification

You can see, in our approach, when we describe the sequential component in UML class diagram, we tend to separate it into two parts, the Dynamic part and Static Part; meanwhile we attach a statechart diagram with the Dynamic part for illustrating its dynamic behavior information. And the Static class has an inner relationship with the Dynamic class.

But why is that? You may want to ask why, why we should separate every component into two parts, why not represent the component in a single class diagram, which is more simple and concise to understand?

Below are some justifications for adopting this method:

1. Separation of Concern, In fact, every component has two major aspects, one is internal datatypes that it represents, and the other is its external behavior that it can perform and show outside. So when we design the component in UML diagram, we try to separate these two parts explicitly, and the Static class focus on the datatypes manipulation, while Dynamic class focus on the behavior information. And you can see the inner relationship between Dynamic and Static part, the Static class describes the internal state and the functional behavior of the component. And Dynamic class can delegate its operation to the Static Class. It is have similar signification with OO technology, which try to separate internal and external concerns by encapsulating something, which don't want outside to know and see.

2. Second, the ultimate purpose of our method is checking and verifying whether the synchronous statechart product is possible or not, or to say whether those composite states can be reached or not. And synchronous product statechart diagram only focus on the interface operations, which the sequential component can perform, and the communication and synchronization relationship between them. So we put all interface operations in the Dynamic part, and we assume a strong link among dynamic class, static class and a simple statechart diagram. The main idea is: the dynamic behavior is an abstract interpretation of the partial abstract data type associated to the class specification, and operations occur in the statechart correspond to operations of dynamic class interface. It is a more concise way and easy to express state transition information, which need to be verified.

3.2.3 Some Assumptions

We consider some assumptions in our approach, and we concentrate only on some essential features; extensions will complete this approach (see Chapter 6). Here, We only consider active and static classes, statechart and collaboration diagrams. We only consider free side-effect models. We should also target imperative models but this increases the notation complexity. To a sequential or a concurrent process we associate an active class stereotyped *process*. At this level the difference between thread and process in UML is not so important. We use algebraic axioms to complete UML diagrams. OCL expressions would be possible but it is rather an operational language than a specification language and it seems to have some lacks [RACH00]. Furthermore we target algebraic languages, and then OCL would require an additional translation as in [HHK98a]. As in process algebra we consider two kinds of components: sequential (the **Bank** and the seat **Reservation**) or concurrent (the global system) component.

3.2.3.1 Sequential Component

Class Diagram

For a sequential component, named *T*, we provide an active class interface (named *DynamicT*) associated with a static class interface (named *StaticT*) and a composition link (named *inner*). The active class has an associated statechart augmented by some comments notation. In the interface description, as usual in UML, the receiver object is implicit.

The static class describes the data used by the active class. It follows the general approach developed in [ARRV00a, ARRV00b]. There is only one generator corresponding to instantiation and axioms are written in a simple and object-oriented way inspired by the formal class model.

Statechart Diagram

The statechart describes the dynamic behavior of the component, the event labeled a transition is assumed to correspond to an operation in the active class interface. To facilitate the component translation we note emissions and receipts as comments notation associated to the statechart diagram. A receipt is associated to a parameter of an internal operation of the active class; an emitted value is produced as the result of an operation implicitly associated to a transition. For instance, in the statechart diagram of **Reservation** component, the ***order*** action emits two values associated to operations ***price*** and ***number***, and the ***requestSeat*** action receives ***n:Natural*** (bank account number) as a argument. We distinguish basis operations: ***new()***, which reach an initial state, and internal operations: ***fail()***, ***payment()***, ***success()***, which link two states. Other operations are observers: ***exists()***, ***enough()***. An operation is a total operation if it is possible in every state, and boolean functions (or predicates) are always considered as total ones. The rule to define partiality of other operation is: *an operation must be defined in every state where it is needed.* For example the ***exists()*** operation is partial, it is only referenced in guards from state ***account check*** in Figure 3.

For each state we consider to have a state predicate ***Pstate***. The semantics of operations is described by axioms. The operations of the dynamic part are delegated (if needed) to operations of the static part in a simple way. The axiom ***P2() /\ exists() /\ enough() => inner(success()) = inner(decrease())*** (where "***2***" is a simple reference of state "***account check***", we will talk about it in Chapter 5) means that the ***success()*** operation, occurring in state ***account check*** and with the guards ***exists*** and ***enough***, will ***decrease*** the bank account of the inner static instance.

3.2.3.2 Concurrent Component

As you can see in the Collaboration Diagram showed above, the information to describe a concurrent component are a collaboration diagram plus some notations to denote synchronizations and internal communications. There are several ways to express concurrency, synchronization and communication. Mainly there are: process algebra expression, temporal logic formula or state machine. Our semantics of concurrency is based on the synchronous product of statecharts associated to the components. We may specify asynchronous messages by the way of buffer (we have data in our components) and synchronous messages. Thus we restrict our presentation to synchronization and communication (the so-called "rendez-vous"). UML uses several notations based on the Harel'statecharts, Petri net notations and messages.

One example of synchronization representation concerns the *order* action of the seat **Reservation** and the *payment* action of the **Bank**. To express this synchronization we use the UML synch state of the left part of Figure 4. To simplify the figures this synchronization will be represented as in the right part of Figure 4.

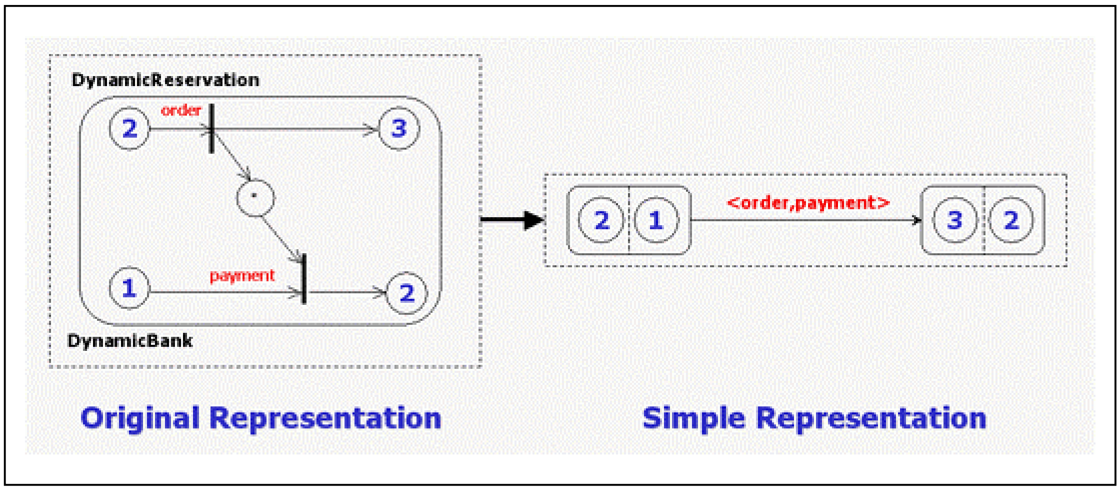


Figure 14: Synchronization Representation

However the drawing of such a diagram, in real case study, becomes too complex. Generally the synchronization rules are based on event names so it is easier to use a class collaboration diagram as in the Figure 6. This gives a better view of the concurrent architecture of the system. Such a kind of concurrency diagram was also suggested in [CPR01a, MM98]. We complete the UML diagrams in the following way:

- To use a class collaboration diagram with some simple notations to note synchronizations and internal communications. Internal communications are described in a comment element attached to this diagram. An emission is prefixed by ! and a receipt is a variable parameter. Of course syntactic compatibility is required between emissions and receipts of a synchronization rule.
- The interaction order in our collaboration diagram is not really useful because such constraints are well defined (and in a more concise way) in the statecharts diagram.
- A last assumption is about the operation signature for the compound system: parameters of an operation are either provided by internal communications or by external communications.

3.2.4 The Description of Work process

Here we used a proper approach based on Graphic Abstract Data Type [Roy01b]. A component is described by a Symbolic Transition System (a STS is a simple form of statechart) and from that an algebraic specification may be built. The semantics of concurrency and synchronization are obtained from the synchronous product of STSs in a similar way than for the synchronous product of automata [Arn94].

First, we built the free product of the two symbolic transition systems. Second we get out the pair of transitions, which are not allowed by the synchronizations. Last, the synchronizations are enriched by communications. An algebraic specification is eventually built from the computed symbolic system. Thus both synchronization and communication are integrated in an algebraic style.

Once we have completed the UML diagrams of sequential component, the generation of the algebraic specification is completely automatic. This is of course a great advantage from a specifier point of view, and also different algebraic languages may be targeted. The approach presented here has two steps:

1. To translate sequential components into algebraic specifications based on its matching UML diagram.
2. To built synchronous product of statecharts of compsite system based on the sequential components, and to generate algebraic specifications for it.

3.3 The Translation Rules from UML to ADT

From the example we show above, each component has two different parts, Dynamic part and Static part, and consequently, when we propose the semantic framework, we try to deal with them respectively, one for static class and another for the dynamic class. And also the translation rules of composite system will be discussed.

Generally speaking, most part of the Dynamic Class Translation Rules are similar with Static Class Translation Rules, but still have some significant difference, the dynamic part is translated by several boolean operations denoting the definedness predicate for each generator operation, the state predicate, and the preconditions according to statechart diagram attached with the Dynamic Class. Here, only different part between them will be mentioned.

Note: We take the Bank Component as an example for all the illustration showed below. And black italic font represents keys word in Larch Prover syntax. And we will divide a LP file into four parts:

- **Heading part:** defined sort, imported data types and used variables
- **Signature part:** describes operator syntax
- **Axiom part:** describes operations semantics
- **Assert sentence:** assert sentence asserts those operations, which can generate the defined

sort, in other words, those operations that return type is defined sort. But there are minor difference between Static Class and Dynamic Class, it will be mentioned in the translation rules specification below.

3.3.1 Static Class Translation Rules

Heading Part

1. We generate an algebraic specification of an abstract data type with sort name **StaticT** (T is defined ort's name).

   set name **sortname**

```
set name StaticBank
```

2. And imports all the data types occurring in the class interface (at least those data types needed for defining the data types of class's attributes).

   declare sort **$\mathbf{datatype_1}$, $\mathbf{datatype_2}$, . . . $\mathbf{datatype_n}$**

```
declare sort StaticBank,List,Real,Natural,Account
```

3. Declare all the variables needed in the axioms specification

   declare variable **$\mathbf{variable_1}$: $\mathbf{datatype_1}$, $\mathbf{variable_2}$: $\mathbf{datatype_2}$, . . . $\mathbf{variable_n}$: $\mathbf{datatype_n}$**

```
declare variable sb:StaticBank,r1:Real,n1:Natural,l1:ListofAccount
```

Signature part

4. The signature has the profiles declared in the static class plus selectors' operations corresponding to the attributes and an instantiation generator. If we have **$\{\mathbf{attr_i} : \mathbf{T_i}\}_{1<=i<=n}$** (where **$\mathbf{T_i}$** is the data type of **$\mathbf{attr_i}$**) as attributes, we generate the following instantiation generator:

   newStaticT : $\mathbf{T_1}$, . . . , $\mathbf{T_n}$ $\rightarrow$ StaticT

```
newStaticBank : List,Natural,Real,StaticBank -> StaticBank
```

5. And the signature of selector function for each attribute in **StaticT** class:

   $\mathbf{attr_{i\,:}}$ StaticT $\rightarrow$ $\mathbf{T_i}$

```
accounts : StaticBank -> List
```

6. The signature of interface operations occurring in **StaticT** class,

   operationName : StaticT, $\mathbf{parameter_1}$, . . . , $\mathbf{parameter_n}$ $\rightarrow$ returnType

```
enough : StaticBank -> Bool
```

where the **StaticT**, which we add into the operation signature as the first parameter, denotes the receiver object of this operation.

Axiom part

7. Axiom part gives a semantics explanation for those operations occurring in the signature part, and when we generate the axioms predicate for each operation, we follow the order as the signature part is generated. And instantiation generator doesn't have a matching axiom part, because its semantics is clear enough with its name, which meas creating a new object of the defined sort.

   For the selector operations, we generate axioms in the following way:

   $\mathbf{attr_{i\,:}}$ (newStaticT(X_1, . . . , X_n)) → X_i

```
price(newStaticBank(r1,n1,l1))=r1;
```

   where **X_i** represents the variable of data type **T_i**

8. We translate the actual axioms, which are described by UML notes element attached with Static Class Diagram, into Algebraic Specification satisfied with Larch Prover syntax.

   (a) adding a **newStaticT(X_1, . . . , X_n)** in first place of the left-hand side conclusion term, which denotes the receiver object, except **newStaticT(X_1, . . . , X_n)** operation and those operation that doesn't occur in the Static Class interface.

   operationName(parameter$_1$, . . . , parameter$_n$) =>
   operationName(newStaticT(X_1, . . . , X_n), variable$_1$, . . . , variable$_n$)

```
exists() => exists(newStaticBank(r1,n1,l1))
```

   (b) replacing each **attr$_i$** token, which occurs in axioms' operations part, the condition expression and the right-hand side conclusion term, by the corresponding **variable$_i$** variable of data type **T_i**. The data type of **attr$_i$** can be found by searching the attributes list of Static Class. For example, in **StaticBank**, attribute **accounts** is **List** data type, **number** is **Natural** data type, and **price** is **real** data type. (and note that, the **variable$_i$** is composed of the first letter of its data type plus the number that it occurs in axioms part.)

   operationName(attr$_i$, . . . , attr$_j$) => **operationName(variable$_i$, . . . , variable$_j$)**

```
has(accounts,number) => has(l1,n1);
```

   (c) for those variables, which are not **attr$_i$** , we will find out its data type by matching it into the operation signature in signature part, and declare this variable explicitly in the head part. For example, we have operation **add(n : Natural) : StaticA** defined in

```
declare variable m:Natural
```

sort **StaticA**, and in axioms part, we have this sentence **add(m) = newStaticA(size, cons(m, contents))**, where **size** and **contents** are attribute and **m** is not a attribute. So we declare:

and the axiom generation:

```
add(m,newStaticA(n1,l1))=newStaticA(n1,cons(m,l1));
```

(This illustration is taken from other example)

Assert sentence

assert sentence asserts those operations, which can generate the defined sort. In static part, only the **newStaticT()** is looked upon as a generator.

assert

***sort* StaticT *generated by* newStaticT;**

```
assert
sort StaticBank generated by newStaticBank;
```

3.3.2 Dynamic Class Translation Rules

Heading Part

1. We generate an algebraic specification of an abstract data type with the same sort of name **DynamicT** (T is defined ort's name). A variable t:**DynamicT** will denote the receiver object.

   ***set name* sortname**

```
set name DynamicBank
```

2. This algebraic specification imports `Boolean` (in Larch Prover, use **Bool** to represent **Boolean** type), `StaticT` as default data type, and all the data types occurring in the interface of the dynamic class.

   ***declare sort* datatype$_1$, datatype$_2$, . . . datatype$_n$**

```
declare sort DynamicBank,Natural,Real,StaticBank,Bool
```

3. We defined all variables occurring in the axioms, while this part can be finished after the analysis of the axioms part (Translation Rule 5).
   ***declare sort* datatype$_1$, datatype$_2$, . . . datatype$_n$**

```
declare sort DynamicBank,Natural,Real,StaticBank,Bool
```

Signature part

4. Translate the class interface into an algebraic signature, and add the necessary operations accordant with Larch Prover syntax, such as definedness predicate, state predicate, and operation precondition predicate, etc. we generate the instantiation generator for **DynamicT** Class like this:
   newDynamicT : StaticT → DynamicT

```
newDynamicBank : StaticBank -> DynamicBank
```

5. The signature of interface operations occurring in **DynamicT** class,
   operationName : DynamicT, parameter$_1$, . . . , parameter$_n$ → returnType

```
payment : DynamicBank,Natural,Real -> DynamicBank
```

   where the **DynamicT**, which we add into the operation signature as the first parameter, denotes the receiver object of this operation.

6. The inner selector:
   inner : DynamicT -> StaticT

```
inner : DynamicBank -> StaticBank
```

   which denotes the **inner** association between **DynamicT** and **StaticT** Class.

7. A definedness predicate:
   DdynamicT: DynamicT -> Boolean

```
DdynamicBank : DynamicBank -> Bool
```

   which is used to verify if a return result from a generator operation is a valid defined sort, and this kind of operation can be used in the state predicate and operation axioms part to define a defined-sort variable. For example, in axioms:

```
account check(payment(db,n,r)) = DdynamicBank(db)/\(middle(db));
```

   which means in order to reach **account check** state after **payment()** operation, **db** should be a valid **DynamicBank** sort and in the **middle** state.

8. One state predicate for each state in the statechart diagram:
   Pstate : DynamicT -> Boolean

```
account check : DynamicBank -> Bool
```

which is used to validate if a defined sort variable is in the **Pstate** or not. It can be used in

```
Csuccess(db) = ((account check(db)/\exists(db)/\enough(db)));
```

the state predicate, operation precondition and operation axioms part to define that a defined-sort variable is in **Pstate**. For example, in axioms:

which means: the precondition of **success()** operation is that **DynamicBank** sort **db** is in **accont check** state, and meanwhile guard condition **exists(db)/\enough(db)** should be satisfied.

9. For each operation one predicate to denote the precondition and noted:
   Coperation: DynamicT, parameter$_1$, . . . , parameter$_n$ → Boolean

```
Cpayment : DynamicBank,Natural,Real -> Bool
```

where **parameter$_1$, . . . , parameter$_n$** represents additional data types required by the operation.

Axiom part

10. We generate axioms for the new operations as following:

(a) For each operation we define the **Coperation** precondition: for each generator we write

Coperation(dynamicT, variable$_1$, . . . , variable$_n$) = exp

```
Csuccess(db) = ((account check(db)/\exists(db)/\enough(db)));
```

where **dynamicT** is a variable of **DynamicT** sort, **variable$_1$, . . . , variable$_n$** are additional variables required by this operation and **exp** is a disjunction of formulas **Pstate(dynamicT)/\guard(dynamicT, variable$_1$, . . . , variable$_n$)** of the different transitions where the operation is possible.

(b) Axioms for the definedness predicate: for each generator operation we write:

DdynamicT(operation(dynamicT, variable$_1$, . . . , variable$_n$))=DdynamicT(dynamicT) /\Coperation(dynamicT, variable$_1$, . . . , variable$_n$)

```
DdynamicBank(success(db)) = DdynamicBank(db)/\Csuccess(db);
```

(c) For each state we define axioms for the state predicates in the following manner: for each generator operation we write

Pstate(operation(dynamicT, variable$_1$, . . . , variable$_n$)) = DynamicT(dynamicT)/\exp

```
middle(fail(db))            =            DdynamicBank(db)/\((account
check(db)/\exists(db)\/enough(db)));
```

where **exp** is the disjunction of the conditions to reach **Pstate** from a transition labeled by generator **operation**. The example showed above means: **middle** state can be reach by **fail()** operation when **DynamicBank** sort **db** is in **account check** state and guard condition **exists/\enough** is not satisfied.

11. Each line in the axioms part of the dynamic class describes an operation's semantics in the dynamic class interface, where it includes two different operations: Generator and Observer Operations, accordingly we have different translations rules for these two kinds of operations.

(a) For generator operation: all the actions, which label the state transitions in the statechart diagram, are generator operations. And we generate axioms predicate for them like this:

DynamicT(t)/\Psource(t)/\Guard(t,*) => axioms line (for this generator operation)

```
DdynamicBank(db)/\account    check(db)/\(exists(db)/\enough(db))    =>
inner(success(db))=decrease(inner(db));
```

where **Psource** is the source state of the state transition where this action take place, and **Guard** is the guard labeling this transition. If there is no source state (for example, the initial and final pseudo-state) we simply generate **Guard(t,*) => axioms line**. Each axiom line in the dynamic class is translated as for the static class (Static Class Translation Rule 8) into the algebraic specification.

In the example showed above, a state transition from source state **account check** to target state **middle**, action is **success()** operation, and guard condition is **exists/\enough**, which means only when this bank account **exists** and has **enough** money to pay for the ticket, **success()** operation can be invoked.

(b) And for observer operation: Usually it is used in guard condition expression of the state transition in the statechart diagram. Then we generate axioms predicate for them like this:

DynamicT(t)/\Psource(t) => axioms line (for this observer operation)

```
DdynamicBank(db)/\account check(db) => enough(db)=enough(inner(db));
DdynamicBank(db)/\account check(db) => exists(db)=exists(inner(db));
```

Where **Psource** is the source state of the state transition, and the guard condition labeled in this transition contains this observer operation, which will be defined.

In the example showed above, a state transition from source state **account check** to target state **middle**, guard condition is **exists/\enough**, which includes observer operation **exists()** and **enough()**. Then we generate the axioms predicate for both of them.

Assert sentence

In dynamic part, the set of generator is the set of internal operations, which return type is the defined sort.

assert

***sort* DynamicT *generated by* newDynamicT, generator$_1$, . . . , generator$_i$;**

```
assert
sort DynamicBank generated by payment,newDynamicBank,fail,success;
```

where **payment : DynamicBank,Natural,Real -> DynamicBank**,
fail : DynamicBank -> DynamicBank, etc, their return types are all **DynamicBank**.

3.3.3 Composite System Translation Rules

From the collaboration diagram (Figure 6) we extract that information concerned with the components, including the synchronization rules and the communications data between components. From that and the sequential component statecharts we generate the synchronous product corresponding to the global dynamic behaviour of the composite system. The principles come from [Arn94, CPR01b]. Each state of the product is a compound state, which has two inner states corresponding to the sequential component states. The transitions of the product are also compound in the way depicted in Figure 7. To take into account the fact that a component may act asynchronously, we use a special nil transition noted "-". Then from this information:

- The state machine product is automatically translated into an algebraic specification.
- If the class has proper operations and axioms, they are translated in a similar way as in the sequential case.

The translation of the state machine product is done as follows:

Heading Part

1. We generate an algebraic specification of an abstract data type with sort name **System**.

```
set name System
```

2. And imports all the data types occurring in the **System** operation interface (and **Bool** is a default data type reserved by Larch Prover syntax, it is not necessary to declare it explicitly.), while **System**, **DynmicA**, **DynamicB**, **StaticA**, **StaticB** are all needed to declare in this part.
   declare sort **datatype$_1$, datatype$_2$, . . . datatype$_n$**

```
Declare sort System,List,Real,StaticBank,DynamicBank,Natural, . . .
```

3. Declare all the variables needed in the axioms specification
   declare variable **variable$_1$: datatype$_1$, variable$_2$: datatype$_2$, . . . variable$_n$: datatype$_n$**

```
declare variable s:System,ls:ListofSeat,r:Real,sb:StaticBank, . . .
```

Signature part

4. The signature contains a **newSystem** generator with profile:
   newSystem : ComponentA, ComponentB → System

```
newSystem : DynamicReservation, DynamicBank -> System
```

   which denotes that the Composite **System** is composed of **ComponentA** and **ComponentB**, and the **newSystem** operation need these two components as parameters.

5. For each kind of transition of the synchronous product we associate a label, which denotes a generator operation. And these generator operations are generated according to the synchronization rules extracted from Collaboration diagram. For example, <**order, payment**>, which means operations pairs **order** and **payment** can be invoked concurrently during the synchronization process. But in fact, there are also asynchronous operations, such as <**requestSeat, -**>, it can be invoked asynchronously in each sequential component respectively. here we use a nil notation "-" to represent that **requestSeat** operation should be synchronized with **"-"** operation, but nothing synchronized actually. We just want to use a uniform synchronization expression to simplify the manipulation process (more details discussed in Chapter 5).
   methodAmethodB : System, parameter$_1$, . . . , parameter$_n$ → System

```
orderpayment : System -> System
```

And how to merge the two operations into integrated one?

Here the profile of these operations is obtained by a merging of the sequential component operation profiles coping with component types and internal communications. And we have two aspect of merging stuff to be considered: the compound operation's name and compound operation's parameters list.

- For the compound operation's name, we just simply combine two operations' name. Such as **<order, payment>** $\rightarrow$ **orderpayment**, while for asynchronous operations, such as **<requestSeat, ->**, we use **left** or **right** to denote its position in the synchronization operation pairs. So here the compound operation's name **<requestSeat, ->** $\rightarrow$ **leftrequestSeat**
- For the compound operation's parameters list, **DynamicA+DynamicB** $\rightarrow$ **System**, and we remove those parameters corresponding to internal communications. The internal data communication information can be retrieved from the comments element of Collaboration Diagram. For example, in the Figure 6, the UML comments notation attached with Collaboration Diagram denotes that **order()** operation emits two value by operations **number()** and **price()**, meanwhile, **payment()** operation receives **number** and **price** as parameters, so these two parameters are internal data communication, which will be neglected in the compound operation's parameters list.

For example the merging of order() and payment() operations:

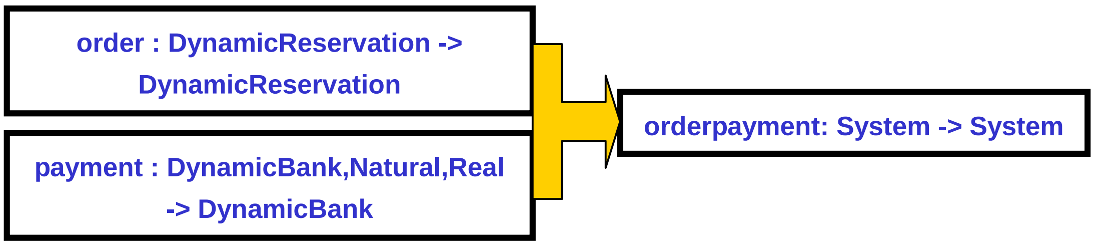

order : DynamicReservation -> DynamicReservation
payment : DynamicBank,Natural,Real -> DynamicBank
orderpayment: System -> System

Figure 15: Merging of two Sequential Component Operations

since there are two internal communications (**price** and **number**) and the **System** type results from the composition of **DynamicReservation** and **DynamicBank**. Note that variables for external communications are not removed (for instance **leftrequestSeat : System Natural -> System**).

9. As the sequential component case, we add following operations in Signature part,

- definedness predicate (**Coperation: System, parameter$_1$, . . . , parameter$_n$ $\rightarrow$ Boolean**),
- the compound state predicates (**PcompoundState : System $\rightarrow$ Boolean**)
- and the operation preconditions for all compound operations (**Coperation: System, parameter$_1$, . . . , parameter$_n$ $\rightarrow$ Boolean**).

Axiom part

10. Also the axioms of the definedness, the preconditions and the state predicates are computed in the same way as the sequential component case.

11. Axioms for the selector operations are defined with the same principles than for the definition of the inner axioms (see Section 4.2) but taking care of asynchronous or synchronous activities and communications between the components.

As you can see the short description above, in the Composite System Axiom translation part it follows most of translation rules as the Dynamic Class dose, then we can reuse the implementation details for Dynamic Class translation rules. But note that there are some minor differences in the representation of Composite statechart diagram, which consists of two sequential component statechart diagrams. For example, in this statechart diagram, we have merging operations, composite guard representation and composite state, etc. but we can just inherit the Class for manipulating the Dynamic Class, and overloading the related operations. It is more easier and integrated way for developing the translation tools. Concrete implementation details will be addressed in chapter 5.

3.4 Conclusion and Summary

The algebraic semantics framework in its actual stage encompasses the formal specifications for the main UML dynamic model elements, such as: Statechart diagram, Collaboration diagram, Classes, Associations (including Compositions). Some other dynamic building blocks of UML can also be incorporated in future by extension, such as Activity diagram, Sequence diagram, Use Case diagram, as well as OCL constraints in the model that can also be translated.

From the formal specifications generated, proofs can be applied over the models and therefore inconsistencies are checked. In future, basing in the formal specifications already achieved, transformations of models can be proved and rapid prototyping from design to code can be implemented.

In order to make clear the final resultant formal specifications for each UML dynamic model element considered, the main translation rules with their result are depicted in the following tables.

3.4.1 Summary table

The rules and saxioms, which numbers are pointed out in these tables, can be found in the corresponding section of the translation from the UML model element to ADT.

Static Class	*Formal Operator*	*Rules and Axioms*
1. sort name	***set name*** **sortname**	**Heading Part 1**
2. imports data types	***declare sort*** **$datatype_1$, $datatype_2$, . . . $datatype_n$**	**Heading Part 2**
3. declare variables	***declare variable*** **$variable_1$: $datatype_1$, . . . $variable_n$: $datatype_n$**	**Heading Part 3**
4. new operation	**newStaticT : T_1, . . . , T_n → StaticT**	**Signature part 4**
5. selector operation	**$attr_{i:}$ StaticT → T_i**	**Signature part 5**
6. interface operation	**operationName : StaticT, $parameter_1$, . . . , $parameter_n$ → returnType**	**Signature part 6**
7. for selector operation	**$attr_{i:}$ (newStaticT(X_1, . . . , X_n)) → X_i**	**Axiom part 7**
8. interface axioms translation	**operationName(parameter1, . . . , parametern) => operationName(newStaticT(X1, . . . , Xn), variable1, . . . , variablen)**	**Axiom part 8 (a), (b), (c)**
Assert sentence	***sort*** **StaticT** ***generated by*** **newStaticT;**	**Assert sentence**

Table 3: Formal Specification for Static Class

Dynamic Class	*Formal Operator*	*Rules and Axioms*
4. new operation	**newDynamicT : StaticT → DynamicT**	**Signature part 4**
5. inner selector operation	**inner : DynamicT -> StaticT**	**Signature part 6**
6. definedness predicate	**DdynamicT: DynamicT -> Boolean**	**Signature part 7**
7. state predicate	**Pstate : DynamicT -> Boolean**	**Signature part 8**
8. operation precondition	**Coperation: DynamicT, $parameter_1$, . . . , $parameter_n$ →**	**Signature part 9**

	Boolean	
9. precondition axioms	**Coperation(dynamicT, variable$_1$, . . . , variable$_n$) = exp**	**Axiom part 8 (a)**
10. definedness predicate axioms	**DdynamicT(operation(dynamicT, variable$_1$, . . . , variable$_n$))=DdynamicT(dynamicT) /\Coperation(dynamicT, variable$_1$, . . . , variable$_n$)**	**Axiom part 8 (b)**
11. state predicates axioms	**Pstate(operation(dynamicT, variable$_1$, . . . , variable$_n$)) = DynamicT(dynamicT)/\exp**	**Axiom part 8 (c)**
12. interface axioms translation	**DynamicT(t)/\Psource(t)/\Guard(t,*) => axioms line**	**Axiom part 11 (a), (b)**
Assert sentence	***assert sort* DynamicT *generated by* newDynamicT, generator$_1$, . . . , generator$_i$;**	**Assert sentence**

*note: the similar parts have been ignored.

Table 4: Formal Specification for Dynamic Class

Composite System	***Formal Operator***	***Rules and Axioms***
1. sort name	***set name* System**	**Heading Part 1**
4. new operation	**newSystem : ComponentA, ComponentB → System**	**Signature part 4**
5. transition actions	**methodAmethodB : System, parameter$_1$, . . . , parameter$_n$ → System**	**Signature part 5**
other operations	same rules as Dynamic Class	**Signature part 6**
selector operations axioms	asynchronous or synchronous activities and communications between the components are taken into account	**Axiom part 8**

Table 5: Formal Specification for Composite System

Chapter Four

Technologies Supporting the Semantic Framework

Tools are needed to assist the formal specification process, besides to prove or to verify some parts. In this chapter the tools and technologies used to automate the generation of the formal specifications from a CASE tool are explained. In the context, the Rational Rose UML CASE tool is used to build the UML models, and we use XMI standard to represent the UML graphic models in a text format, Java language is used to implement the translations rules, which are defined in semantics framework proposition in last chapter, Larch Prover interprets the formal specifications generated, and to conduct validations and verification over them, which can lead to early detection of errors and inconsistencies in the software development process. Each of these technologies and their integration are explained as follows.

4.1 The Practical Context to apply the Framework

In order to allow automatic generation of the formal specifications from a CASE tool based on the translation rules described (in section 3.3), some technologies and tools are used in a suitable integrated way. First, the Rational Rose UML CASE tool is used to input a UML model. From this user model, a XMI file, which records all the information of the UML diagram, will be generated by Unisys Rose XML tools add-on for Rational Rose [Uni98], and then ASCII files containing the formal specifications following Larch Prover syntax are generated by XMI2LP, it is a tool developed in Java language with the XML4J (XML file parser for Java) Java package [IBM98]. This generation is automated through a set of APIs (Application Programming Interface) built in Java Package, from which functions can be called by XMI2LP tool. The Java source code invokes these APIs functions in order to be able to access XMI file repositories, from which all the information about the user model can be recovered.

Larch Prover ends this process by interpreting the formal specifications and conducting verification and validation to prove properties and detect inconsistencies about the UML models. Figure 10 shows a scheme of the integration among these different technologies.

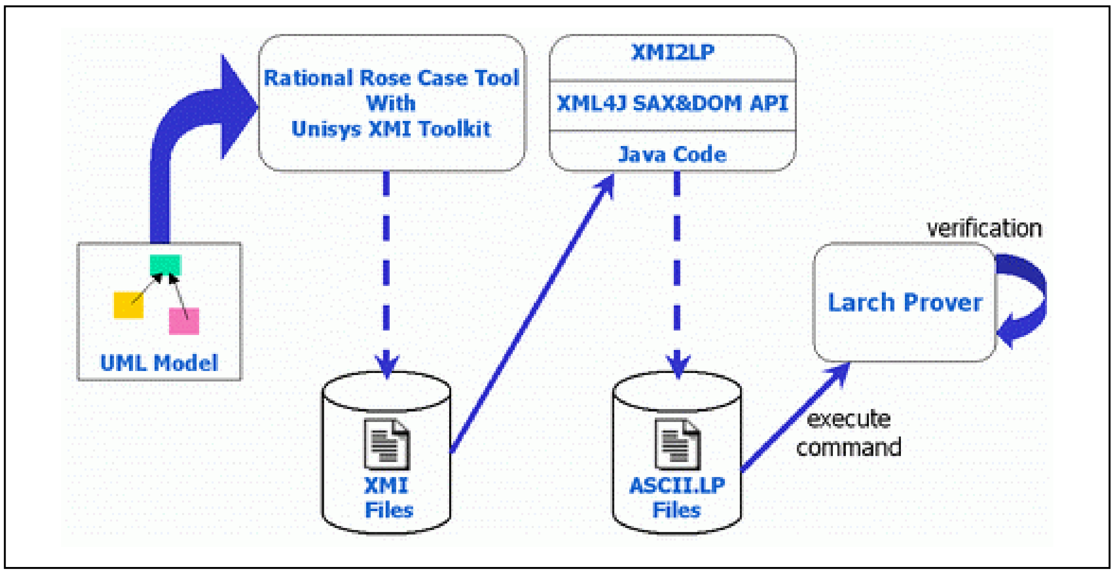


Figure 16: The Tool's Work Process

We can see from the illustration of tool’s work process and short description above, after user input the UML model, all of rest work are totally automatic. This is of course a great advantage from a user point of view, and meanwhile different algebraic languages may be targeted. It makes the proof work more convenient and flexible. In the next sections, each one of these technologies is described.

4.2 Related Tools

4.2.1 Larch Prover

Larch itself is not in fact a language but an approach to define formal specifications being composed by a family of languages and tools. Larch Prover (LP) [GG89], the theorem prover of the Larch family is a set of proving tools that includes: rewriting, critical pair computation, Knuth-Bendix completion, proof by induction, proof by contradiction, and proof by case. LP has simple syntax and semantics, allows the definition of algebraic specifications to describe Abstract Data Types, and allows using rewrite rules to prove properties.

Larch Prover is based on Larch Shared Language (LSL). LSL is a two-tier language of the Larch family which has a top tier that is a behavioral interface specification language (BISL) tailored to a specific programming language, and a bottom tier that is used to describe the mathematical vocabulary used in the pre- and post-condition specifications. Besides the fact that LP is based on LSL it can also uses its own input syntactic format to the formal specifications that is the one followed in this work.

LP allows defining existential propositions (with the **\E** prefix), universal propositions (prefix **\A**) and propositions with usual logical connectors. It also supports first order predicate calculus with equality. The main principle behind LP is the rewrite process: each rule defined by an axiom is rewritten based on an operation in a process that goes until it can be concluded (terminated) or some inconsistency can be detected.

The complete command of LP uses a well-known algorithm: the Knuth-Bendix completion algorithm. This algorithm computes all the critical pairs and adds them in the system. The process stops with an inconsistency, which implies that the system is not consistent. Sometimes the process terminates without inconsistency. Otherwise the system does not terminate. The use of LP to proceed to proofs will be presented in chapter 5, section 5.4.

Other important aspects about LP are that it does not support generality nor partial algebras and the only predefined type is Boolean. The semantics of the LP operations is expressed in axioms written through equations determining equality between terms.

4.2.1.1 A Sample Proof Example

After all, Larch Prover is not a well-known general tool for most of people, as well as its syntax. In order to get a sensitive understanding about Larch Prover at the first sight, we illustrate how to use LP by presenting a sample proof along with explanatory comments. The proof shows the basic operators used in LP and how to conduct a proof process by using several categories axioms.

You can type input command directly in response to LP’s prompts, or you can create a file of LP commands, below the ***set1.lp*** is a sample file.

```
%-----------------
%FileName: set1.lp
%-----------------

declare sorts E, S
declare variables e, e1, e2: E, x, y, z: S
declare operators
  {}:              -> S
  {__}:          E    -> S
  insert:        E, S -> S
  __ \union __:    S, S -> S
  __ \in __:      E, S -> Bool
  __ \subseteq __: S, S -> Bool
  ..

set name setAxioms
assert
  sort S generated by {}, insert;
  {e} = insert(e, {});
  ~(e \in {});
  e \in insert(e1, x) <=> e = e1 \/ e \in x;
  {} \subseteq x;
  insert(e, x) \subseteq y <=> e \in y /\ x \subseteq y;
  e \in (x \union y) <=> e \in x \/ e \in y
  ..
set name extensionality
assert \A e (e \in x <=> e \in y) => x = y
  ..

set name setTheorems
prove e \in {e}
prove \E x \A e (e \in x <=> e = e1 \/ e = e2)
prove x \union {} = x
prove x \union insert(e, y) = insert(e, x \union y)
prove ac \union
```

Table 6: A Larch Prover Sample Proof

Parts Description

The first three commands in ***set1.lp*** declare symbols for use in axiomatizing the properties of sets of elements. The first declare command introduces names for two sorts, `E` and `S`. LP predefines the boolean sort `Bool`. The second command introduces variables ranging over `E` and `S`. These variables will be used when stating axioms and conjectures. The third command introduces symbols for the operators whose properties we will axiomatize. Two periods (..) mark the end of the command. The next several commands in ***set1.lp*** axiomatize the properties of finite sets of elements. And the last part `setTheorems` is sample conjectures we try to prove. The set command directs LP to assign the names `setAxioms.1`, `setAxioms.2`, ... to the axioms introduced by the subsequent `assert` commands. When multiple axioms are asserted in a single command, they are separated by semicolons.

Logical Symbols

The axioms are formulated using declared symbols (for sorts, variables, and operators) together with logical symbols for equality **(=)**, negation **(~)**, conjunction **(/\)**, disjunction **(\/)**, implication **(=>)**, logical equivalence **(<=>)**, and universal quantification **(\A)**. LP also provides a symbol for existential quantification **(\E)**. LP uses a limited amount of precedence when parsing formulas: for example, the logical operator **(<=>)** binds less tightly than the other logical operators, which bind less tightly than the equality operator, which bind less tightly than declared operators like **\in** and **\union**.

Axioms CategoriesAxioms Categories

- **Induction rules:** It provides the basis for definitions and proofs by induction. For example, the first axiom, `sort Set generated by {}, insert`, asserts that all elements of sort S can be obtained by finitely many applications of insert to `{}`.
- **Explicit definitions:** The second axiom, `{e} = insert(e, {})`, is a single formula that defines the operator `{__}` (as a constructor for a singleton set).
- **Inductive definitions:** The next two pairs of axioms provide induction definitions of the membership operator `\in` and the subset operator `\subseteq`. Inductive definitions generally consist of one formula per generator.
- **Implicit definitions:** The final formula, `e \in (x \union y) <=> e \in x \/ e \in y`, in the first assert command, together with the other axioms, completely constrains the interpretation of the \union operator.
- **Constraining properties:** The second assert command formalizes the principle of extensionality, which asserts that any two sets with exactly the same elements must be the same set.

Proof Process

Now we take a look with the proof process of the first theorem in ***set1.lp***.

```
set name setTheorems
prove e \in {e}
qed
```

The prove command directs LP to initiate the proof of a conjecture, and the **qed** (means qualified) command directs LP to confirm that its proof is complete. LP proves this conjecture automatically by using the user-supplied axioms as rewrite rules. When using a formula as a rewrite rule, either LP rewrites terms matching the entire formula to true or, when the principal connective of the formula is = (equals) or <=> (if and only if), LP rewrites terms matching the left side of the formula to terms matching the right. Occasionally LP will reverse the order of terms in an equality to ensure that the resulting set of rewrite rules does not produce nonterminating (i.e., infinite) rewriting sequences. Here's how LP proves the first conjecture:

```
e \in {e} -> e \in insert(e, {})    by setAxioms.2
          -> e = e \/ e \in {}      by setAxioms.4
          -> true \/ e \in {}       by a hardwired axiom for =
          -> true                   by a hardwired axiom for \/
```

4.2.2 The Rational Rose UML CASE Tool

Rational Rose provides the software developer with a complete set of visual modeling tools for development of robust, efficient solutions to real business needs, such as in the client/server, distributed enterprise and real-time systems environments, telecommunications, distributed web-based services etc. Rational Rose products share a common universal standard modeling language accessible to non-programmers wanting to model business processes as well as to programmers modeling applications logic. In out context, we use it to input class diagram, statechart diagram and collaboration diagram, etc, by end user.

4.2.2.1 What can it do?

Model-driven development with UML
Rational Rose is the award-winning model-driven development tool, which is part of Rational Software's comprehensive and fully integrated solution designed to meet today's software development challenges. No matter you're a Software Developer, Project Manager, Engineer or Analyst looking for proven ways to build better software faster, Rational Rose is the appropriate tool. You are not oblige to be a programmer before using the Rational Rose, it is a common

Unify development teams
By integrating the modeling and development environments using the Unified Modeling Language (UML), Rational Rose enables all team members to develop individually, communicate collaboratively and deliver better software.

Create robust system architecture
With the ability to create resilient, component-based architectures, Rational Rose lets software processes evolve in a controlled, managed and identifiable way, reducing costs and accelerating time-to-market.

One tool for all your technology needs
Rational Rose offers seamless integration with all of the leading IDEs and latest technologies, maximizing the speed and simplicity of your development efforts.

- Visualize your application as it really is - or as you want it to be.
- Specify the complete structure or behavior of your application.
- Create a template that guides you as you construct your application.
- Build in quality throughout the development lifecycle.
- Document all the decisions you have made along the way.

in fact, other CASE tools can also be taken into account, such as ArgoUML, but Rational Rose is a Leading Visual Modeling Tool in this field, and widely used in industrial field, that's the main reason we choose it as the UML CASE tool. We want the generality.

4.2.2.2 Who use Rational Rose?

It can generate code in C++, Java, CORBA IDL, Visual Basic, and Oracle8 DDL automatically; a single-language Professional edition; and a Modeling Edition with UML (unified modeling language).

- Teams of software developers and architects
  1. Who need to develop, communicate, or understand a software architecture
  2. Using C++, Java, Ada, Visual Basic, PowerBuilder, Smalltalk, IDL, Oracle8 or Forté (Delphi, Centura, Dynasty, JBuilder, Café via 60+ RoseLink Partners)
- Business Analysts and Software Analysts
  1. Who communicate with users and/or software development teams
- Other development team members, including documentation writers and QA engineers
  1. Who need to understand the architecture of a software system.

4.2.3 The Unisys Rose XML Toolkit

Rational Rose features include expanded round-trip engineering, support for UML 1.3, and built-in team development. Developers also can publish Rose diagrams to the Web or reuse them in other environments via OMG's XMI (XML metadata interchange). And The Unisys Rose XML Toolkit is a Rational Rose add-in developed by Unisys Company, used to generate XMI file from a UML diagram automatically. And it support XMI standard V1.0 and V1.1 (UML 1.3 [Uni98]). XMI support gives Rose users the capability not only to save a model in XMI format, but also to open an XMI model in other CASE tools, which support same version XMI standard. Any Rose customer can use this in exchanging information between the new Rational Rose 2000 visual modeling environment and any Rose edition or other tools and environments that require XMI. (It also supports Rose 98i, sp1).

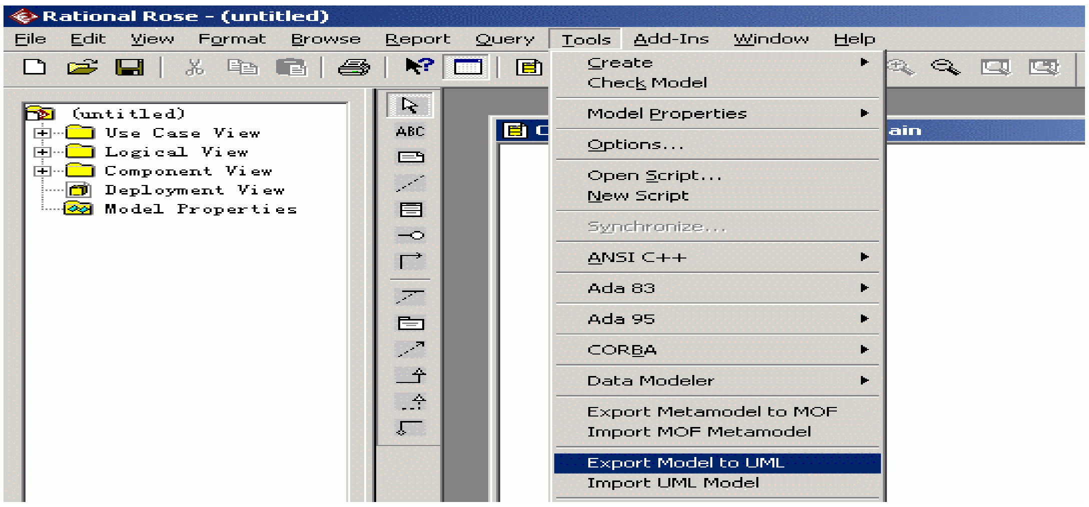


Figure 17: The menu option to run the translation from UML to XMI

4.2.4 XML4J parser

In fact, The XMI format generated form Unisys Rose XML Toolkit, which represents the UML graphic models, is a XML [XML00] (eXtensible Markup language) file format at first; it is a more specialized standard, which is defined for the Metadata Interchange in Visual Modeling field. Consequently it follows the same framework and structure with XML standard. XML Parser for Java (XML4J [IBM98]) is a validating XML parser developed in Java. The package (com.ibm.xml.parser) contains classes and methods for parsing, generating, manipulating, and validating XML documents. Because a XMI file is a XML file at first, we choose it as API for building the XMI document parser.

XML4J version 3.2.0 incorporates the following attributes:

1. W3C XML Schema Recommendation 1.0 support
2. SAX 1.0 and SAX 2.0 support
3. Support for DOM Level 1, DOM Level 2, some features of DOM Level 3 Core Working Draft
4. JAXP 1.1 support

The parser uses both the DTD and the XML document to create a Document Object Model (DOM) tree, which presents the document hierarchically. The DOM provides a group of APIs, which allow access to the elements within the tree. Using the DOM APIs, any element within the XML document can be accessed, changed, deleted or added. Also, the XML parser uses the DTD to validate the document, which involves ensuring the XML document, follows all of the rules specified in the DTD. For example, the DTD rules can specify the valid set of tags, the valid element nesting rules and the attributes, which are associated with a particular element.

Detail information about XML, its DOM structure, DTD and also XMI technology will be talked about in section 4.4.1.

4.2.5 XMI2LP translations Tool

This is a prototype processing tool developed for my thesis work, and XMI2LP means to translate a XMI file into LP syntax file, and it is developed in Java with the XML4J parsing APIs [IBM98]. When we talk about "parsing", we are just talking of an operation that, for example, breaks down a text into recognized strings of characters for further analysis. With the previous version of JAXP 1.0.x (Java API for XML Parsing), we can only open and parse an XML document. When we talk about "processing", we are talking of operations that will allow not just to parse, but also to apply some kind of transformation to the text. We define several classes related to the different parts of the translation. `XMI2LP.java` is the main file for handling the XMI file translation. `CharTool.java` implements java utilities for manipulating string and character variable. `Transition.java, OperationType.java, AttributeType.java, Datatype.java,` and `Association.java` are elements used in XMI2LP tool. A representation of the synchronous product of statecharts was done in XML format, with two new tags. Then there are also classes to implement the translation of the concurrent case, the `CS.java` implement the translation rules for the composite system. Concrete implementation details and tools structure of these tools will be discussed in chapter five.

4.3 XMI (XML Metadata Interchange) Standard

4.3.1 What is XML and Its Capabilities?

To get to know what is XMI? We should take a look at its parents, a very hot and popular standard based on web world - XML. The numbers of applications currently being developed that are based on it, or make use of it, First of all, it is kind of Document, while the word "document" refers not only to traditional documents, like this one, but also to the myriads of other XML "data formats". These include vector graphics, e-commerce transactions, mathematical equations, object meta-data, server APIs, and a thousand other kinds of structured information. It presents a new standard for representing data in a vendor neutral format. It separates the content of the data from the presentation of it. It is a promising format for exchanging data between software systems.

How can it get these advantages?

XML (eXtensible Markup Language) is a markup language for documents containing structured information. It is designed to improve the functionality of the Web by providing more flexible and adaptable information identification. It is called extensible because it is not a fixed format like HTML (a single, predefined markup language). Instead, XML is actually a ‘meta-language’ --a language for describing other languages, which lets you design your own customized markup languages for limitless different types of documents. XML can do this because it's written in SGML, the international standard meta-language for text markup systems (ISO 8879).

Structured information contains both content (words, pictures, etc.) and some indication of what role that content plays (for example, content in a section heading has a different meaning from content in a footnote, which means something different than content in a figure caption or content in a database table, etc.). Almost all documents have some structure.

A markup language is a mechanism to identify structures in a document. The XML specification defines a standard way to add markup to documents.

What does "well-formed" mean?

XML consists of two parts: documents and DTDs (Document Type Declaration). Documents contain the information as a set of tags, while DTDs specify the rules for how tags may be used in a document. Both the document and DTD work together to provide meaningful information. Then a document, which is well-formed, is easy for a computer program to read, verify, and ready for network delivery according to the DTD definition. Specifically, in a well-formed document: All the begin-tags and end-tags match up Empty tags use the special XML syntax (e.g. <empty/>) All the attribute values are nicely quoted (e.g. <a href="http://www.textuality.com/xml.html">) All the entities are declared (entities are reusable chunks of data, much like macros, part of XML's inheritance from SGML).

What does "valid" mean?

In XML, validation means exactly the same thing it does in SGML. A valid document must have a document type declaration, which is a grammar or set of rules that define what tags can appear in the document and how they must nest within each other. The document type declaration also is used to declare entities, re-usable chunks of text that can appear many times but only have to be transmitted once. A document is valid when it conforms to the rules in the document type declaration. Validity is useful because an XML-savvy editor can use the type declaration to help (and in fact require) users to create documents that are valid; such documents are much easier to use and (especially) re-use than those which can contain any old set of tags in any old order.

4.3.1.1 XML Structure

Each XML Document consists of elements specific to that document. Figure 12 shows the structure of as XML element. An element with content has a start tag and an end tag, with the content in between the two tags. Elements without content, often used for structuring, can have a start tag with a slash (/) before the greater than sign (>) to denote that no content exists. Elements can be organized into a structure, much like files are today; the nesting is reflected by the position of the start and end tags.

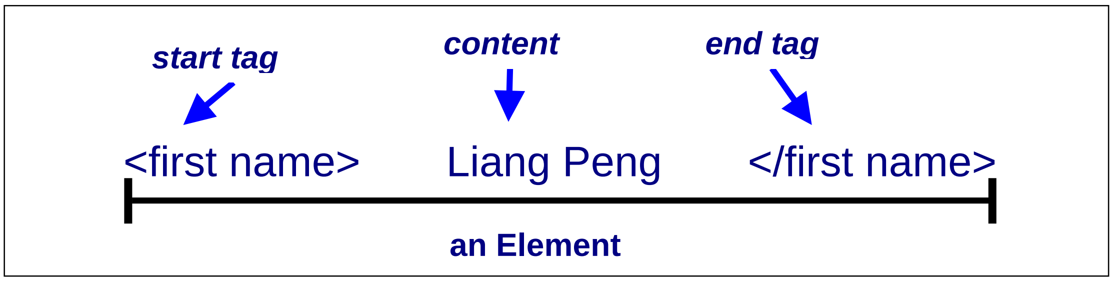


Figure 18: XML Element structure

DOM & SAX

Talking about XML file structure and its processing, we can’t neglect the DOM (Document Object Model)&SAX (Simple API for XML), the most important APIs for XML parsing. SAX is a popular Simple API for XML; it yields a sequence of events corresponding to XML input, incorporating support for Namespaces, for filter chains, and for querying and setting features and properties in the parser. DOM is a standard. It yields a tree representation of the document used by applications at runtime to query and update information within an XML document. In fact, DOM and SAX present two different methods to traversing a XML document, DOM creates a tree structure, where you can insert, remove and alter the nodes in the tree from your application. SAX is an event driven model, which launches events towards your application every time it parses a node. DOM and SAX can be accessed from languages like Java, VB, and ASP etc. In my work, I choose DOM structure as the APIs for processing XML file. And we will present the DOM structure from a sample XML file for Electronic Catalog.

```xml
<?xml version=”1.0”?>
<catalog season="fall">
       <name>Wally's Fall Outdoors Apparel</name>
       <item>
              <itemname>Wool Jacket</itemname>
              <type>
                     <typename>Male</typename>
                     <cost>$50.00</cost>
                     <description>Lite weight Wool Jacket</description>
                     <number>490195M</number>
                     <weight unit="pound">1.5</weight>
                     <shippingcost />
              </type>
              <type>
                     <typename>Female</typename>
                     <cost>$57.50</cost>
                     <description>Unlined Lite weight Wool Jacket</description>
                     <number>490394W</number>
                     <weight unit="pound">1.2</weight>
                     <shippingcost />
              </type>
       </item>
</catalog>
```

Table 7: XML Document for Electronic Catalog

This XML document contains information about two wool jackets, a male version and a female version. Within each `<type> </type>` tag set is the information for each item (jacket).

A Document Type Definition (DTD) should accompany an XML document to be considered valid (W3C propose that XML schema will replace DTD in the near future, but it is not yet a recognized standard, so here we still use DTD to represent the meta-structure of a XML file). The DTD contains the structure of the XML document and any rules about the relationship between elements and any rules particular to an element. The DTD expresses the hierarchy and the nesting of elements within the structure.

The DTD that defines the structure of the catalog document is illustrated in Table 8.

```
<!ELEMENT catalog (name,item*) > <!ELEMENT name (#PCDATA) >
  <!ATTLIST catalog season (winter|spring|summer|fall) #REQUIRED>
<!ELEMENT item (itemname,type*) >
  <!ELEMENT itemname (#PCDATA) >
  <!ELEMENT type (typename,cost,description,weight,shippingcost) >
    <!ELEMENT typename (#PCDATA) >
    <!ELEMENT cost (#PCDATA) >
    <!ELEMENT description (#PCDATA) >
    <!ELEMENT weight (#PCDATA) >
      <!ATTLIST weight unit (pound|kilogram|gram|ton) #REQUIRED>
    <!ELEMENT shippingcost (#PCDATA) >
```

Table 8: Document Type Definition

Where the [*], [+], [?] are qualifiers for the occurring times of each elements in DTD definition, and these three qualifiers are used most frequently in DTD (there are some other qualifiers in W3C XML/SGML standard).
[*] means: the element can occur any times, including 0;
[+] means: the element must occur more than one times, including 1;
[?] means: the element can occur only 0 or 1 time inside its parent tag.

Figure 13 contains an example of the DOM tree for this document. It does not represent the content within the tree structure; the tree must be visualized as a grove or forest of trees representing that content rather than this single structure. Each rectangle in Figure 13 represents a node in the tree and each oval represents an attribute. To keep the figure simple, only the element and attribute names are included.

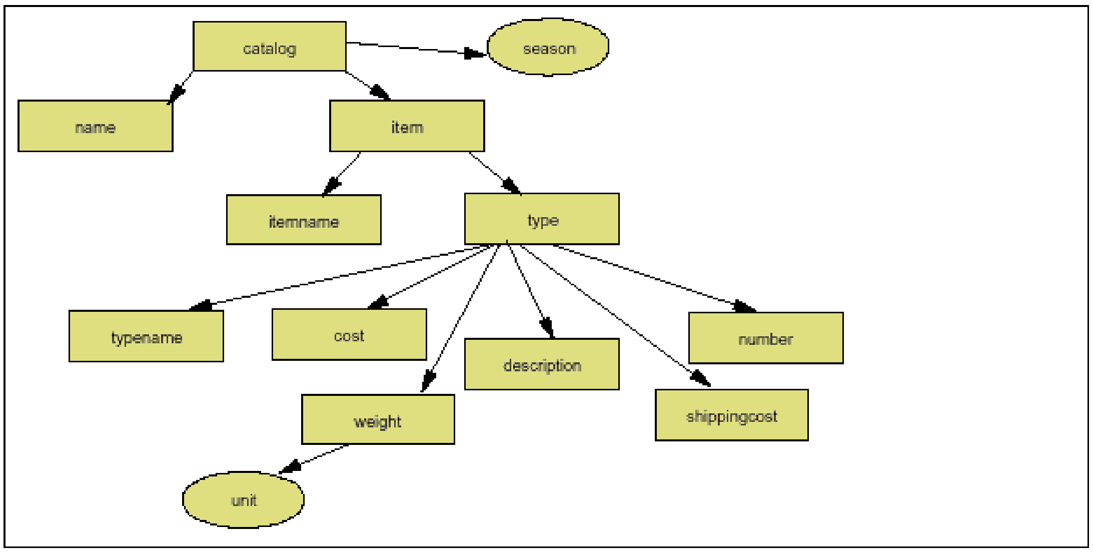


Figure 19: Document Object Model

The DOM provides the methods to access the elements within the tree. By following the tree (hierarchical) structure, the methods allow traversing the tree using method calls for the parents and children. The DOM provides methods to traverse the tree created by the parser. The methods access the elements within the tree using the parent-child relationship.

4.3.2 What is XMI?

The XML Metadata Interchange Format (XMI) specifies an open information interchange model that is intended to give developers working with object technology the ability to exchange programming data over the Internet in a standardized way, thus bringing consistency and compatibility to applications created in collaborative environments. By establishing an industry standard for storing and sharing object-programming information, development teams using various tools from multiple vendors can still collaborate on applications. The proposed standard will allow developers to leverage the web to exchange data between tools, applications, and repositories to create secure, distributed applications built in a team development environment.

And the main purpose of XMI is to enable easy interchange of metadata between modeling tool (based on the OMG UML) and metadata repositories (OMG MOF based) in distributed heterogeneous environments. XMI combines the three key industry standards:

- XML - eXtensible Markup Language, a W3C standard
- UML - Unified Modeling Language, an OMG modeling standard
- MOF - Meta Object Facility, an OMG metamodeling and metadata repository standard

The integration of these three standards into XMI marries the best of OMG and W3C metadata and modeling technologies, allowing developers of distributed systems to share object models and other metadata over the Internet. XMI, together with MOF and UML form the core of the OMG metadata repository architecture as the figure shows. The UML standard defines a rich, object oriented modeling language that is supported by a range of graphical design tools. The MOF standard defines an extensible framework for defining models for metadata, and providing tools with programmatic interfaces to store and access metadata in a repository. XMI allows metadata to be interchanged as streams or files with a standard format based on XML. The complete architecture offers a wide range of implementation choices to developers of tools, repositories and object frameworks. XMI in particular lowers the barrier to entry for the use of OMG metadata standards.

The standard covers the transfer of UML models and MOF Meta models. It identifies standard XML DTD's to allow the exchange of UML and MOF information. Follow on proposals may cover additional domains such as data warehousing, component-based development, and web metadata. XMI will also enable the automatic generation of XML DTDs for each Meta information model.

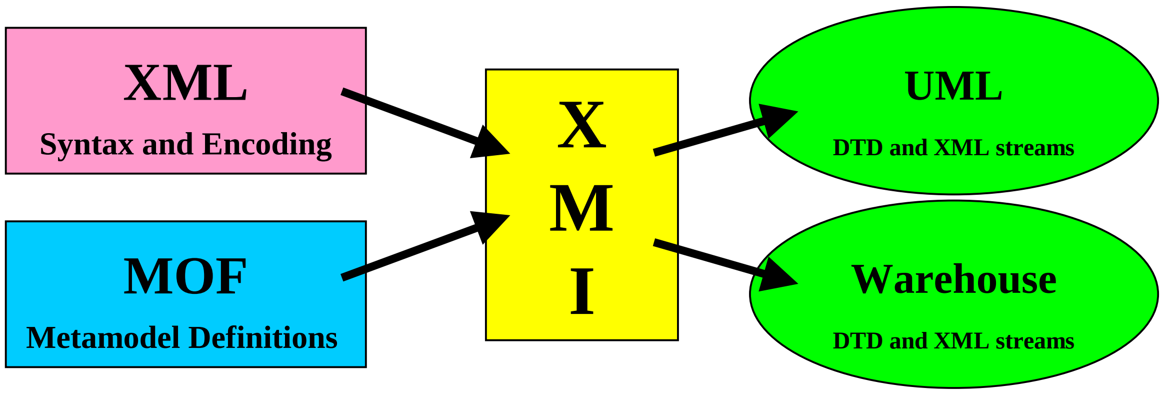


Figure 20: XMI Simplified schema

4.3.3 Why XMI?

So why do we choose XMI format as the interchange standard for describing UML diagram information? There are some alternatives provide besides this, such as the internal representation of UML diagram of Rational Rose, accessing by DLL functions. There are some justifications for that:

- **File/Stream based interchange format**
  XMI is intended to be a "stream" format. Which means, it can either be stored in a traditional file system or streamed across the Internet from a database or repository. It is

easier to manipulate and exchange over the network.

- **XMI is a sub branch of XML technology**
  In fact, XMI standard inherits dozens of benefits and advantages from XML standard, which has been an exploding technology in industry fields. XML is W3C open standard, and includes International ISO character sets. XML has been supported and admitted widely by industry fields, including Web, publishing, repositories, modeling, databases/ warehouses, services, financial, health care, semiconductors, which provide a good basic for the development and application of XMI standard.
  In the meantime, XML is system-independent, vendor-independent, proven with HTML on the web. Its metadata can be delivered via the web, and also validation, tool support, low cost of entry, are taken into account.

- **A universal standard for interchange among a set of application development tools**
  It allows the exchange of objects and software assets throughout your application development environments from the OMG's Object Analysis and Design Facilities. These objects are more commonly described as UML (Unified Modeling Language) and MOF (Meta Objects Facility). XMI is the new industry standard way of doing this, which avoids creating a variety of proprietary formats, each specific to a vendor tool. The diagram below shows the open interchange of XMI, where the major types of application development tools interchange their information using XMI as the standard. These applications include:

- Design tools, including object-oriented UML tools such as Rational Rose and Select Enterprise.
- Development tools, including integrated development environments like Visual Age for Java and Symantec Café.
- Databases, Data Warehouses and Business Intelligence tools, including IBM DB/2, Visual Warehouse, Intelligent Miner for Data, and Oracle/8i.
- Software assets, including program source code (C, C++, and Java) and CASE tools such as TakeFive's SniFF+.
- Repositories, including as IBM VisualAge TeamConnection and Unisys Universal Repository.
- Reports, report generation tools, documentation tools, and web browsers.

To participate in this architecture, vendors only needs to add XMI support to their product to leverage access to all the other tools. The system is open and everyone can participate immediately with XMI-enablement.

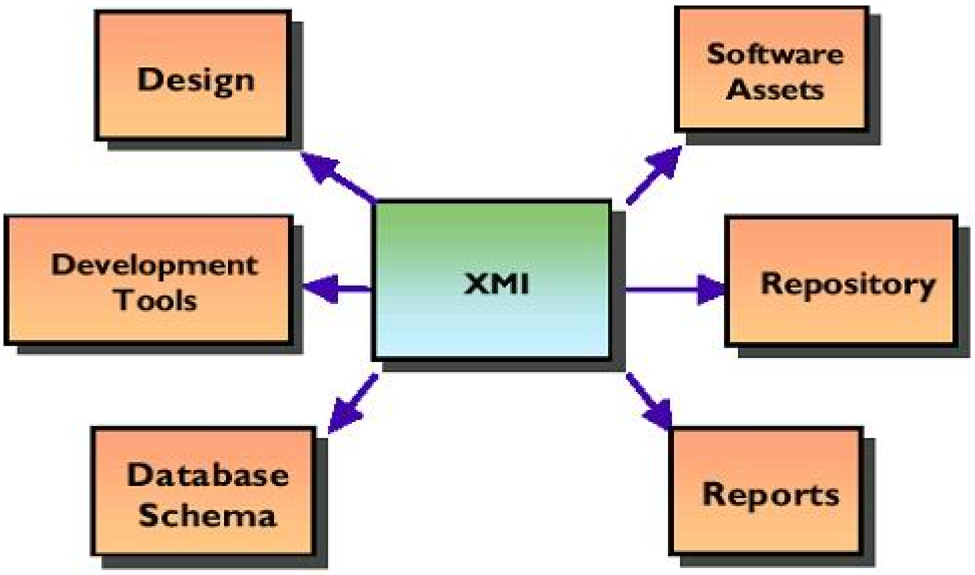


Figure 21: Open Interchange with XMI

4.3.4 XMI file framework

Below is a fragment of UML DTD, which defines the Class element.

```
<! ELEMENT Class (name, visibility, isRoot, isLeaf, isAbstract, isActive, XMI. extension*,
                 constraint*, requirement*, provision*, stereotype*,
                 elementReference*, collaboration*, partition?,
                 template?, view*, presentation*,
                 namespace?, behavior*, binding?
                 implementation*, generalization*, specialization*,
                 parameter*, structuralFeature*, specification*,
                 associationEnd*, participant*, createAction*,
                 instance*, classifierRole*, realization*,
                 classifierInState*, taggedValue*, ownedElement*,
                 feature*)?>
   <! ATTLIST Class XMI.element.att; XMI.link.att;>
   <! ELEMENT name (# PCDATA | XMI. reference)*>
   <! ELEMENT feature (Feature| StructuralFeature| Attribute| BehavioralFeature|
                       Operation| MethodReception)*>
   ...
```

Table 9: UML DTD Fragment

Where the `(name, visibility … )` are sub-elements belong to `Class` element. And

`XMI.element.att; XMI.link.att` are attributes belong to `Class` element, then the definition for each sub-element, `name`, `feature`, etc.

And this is a simple XMI framework eliding most of trivial things, which are not so valuable for our transformation work. We just show a main framework, and then we can get a more legible framework for building the processing tool in Java. (the number followed in each line denotes the tag level in a XMI file)

```
<XMI> 1
  <XMI.header> 2
  </XMI.header>
  <XMI.content> 2
   <!-- ==================== [Model] ==================== -->
   <UML:Model> 3
     <!--  ==================== [Class] ==================== -->
     <UML:Class> 5
      <UML:Classifier.feature> 6
       <!--  ==================== [Attribute] ==================== -->
         <UML:Attribute>
         </UML:Attribute>
         ..
       <!--  ==================== [Operation] ==================== -->
       <UML:Operation> 7
        <UML:BehavioralFeature.parameter> 8
         <UML:Parameter> 9
         </UML:Parameter>
         .. <!-- the last one is return type -->
        </UML:BehavioralFeature.parameter> 8
       </UML:Operation> 7
       ..
      </UML:Classifier.feature>
     </UML:Class>
     <!--  ==================== [StateMachine] ==================== -->
     <UML:StateMachine> 5
      <UML:StateMachine.top> 6
       <UML:CompositeState> 7
        <UML:CompositeState.subvertex> 8
         <UML:SimpleState/> 9
         <UML:Pseudostate/> 9 <!-- this line denotes the initial state -->
        </UML:CompositeState.subvertex> 8
       </UML:CompositeState> 7
      </UML:StateMachine.top> 6
      <UML:StateMachine.transitions> 6
       <!-- ==================== [all transitions] ==================== -->
       <UML:Transition> 7
           ..
           <!-- the action related with this transition -->
           <UML:UninterpretedAction> 11
            <UML:Action.target>
             <UML:ObjectSetExpression/>
            </UML:Action.target>
           </UML:UninterpretedAction>
           ..
       </UML:Transition>
       ..
      </UML:StateMachine.transitions> 6
     </UML:StateMachine>  5
     <!-- ==================== [all datatypes] ==================== -->
     <UML:DataType/> 5
     ..
     <!-- ==================== [all comment elements] ==================== -->
     <!-- but most important is the one whose "annotatedElement=classname"-->
     <UML:Comment> 5
     </UML:Comment>
     ..
   </UML:Model>
  </XMI.content>
</XMI>
```

Table 10: Simple XMI framework

From this Simple XMI framework sketched above, we can see, from more general point of view, the basic structure of a XMI file, and how it can be used to express information of UML diagram, for example, `UML:Attribute` and `UML:Operation` are sub-element of `UML:Class` element, and they can occur 0 or any times inside `UML:Class` element according to the rules, which has been defined in its matching UML DTD file. (Here it is a Class Diagram with statechart diagram).

4.4 Conclusion

In this chapter it was reported how the translation process from UML to algebraic specifications describing ADTs could be automated. The work realized to this automation took into account the integration of Rational Rose, XMI interchange format and XML4J Parser in a suitable way. In the past there was already a project [MA00] developed by

EMOOSE students at Ecole des Mines de Nantes, France, which made use of some of these technologies. The subject of the project was “Translating XMI specifications into UML models”. The purpose is to instantiate automatically the UML meta-model from XMI specifications. A transformation of designs done in XMI to UML modeling language was defined. This project was used in my work as the basis to the development of the Java source code and to perform its integration to the Rational Rose.

And in fact, ArgoUML, another UML CASE tool developed totally in Java language was a better choice for my work, which also provides ability of generating XMI file from UML models, and XML4J parser is a internal component package used by this UML CASE tool, which will be more comparable to extend ArgoUML as the tool for generating Algebraic Specification. But problem is the latest ArgoUML version doesn’t support latest XMI proposed standard XMI 1.1 (maybe it is only a free software developed by fun), and XMI is still a changing standard, we want to catch up with the tide. Unisys XMI toolkit satisfy this requirement and associated with widely used Rational Rose CASE tool.

Chapter Five

Concrete Implementation of the Semantic Framework

In Chapter three and four we've talked about the semantics framework proposed for my work and related technologies contribute to the translation work, and we mentioned the advantage of automatic generation of algebraic specification from a specifier point of view. Then consequence problem is how we can generate the algebraic specification from UML diagrams automatically?

In this chapter, the implementation details of the translation tools will be discussed, including tools architecture, how to access valuable elements in XMI file with XML4J parser, how to generate every part of Algebraic Specification and how to generate the concurrent product based on two sequential components, etc.

And in order to demonstrate how the translation process from UML to algebraic specifications describing in LP syntax works in practical example, two concrete UML sequential components developed in the UML CASE tool - Rational Rose are presented. And after the generation of Algebraic Specification, we will point out the deadlock in the composite statechart diagram by using Larch Prover verification.

5.1 Access XMI information

In chapter five, we talked about several technologies, including XMI, which dedicate to the integration of translation tools. Well, a normal XMI is generated according to the rules defined in UML DTD; it contains huge information for describing a UML model. So how to extract valuable information we really need from a XMI file? For example for a class diagram, its attributes, operations' information, and also its matching statechart diagram are crucial stuff we concern with compared to other parts. How to access this information? Below we show the concrete implementation with explanations.

Initial operation:

```
// creating parse document
DOMParserWrapper parser =
    (DOMParserWrapper)Class.forName(parserWrapperName).newInstance();
// where the uri represents the XML file to be parsed
Document document = parser.parse(uri);
```

In the initial operation, we create a `parser` object, which is used to parse XML file, and create the `document` object, which is used to represent the file to be parsed.

Traverse Class Elements

```
// traversing all class elements in this XMI file
NodeList classes = document.getElementsByTagName("UML:Class");
// traversing each class node
traverseClasses(classes);
```

and we use the operation `getElementsByTagName(String tagname)` to get a `NodeList` of all the elements matching the searching name condition. For example, all the elements whose tagname is “`UML:Class`”

Traverse sub-elements inside super-element

```
// separate class list into single one
for (int i = 0; i < classes.getLength(); i++) {
          Element classe = (Element)classes.item(i);
// get attributes information inside a class element
NodeList attributes = classe.getElementsByTagName("UML:Attribute");
// save the attributes information into attributesList hashtable
for ( int i = 0; i < attributes.getLength(); i++ ){
   Element attribute = (Element)attributes.item(i);
   addToGlossary(attribute);
   AttributeType at = new AttributeType();
   at.type = getElementAttribute(attribute,"type");
   at.name = getElementAttribute(attribute,"name");
   attributesList.put(at.name,at);
}
```

Table 11: Java code for accessing XMI information

and for the `attributes` and `operations` information inside each `Class` element, we should extract them just within this `Class` element domain, which is designated by tag `<UML:Class>` and `</UML:Class>`. Otherwise which `Class` these `attributes` and `operations` belong to will be ambiguous.

From the code above, we can see that, the most important operation is `getElementsByTagName(String tagname)`. And note that this operation is an overloading operation, which can be applied both in `Element` and `Document` Class, they are both abstract interface, which extends from `Node` interface.

5.2 For Sequential Component

This tool (named XMI2LP) can transform a XMI file (generated by Unisys Rose XML tools add-on for Rational Rose) into LP syntax file, and it is especially for Sequential Component case. And this tool is a fundamental part during the translation process, because in the composite system, we will reuse this tool to help in generating the LP syntax file over the temporary XMI file. Now we give a detailed specification to this tool.

5.2.1 Tools Architecture

Generally speaking, the objective of this tool is to implement the transformation rules mentioned in Chapter Three – a Semantics Framework Proposition). And how to use the mapping relationship between UML diagram and XMI file to generate LP syntax? I think the Statechart diagram and Axioms information inside every class is most complicated stuff to manipulate.

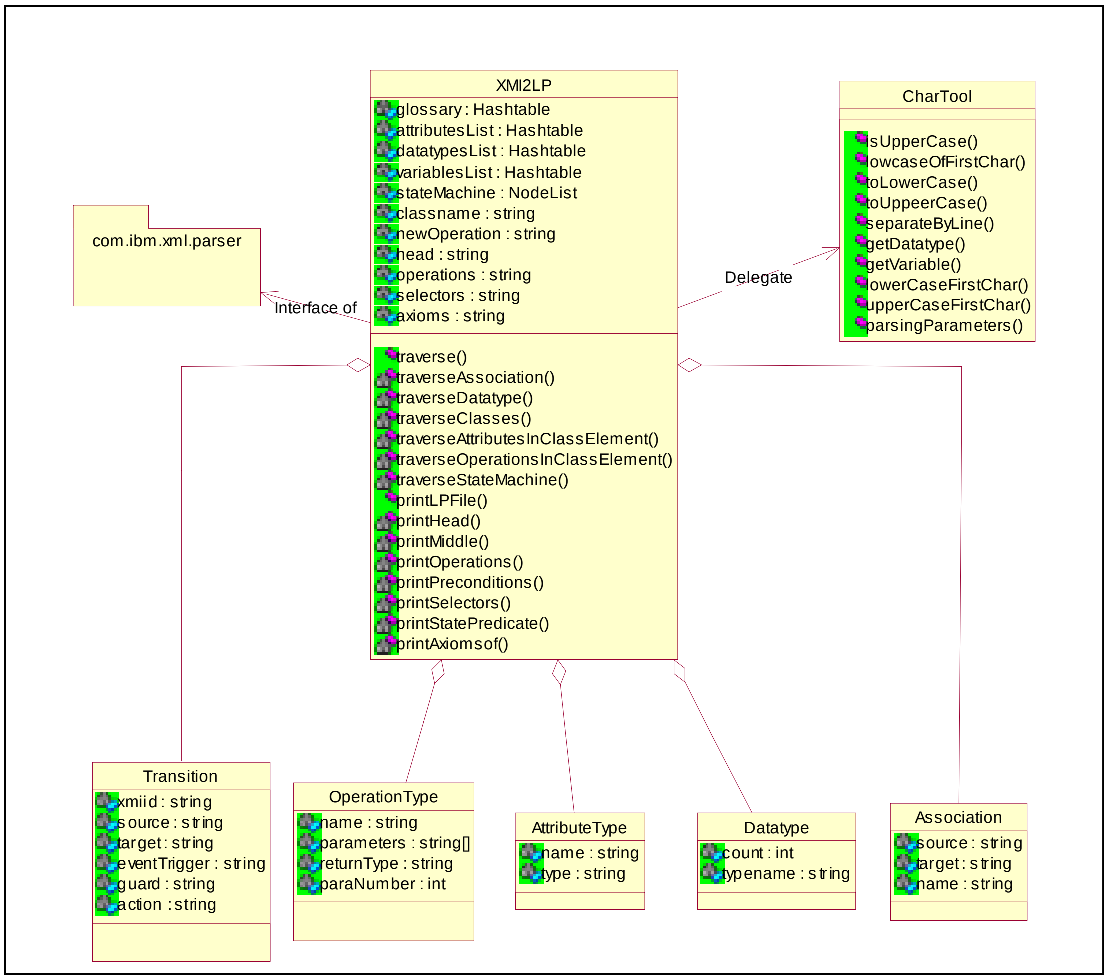


Figure 22: XMI2LP Tool Architecture

1. `XMI2LP.java`: is the main file for handling XMI file. It is in charge of traversing the whole XMI file, and generating LP syntax file as output. The operations `traverse()`, `traverseAssociation()`, `traverseClasses()`, `traverseAttributesInClassElement()`, `traverseOperationsInClassElement()`, etc, are traversing functions, which are used to save the element information into related data structure, such as `OperationType`, `AttributeType` etc. And accordingly, `printLPFile()`, `printHead()`, `printMiddle()`, `printOperation()`, `printPrecondition()`, etc, are operation used to output every part of a LP syntax file.

2. `CharTool.java`: Java utility file for manipulating string and character variable. Some operations are written especially for the variable handling of String and Char data types in XMI2LP. For example `separateByLine()` is used to separate a String variable in comment element into single line String by Carriage Return character. And `lowcareOfFirstChar()` is used to make the first char of a String into lowcase.

3. `Transition.java`, `OperationType.java`, `AttributeType.java`, `Datatype.java`, and `Association.java` are aggregation elements used in XMI2LP file. They are used to save the element information, which is necessary for later LP syntax generating. For example, below is the Java code defining the Class `OperationType`, which is used to save an Operation Element information inside a Class element.

OperationType.java

```
class OperationType {
   String     name;
   String     parameters[];
   String     returnType;
   int       paraNumber;
   OperationType(){
      parameters = new String[5];
   }
}
```

where String variable `name` saves the operation's name; string array `parameters[]` is used to save the parameters' list of this operation; `returnType` is used to save the return type of this operation, and int variable `paraNumber` saves the parameters' number of this operation. Normally, we assume that the parameters number will not exceed 5, and we initialize a parameters String array of five.

5.3 For Composite System

For the Composite System, we build the concurrent product on the base of two sequential components – `ComponentA` and `ComponentB`, and also the Collaboration Diagram of theirs. Because we use XMI (UML XMI 1.1) file format to express UML diagram information, and in the sequential component case, tool XMI2LP use XMI file format as input, and LP syntax file as output, it will make the work easier to get a XMI format of the Composite System. And we generate the XMI file of Composite System based on these two sequential components XMI files with the XMI file of Collaboration Diagram. Below is the illustration for the XMI file generation of Concurrent System (*note: Here we say Concurrent System represents the Composite Statechart diagram of the Composite System).

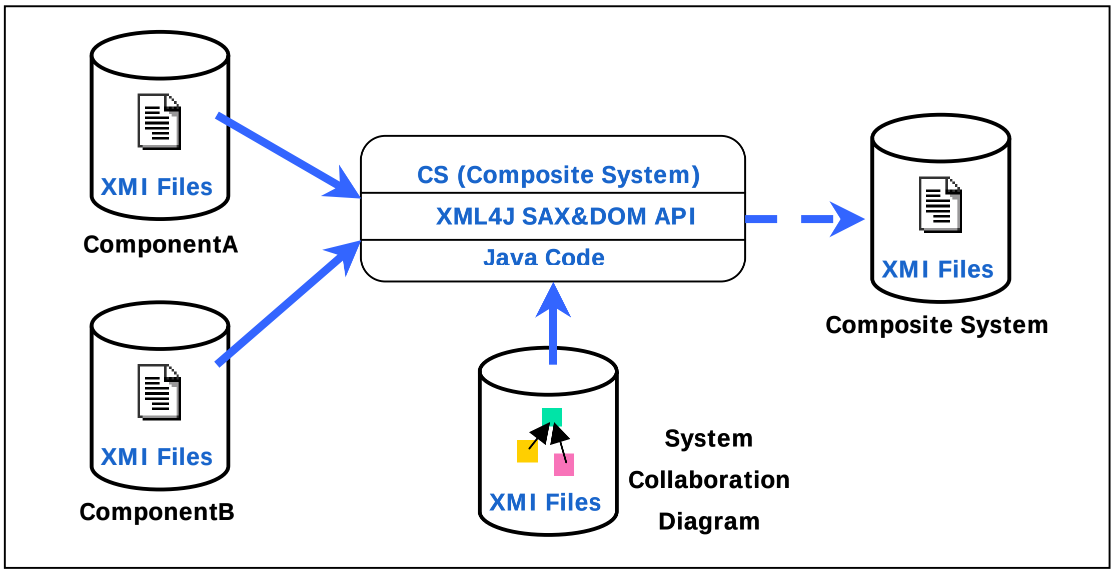


Figure 23: Concurrent Product Generation Process

There are some problems should be resolved, one problem is how to describe the concurrent structure. Here we use the notation **<operation1, operation2>** to express synchronized operations, which means **operation1** and **operation1** should be synchronized during the communication between two components, and note that **operation1** is a operation of `ComponentA`, and **operation2** is a operation of `ComponentB.` The other problem is that each state of the concurrent product is a compound state, which has two inner states corresponding to the sequential component state. In the same way, the transitions of the concurrent product are also compound, and then we should find some way to express that information in the XMI file. All these problems will be discussed in the section below – Driving Technologies.

5.3.1 Driving technologies

Now we move our attention into the detailed implementation. And there are some difficulties we should resolve, for example, how could we represent two states in one compound state, because in standard UML notation, we don't have this kind of notation for representing a compound state and transition. And how to generate those compound states based on the sequential statechart diagrams? How about generation algorithm? Synchronization representation is also a problem.

And we should introduce some new XMI tags for representing particular information in the Concurrent System diagram; our purpose is not changing the UML XMI standard proposed by OMG, but just adding our own syntax. Because our intention of generating this temporary XMI file is not composing a real XMI file, but translating the Concurrent state machine into an algebraic specification automatically by reusing what we have done in the sequential components case. And in this context, this XMI format is a middleware in the work process. At last we translate this temporary XMI file into LP syntax file by tool XMI2LP.

5.3.1.1 New XMI tags and some Convention in my approach

Composite State

Here we combine two simple state "1" (simple representation of sequential component state - `empty`) and "1" into a composite state "11", which has the same tag and syntax with XMI standard.

```
<UML:SimpleState xmi.id="S.1" name="11" visibility="public" isSpecification="false" outgoing=""
incoming="G.7 G.2" />
```

Guard Specification

We introduce two new tags <UML:GuardA.expression> and <UML:GuardB.expression> to describe guard information of the composite transition, where "noguard" means there is no guard condition for this operation.

```
- <UML:Transition.guard>
  - <UML:Guard xmi.id="GU.31" name="" visibility="public" isSpecification="false">
      - <UML:GuardA.expression>
          <UML:BooleanExpression language="" body="2<=size" />
        </UML:GuardA.expression>
      - <UML:GuardB.expression>
          <UML:BooleanExpression language="" body="2<=size" />
        </UML:GuardB.expression>
    </UML:Guard>
  </UML:Transition.guard>
```

xmi.id specification

`xmi.id` is a very important attribute in XMI standard, which label a exclusive identifier for each Element in XMI file, such as `Class`, `Operation`, `Attribute`, etc. And here, what we create is only a temporary XMI file, in other words a section of real XMI file, which only describes the StateMachine information of a XMI file. It is not necessary to distinguish `state` and `transition` elements from all the standard UML elements, which don't appear in this part. Conventionally, we use the uppercase of initial letter of each tagname plus sequential natural number as the `xmi.id` value. For example, we use "S.1" as the `xmi.id` of the first composite state "11".

Synchronized Operation

For each transition of this product, we associate a generator named by a composite operation name. The example shows a composite operation combined with **order** and **payment**. About detailed information of the merging of two operations, see section 3.3.3.

```
<UML:UninterpretedAction xmi.id="UA.20" name="orderpayment" visibility="public"
isSpecification="false" isAsynchronous="false">
```

5.3.1.2 Generation of Concurrent Product

How to generate the concurrent product based on the sequential component state machine? We use a spiral and probing algorithm to generate it. First of all, we put the initial Composite State `(1,1)`, which consist of simple initial state - `[1]` of **ComponentA** and simple initial state - `[2]` of **ComponentB,** (about the simplification of composite state can be found in Section 3.2.3.2 and Section 5.3.1.1) into the `StateList`, and then program probes ahead by testing each possible condition where the Composite State can be lead to another new Composite State. For example, in the first round, Simple State `[1]` can be lead to state `[1]` by **operation1**, and Simple State `[1]` can be lead to state `[2]` by **operation2,** meanwhile the operations' pair **<operation1, operation2>** is a pair of legal synchronized operations in the synchronization rules. Then we add Composite State `(1,2)` into the `StateList`.

1st time	2nd time	3rd time	4th time	5th time	6th time
(1,1)	(1,1)	(1,1)	(1,1)	(1,1)	(1,1)
	(1,2)	(1,2)	(1,2)	(1,2)	(1,2)
		(2,3)	(2,3)	(2,3)	(2,3)
		(2,2)	(2,2)	(2,2)	(2,2)
			(1,3)	(1,3)	(1,3)
			(3,3)	(3,3)	(3,3)
			(2,1)	(2,1)	(2,1)
				(3,2)	(3,2)

Figure 24: Algorithm for the Concurrent System Generation

Probing process will terminate until no satisfied Composite State can be found (in the n and n+1 round, `StateList` result are the same) or all possible Composite States have been found in the `StateList`. The figure above depicts a visual procedure of this algorithm.

5.3.2 Tools Architecture

We develop a separate tool `CS` (Composite System) especially for building the composite system. But we still need analyze State machine information of the sequential component, and we can reuse some methods and attributes of XMI2LP for the sequential case. And because we could not predict how many components the Composite system will be composed of, we suppose that each component should be dealt with separately.

In the Tool architecture diagram below, those operations in `Component` Class is used to analyze StateMachine information of a single sequential component, for example, `State`, `transition` information, etc, and `CS.java` builds Composite System based on the information get from Class `Component`. Below is the tool architecture in UML class diagram.

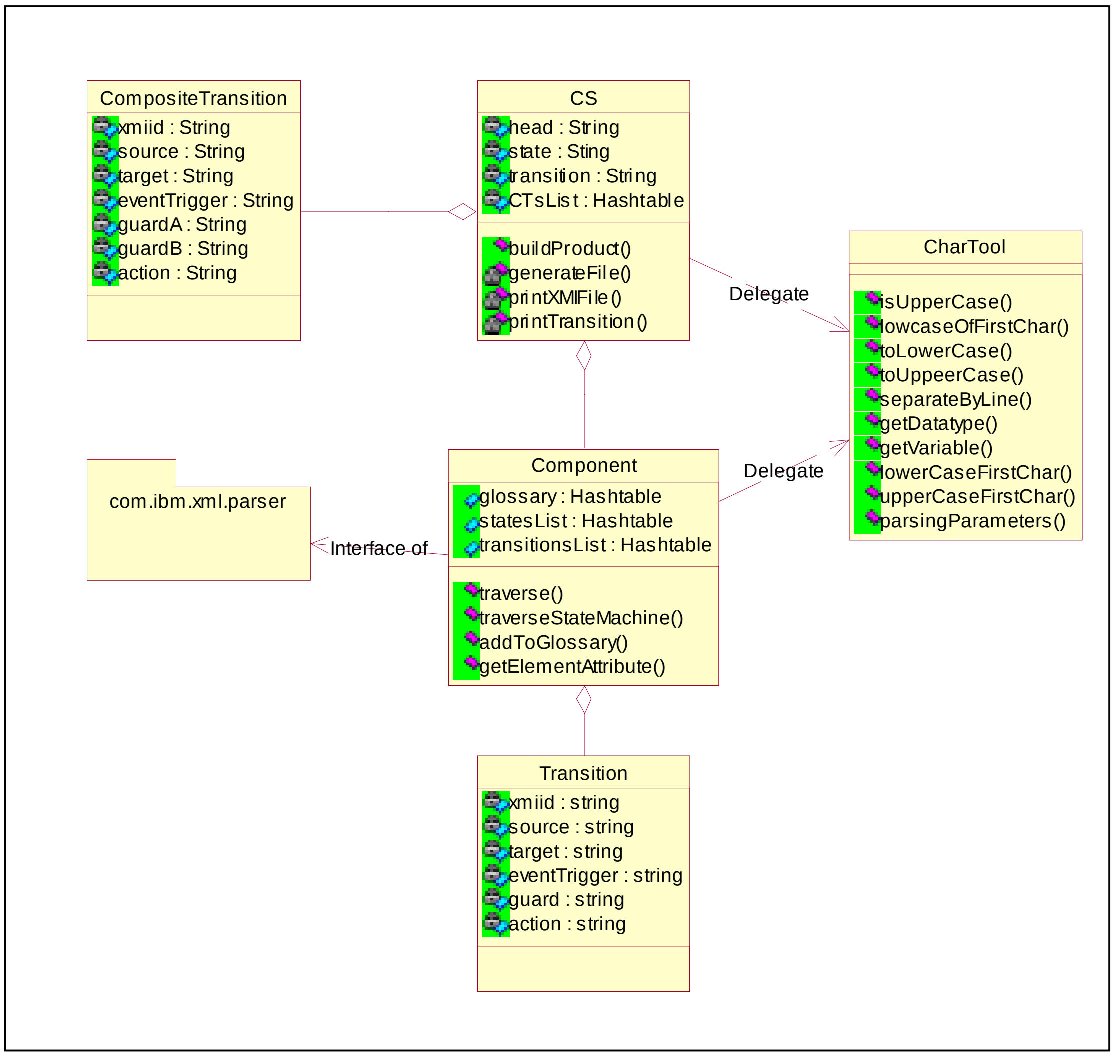


Figure 25: CS Tool Architecture

1. `Component.java:` is java file for handling single `Component`, such as `ComponentA` and `ComponentB`. `traverse()` and `traverseStateMachine()` operations are in charge of traversing the State Machine of the component, and saving the `state` and `transition` information into `StateList` and `TransitionList` Hashtable for the use of `CS.java`.

2. `CS.java:` is the main file for generating XMI file of Composite System, where the `buildProduct()` operation is used to generate the internal data representation of concurrent system. And operation `generateFile()` transforms the internal data representation into XMI representation and saves it.

3. `Transition.java`, `CompositeTransition.java`: are aggregation elements used in `Component` and `CS` Class. They are used to save the transition information in the State machine. And `CompositeTransition` Class is more dedicated to the Composite case,

CompositeTransition.java

```
class CompositeTransition {
   String     xmiid;
   String     source; // source state of the transition
   String     target; // target state of the transition
   String     eventTrigger; // event trigger this transition
   String     guardA;
   String     guardB;
   String     action; // action labeled the transition
}
```

which adds two attributes `guardA` and `guardB` for saving guard condition over **ComponentA** and **ComponentB** respectively.

5.4 Extensions and Related Works

In this thesis research work, due to the time limitation we are able to translate only some UML constructions, and extensions are still needed. They are some previous works, which have shown that some restrictions are easy to consider [RACH00, LB98] then some hypothesis does not really restrict expressiveness. We already studied some extensions like super-state and aggregation state. And we have studied inheritance for static classes in [ARRV00a], for dynamic inheritance a way to solve anomalies may be used [Mes93]. More work should be done to cover a great part of UML.

A related approach is [RACH00], it uses labeled transition system and the algebraic language – CASL (The Common Algebraic Specification Language). One very important difference is that we use (STS) Symbolic Transition Systems in our approach. This avoids many problems of LTS (Labeled Transition Systems). Unlike LTS, our STS labels are operation calls with variables and guards. This concept is related to machines where states and transitions are not necessarily unique objects. A state may represent a set of either finite or infinite objects and a transition collects several state changes. This kind of state machine avoids the state and transition explosion problems and makes dynamic behaviors more readable. But make proof of temporal properties more difficult, however we have described such a way in [Roy01b].

One important work is [HHK98b], we used a more constructive approach for the static diagrams, see [ARRV00a, ARRV00b]. For the dynamic diagram they use pre- and post-conditions written in OCL; this is an interesting alternative. However the main problem would be concurrency and verification, especially temporal verifications.

The notion of component we use is rather linked to component in UML-RT [SR98] than to implementation component of UML, or EJB, Active-X and so on. Thus we need specific notations to define the dynamic interface of a component, its communications with others and concurrency. The present work and UML-RT partly address the same issues: architectural design, dynamic components and reusability. However, UML-RT is at the design level and real time whereas our approach is rather concerned about (formal) specification and logical time issues [CPR01a]. There are also some other difference, mainly at the communication level, but the major one is that, to the contrary of UML-RT, we provides a uniform way to specify both data types and behaviors.

Chapter Six

Conclusion and Future Work

The formalization of Object Oriented analysis and design modeling languages has been claimed as a means to allow rigorous analysis, software comprehension and to guarantee consistency in all software development phases. The rigor imposed by formalization can also support early detection of errors in the development process what avoids that errors are carried till the implementation of the systems.

Even though UML is adopted as the standard Object Oriented modeling language for analysis and design it is not yet formalized.

The thesis of this research has been formalizing UML through the use of a formal abstract language and also giving support to proceed to checks and validations on the formalized models, which brings several contributions to software engineering and reengineering processes. Moreover formalization makes many ambiguities in the semantics arise, and be able to help in solving them.

6.1 Contribution

The main contribution of this work is to provide a basis to achieve a final UML formalization approach that can be used to support software engineering as well as software reengineering efforts. Formalization plays an important role in software engineering and reengineering environments in the sense that it can help in guaranteeing consistency in many stages: among model elements used in a model, between diagrams used to model a system, and between design and implementation through the refinement of models into code (and in the other way around: recovering design from code). Moreover it can contribute towards the specification of a final and unambiguous semantics to UML model elements.

In the semantic framework proposed in this thesis, the main concrete advantage taken is the early detection of errors that can be achieved in the analysis and design phases considering the software development life cycle, especially for checking the deadlock of the dynamic system. Avoiding that errors are carried till the source code, that is really cost effective since errors in the implemented system require more effort and high cost to be eliminated.

In the context explained, many other contributions can be provided in future having the semantic framework as a basis:

- Improving OO legacy systems can be based on formal specifications in order to preserve semantics. Transformations of models based on refinements steps can be performed based on formal proved transformations.
- The formal specifications can make the link between design and implementation. Rapid prototyping generating source code from formal specifications has more chances to make it suitable to the system requirements.
- Ambiguities in UML semantics are solved through formalization.
- System quality and consistency are proved through the application of proofs in the formal specifications generated.

6.2 Conclusion

The presented approach in my thesis suggests a method to present UML component specifications based on previous work around LOTOS and some algebraic stuff. We have shown that a translation of these diagrams is possible and automatic into an algebraic context. This approach is based on a homogeneous semantics for both data types and concurrency; this is a great advantage for verifications. There are other approaches related to this but often they use labeled transition system. One problem is the state and transition explosion problem, which ruins the ability to use model-checkers. Our approach is based on symbolic transition system and this has several advantages. It provides abstraction and readability and this remains close to UML statecharts. We also have means to prove temporal properties and in some case it even allows automatic proofs. We have implemented a tool in Java to run this translation. It uses current object-oriented technology and XMI standard to achieve portability.

6.3 Future work

6.3.1 About Tools

First of all, some problems with the translation tool will be mentioned.

- In the semantic framework presented in this dissertation, because of the limited time available to its development, only part of model elements in UML dynamic part are formal described. Concerning the dynamic Diagram of UML, other model elements (or variations of them) are still to be considered in the formalization. The future will extends our approach to additional UML features: state with activities, complex message and activity diagram and Constraints written in OCL. We also begin to design more complex and real examples and to translate them with our tool thus verifications will become possible.

- And this is only a prototype tool to fulfill testing idea, and it was built especially upon some certain case, for example, there are classes for the dynamic and static part manipulation in the Class diagram respectively. If we change the UML diagram framework, this translation tool will not function any more. So we should improve this tool to be able to deal with any conditions not only different Class Diagram structure, but also different dynamic constructions. After all, this tool introduces some new idea about implementation and presents feasible pattern for the future work.
- In the concrete application section, we proposed a method to generate a temporary XMI file, which follows the XMI syntax. Considering that we only need a temporary XMI syntax file (or part of it actually, because it only includes the statechart diagram part), not a whole representation, we generate these XMI tags and elements by ourselves. It means we create our own functions to write and save a XMI file, but not the universal SAX (Simple APIs for XML) interface. It will make the extension of future work more difficult. To improve it, the XMI Framework is a good choice, which provides a simple Java APIs for saving and loading XML Metadata Interchange (XMI) files and creating XMI DTDs. It supports XMI version 1.0 and version 1.1. You can use the framework object model to represent your data and models, or you can use your own classes. You can also generate Java code from framework models and UML XMI files. You can use any XML parser that supports the JAXP 1.0 interface.

6.3.2 About Semantics Coverage

In this thesis it was presented an approach of a UML formalization method that has been developed making use of algebraic specifications to describe ADTs.

Moreover it is considered the core semantics concerning each model element. Many other points can be considered in order to extend the framework:

- Extensions to the core concepts described are needed in order to have complete semantics specifications for the Behavior Aspects of UML.
- Formalization of the remaining UML dynamic model elements needs to be considered.
- Model transformations need to be formal proved. The translation tools should catch up with the latest standard adopted in the implementation (for example, XMI). This is one of the most important points to achieve with formalization. Through proved transformations, reengineering and forward engineering efforts encompassing model refinements can be supported.

In fact, the main point to consider now is how the results of the proofs and checks obtained in Larch Prover can be demonstrated in the CASE tool to allow end user direct access. And tools can help them solve the appearing problem automatically. I mean correcting the error associations between classes, pointing out the interface operations, which lead to deadlock, etc.

As there was a real time constraint in order to develop this semantic framework, many of these points suggested as future extensions are still under investigation by the collaborators of this work. It is hoped that these extensions as soon as they are achieved, they can be published and widely spread through the interested software engineering and academic community.

References

[AA00] Jose Luis Fernandez Aleman, Ambrosio Toval Alvarez. ***Formally Modeling and Executing the UML Class Diagram.*** In Rodriguez, M.J., Paderewski, P. (eds.): Proc. of the V Workshop MENHIR (Models, Environments, and Tools for Requirements Engineering), Universidad de Granada, Spain (March 2000).

[ADV99] Verónica Argañaraz, Ilse Dierickx, and Aline Vasconcelos. ***A Pattern Representation Tool with UML***. EMOOSE – European Master of Science in Object Oriented Software Engineering. Ecole des Mines de Nantes, France. Vrije Universiteit Brussel (VUB), Belgium. February 1999.

[Arn94] André Arnold. ***Finite Transition Systems***. International Series in Computer Science. Prentice-Hall, 1994. ISBN 0-13-092990-5.

[ARRV00a] Pascal André, Annya Romanczuk, Jean-Claude Royer, and Aline Vasconcelos. ***An Algebraic View of UML Class Diagrams***. In H. Sahraoui C. Dony, editor, Acte de la conférence LMO'2000, pages 261–276, January 2000. ISBN 2-6462-0093-7.

[ARRV00b] Pascal André, Annya Romanczuk, Jean-Claude Royer, and Aline Vascon-celos. ***Checking the Consistency of UML Class Diagrams Using Larch Prover***. In T. Clark, editor, Proceedings of the third Rigorous Object-Oriented Methods Workshop, BCS eWics, ISBN: 1-902505-38-7, January 2000. http://www.ewic.org.uk/ewic/workshop/view.cfm/ROOM2000.

[BRJ99a] Grady Booch, James Rumbaugh, and Ivar Jacobson. ***The Unified Modeling Language User Guide***. Rational Software Corporation. Copyright 1999 by Addison Wesley Longman, Inc.

[BRJ99b] Grady Booch, James Rumbaugh, and Ivar Jacobson. ***The Unified Modeling Language Reference Manual***. Rational Software Corporation. Copyright 1999 by Addison Wesley Longman, Inc.

[CE97] Tony Clark and Andy Evans. ***Foundations of the Unified Modeling Language***. In NFM97: 2nd BCS-FACS Northern Formal Methods Workshop, Ilkley, UK, September 1997.

[CHS+97] Ercüment Canver, Friedrich von Henke, Detlef Schwier, Marie-Claude Gaudel, Nicolas Guelfi, Olivier Biberstein, Didier Buchs. ***Comparison of Object-Oriented Formal Methods***. Universität Ulm 1997.

[CP99] Miro Casanova Paes, ***Formal Representation of UML***. EMOOSE – European Master of Science in Object-Oriented and Software Engineering Technologies. Ecole des Mines de Nantes, France. Vrije Universiteit Brussel (VUB), Belgium. February 1999.

[CPR01a] Christine Choppy, Pascal Poizat, and Jean-Claude Royer. ***Specification of Mixed Systems in KORRIGAN with the Support of an UML-Inspired Graphical Notation***. In FASE'2001, Lecture Notes in Computer Science. Springer-Verlag, April 2001.

[DW98] Desmond D'Souza and Alan Wills. ***Objects, Components and Frameworks With UML: The Catalysis Approach***. Addison-Wesley, 1998.

[EAE99] B.Krieg-Bruckner E. Astesiano and H.-J. Kreowski Eds., editors. ***Algebraic Founda-tions of System Specification.*** IFIP State-of-the-Art Reports. Springer Verlag, 1999. ISBN 3-540-63772-9.

[EBFLR98] A.Evans, J-M. Bruel, R. France, K. Lano, and B. Rumpe. ***Making UML Precise***. OOPSLA'98 Conference on object-Oriented Programming Systems, Languages, and Applications. Vancouver, October 1998.

[FELR97] Robert France, Andy Evans, Kevin Lano, and Bernhard Rumpe. ***The UML as a Formal Modeling Notation***. OOPSLA'97 Workshop on Object-oriented Behavioral Semantics, p. 75-81. Atlanta, Georgia, USA, October 1997.

[GG89] Stephan Garland and John Guttag. ***An Overview of LP, the Larch Prover***. In Proc. of the third International Conference on Rewriting Techniques and Applications, volume 355 of Lecture Notes in Computer Science. Springer-Verlag, 1989.

[GG91] Stephen J. Garland and John V. Guttag. ***A Guide to LP, the Larch Prover***. MIT Laboratory for Computer Science, December 1991.

[HHK98a] Ali. Hamie, J. Howse, and S. Kent. ***Interpreting the Object Constraint Language***. In Proceedings of Asia Pacific Conference in Software Engineering. IEEE Press, January 1998.

[HHK98b] Ali Hamie, John Howse, Stuart Kent. ***Modular Semantics for Object-Oriented Models***. Proceedings of Northern Formal Methods Workshop, eWics Series, Springer Verlag. September 1998.

[HJ95] I. Houston and M. Josephs. ***The OMG's Core Object Model and compatible extensions to it***. Computer Standards and Interfaces, vol 17, nos 5 – 6, 1995.

[HR87] Horst Reichel. ***Initial Computability Algebraic Specifications and Partial Algebras***. International Series of Monographs on Computer Science No. 2. Oxford Science Publications – 1987.

[IBM98] IBM. XML4J. ***Technical report***, 1998. http://www.alphaworks.ibm.com/tech/xml4j.

[Jen97] Jensen.K, ***Coloured Petri Nets: Basic Concepts, Analysis Methods, and Practical Use***. Berlin, Germany: Springer-Verlag, 1997.

[LB98] K. Lano and J. Bicarregui. ***Semantics and Transformations for UML Models***. UML'98 International Workshop. Mulhouse, France. June, 1998.

[MA00] Marc Segura, ***Translating XMI specifications into UML models***, Project for EMOOSE course Spec-Training, Ecole des Mines de Nantes, Dec, 2000

[MEY97] Bertrand Meyer. ***Object-Oriented Software Construction, 2nd Ed***. Prentice-Hall, En-glewood Cliffs, NJ 07632, USA, second edition, 1997.

[MM98] Michael J. McLaughlin and Alan Moore. ***Real-time extensions to UML***. Dr. Dobb's Journal of Software Tools, 23(12):82, 84, 86–93, December 1998.

[PCR99] Pascal Poizat, Christine Choppy, and Jean-Claude Royer. ***Concurrency and Data Types: a Specification Method. An Example with LOTOS***. In J. Fiadero, editor, Recent Trends in Algebraic Development Techniques, Selected Papers of the 13th Workshop on Algebraic Development Techniques, WADT'98, volume 1589 of Lecture Notes in Computer Science, pages 276–291. Springer-Verlag, 1999.

[MW93] M. Ward. ***Abstracting a Specification from Code***. Journal of Software Maintenance: Research and Practice, vol 5, 1993, pp. 101- 122.

[Par72] D.Parnas, ***A Technique for software module specification with examples***. Communications of the ACM 15,5 (1972), 330-336.

[PRR] Liang Peng, Annya Romanczuk, and Jean-Claude Royer. ***A Practical Translation of UML Components into Formal Specifications***. Groupe Objets, Composants, Modèles, Ecole des Mines de Nantes, Equipe Génie Logiciel, Méthodes et Spécifications Formelles, IRIN - Université de Nantes, submitted to TOOLS EASTERN EUROPE 2001.

[RAC94] Jean-Claude Royer, Pascal André, Dan Chiorean. ***Object Design with Formal Classes***. MSF – IRIN – Université de Nantes. April 1994.

[RACH00] Gianna Reggio, Egidio Astesiano, Christine Choppy, and Heinrich Hussmann. ***Analysing UML active classes and associated state machines – A lightweight formal approach.*** In Tom Maibaum, editor, Proc. Fundamental Approaches to Software Engineering (FASE 2000), Berlin, Germany, volume 1783 of LNCS. Springer, 2000.

[Roy01a] Jean-Claude Royer. ***An Operational Approach to the Semantics of Classes: Application to Type Checking***. Programming and Computer Software, to appear 2001. ISSN 0361-7688.

[Roy01b] Jean-Claude Royer. ***Formal Specification and Temporal Proof Techniques for Mixed Systems***. In Proceedings of the 15th IPDPS 2001 Symposium, FMPPTA, San Francisco, USA, April 2001. IEEE Computer Society.

[Roy99b] Jean Claude Royer. ***UML and ADT: A First Approach to Semantics and Verifications***. IRIN – Université de Nantes. June 1999. Internal Document.

[SR98] Bran Selic and Jim Rumbaugh. ***Using UML for Modeling Complex Real-Time Systems***. Technical report, Rational Software Corp., 1998.

[VA99] Aline Vasconcelos, ***Formalization of UML Using Algebraic Specification***. EMOOSE – European Master of Science in Object-Oriented and Software Engineering Technologies. Ecole des Mines de Nantes, France. Vrije Universiteit Brussel (VUB), Belgium. September 1999.

[UML99] ***OMG Unified Modeling Language Specification***. UML Semantics. Version 1.3. January 1999.

[Uni98] ***Unisys Corp. et al. XML Metadata Interchange (XMI)***, October 1998. ftp://ftp.omg.org/pub/docs/ad/98-10-05.pdf.

[Wir90] Martin Wirsing. ***Algebraic Specification***, volume B of Handbook of Theoretical Computer Science, chapter 13, pages 675–788. Elsevier, 1990. J. Van Leeuwen, Editor.

[XML00] ***eXtensible Markup Language (XML) 1.0*** (Second Edition), W3C Recommendation 6 October 2000