FRAMEWORK FOR AUTOMATIC VERIFICATION OF UML DESIGN MODELS: APPLICATION TO UML 2.0 INTERACTIONS

Vitor Nunes de Lima

A thesis
in
The Department
of
Information Systems Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Applied Science in Information System Security
Concordia University
Montréal, Québec, Canada

January 2010

Your file Votre référence
ISBN: 978-0-494-67140-5
Our file Notre référence
ISBN: 978-0-494-67140-5

Abstract

Framework for Automatic Verification of UML Design Models: Application to UML 2.0 Interactions

Vitor Nunes de Lima

Software-intensive systems have become extremely complex and susceptible to defects and vulnerabilities. At the same time, the consequences of software errors have also become much more severe. In order to reduce the overall development cost and assure the security and reliability of the final product, it is of critical importance to investigate techniques able to detect defects as early as possible in the software development process, where the costs of repairing a software flaw are much lower than at the maintenance phase. In this research work, we propose an approach for detecting flaw at the design phase by combining two highly successful techniques in the information technology (IT) industry in the field of modeling languages and verification technologies. The first one is the **Unified Modeling Language** (UML). It has become the de facto language for software specification and design. UML is now used by a wide range of professionals with very different background. The second one is **Model Checking**, which is a formal verification technique that allows the desired properties to be verified through the inspection of all possible states of the model under consideration. Despite the fact that Model Checking gives significant capabilities to developers in order to create a secure design of the system, they are still not very popular in the UML community. There are many challenges faced by UML developers when it comes to combine UML with model checking (e.g., developer are not familiar with formal logics, the verification result is not in the UML notation, and the generation of the model checkers code from UML models is a problematic task). The proposed approach addresses these problems by implementing a new verification framework with support to property specification without using the complexity of formal languages, UML-like notation for the verification results, and a fully automatic verification process.

Acknowledgments

I would like to express my sincere gratitude to my thesis supervisors Prof. Mourad Debbabi and Prof. Lingyu Wang at the Concordia Institute for Information System Engineering. Their expertise, advices and guidance had a major influence for the success of the thesis. I would like also to thank Ericsson Canada Software Research, especially Dr. Makan Pourzandi, for funding and scientific cooperation that made possible this research initiative. Also, a very special thanks to all MOBS2 team colleagues for help, hard work and support.

Last but certainly not least, I would like to dedicate my thesis to my parents and rest of my family for great encouragement and valuable moral support. My thesis is especially dedicated to my beloved wife Juliellen for her irreplaceable love, friendship and support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software-related systems are nowadays part of our everyday life. From simple electronic gadgets to complex satellites, people lives surrounded by these technological innovations. It is evident all the benefits brought by this scientific progress. On the other hand, software products have become extremely complex and susceptible to defects and vulnerabilities. At the same time, the consequences of software errors have also become much more severe. Consequently, the software engineering discipline must now play a predominant role in the process of building secure and reliable software.

The construction of complex software-related systems includes, in summary, requirements engineering, design, code implementation and testing. Requirement engineers typically prepare a requirements document in order to describe the required or expected behavior of the new software. Usually, this document is written in prose with no formalism [5]. The requirements state the anticipated behavior of a system

component in reaction to a sequence of external stimulus. The requirements document is then used, at the design phase, as input for designing models that reflect the desired properties of the system. Developers use these models to build the code at the implementation phase. The models may also serve as input in designing test cases to test the code at testing phase [5].

A great need in software development process is to advance error detection to early phases of the software life cycle. It has been shown that costs of repairing a software flaw during maintenance are approximately 500 times higher than fixing them at early design phase [6]. Figure 1 gives a clear idea of how error introduction, detection and repair costs are distributed through the developments process.

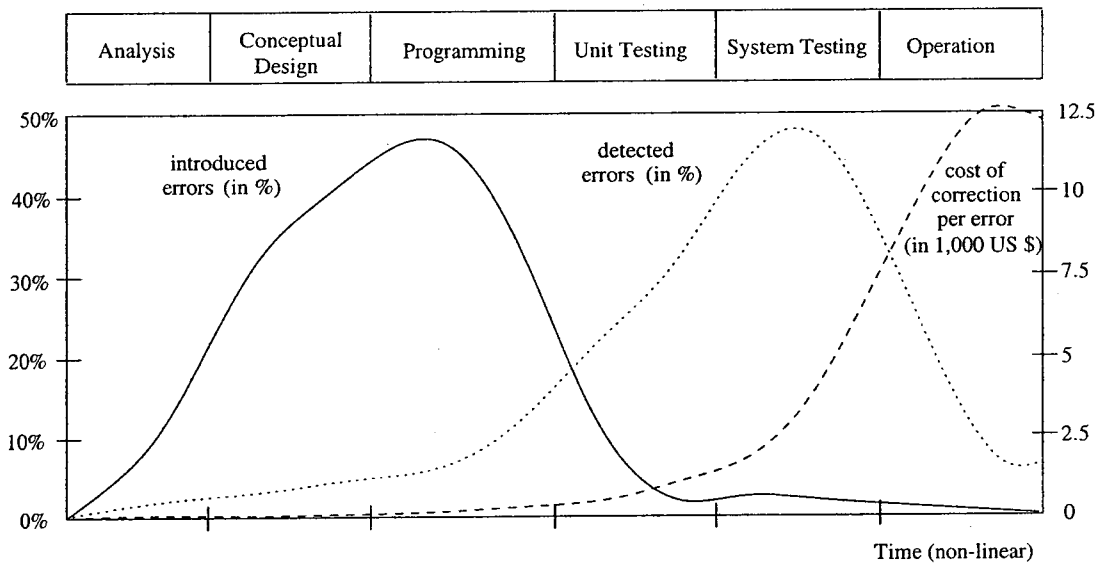


Figure 1: Software lifecycle and error introduction, detection, and repair cost [6]

From the figure, only 15% of flaws are detected at the initial design phase, whereas the cost of fixing them at this phase are extreme low when compared with the cost of corrections at the testing stage where most of the errors are found. E.g., if a design

model contains errors, the errors are transited to the developers during the coding and to the test engineers during the test design. Any code or test case construct will likely present the same errors as the model. These defects may not be revealed until the acceptance test, or worse, after the system is fully installed [5]. Therefore, in order to reduce the overall development cost and assure the security and reliability of the final product, it is of critical importance to investigate techniques able to detect defects as early as possible in the software development process.

For the purpose of advancing error detection, the verification of design models appears as one of the most prominent solutions. Briefly, the objective of this verification is to determine whether the design models actually possess the desired properties specified at the requirement phase. In the state-of-the-art, there are many verification techniques targeting at design models. Including informal approaches such as peer reviewing and testing, as well as formal methods, static analysis, theorem proving, model checking, etc. Different approaches are usually more applicable to different areas.

In this research work, we combined two highly successful techniques in the information technology (IT) industry in the field of modeling languages and verification technologies. The first one is the **Unified Modeling Language** (UML) [44]. It has become the de facto language for software specification and design. UML is now used by a wide range of professionals with very different background, e.g.: software architects, database professionals, business planner, software developer and etc. The second one is **Model Checking**. It is a formal verification technique which allows the desired properties to be verified through the inspection of all possible states of

the model under consideration. The attractiveness of model checking is the following: it is completely automatic, it offers counterexamples in case a model fails to satisfy a property assisting as a valuable debugging in information, and finally, the performance of model-checking tools has proven to be mature since they have been used by a number of successful industrial applications [6]. The combination of UML and Model Checking gives significant capabilities to developers in order to create a secure design of the system and, at the same time, verify the correctness of models at the beginning of the software life cycle.

Even though the model checkers have been successful in implementing high-performance verification algorithms, they are still not very popular in the UML community. There are many challenges faced by UML developers when it comes to combine UML with model checking. First, most of developers are not familiar with formalism used by model checkers. Concepts like temporal logics and labeled transition system are some prerequisites to use model checking techniques. However, most of developers are resistant to get used to these concepts. Second, even if the developers are able to deal with the formalism, the output from the verification tool is not similar to the UML notation and it is not easy be understood in the UML context. Finally, the generation of model checkers code is a very problematic task. It demands specialized knowledge such as: (1) a detailed understanding of the semantics of the UML diagrams in order to extract the correct behavior from the models. (2) an advanced knowledge about the semantics and syntax of the model checker language to guarantee that the generated code reflects exactly the behavior extracted from the UML model. In addition to that problems, a simple UML model may require many lines of model checker code,

which makes it a very tedious and demanding task when the code generation is done manually.

The approach proposed in this thesis addresses all these problems by implementing a new verification framework with support to property specification without using the complexity of formal languages, UML-like notation for the verification results, and a fully automatic verification process. As the model checking verification process is completely automatic, this mechanism can be incorporated into the development methodology without the need to give training to users about the mathematical foundations and the verification algorithms. The result of this approach is an efficient tool for advancing error detection. This tool inherits the rigor and soundness of formal techniques (which is of vital importance when it comes to verification of security-critical and high-reliable software) and, simultaneously, it hides its complexity. Moreover, since it targets at the early stages of the development process, it can reduce considerably the overall development cost.

This thesis is part of the research initiative supported by Ericsson Canada Software Research. This cooperation program aims at developing a Model-Based Framework for Engineering Secure Software and System (MOBS2)[1]. The targeted security concerns are: capturing security requirements, specification and design of security mechanisms, verification and validation of security properties/policies, and automatic generation of secure code. Appropriate security profiles and UML language extensions will be used in the capture of security requirements as well as the specification and design of security solutions.

[1] http://mowglish.ciise.concordia.ca

In the following, we enumerate the objectives of this research work along with the proposed approach and the framework designed to achieve these goals.

1.1 Objectives

This main objective of this thesis consist in proposing a mechanism to advance error detection in the software development process by creating verification framework capable of analyze UML design models using model checking techniques and provide meaningful and easy-to-understand results. This framework needs to provide support for properties specification without using the complexity of formal languages. In addition, the verification procedures need to be transparent to the developer, meaning that the proposed approach should be as much as possible automated in order to hide the complexity of model checking.

More specifically, the detailed goals are the following:

- Investigate the state of the art approaches in the fields of properties specification and software verification at the design level.
- Provide alternatives for the property specification using behavior diagrams, and propose a new language on top of formal logics.
- Define the translation rules of UML models and properties into the input language of the model checker.
- Prototype the approach into a framework for verification design models, and incorporate it into a Integrated Development Environment (IDE).

- Conduct case studies with the objective of demonstrating the feasibility and effectiveness of the proposed approach.

1.2 The Approach

As previously mentioned, our approach comes from the harmonious combination of two successful techniques, software modeling with UML and formal verification using model checking. Figure 2 depicts the overview of our approach.

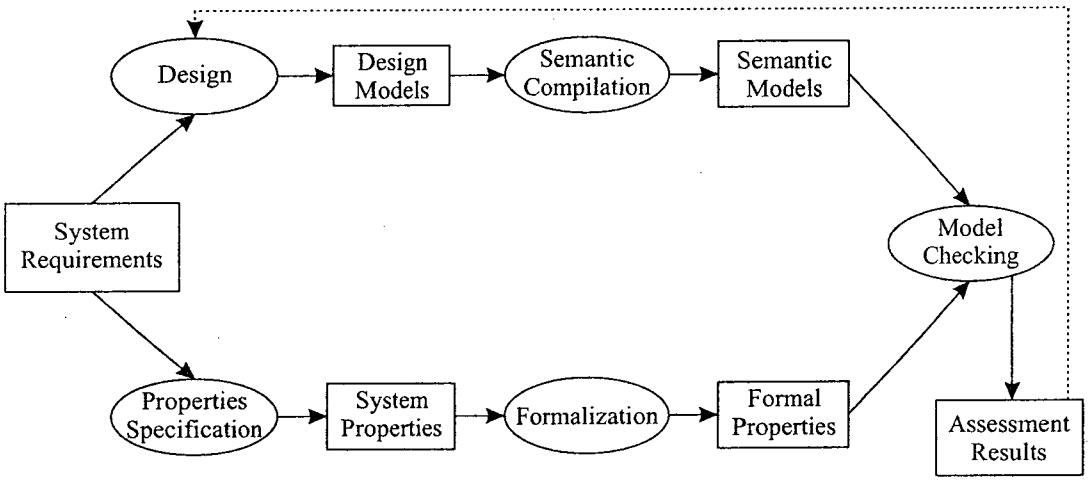


Figure 2: Approach Overview

The verification process stars with the *System Requirements* developed by the requirement engineers. Typically, this artifact is a well-structured document containing all what the system should and should not do. The *System Requirements* are then used as input to two important activities: *Design* and *Properties Specification*.

The *Design* has the purpose of translating the requirements into a specification that describes how to implement the system [32]. The output of this activity is the *Design Models* having the structural and behavioral specification of the system. In

this thesis, we assume UML as the modeling language used in this activity, since it has proved to be very powerful, versatile and well-accepted by the industry.

The *Design Models* needs to be accompanied with a specification of properties of interest to be verified. At the *Property Specification* activity, we provide developer with alternatives (e.g., UML profile, templates, etc) to facilitate the task of writing properties without the need to uses formal languages.

The subsequent steps of the proposed approach are all automatic. The verification engine receives the models and the properties and transforms them into *Semantic Models* and *Formal Properties*, respectively. The *Semantic Models* are derived from the models by following the OMG UML specification [44]. For the *Formalization* of system properties, Linear Temporal Logics (LTL), Computational Tree Logic (CTL), and automata-based properties are used as the underlying formal language.

The *Model Checking* algorithms are implemented by the existing model checking tools. In this work, we decided to use **SPIN** which is a well-known model checker. SPIN is one of most popular and powerful tool for detecting software defects in concurrent system designs. It has been developed at Bell Labs in the eighties and nineties. In 2002 SPIN was recognized by the Association for Computing Machinery (ACM) with its most prestigious *Software System Award.* The tool has been applied in several application from verification of complex call processing software that is used in telephone exchanges, to the validation of intricate control software for interplanetary spacecraft [24].

Finally, the results are presented as an easy-to-understand graph automatically generated by analyzing the model checker's outputs. These results are then utilized

by the developers to refine the design models in order to remove all the found errors.

This main contributions introduced in this approach are the followings: (1) In the *property specification* activity, the new proposed alternatives to write properties are based on UML state machine diagrams and natural language, which are appealing to developers. Moreover, the work of formalization and translation to the input language of the model checker is completely transparent to the developer. (2) The *semantic compilation* is also automatic and transparent to the developer. In addition, our approach can handle the new element of UML 2.0 interaction, which allows developer to verify very complex scenarios with non-straightforward execution trace. Finally, (3) in the assessment results, our approach address the problem of output generated by the verification tool not being similar to the UML notation. Our tool generates graphical results that can be easily compared to model being analyzed. Consequently, a developer can quickly identify the problem when reading results.

1.3 Framework

Our framework demands an underlying UML modeling tool where the developers can create the UML design models. We have chosen IBM Rational Software Architect[2] (RSA) as the environment for development, since it contains a very powerful UML modeler. In addition, it can be augmented with Eclipse plug-ins, which allows the verification engine to be embedded into the development environment.

Figure 3 shows the structure of the main components of the framework. We decided to divided our tool into three layers. The first one is the *Modeling Layer*. This is

[2]http://www.ibm.com/software/awdtools/architect/swarchitect/

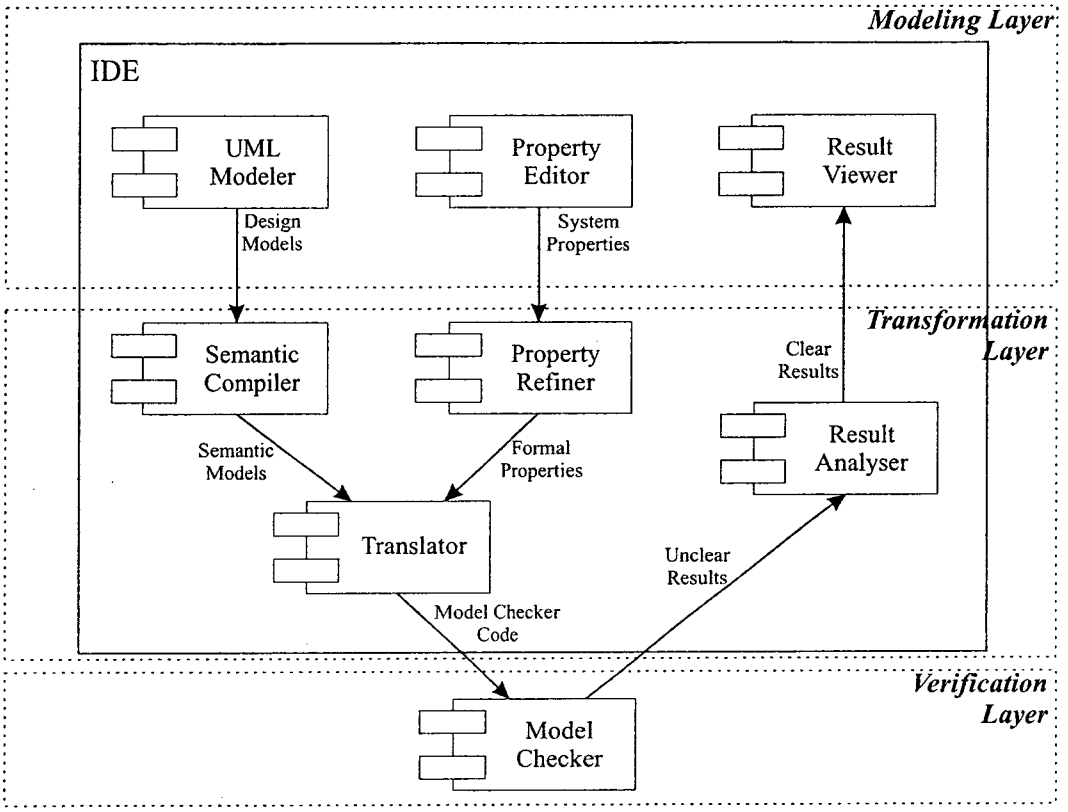


Figure 3: Framework Components Overview

the layer that interacts directly with the user (the developer). It is composed of *UML Modeler*, *Property Editor* and *Result Viewer*. The *UML modeler* comes as part of the IBM RSA and its elements are accessed using eclipses plug-ins. We developed the *property editor* and the *Result Viewer* to facilitate the specification of system properties and to present the result in a friendly manner, respectively. The second layer is responsible to translate models and properties into the input language of the model checker by extracting the semantic models (*Sematic Compiler*), formalizing the properties (*Property Refiner*), and translating then into the model checker code (*Translator*). Moreover, this layer receives the output from the model

checker and analyzes it (*Result Analyzer*) in order to produce meaningful and easy-to-understand results. Finally, the third layer contains the actual verification engine. Herein, we consider only model checking tools, but this layer can expanded to also include other verification tools such as theorem provers or static analyzers. It is import to mention that the first two layers are into the IDE. This feature makes the incorporation of the verification mechanism very smooth and reduces considerably the work with configuration and training.

1.4 Related Publication

The significance of this research work gained scientific visibility with the following publications:

- In [57], we present an extensive survey about usability of security specification approaches for UML Design. It shows how the main adopted can be used for property specification along with a comparative study. The details of this study can be found in chapters 2 and 3.
- In [37], we present an efficient technique for formal V&V of UML 2.0 sequence diagrams. This paper focus on the semantics of UML sequence diagrams and its verification using SPIN model checker. The details about the proposed approach can be found in chapter 4.

1.5 Structure of the Thesis

The reminder of the thesis is organized as follows:

- Chapter 2 provides a review of literature. It shows the theoretical background about the OMG Unified Modeling Language, and software verification. It also depicts the state-of-the-art survey of property specification and verification of UML diagram.

- Chapter 3 shows different alternatives for writing properties. It shows how UML artifacts can be used for property specification. In the end, it presents the new alternatives using UML behavior diagrams and a new language on top of the formal logics.

- Chapter 4 presents the application of our approach in UML interactions. It starts presenting the semantics of UML interactions. Then it shows how to compile interaction semantics into PROMELA semantics, which is used as the input for the SPIN model checker. Subsequently it presents a case study to demonstrating the feasibility and effectiveness of the proposed approach.

- Chapter 5 gives a detailed description of the tool implemented to support our approach.

- Chapter 6 presents the summarizing conclusion of the thesis.

Chapter 2

Review of Literature

This chapter introduces the main concepts needed to support the work developed in this thesis. We present the theoretical background on modeling languages (with focus on UML), starting from the classification of UML diagrams to the different views of the models. In addition, the foundation on software verification is also presented. Finally, the state-of-the-art survey on property specification in UML and UML models verification is detailed in the end of this chapter.

2.1 The OMG Unified Modeling Language

Nowadays, models appear constantly in our routine. Any person, even with no modeling background, is used to read models representing, for example, driving directions, furniture assembling instructions, device safety procedures, and so on. Models are an appealing way of representing a system in many different fields. It is not a surprise that modeling languages are increasingly becoming more and more important in software engineering. Modeling abstracts a real system to a level where only the essentials

aspects matter. It provides means of understanding extremely complex software, as well as it makes the communication among the development team much more efficient and effective [58]. Hereafter, we introduce the modeling language which has become the de facto standard language for software specification and design: UML.

The Unified Modeling Language (UML) is a language and notation system used to specify, construct, visualize, and document models of software systems. Before UML, software developers used to have a collection of mismatched diagram techniques, notation, and semantic approaches [36]. The creation of UML came as solution in order to have an *unified* notation and semantic model. UML covers a wide range of applications and is suitable for technical (concurrent, distributed, time-critical) systems and so-called commercial systems [62]. It is now used in many different ways by people with very different backgrounds. Weilkiens and Oestereich, in [62], enumerate some interesting examples of professionals using UML:

- business planners, as a language to specify the planned operation of a business process, perhaps in concert with a business process language such as the Business Process Modeling Notation (BPMN) [45].
- consumer device engineers, as a way to outline the requirements for an embedded device and the way it is to be used by an end user.
- software architects, as an overall design for a major stand-alone software product.
- IT professionals, as an agreed-on set of models to integrate existing applications.
- database professionals, to manage the integration of databases into a data warehouse, perhaps in concert with a data warehousing language such as the Common

Warehouse Metamodel (CWM) [46].

- software developers, as a way to develop application that are flexible in the face of changing business requirements and implementation infrastructure.

UML is now at the version 2.2. A major update has been done at version 2.0 compared to the version 1.x. UML 2.0 improved behavioral modeling by deriving all behavioral diagrams from a fundamental definition *behavior*, In contrast to UML 1.X where different behavioral models were completely independent [42]. It also improved relationship between Structural and Behavioral Models. Now UML allows to designate that (for example) a State Machine or Sequence is the behavior of a class or a component.

The new version of UML goes beyond the Classes and Objects modeled by UML 1.x to add the capability to represent not only behavioral models, but also architectural models, business process and rules, and other models used in many different parts of computing and even non-computing disciplines [42].

2.1.1 UML Diagrams

There is a wide range of UML diagrams with different capabilities. The OMG UML specification classifies the models into two main categories: *structural* and *behavioral* diagrams. A Structural model shows the static structure of the objects in a system [44], i.e., how the elements are composed. A behavioral model shows the dynamic behavior of the objects in a system, including their methods, collaborations, activities, and state histories [44]. Unhelkar, in [58], proposed an additional classification for

UML diagrams based on the *time* dependency of each diagram. He suggests that UML models can have either a *static* or a *dynamic* nature. Dynamic models are those which display various states of elements and the events that causes state changes, and those diagrams which are *frozen* in time are then static.

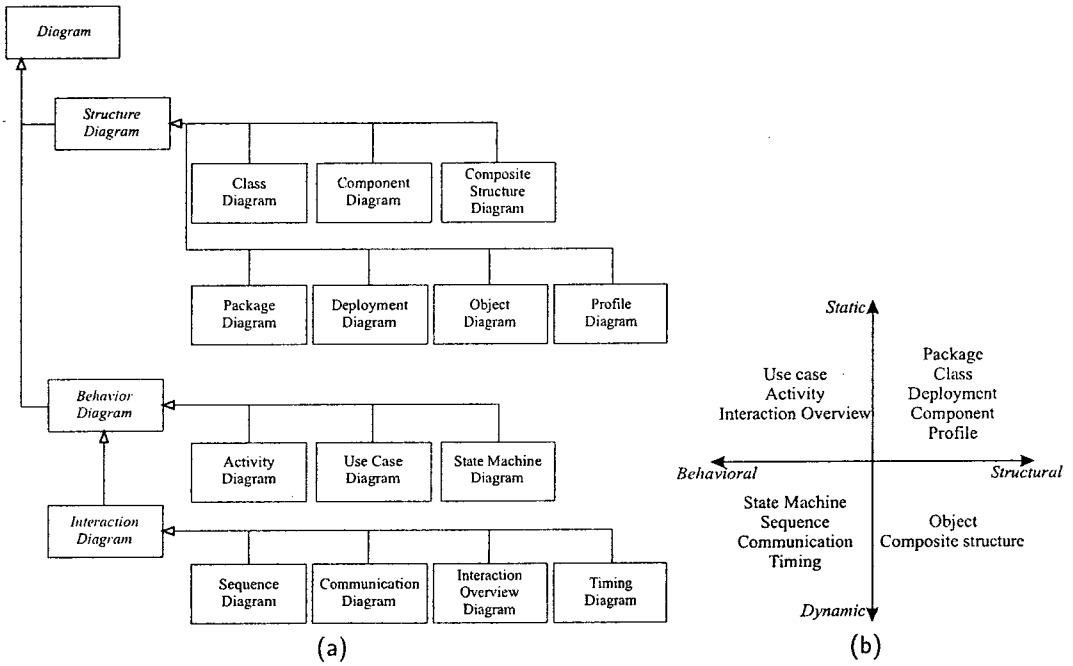


Figure 4: (a) OMG classification of UML diagrams, (b) Diagrams classification including structural and characteristics as well as their static versus dynamic nature [58]

To illustrate the different applications of UML diagrams, figure 5 depicts a hypothetical situation where the system needs to implement two use cases (*login* and *logout*). This requirement is shown in figure 5(a) by the *Use Case diagram.* In order to implement these use cases, a developer can decide to define two classes which are: *User* and *Authenticator.* The static structure of these classes is shown in figure 5(b) as a *Class diagram.* The interaction among the instances of the classes in the login scenario is presented as a *Sequence diagram* in figure 5(c). This diagram shows that

a database with user credentials should also be implemented in this system. Finally, internal behavior of the authenticator is specified using a *State Machine Diagram.*

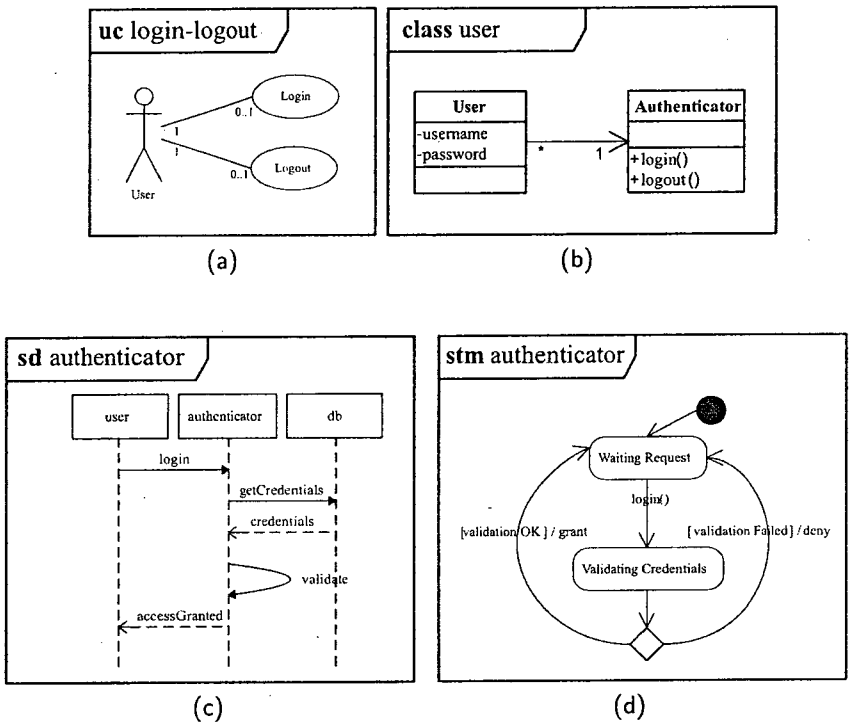


Figure 5: Example of using UML diagrams

Table 1 provide a brief description of all diagrams. It is possible to see that each diagram has a different purpose and a precise strength for particular tasks inside the software development process. Choosing the right set of diagrams to model a system is very important to make the design understandable and approachable [58].

2.1.2 Views of the Model

There are many ways to break up UML diagrams into perspectives or views that capture a particular aspect of a system. In this research work, we follow the Kruchten's

UML Diagrams	Represents
Use case	system functionality from the user's viewpoint
Activity	a sequence of actions of a flow within the system
Class	class, entities, business domain, database
Sequence	interactions between objects
Interaction Overview	interactions at a general high level
Communication	interactions between objects
Object	objects and their links
State Machine	the run-time life cycle of an object
Composite Structure	component of object behavior at run-time
Component	executables, linkable libraries, etc.
Deployment	hardware nodes and processor
Package	subsystems, organization units
Timing	time concept during object interactions
Profile	UML extensions

Table 1: UML diagrams [58]

4+1 view model [31] to describe the role of each diagram in the overall model. This approach has become a de facto standard to classify the views of the model. The 4+1 view model organizes a description of a software architecture using five concurrent view, each of which address a specific set of concerns [31], as shown in Figure 6 .

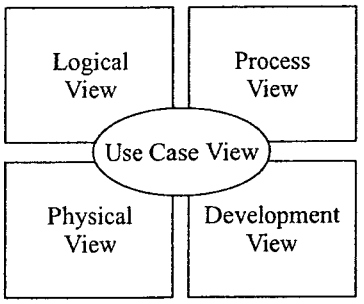


Figure 6: Kruchten's 4+1 view model

Each view is defined in the following [40]:

- The *logical view* describes the abstract description of a system's parts. Used to model what a system is made up of and how the parts interact with each other. The UML diagrams typically used in this view are class, object, state machine,

and interaction diagrams.

- The *process view* describes the processes within your system. It is particularly helpful when visualizing what must happen within your system. This view typically contains activity diagrams.
- The *development view* describes how your system's parts are organized into modules and components. It is very useful to manage layers within your system's architecture. This view typically contains package and components diagrams.
- The *physical view* describes how the system's design, as described in the three previous view, is then brought to life as a set of real-world entities. The diagrams in this view show how the abstract parts map into the final deployed system. This view typically contains deployment diagrams.
- The *use case view* describes the functionality of the system being modeled from the perspective of the outside world. This view is needed to describe what the system is supposed to do. All of the other view rely on the use case view to guide them. This view typically contains use case diagrams, descriptions, and overview diagrams.

2.2 Software Verification

During the recent years, a number of approaches have been proposed to insure the correctness of software-intensive systems. These techniques are either informal or formal as well as manual or automated. *Software verification* approaches focus on checking if a product is being built correctly, i.e., they make sure that the program

functions have the expected behavior. We also investigate *software validation* techniques. These approaches ensure whether the software meets the user's needs, i.e., they check if the correct product is being built.

Among all the proposed approaches, formal methods are now getting a considerable attention from the research community because of their rigor and soundness. This attribute has even more importance in the software security development where the need of vulnerabilities-free software is main goal. Next sections describes the main formal verification approaches as well as the formal language used in these approaches.

2.2.1 Formal Methods for Verification

Due to the fact that software architectures are becoming increasingly complex, it is each time more difficult to assure the satisfaction of all required properties only by using of test-based techniques. In order to overcome this problem, *formal methods* appears as very important option to guarantee high-quality of software-based systems. Formal methods are techniques based on logics, set theory, and algebra for the specification of software systems models and verification of models' properties [15]. The use of formal methods has become widespread, especially during early phases of the software development process. The concept is to create an abstract model of a software system which can be used to verify whether the software under development satisfies a given set of properties. Indeed, the detection and prevention of faults is one of the main motivation for using formal methods. Verifying a formal system specification can help to detect many design flaws; furthermore, if the specification is given in an executable language, it may also be exploited to simulate the execution of the

system, making the verification of properties easier (*early prototyping*) [15]. *Static Analysis*, *Theorem Proving* and *Model Checking* are the three major approaches in the state-of-the-art. These approaches are described below.

Static Analysis

Static Analysis techniques are those approaches that scan for errors using the static specification of a system. In other words, any tool that analyzes a code without executing it is performing static analysis [13]. Static analysis tools are often compared with spell checkers. The latter can rise an alert when the writer makes a well-known mistake, nevertheless they are unable to interpret a text and detect a misuse of certain word (e.g., spell checkers may not tell that you should have used *meet* instead of *meat*). Currently, there exist many commercial and open-source static analysis tools and they are widely used in the software development process. Type checking, style checking, vulnerability finding, security review are only few examples of the applicability of static analysis.

Despite its popularity, there is a common complaint against static analysis tools regarding the number of *false positives*. A false positive is a defect reported by the verification engine where no problem actually exists. Since the runtime behavior is not used in this technique, at many points the tool is not able to determine whether an unclear situation has a real bug. Consequently, it gives several alarms. A very important motivation to generate many false positives is to minimize the number of *false negatives*. A false negative is the most undesired situation where a real problem actually exists, but the verification engine does not report it. On the other hand, a

large number of false positives can induce the developer to overlook some severe bugs.

Theorem Proving

Theorem proving tool is an application that, given a calculus (for some logic) and a formula, attempts to find a proof by repeatedly applying the inference rules of the calculus [54]. There are two main types of theorem provers: *Interactive Theorem Provers* (ITPs) and *Automated Theorem Provers* (ATPs). ITPs are tools where the application of operations and inference rules is performed manually by executing commands and creating command scripts. Whereas ATPs algorithms perform automatic search for a proof. There also rare cases where due to the formalism involved (e.g. hidden circular references leading to a logical paradox), a given conjecture cannot be either proven or disproven [11]. The high level of complexity of this technique is one of the main reasons why it is not yet widely used for the verification of software-related systems.

Model Checking

Model checking is a formal verification approach for detecting behavioral flaws (including safety, reliability, and security-related flaws) of software systems based on suitable models of such systems [15]. Model checking is fully automatic, has a very good coverage, produces valuable results (counterexamples), and it can uncover software defects that might go undetected using other verification techniques. However, there are still some shortcomings in model checking. Indeed, there are still several barriers to fully integrate it into software development process. More specifically, model

checking has a major scalability issue; also, there is still a gap between model checking concepts and notations and the models used by engineers to design systems [15].

Unfortunately, the number of states may proliferate even for relatively simple programs, making the model checking approach computationally very expensive. However, space search algorithms allowing more than 10^{20} states have been available for several years now, and today's model checkers can easily manage millions of state variables. In addition, a number of techniques have been developed to prevent state explosion and to enable formal verification of realistic programs and designs [15].

The theoretical concept supporting model checking techniques is state reachability analysis, which has the advantage of being conceptually simple. Basically, the verifier states the properties it would like the program to possess; then a model checker tool searches the program state space looking for *error states*, where the specified properties do not hold [15]. It is important to mention that the property specification prescribes *what* the system should do and what it should not do, while the model description address *how* the system behaves. If the tool is able to find a state for which the property under consideration fails, it provides a counterexample that indicates how the model could reach the undesired state. The counterexample describes an execution path from the initial system state to a state that violates the property being verified. With a simulator, the user can replay the violating scenario, obtain useful debugging information, and adapt the model (or the property) accordingly [6]. Figure 7 shows an overview of the model checking approach.

From figure 7, it is possible to see that the model checking approach has a very good similarity with the approach proposed in figure 2. Both approaches have activities

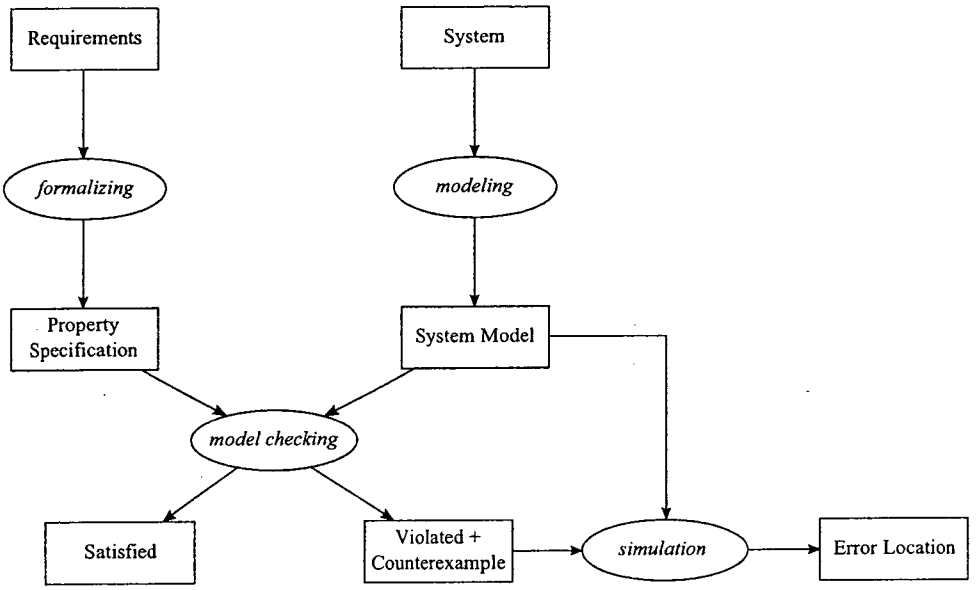


Figure 7: Model Checking Approach Overview [6]

to generate system properties and design models. The assessment results from figure 2 would encapsulate the elements after the model checking in figure 7 (i.e. satisfied, violated + counterexample, simulation and error location). This was one of the main reasons why we decided to choose model checkers as the verification engine supporting the error detection in UML diagrams. In the following, we introduce the SPIN Model Checker, which is verification tool chosen for this research work.

SPIN Model Checker

SPIN is a generic verification system that supports the design and verification of asynchronous process systems [25]. SPIN verification algorithms focus on proving the correctness of process interactions. These interactions can be specified using SPIN rendezvous primitives that allow specification of asynchronous messaging passed through buffered channels and accessed by shared variables. The name SPIN was originally

chosen as an acronym for Simple PROMELA Interpreter, since the specification language that it accepts is called PROMELA. SPIN can be used in two basic modes: as a simulator or as a verifier. In simulation mode, SPIN is used to get a quick impression of the types of behavior that are captured by a system model, as it is being built. This can be of considerable help in the debugging of models. However, no amount of simulation can prove the satisfaction or not of a given property; only a verification run can do so [24]. In the verification mode, SPIN checks if there is at least one execution path that leads to an undesired state, the it uses the simulation mode to display the error trace.

The internal structure of SPIN is shown in figure 8. SPIN comes with a graphical front-end: XSPIN. The PROMELA parser is able to receive properties written in Linear Temporal Logic (LTL) and translate them into the PROMELA language. For the verification purpose, SPIN generates an optimized C code from the PROMELA specification. This code needs to be compiled by a C compiler and the output of the executable file is the result of the verification. If a property fails, an execution trace is generated from the verifier and it can be used guide a simulation showing the steps that lead to undesired situation.

2.2.2 Temporal Logics

Temporal logic is a form of logic specifically tailored for statements reasoning which involve the notion of order in time [8]. Although the term *temporal* suggest a relationship with the real-time behavior, this is only true in an abstract sense. A temporal logic allows the specification of *relative order* of events. It does not support any

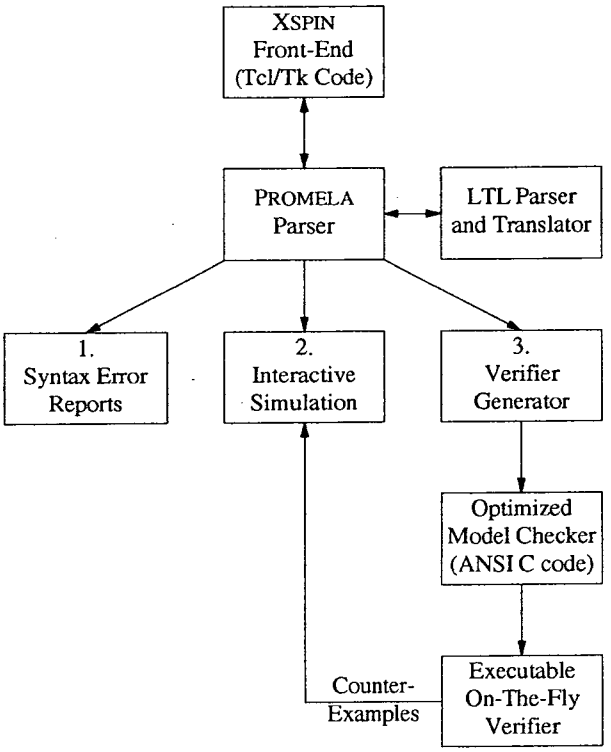


Figure 8: SPIN structure [25]

means to refer to the precise timing of events. In terms of transition systems, neither the duration of taking a transition nor state residence times can be specified using the elementary modalities of temporal logics. Instead, these modalities allows the specification of the order in which state labels occur during an execution, or to assess that certain state labels occurs infinitely often in a (or all) system execution [6].

Temporal logic also offers concepts immediately ready for use. Its operators mimic linguistic constructions (the adverbs "always", "until", the tenses of verbs, etc.) with result the natural language statements and their temporal logic formalization are fairly close. Finally, temporal logic comes with a *formal semantics*, an indispensable specification language tool [8].

In the following, we describe the two most commonly used temporal logics in model checking tools: Linear Temporal Logic (LTL), a temporal logic that is based on a linear-time perspective, and Computational Tree Logic (CTL), a logic that is based on a branching-time view.

Linear Temporal Logic (LTL)

Linear Temporal Logic is a logic for reasoning about properties of computational paths. A LTL formula, in the context of a given execution, considers only one possible future in a moment. It cannot examine alternative executions which split of the execution when a nondeterministic choice is possible [8].

The basic ingredients of LTL-formulae are atomic propositions, the Boolean connectors like conjunction $\wedge$, negation $\neg$, and two basic temporal modalities $\bigcirc$ (next) and $\mathbf{U}$ (until). The $\bigcirc$ operator is a unary prefix operator. The formula $\bigcirc\phi$ holds at the current moment, if ϕ holds in the next "step". The $\mathbf{U}$ operator is a binary operator. The formula $\phi_1 \mathbf{U} \phi_2$ holds at the current moment, if there is some future moment for which ϕ_2 holds and ϕ_1 holds at all moments until that future moment [6].

The until operator allows deriving the temporal modalities eventually (denoted $\Diamond$) and always (denoted $\Box$) as follow:

$$\Diamond\phi \stackrel{\text{def}}{=} \text{true}\mathbf{U}\phi \tag{1}$$

$$\Box\phi \stackrel{\text{def}}{=} \neg\Diamond\neg\phi \tag{2}$$

Figure 9 shows the semantics of temporal operators for the case where the arguments are just atomic propositions.

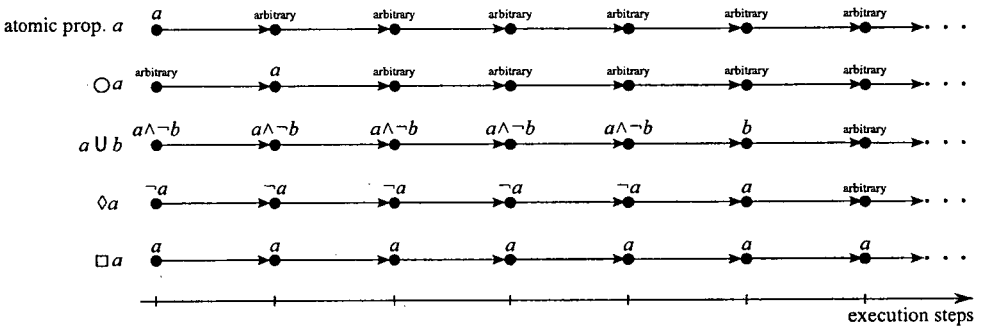


Figure 9: Semantics of Temporal Operators

Computational Tree Logic (CTL)

Computational Tree Logic is a branching-time logic. The semantics of this temporal logic is not based on a linear notion of time, but on a branching notion of time (an infinite *tree* of states). Branching time means that at each moment there may be different future paths. The semantics of branching temporal logic is defined in term of an infinite, directed *tree* of states. Each traversal of the tree starting from its root represent a single path [6].

The temporal operator in branching-temporal logic permit the expression of properties of *some* or *all* computations that start in a state. In order to allow this kind of expressions, CTL supports the existential (denoted $\exists$) and universal (denoted $\forall$) path quantifiers. Figure 10 presents the semantics of the CTL operators.

- In figure 10(a), there are two paths that is possible to reach a white circle. Since the condition to $\exists\Diamond$ *white* be true is to have at least one path reaching a white

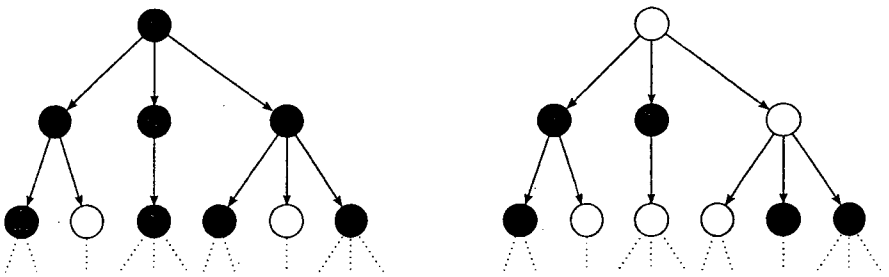
circle, this property holds in that tree.

- In figure 10(b), there is one path that the color is always white. This makes the property $\exists\Box\, white$ true in that scenario.
- In figure 10(c), for all paths there is at least one white color. This is condition for $\forall\Diamond\, white$ to be true.
- In figure 10(d), all the colors are white, then $\forall\Box\, white$ holds in this scenario.
- In figure 10(e), there is a path where the colors are gray until the first black. In other words, grays is true until black. Therefore, $\exists(gray\ \mathbf{U}\ black)$ is true.
- In figure 10(f), the condition grays is true until black holds in all path. Then $\forall(gray\ \mathbf{U}\ black)$ is true.

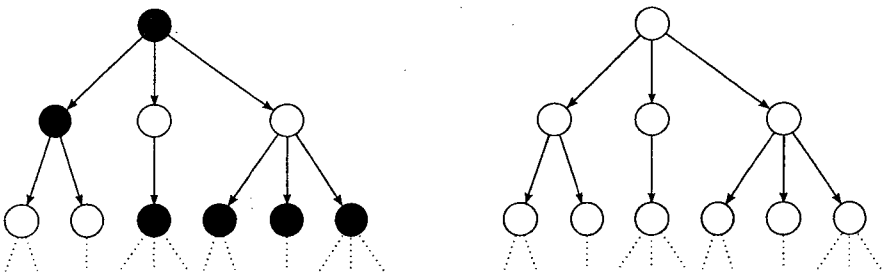
Expressiveness of CTL vs. LTL

Even though CTL and LTL allow the specification of many important properties, those logics are not comparable in terms of their expressiveness. In other words, there are properties that can only be expressed using LTL, whereas some other properties can only be expressed in CTL. Below are some examples of the difference between LTL and CTL.

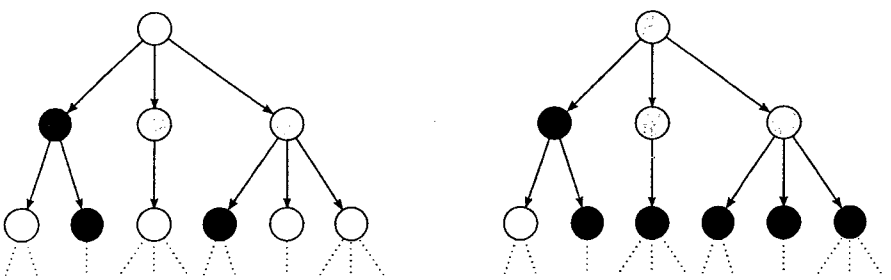
- LTL $\not\subseteq$ CTL
  - $\Diamond\Box a \in$ LTL, but $\Diamond\Box a \notin$ CTL (idea: $\forall\Diamond\forall\Box a \neq \forall\Diamond\Box a$)
- CTL $\not\subseteq$ LTL
  - $\forall\Box\exists\Diamond a \in$ CTL, but $\forall\Box\exists\Diamond a \notin$ LTL (idea: no $\exists$ in LTL)



(a) $\exists\Diamond\, white$ (b) $\exists\Box\, white$



(c) $\forall\Diamond\, white$ (d) $\forall\Box\, white$



(e) $\exists(gray\ \mathbf{U}\ black)$ (f) $\forall(gray\ \mathbf{U}\ black)$

Figure 10: Semantics of CTL operators

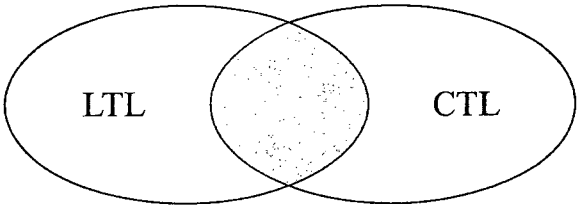


Figure 11: Expressiveness of CTL vs. LTL

There are still many debates regarding the complexity of the verification algorithms in both languages. However, in practice there is no measure that can reliably tell which method can solve a given problem more efficiently [24]. CTL model checkers have been very popular in the development of the early tools for hardware verification, while LTL model checkers have become dominant in applications of software verification [24]. The logic itself, though, is not the only factor that determines the success of a model checkers. Implementation issues as well as the domain where the tool is being applied could also be mentioned as important factors to its performance.

2.3 Property Specification for UML Design

In this section, we present a survey on the main state of the art contributions that are related to specifying and designing security for UML.

The UMLSec approach by Jürjens is among the first efforts in extending UML for the development of security-critical systems [27]. It provides a UML profile where general security requirements such as secrecy, integrity, fair exchange, etc are encapsulated using UML stereotypes and tagged values. It also defines a tailored formal semantics to formally evaluate UML diagrams against weaknesses. In order to analyze security specifications, the behavior of a potential adversary that can attack

various parts of a system is formally modeled. However, UMLSec lacks in expressiveness since security properties are predefined using UML stereotypes and tagged values. This framework cannot be used to specify user-defined properties.

Pavlich-Mariscal *et al.* propose an aspect-oriented approach to model access control policies [48]. They augment UML with new diagrams to represent Role-Based Access Control (RBAC), Mandatory Access Control (MAC) and Discretionary Access Control (DAC) schemes, that are separated from the main design. MAC, DAC and RBAC are decomposed into security features which represent specific elements of an access control policy, e.g. permissions, MAC security properties, delegation rules, etc. This is the only approach that combines MAC, DAC and RBAC into a set of security diagrams separated from the main design. Modeling security as aspects reduces the scattering of access control definitions in the entire application. It is also possible to make changes to the design without impacting the entire security of the application. Moreover, Pavlich *et al.* supports an AOP [29] code generation to enforce access control policies at execution time. However, this approach is limited to access control policies.

Zisman proposes a framework to support the design and verification of secure peer-to-peer applications [64]. The design models and security requirements are specified using UMLSec. The modeling of abuse cases to represent possible attack scenarios and potential threats helps designers to identify the security properties to be verified in the system. In addition, this approach artifacts expressing properties to be verified by defining a graphical template language. It also allows verification of the models against the properties and visualization of the verification results.

Lodderstedt *et al.* (SecureUML) propose an approach to model RBAC policies for model-driven systems [38]. It also provides additional support to specify authorization constraints related to the state of the system. In contrast to other approaches, SecureUML proposes a general schema for building systems by combining design modeling languages with a security modeling language; it does not fix one particular design modeling language. However, it only focuses on specifying RBAC model, and does not support secure code generation.

The approach of Doan *et al.* incorporates RBAC, MAC and lifetimes into UML for time-sensitive application design [17]. The main focus of this approach is that the process of designing and integrating security in a software application captures not only the current design state, but allows tracking the entire design evolution process via the creation and maintenance of a set of design instances over time. The design tracking allows a software/security engineer to recover to an earlier design version that satisfies specific security constraints.

Montangero *et al.* (For-LySa, DEGAS project) present two UML profiles to model authentication protocols [41]: the Static For-LySa profile which describes how the authentication protocol concepts (Server, Principals, Keys, Messages, etc.) can be modeled using UML class diagrams, and the For-LySa profile which models the dynamic aspects of the protocol in sequence diagrams, as well as the information needed to analyze the protocol. In order to validate a protocol, the approach For-LySa defines a specification language with semantics to write pre/post conditions and invariant constraints. This approach focuses only on the modeling of authentication protocols.

The approach of Ray *et al.* uses parameterized UML diagrams to model RBAC

and MAC frameworks and then compose them manually to produce a hybrid access control policy [50]. It is the first approach that attempts to combine RBAC and MAC. However, it focuses only on how to model RBAC and MAC systems in UML without considering how this approach can be used to design a secure software system. In another effort [56], Ray *et al.* integrate RBAC and MAC policies into an application using an aspect-oriented approach to separate access control features from other application features.

Alghathbar and Wijesekera (AuthUML) propose a framework to incorporate access control policies into use case diagrams only [4]. The aim of AuthUML is analyzing (not necessarily modeling) access control policies during the early stages of the software development life cycle before proceeding to the design modeling to ensure consistent, conflict-free and complete requirements.

Popp *et al.* propose an extension to the conventional process of developing use case oriented processes [49]. In addition to modeling security properties with UML, this approach provides a method to incorporate these security aspects into a use case oriented development process.

Painchaud *et al.* (SOCLe project) provide a framework that integrates security into the design of software applications [47]. It also includes verification of UML specifications and a graphical user interface tool that allows the designer to visualize the verification results and to inspect the diagrams' execution graph. But in this approach, security policies are simply specified using the Object Constraint Language (OCL) constraints.

Ledru *et al.* (EDEMOI project) aim at modeling and analyzing airport security [34]. The security properties are first extracted from natural language standards and documents, and integrated into UML diagrams as stereotypes in a UML profile. The UML specifications are then translated into formal models for verification purposes. This approach is not general enough to be used for software development. Epstein and Sandhu's work is one of the first approaches that investigate the use of UML to model RBAC policies [18]. However, it is limited to only one specific RBAC model which is the RBAC Framework for Network Enterprises (FNE). The FNE model contains seven abstract layers that are divided in two different groups. This approach allows to present each of the FNE model's layers using UML notation by defining new stereotypes. This approach can assist the "role engineering" process, however, it does not include subtle properties of RBAC such as separation of duty constraints and it does not provide a method for deriving roles. In addition, there is no formal semantics for verifying UML models.

Ahn and Shin propose a technique to describe the RBAC model with three views: static view, functional view and dynamic view using the UML diagrams [1]. This approach focuses only on the way that UML elements can be used to model RBAC policies rather than taking a larger view of examining secure software design. It does not provide a systematic modeling approach that can be used by developers to create applications with RBAC models.

Brose *et al.* extend UML models to support the automatic generation of access control policies for CORBA-based systems [9]. They specify both permissions and

prohibitions on accessing system's objects since the analysis phase in use case diagrams. The UML design is used to generate an access control policy in VPL (View Policy Language) that is deployed together with the CORBA application.

Vivas *et al.* propose an approach for the development of business process-driven systems where security requirements are integrated into the business model [61]. Security requirements are first stated at the high level of abstraction within a functional representation of the system given by UML diagrams using tagged values. Next, the UML specification is translated into XMI representation that allows automatic processing of the specification. Finally, the resulting specification is translated into a formal notation for consistency checking, verification, validation and simulation.

Fernandez provides a methodology to build secure systems using patterns [19]. The main idea of this approach is that security principles should be applied through the use of security patterns at every stage of the software development process (requirements, analysis, design and implementation stages). At the end of each stage, audits are performed to verify that the security policies are being followed.

Chan and Kwok [12] propose a design methodology for e-commerce systems to specify design details for three processes: Risk, Engineering, and Assurance, which represent the main areas of security engineering in the systems security engineering capability maturity model (SSE-CMM) on which this methodology is based. A security design pattern is used to specify each of those processes.

2.4 Verification of UML Design Models

During the recent years, many techniques have been proposed for verification and validation of UML diagrams, e.g., static analysis, theorem proving, model checking, etc. Those approaches have different strengths in different areas. Since model checkers provide automated tools for verification of a given behavioral property, they have often been used in behavioral diagrams to ensure whether the system meets the pre-defined requirements. Though, most of the proposed approaches target only activity and state machine diagrams [10, 21, 22, 28, 33, 39, 53, 55]. There are some approaches targeting sequence diagram [3, 60]. However, when it comes to interactions, it is important to analyze the type of messages being exchanged, as well as their source and destination, and their send and receive events. The proposed approaches targeting sequence diagram mainly focus on getting a formal representation of interactions, and they miss a well-defined methodology to analyze all these important elements. Moreover, those works either do not take into account UML combined fragments (components newly introduced to UML 2.0 that allow designers to describe a number of traces in a compact and concise manner [44]) or their semantics models are not in accordance with the semantics defined in the UML 2.0 specification.

Alawneh [3] proposes a framework for V&V of some popular UML diagrams (Class, State Machine, Activity and Sequence diagrams). In this approach, a semantics model called configuration transition system (CTS) is extracted from behavioral diagrams and then translated into NuSMV [14] code. This approach allows V&V of behavioral models against properties written in computational tree logic (CTL). Even though

this approach is dealing with some UML 2.0 sequence diagrams elements, the proposed semantics model is not in full accordance with the standard semantics specified in [44] due to the lack of send and receive events. As a consequence, some traces cannot be captured in this approach.

Leue [35] presents a detailed description of the translation of Message Sequence Charts (MSCs) [26] into PROMELA. Since the MSCs are the basis of UML sequence diagrams [52], many of the proposed translation decisions can be applied to sequence diagrams. However, the proposed approach deals only with the basic components and decisions; consequently, its PROMELA representation of MSCs does not cover the behavior of combined fragments introduced in UML 2.0 interactions diagrams.

Amstel [59, 60] proposes a set of techniques to improve the quality of sequence diagrams. One of these techniques is trace analysis by using model checkers. To obtain PROMELA code from sequence diagrams, this technique provides a translation scheme that is based on [35]. However, since this approach is intended to UML 1.5 diagrams, combined fragments are not taken into account. In addition, the authors do not propose a mechanism to use source and destination for writing formal properties.

In [16, 30, 53], they determine whether a given interaction can be successfully executed in a system where the behavior is specified using state machines. These works assume sequence diagrams as properties to be verified.

Chapter 3

Property Specification for UML Design

In this chapter we investigate properties specification for UML design. It starts with the detail explanation of the main adopted approaches in the area. Section 3.1 presents how to specify properties using the existing UML artifacts. Section 3.2 and section 3.3 show how to define properties by extending the UML meta-language and creating a new meta-language, respectively. In the section 3.4, we compare these approaches through a usability discussion. Finally, in section 3.5, we address the problems of the existing approaches by providing our new mechanism to specify properties.

From the state of the art presented in the previous chapter, three main UML artifacts can be used for property specification: (1) stereotypes and tagged values, (2) OCL, and (3) behavior diagrams. In addition, two other approaches can be used: (1) extending the UML metalanguage or (2) creating a new metalanguage. Table

2 summarizes the use of these approaches to specify security requirements by the contributions presented previously.

Contributions	Stereotypes and tagged values	OCL	Behavior diagrams	Extending UML metalanguage	New meta-language
UMLSec [27]	✓				
P. Mariscal *et al.* [48]	✓			✓	
SecureUML [38]	✓	✓			✓
Zisman [64]			✓		
SOCLe [47]		✓			
For-LySa [41]	✓				
Epstein and Sandhu [18]	✓				
Brose *et al.* [9]	✓				
Ahn and Shin [1]		✓			
AuthUML [4]		✓			

Table 2: The Use of Security Specification Approaches in the State of the Art

In the following, we present each of those approaches and explain how it can be used for security specification. The activity diagram of Figure 12 will be used throughout the following subsections to show how security requirements can be specified for UML design. The diagram specifies the behavior related to the admission of patients in a medical institution. This example is a simplified version of the business process used in [51]. The activity diagram consists of three main partitions: (1) Patient who starts the activity by filling out an admission request, (2) Administration area where insurance and cost information are collected, and (3) Medical area which is responsible for admission tests, exams, medical evaluations and sending the medical results to the patient.

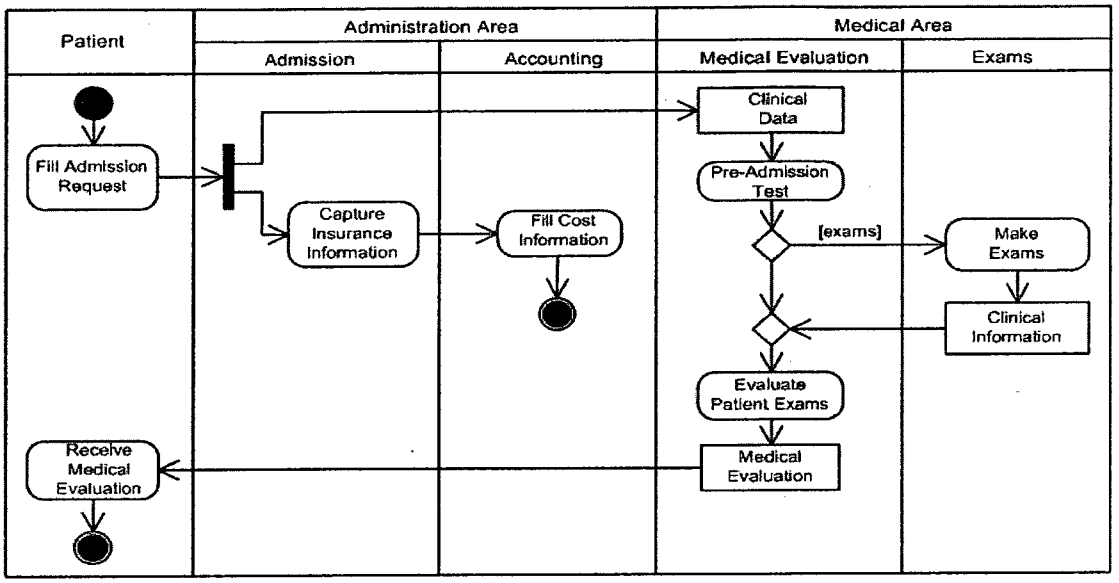


Figure 12: An Activity diagram: admission of patients in a medical institution

3.1 Property Specification Using UML Artifacts

3.1.1 Stereotype and Tagged Values

Stereotypes are provided as a mechanism for extending the UML meta-language. Therefore, a stereotype is considered as a user-defined meta-element. Its structure matches the structure of an existing UML meta-element which is referred to as "base class". In that sense, a stereotype represents a subclass (subtype) of the base class. It has the same form but with a different intent. A stereotype can have tagged values used to define the additional information needed to specify the new stereotype intent. Besides, constraints can be defined on both the base class attributes as well as the tagged values. Code generators and other tools, such as those used for verification and validation, reserve special treatment to stereotypes.

Use for Property Specification: System requirements are specified by attaching stereotypes along with their associated tagged values to selected elements of the design (e.g., subsystems, classes, etc.). Thus a specific profile should be created by some expert for the specification of these stereotypes. The compiler used to parse UML diagram is then modified such that it can read and interpret the stereotypes annotating the design. This interpretation consists in generating a formal representation of the property corresponding to the used annotation. This requirement is generated on the basis of the intent of the expert while taking into consideration the specificities of each design. In addition, a formal semantics is associated with the design. Then, the formal properties together with the formal semantics are provided as inputs to a verification tool (usually a model checker). The result of verifying the property on the design is translated into some representation that any non-expert developer can understand. Some stereotypes are parameterized over the adversary type. These stereotypes are used to specify security properties that need to be verified against a specification of an attacker (adversary). Fair exchange, secrecy, and authenticity are examples of these properties. The adversary type specifies the adversary's computation capabilities and initial knowledge. Figure 13 shows how stereotypes can be used to specify security requirements on the UML design of Figure 12. The used stereotypes are Privacy, Auditing, Access Control, Critical, Integrity, and Non Repudiation. For example, the stereotype Privacy is attached to the Patient partition to specify that unauthorized disclosure of sensitive information about the patient is not permitted.

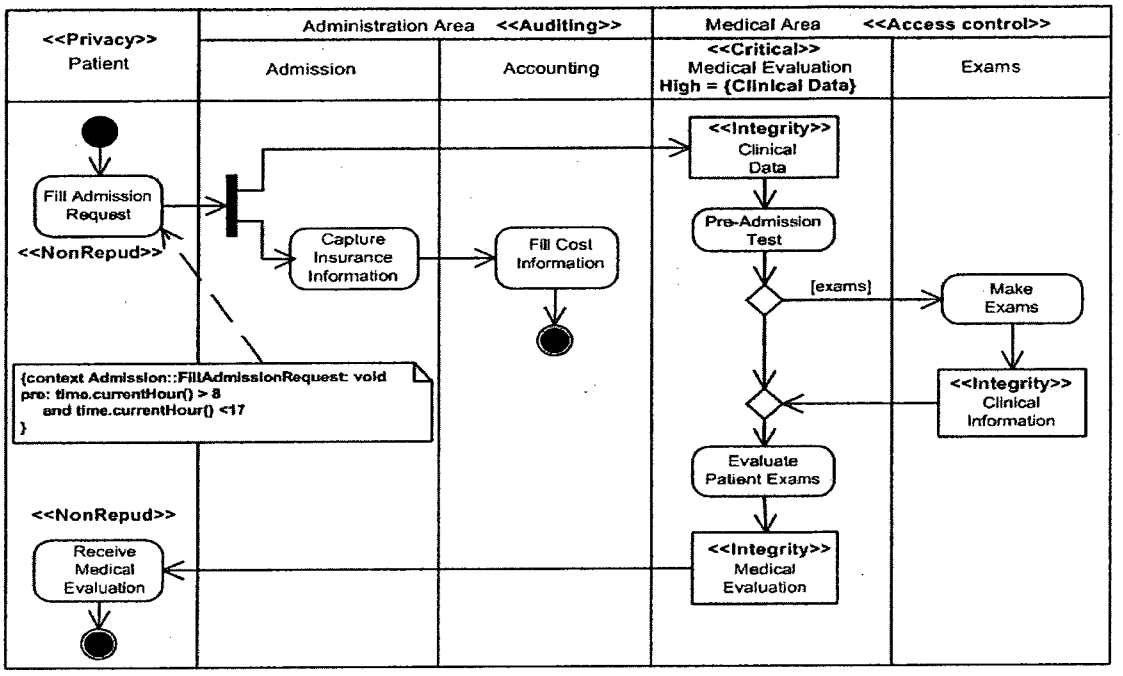


Figure 13: An example of specifying properties using stereotypes

3.1.2 Object Constraint Language (OCL)

The OCL is a formal language used to express constraints over UML diagrams. These constraints mainly specify those conditions that must be satisfied by the system being modeled. The OCL is mainly used to specify application specific requirements for UML models. In addition it is used to specify invariants of the UML meta-language. More precisely, the main purposes for which OCL can be used are the followings: (1) To specify invariants on classes and types in the meta-language, (2) to specify type invariant for Stereotypes, (3) to describe pre and post conditions on operations and methods, and (4) to describe guards [43].

Use for Property Specification: Since OCL is a language for constraints specification, it is natural to be used for property specification. OCL has been used for

security specification following three main directions. First, for the security profiles extending UML for security specification, OCL is used to define constraints on elements described by stereotypes and tagged values. Second, for those stereotypes used for the specification of access control properties, OCL can be used by the designer to define access control constraints (pre conditions and authorization guards). Third, some OCL extensions [63] allow the specification of temporal logic formulas and thus are used to specify security requirements in temporal logics, e.g., LTL, CTL, etc. Figure 13 shows how OCL can be used to specify a constraint on the action "Fill admission request". This constraint restricts the execution of this action to the working hours. This will protect the system from malicious use during nights. The condition start by specifying its context, i.e., the method on which it is applied, which is the method FillAdmissionRequest of the class Admission. Then the constraint specifies the pre-condition to be satisfied before executing the controlled method.

3.1.3 Behavior Diagrams

Behavior diagrams are UML diagrams used to depict the behavior features of the system under design. These include activity, state machine, and use case diagrams as well as four interaction diagrams.

Use for Property Specification: Behavior diagrams can be used for property specification in two ways. The First one is to specify the behavior that 'MUST' be observed by the system and the second one is to specify the behavior that 'MUST NOT' be observed by the system. The later has been investigated by some recent

contributions [64] where the used diagrams are called "Abuse cases diagrams". Figure 14 shows an example of an activity diagram specifying the behavior that must be followed by the system after filling the cost information until sending the medical evaluation to the patient. This behavior is required for enforcing faire exchange between patients and the medical institution. Enforcing this behavior inside the original design of Figure 12 results to the new design presented in Figure 15. This represents one possible scenario of using behavior diagrams to enforce security requirements. A non-security expert designer will use this "safe design" and integrate it inside its original design. Another possible scenario is when the behavior diagram, specifying a security requirement, is used to verify, through model checking or theorem proving, whether the design satisfies or not the security requirement. In this case, the diagram is translated into a (1) transition system (finite state machine or automata, etc.) or (2) a logic formula, both expressed in the input language of the target verification tool. Indeed, many contributions establishing the correspondence between transition systems and temporal logics can be found in language theory [7]. A third possible scenario is the use of behavior diagram to specify security aspects. Indeed, aspects [29] are usually defined by specifying a behavior that is inserted before or after some execution point. Thus this behavior can easily be specified by a behavioral diagram. However, the weaving of aspects and the original design can be performed on the level of design by weaving UML diagrams or postponed to the implementation phase. In the later case, the weaving is performed on selected files of the source code and the actual aspects expressed in existing aspect languages, e.g., AspectJ, and resulting from the refinement of their initial behavior diagrams.

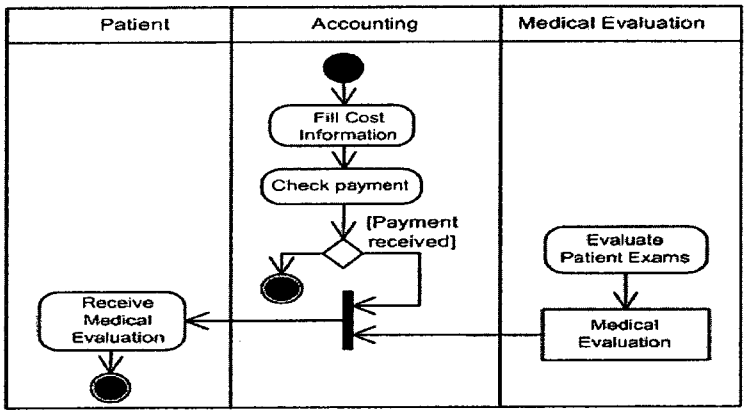


Figure 14: Fair exchange requirement inside medical application

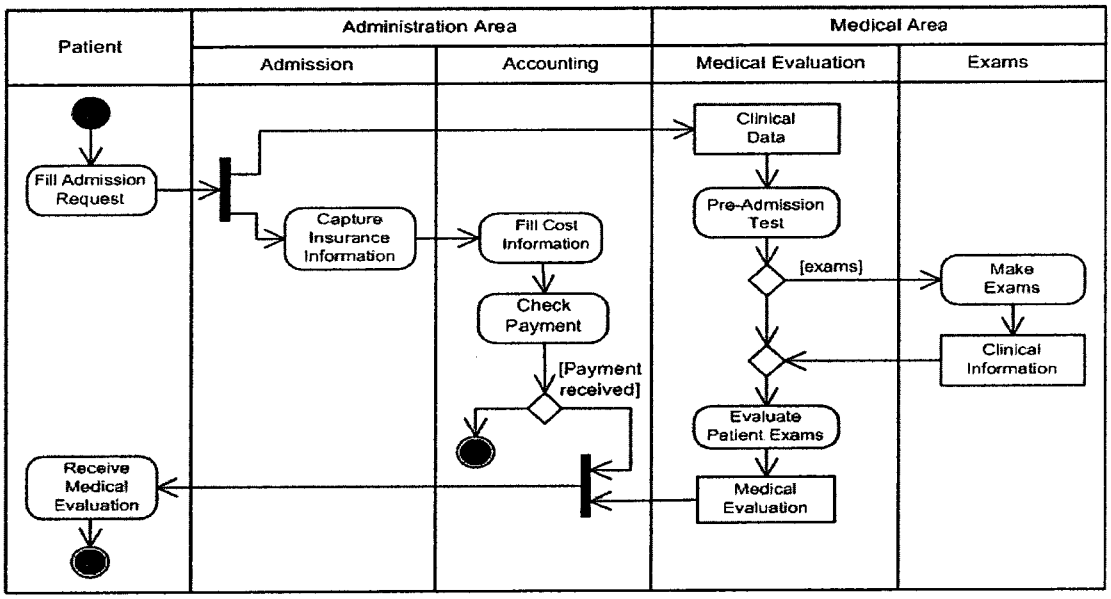


Figure 15: Enforcing the security requirement of Figure 14 in the activity diagram of Figure 12

3.2 Property Specification by Extending the UML Meta-language

In this approach, the UML meta-language is directly extended by a meta-language specification language as MOF (the Meta-Object Facility) [44]. The MOF defines

a simple meta-metamodel, and the associated semantics, allowing the description of metamodels in various domains including the domain of object design and analysis. Extending the UML meta-language (meta-model) is usually needed when extension mechanisms provided by UML (mainly stereotypes) are not appropriate for the target extension or when the resulting complexity is not tolerated.

Use for Property Specification: The two reasons stated above are the same motivating the extension of UML meta-language for property specification. Although, stereotypes allow the specification of a wide range of security requirements, they are not appropriate for specifying structured security policies: Those that are usually specified using well structured specification languages. Access control properties and security aspects are the main requirements for which it is better to have dedicated meta-elements than using standard UML meta-elements annotated by stereotypes and tagged values.

3.3 Property Specification by Creating New Meta-languages

In this approach, a new meta-language is defined using a metalanguage specification language as MOF. The motivations of creating a new meta-language are the same as those of extending the UML metalanguage. The vocabulary used by the meta-elements defined by the new meta-language have domain-specific intuition and are much more precise than the one used for UML meta-elements. Thus, the interfaces needed for manipulating the new meta-elements are too simpler compared to those required for UML design.

Use for Property Specification: The motivations of creating new meta-languages for property specification are exactly the same of extending the UML meta-language for property specification. Indeed, the approach is used for the same objectives and allows the specification of almost the same security requirements.

3.4 Usability Discussion

This section discusses the usability of each property specification approach using the following criteria for evaluation: *expressiveness*, *tool support*, *verifiability*, and *complexity*.

3.4.1 Stereotypes and Tagged Values

In the following we discuss the usability of stereotypes and tagged values for security specification.

- Expressiveness: UML artifacts provided by standard UML mainly stereotypes and tagged values are the most used by the majority of the contributions. Among these contributions, we can cite: UMLSec [27] by Jürjens which provides a UML profile and an open-source tool for specifying security requirements such as secrecy, integrity, authenticity, fair exchange, role-based access control, secure communication links, and secure information flow. Stereotypes are used by Pavlich-Mariscal et al. [48] and Basin et al. [38] for specifying access control policies and by Montangero et al. [41] for modeling authentication protocols. These contributions show that various security requirements have been specified

using stereotypes and tagged values.

- Tool Support: Has an excellent tool support since any standard UML modeling framework supports profile specification.

- Verifiability: A lot of work is done in background to generate a formal semantics for the UML design, formally specify the security requirement, verify the property against the design, and show the verification result to the end user (UML designer). The later usually consists in displaying counter examples and providing advices to improve the design and fix the vulnerabilities.

- Complexity: The complexity of the information related to stereotypes and tagged values added for security specification, depends on the number of stereotypes and tagged values attached to each UML element. For example, if different security stereotypes are associated with the same UML element then it will be complex for the user to select all these stereotypes and edit the associated tagged values. In this case, the security profile designer has the responsibility of compacting as possible the architecture of his profile design.

3.4.2 OCL

The OCL is also used by many of the surveyed contributions to express formal constraints in the specification of security properties. This is due to the fact that OCL is part of the UML standard, and by its formal nature, it allows precise specification of security constraints. The approach of Painchaud et al. (SOCLe project) [47] is based on temporal logic extension of OCL for security specification. OCL has been also

used by [38] to specify additional authorization constraints related to the state of the system. As we mentioned previously, it is natural to use OCL for security specification. However, it is important here to distinguish between using OCL as a support for some security specification artifact as stereotypes and behavior diagrams, and using it as security specification language. In the former case, the use of OCL improves the usability of any specification artifact by allowing the definition of constraints over the UML design entities. Accordingly, we focused our usability evaluation on the later case. In the following we discuss the usability of OCL for security specification.

- Expressiveness: As a security specification language, the standard OCL [44] is limited to specifying pre and post conditions and invariants that should be satisfied by the application behavior. However, some OCL extensions allow the specification of temporal logic properties.

- Tool Support: Standard OCL benefits from the support of different tools provided by standard UML modeling frameworks. However, the usability of OCL extensions is limited by the availability of tools supporting the specification and the compilation of security requirements.

- Verifiability: Once compiled and analyzed by the tool, security requirements specified using OCL extensions are systematically provided as input formulas for verification tools (model checkers and/or theorem provers). However, as for stereotypes, a lot of work is done in background to generate a formal semantics for the UML design, verify the properties against the design, and show the verification result to the end user (UML designer).

- Complexity: The complexity introduced by this approach depends on the number of OCL expressions added to specify security properties and whether they are crosscutting the application functionalities design or separated from them.

3.4.3 Behavior Diagrams

We notice the lack of using behavioral diagrams for security specification among the surveyed approaches. In fact, only the approach of Zisman et al. [63] that proposes the modeling of abuse cases to represent possible attack scenarios and potential threats to the system security. In the following we discuss the usability of behavior diagrams for security specification. We distinguish in our discussion between the use of behavior diagrams to specify security requirements for the sake of verification and their use to specify security aspects for the sake of security enforcement or hardening.

- Expressiveness: Is limited to specify those security requirements that are naturally expressible via transition systems. These include mainly attack scenarios and dynamically enforceable security requirements. As for security aspects specification, behavior diagrams are very useful for specifying advices behavior. However, stereotypes should be defined to allow the specification of patterns needed for the definition of pointcuts.

- Tool Support: Behavior diagrams benefit from a wide tool support. However, tool support for this approach depends also on the tool support of stereotypes.

- Verifiability: When used for security requirements specification, behavior diagrams are translated to transition systems or logical formulas in order to be

verified on the system design. While the former translation is almost systematic, the later is limited to those diagrams satisfying some structural constraints (e.g., determinism) and constrained by the availability of translation algorithms in language and logic theory. As for stereotypes and OCL, a lot of work is done in background to generate a formal semantics for the UML design, verify the properties against the design, and show the verification result to the end user (UML designer). When used for security aspects specification, as for the first approach, a lot of work is done in background to (1) identify diagram entities (e.g., methods/actions) matching the specified patterns and (2) weaving diagrams specifying advices and those specifying the system behavior.

- Complexity: Relatively acceptable since the behavior diagrams specifying security requirements are separated from those specifying the system behavior and are easily distinguishable from them. The complexity of security aspects specification is comparable to that of security requirements specification.

3.4.4 Extending the UML Metalanguage

Only few contributions [48] have investigated the extension of the UML metalanguage for security specification. This is due to the fact that this kind of modification requires a high expertise and knowledge of the UML meta-language and its objectives. Indeed, the extension may require the modification of the whole metalanguage which is too complex. In the following we discuss the usability of extending the UML metalanguage for security specification.

- Expressiveness: Comparable to that of stereotypes.
- Tool Support: The extension is heavyweight so that "may require one to extend the CASE-tool itself, in particular the storage components, i.e., the repository, and the visualization components" [38]. This impacts negatively the portability of any extension since any UML modeling framework is heavily modified to allow the use of the new meta-elements and their interpretation.
- Verifiability: A lot of work is done in background to generate a formal semantics for the UML design, verify the properties against the design, and show the verification result to the end user. However, if the extension targets some low-level policy specification language or AOP language, then the effort spent in background is limited to parsing the specification and translating it to the target language.
- Complexity: The complexity is comparable to that of using behavior diagrams.

3.4.5 Creating a New Metalanguage

As for the previous approach, only few contributions [38] have investigated the creation of new meta-languages for security specification. In the following we discuss the usability of creating a new metalanguage for security specification.

- Expressiveness: Comparable to the expressiveness of extending the UML metalanguage.
- Tool Support: Better than that of extending the UML metalanguage and comparable to that of stereotypes. In addition, the compiler needed to parse the

specification can be easily plugged in to the UML modeling framework.

- Verifiability: Better than that of the verifiability of extending the UML metalanguage. Indeed, the security specification is exclusively based on the new meta-elements and thus is easier to parse and translate.
- Complexity: Comparable to the complexity of extending the UML metalanguage.

3.5 Our Approach for Property Specification

3.5.1 State Machine-Based Properties

In table 2, we can see that only one approach has investigated the use behavior diagrams to express system properties. There are few approaches attempting to define how UML behavior diagrams can be used as properties. In [64], it is mentioned that state machines can be used to represent properties, however it does not provide details how to use these diagram as a input to the model checker.

As previously cited, behavior diagrams can be translated into transition system o logic formula. Since UML state machines are semantically close to transition system, we decided to provide developers with a mechanism to write using these diagrams by implementing the approach depicted as an activity diagram in figure 16.

The approach starts from the developer creating a new state machine property. This state machine has some special characteristics. First, it needs to be marked with *property* stereotype. It is important to ensure that what is described in there is part of the system functional design. Then, it needs one, and exact one, final state. The

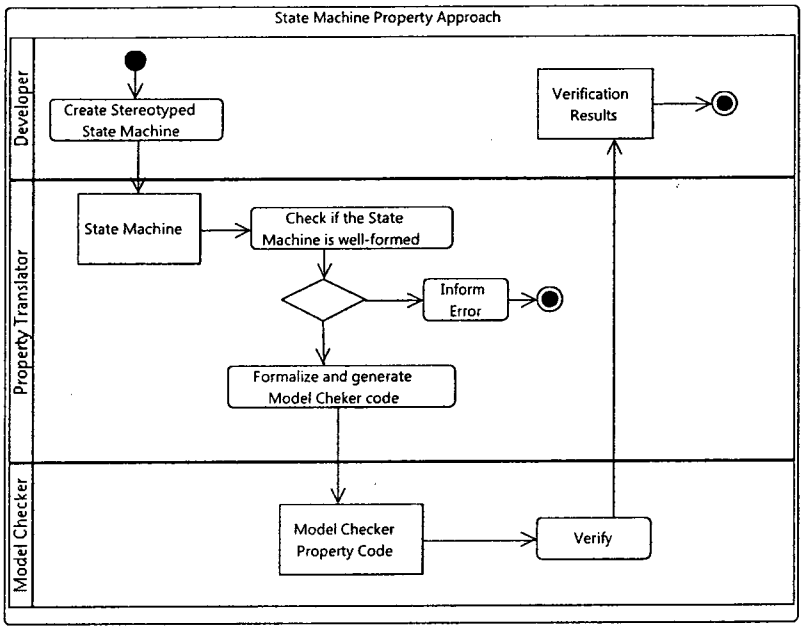
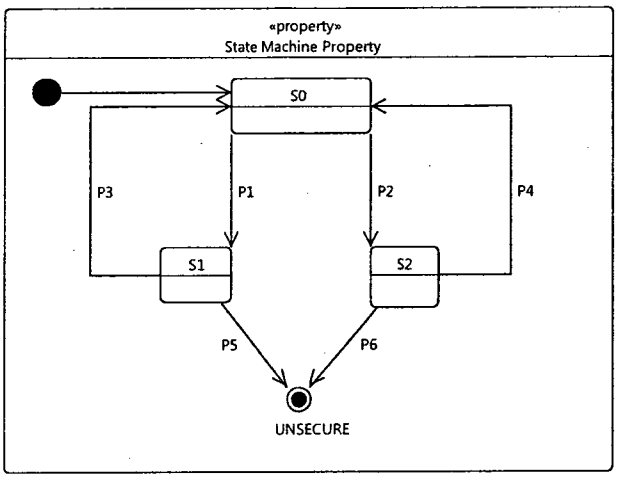


Figure 16: State Machine Property Approach

final state symbolizes the *unsecure* state, meaning that, if the state machine reach the final state, it is in an unsecure situation and the property failed. Moreover, the transitions are triggered by propositional formulas assigned to its label. These are logical formulas composed by system atomic propositions. A transition is fired if an outgoing transition of the current has its proposition formula equal to true. All this characteristics are checked automatically and, if any error is found, the problem is informed to the developer.

Once the state machine if defined and well-formed, it is formalized and translated to never claims statements in PROMELA. This statement can be used to determine a behavior that must never happen. It is executed in parallel with the based model (verify action in figure 16). If it reach its final state (that is, its closing curly brace), it means that the property failed and the model checker provides a counterexample

from the beginning of execution up to that point.



(a)

```
1 never {
2  goto S0;
3 S0:
4  if
5   ::(P1) -> goto S1;
6   ::(P2) -> goto S2;
7   ::else -> goto S0;
8  fi;
9 S1:
10 if
11  ::(P3) -> goto S0;
12  ::(P5) -> goto UNSECURE;
13  ::else -> goto S1;
14 fi;
15 S2:
16 if
17  ::(P4) -> goto S0;
18  ::(P6) -> goto UNSECURE;
19  ::else -> goto S2;
20 fi;
21 UNSECURE: skip}
```

(b)

Figure 17: (a) A generic state machine property, (b) respective never claim statement used by SPIN

3.5.2 MOBS2 Language for Property Specification

In order use all the expressiveness of formal logics and, at the same time, reduce the gap between them and the natural language, we propose a new language on top of

LTL and CTL to allow developers to write properties in a higher level. To achieve this goal, we define the following two requirements to this language:

1. This language needs to express its operator in a way that a developer familiar with propositional logic will be able use then.

2. Properties written using this language need to be easy to read (even for a non-expert), so it can be attached to the regular documentation of the system.

First, we define the following mapping rules for the operators:

Operator	MOBS2 Language
$\Box$	ALWAYS
$\Diamond$	EVENTUALLY
$\mathbf{U}$	UNTIL
$\bigcirc$	NEXT
$\forall$	FORALL
$\exists$	EXIST
$\rightarrow$	IMPLY
true	any

Table 3: Mapping rules from logical and temporal operators to the MOBS2 Language

Even though these mapping rules are simple, it improves significantly the readability of the final property.

To improve even more the new language, we attempted to remove the repetition of boolean operators by identifying some patterns that appears very often in many properties. We found that property for interaction diagrams have very often an *and* combination of the sender, message and receiver. The new language allows the developer to write this situation as simple English sentences like "sender sends message to receiver" or "receiver receives message from sender".

The grammar rule for this language is defined below:

<consts> ::= any

<prop> ::= <consts>| P_1 | P_2 | P_3 ... (atomic propositions)

|!<prop>

|<prop>'&&'<prop>

|<prop>'||'<prop>

|<prop> IMPLY <prop>

|<prop> sends <prop> to <prop>

|<prop> receives <prop> from <prop>

<tempExpr> ::= <prop>

| ALWAYS (<tempExpr>)

| EVENTUALLY (<tempExpr>)

| (<tempExpr> UNTIL <tempExpr>)

| NEXT (<tempExpr>)

| FORALL (<tempExpr>)

| EXIST (<tempExpr>)

This is an abstract grammar. It can represent both LTL and CTL formulas. In practice, the scope the properties that can be specified will be restricted by the input language of the model checker.

Chapter 4

Verification and Validation of UML 2.0 Interactions

In this chapter, we apply our framework in one important type of UML diagrams: interaction diagrams. In Section 4.1, we briefly introduce the semantics of UML interactions. Section 4.2 discusses how to get PROMELA code according to the semantics of UML 2.0 interactions. In Section 4.3, the mechanism for V&V using source/destination and send/receive events is presented. In Section 4.4, we present a scenario applying our approach, as well as experimental results.

UML interaction diagrams come in different variants (see figure 4(a)). The most common variant is the Sequence Diagram that focuses on the exchanging of messages among lifelines. Communication diagrams is very similar to sequence diagram but is focused on the objects and the number of messages between them, rather than on a sequence of events [36]. Interaction overview presents an interaction in a way that

highlights the overview of the control flow. Timing diagram focuses on showing timed-based message events of the system. Herein, we illustrate out approach using the most popular variant of interaction diagrams which is sequence diagram. Nonetheless, since these diagrams share the same metamodel, the presented approach can be applied to any of them.

UML sequence diagrams are behavioral diagrams used to specify interactions among system entities in many different situations. They are used to get a better grip of an interaction situation for an individual designer or for a group that needs to achieve a common understanding of the situation [23]. Along with class diagrams and use case diagrams, sequence diagrams are the most popular diagrams of UML [60].

The main contribution of this work is to provide an efficient mechanism to be able to track the execution state of an interaction, which allows designers to write relevant properties involving send/receive events and source/destination of messages using LTL. This mechanism was implemented in such a way that allows the designers to select the portion of the information that is relevant to their properties. Consequently, it gives them flexibility to write very expressive properties. Another important contribution is the definition of the PROMELA structure that provides a precise semantics of most of the newly UML 2.0 introduced combined fragments, allowing the execution of complex interactions. It allows the developer to simulate much more complex sequence diagrams, with non-straightforward execution trace. The result of these contributions is an efficient approach which is capable of detecting more flaws on more complete and complex interactions.

The proposed approach creates a PROMELA-based model from UML interactions

expressed in sequence diagrams, and uses SPIN model checker to simulate the execution and to verify properties written in Linear Temporal Logic (LTL) or state machines. PROMELA/SPIN was chosen because it provides important concepts (sending and receiving primitives, parallel and asynchronous composition of concurrent processes, and communication channels) that are necessary to implement sequence diagrams [35]. This makes the implementation easier since the communication primitives and channels are already available in PROMELA and it does not need any extra effort to implement them. The whole technique is implemented as an Eclipse[1] plugin, which hides the model-checking formalism from the user and allows the V&V engine to be embedded into the development environment.

4.1 Semantics of UML Interactions

Interaction in UML is considered a pattern of message exchanges to accomplish a specific purpose [52]. The semantics of an interaction is given as a pair of sets of traces [**P**,**I**] [44]. **P** represents valid traces, whereas **I** is the set of invalid traces. A trace is a particular execution history, i.e., it is a sequence of event occurrences. Equation 3 shows the notation for traces.

$$\textbf{Trace } t = \langle e_1, e_2, \ldots, e_n \rangle \tag{3}$$

The set of traces of an interaction is mechanically built from the the semantics of its constituent interaction fragments by ordering and combining them using the weak

[1] http://www.eclipse.org/

sequencing operator. Usually all traces are valid; invalid traces are those associated only negative combined fragments. The next sections show how to extract the semantics of interactions in different scenarios as well as the corresponding PROMELA code.

4.2 Translation of UML 2.0 Combined Fragments into PROMELA

In this section, we briefly present the PROMELA representation of the basic elements of sequence diagrams as defined in [35,59]. Then we present the trace semantics of the most popular combined fragments and their respective PROMELA code that correctly simulates the execution traces. The composition of the presented translation rules allows the simulation of complex interactions with interesting and non-straightforward execution trace.

4.2.1 Basic Elements

The work presented in [35] specifies how to translate basic elements of MSCs into PROMELA and [3] shows that this schema can be reused for basic elements of sequence diagrams. The translation rules for basic elements presented here are based on the work proposed in those approaches, and they will be the basis for the next (and more complex) interaction elements. The PROMELA elements used for representing basic components of interactions are: (1) `proctype`: it is used for declaring new process behavior, (2) `mtype`: it defines symbolic names of numeric constants that are

used as messages in the communicating process. (3) `cham`: it declares and initializes communication channels. Finally, (4) `!/?` operators: Theses symbols are used for sending/receiving messages to/from channels, respectively.

Table 4 provides the PROMELA representation of the basic elements shown in figure 18. The semantics of this interaction is the single trace $\langle !m, ?m \rangle$ [23].

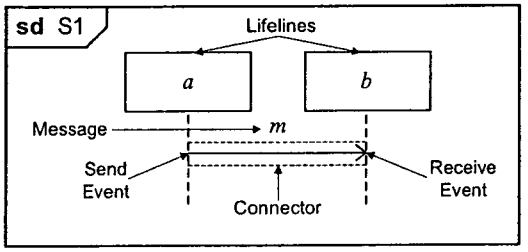


Figure 18: Simple Sequence Diagram

UML element	PROMELA element	PROMELA statement
Lifeline	Process	`proctype{...}`
Message	Message	`mytpe = {m1,...mn}`
Connector	Communication channel for each message arrow	`chan chanName = [1] of {mtype}`
Send and Receive events	Send and Receive operations	Send ⇒ `ab!m`, Receive ⇒ `ab?m`

Table 4: Mapping of basic UML Sequence Diagrams into PROMELA

4.2.2 Interaction Fragments and Weak Sequencing Combined Fragments

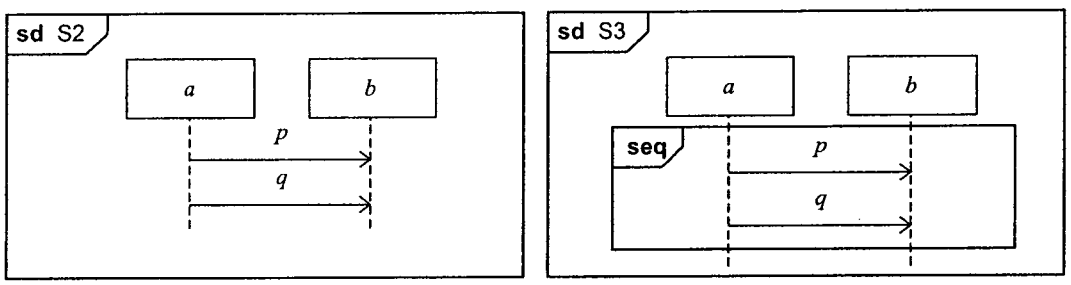
An Interaction Fragment is an abstract notion of the most general interaction unit. In other words, it is a piece of an interaction [44]. In figure 19(a) two messages (p and q) are sent from a to b. Each message has the semantics given for the message in figure 18. The vertical positions of events represent their order on each lifeline. However, the two lifelines are independent [23]. Thus, the possible execution traces can be derived from figure 19(a) by using the weak sequencing operator (**seq**) defined in [44]. This

operator is also used in weak sequencing combined fragment as shown in figure 19(b). The operator **seq** defines the set of traces with the following constraints [23,44]:

1. The order of events within each of the operands is maintained in the result.
2. Events on different lifelines from different operands may come in any order.
3. Events on the same lifeline from different operands are ordered such that an event of the first operand comes before that of the second operand.

For the interactions in figure 19(a) and 19(b), we get the following result:

$$S2 = \langle !p, ?p \rangle \textbf{ seq } \langle !q, ?q \rangle = \{\langle !p, ?p, !q, ?q \rangle, \langle !p, !q, ?p, ?q \rangle\} \tag{4}$$




```
1 /*Messages declaration*/
2 mtype = {p,q};
3 /*Channels declaration*/
4 chan ab_p = [1] of {mtype};
5 chan ab_q = [1] of {mtype};
6 /*Lifelines Specification*/
7 proctype a(){ab_p!p; ab_q!q;};
8 proctype b(){ab_p?p; ab_q?q;};
9 /*System Instantiation*/
10 init{atomic{run a(); run b();}}
```

(c)

Figure 19: (a) Simple Interaction Fragment, (b) Weak Sequencing Combined Fragment and (c) their corresponding PROMELA Code

PROMELA Representation:

The communication primitives available in PROMELA naturally implements the **seq** operator following the translation map shown in table 4. This is one of the main reasons for choosing PROMELA/SPIN for model checking of sequence diagrams. Fig 19(c) shows the PROMELA code for the interactions in figure 19(a) and figure 19(b). In line 2 the messages are declared, lines 4 and 5 represent the channels on which the messages are sent, lines 7 and 8 specify the lifelines using process, and line 10 is the instruction to instantiate the system.

4.2.3 Alternative and Option Combined Fragments

Alternative and Option combined fragments represent a choice of behaviour in sequence diagrams. Alternative and Option operators are denoted as **alt** and **opt**, respectively [44]. The **opt** operator designates that the combined fragment represents a behaviour choice where either the sole operand happens or nothing happens. An option is semantically equivalent to an alternative combined fragment where there is one non-empty operand and the second operand is empty [44]. The set of traces that defines a choice is the union of the traces of the operands [23,44]. Eq. 5 shows the set of traces of the interaction in figure 20(a).

$$S4 = \langle !p, ?p \rangle \ \textbf{alt} \ \langle !q, ?q \rangle = \{ \langle !p, ?p \rangle, \langle !q, ?q \rangle \} \tag{5}$$

PROMELA Representation:

Alternative and Option operator are represented as *if* condition in PROMELA. The guard variable is declared globally to enforce all lifelines to get the same decision at the choice point. The non-deterministic behaviour is implemented at the set-up time by assigning different values to the guards using *if* statement with two executable conditions (lines 13 and 14 of figure 20(b)). At execution time, SPIN randomly chooses an option and continues the simulation. In exhaustive mode, SPIN will simulate all possible system decisions and it will provide all traces shown in Eq. 5. figure 20(b) presents the PROMELA code corresponding to the model in figure 20(a).

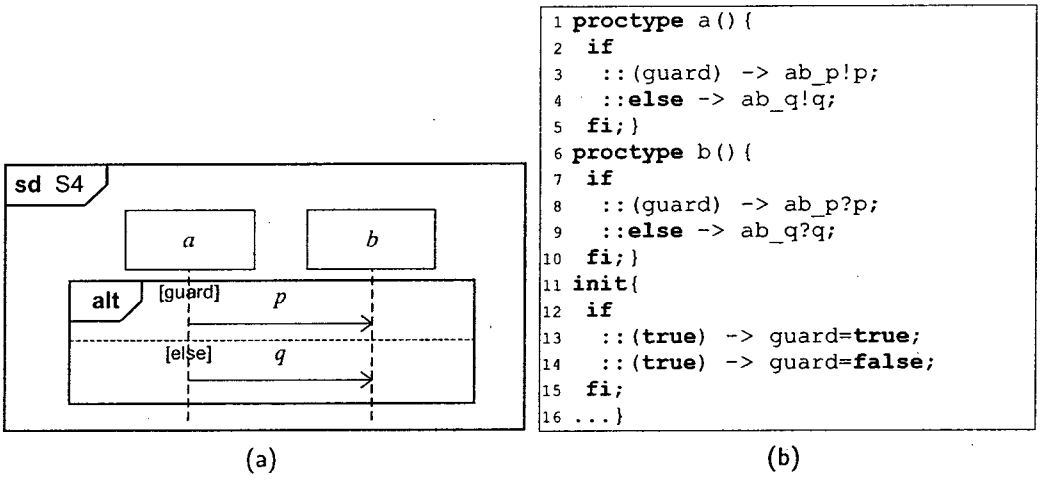


Figure 20: (a) Alternative Combined Fragment, (b) Respective PROMELA code

4.2.4 Parallel Combined Fragments

A Parallel Combined Fragment, denoted by **par** operator, represents a parallel merge between the behaviours of the operands. The events of the different operands can be interleaved in any way as long as the ordering imposed by each operand as such is

preserved [44]. Its set of traces describes all the ways that events of the operands may be interleaved without obstructing the order of the events within the operand [23]. Eq. 6 shows the set of possible traces of the diagram in figure 21(a).

$$S5 = \langle !p, ?p \rangle \textbf{ par } \langle !q, ?q \rangle = \{\langle !p, ?p, !q, ?q \rangle, \langle !p, !q, ?p, ?q \rangle, \langle !q, !p, ?q, ?p \rangle, \langle !q, ?q, !p, ?p \rangle, \langle !q, !p, ?p, ?q \rangle, \langle !q, !p, ?q, ?p \rangle\} \quad (6)$$

PROMELA Representation:

Parallel behaviour can be implemented using sub-instances of the lifelines covered by the parallel fragment (figure 21(b), lines 9 and 11). The new element is instantiated right before the main process starts the parallel activities. The actions inside the parallel fragment are divided among the main process and its sub-instances. Each one executes one operand. A synchronism mechanism should be implemented to ensure that no event after a combined fragment will overtake an event in it. This synchronism is done with token messages that will be sent from the subprocess to the main process right before finishing its execution (figure 21(b), lines 10 and 12). The main process must wait for all tokens before continuing the execution (figure 21(b), lines 4 and 8). figure 21(b) shows the PROMELA code of the model in figure 21(a).

4.2.5 Loop Combined Fragments

The operator **loop** indicates that the combined fragment represents a repetition structure. The loop operand will be repeated a certain number of times according to the

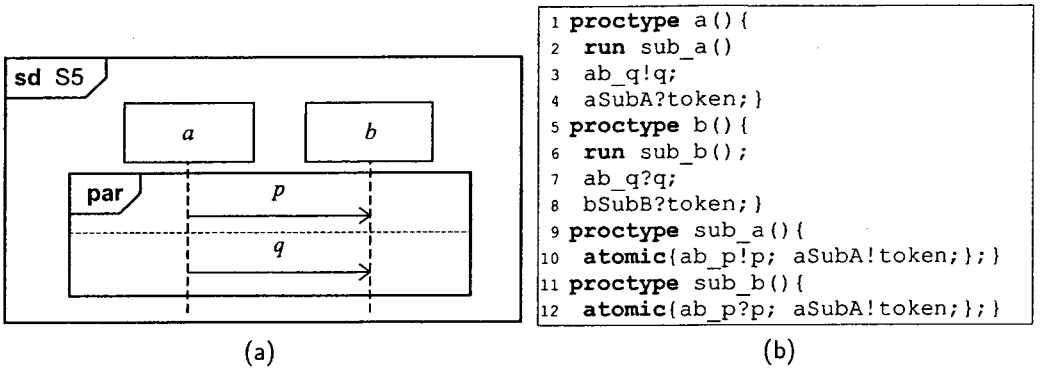


Figure 21: (a) Parallel Combined Fragment, (b) Respective PROMELA Code

values defined by the designer. The loop construct represents a recursive application of the **seq** operator where the loop operand is sequenced after the result of earlier iterations [44].

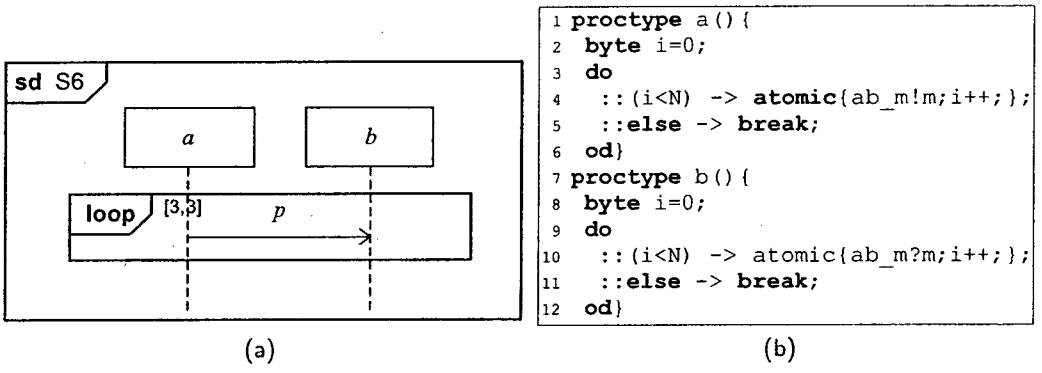


Figure 22: (a) Loop Combined Fragment, (b) Respective PROMELA Code

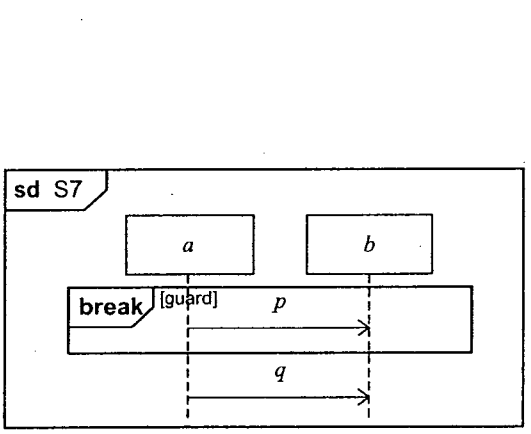
PROMELA Representation:

Our PROMELA implementation of loop works with a fixed number of repetition. PROMELA defines *do* operator as a repetition construct. Loop fragments are implemented by declaring a global variable with the total number of repetition, and a *do* structure in each lifeline covered by the fragment. Fig 22(b) presents the PROMELA

code of the model in figure 22(a).

4.2.6 Break Combined Fragments

The interaction operator **break** shows a combined fragment representing a breaking scenario. If the guard condition is true, the operand scenario is performed instead of the remainder of the enclosing interaction fragment [44].



(a)

```
1  proctype a(){
2   if
3    ::(guard) -> atomic{ab_p!p;
4                        goto L0;};
5    ::else -> skip;
6   fi;
7   ab_q!q;
8   L0:skip;}
9  proctype b(){
10  if
11   ::(guard) -> atomic{ab_p?p;
12                       goto L1;};
13   ::else -> skip;
14  fi;
15  ab_q?q;
16  L1:skip;}
```

(b)

Figure 23: (a) Break Combined Fragment, (b) Respective PROMELA Code

PROMELA Representation:

The **break** operator can be simulated with *goto* statements in PROMELA. If the guard condition is true, the action inside the break combined fragment is performed, then the execution jumps to the end (lines 4 and 12 of figure23(b)). The non-deterministic behavior is implemented in the same way as in alternative and option combined fragments. figure 23(b) shows the PROMELA code for the break combined fragment in figure 23(a).

4.3 Using Source/Destination and Send/Receive Events for Sequence Diagrams V&V

In the previous section, we provided what is needed to simulate the execution of sequence diagrams by covering the most important combined fragments. However, the main objective of using PROMELA-based model is not to simulate the execution of sequence diagrams, but the verification of formal properties. When it comes to verify formal properties on SPIN, it is impossible to determine whether a send or receive event has occurred. Indeed, the system state does not change when messages are sent over channels [60]. To overcome this obstacle, [60] proposed a flag-based technique to mark an occurrence of a send/receive event. This section presents an extension of this approach that is able to determine *who is sending/receiving what to/from whom* at any time of the execution. This information is very useful when one wants to write properties to be verified. We also show how to write LTL properties using this approach.

4.3.1 Tracking the execution state

The first step toward the formal V&V of sequence diagrams is to keep track of the actions performed by the entities in the interaction. In other words, it is essential to be aware of all event occurrences during the execution. In [60], the authors suggest tracking of sending and receiving events of messages by using flags associated with the respective event (e.g., using the flag "Sx" for "sending message x"). In spite of the fact that they improved the set of properties that can be verified, many other

properties are still not covered since they require the information of the entities the are interacting (e.g., the following constraint could be specified to a particular system: "*Alice* is not supposed to receive a request from *Bob*"). In [3], the authors define the concatenation of sender, message and receiver as one action, but they do not include send and receive events. Even though it provides the entities information, it does not give the flexibility to write properties looking only at a particular element in the model. (e.g., *Server* does not send anything to anyone without signing). This flexibility is important because systems usually have many entities, but only some of them are really critical. To address these weaknesses, we define a state transition system such that the transitions are trigged by the send and receive events of the interaction and each state is characterized by a 4-tuple consisting of the following fields:

1. Lifeline that performed the last action.
2. Last performed action (send or receive).
3. Message used in the last action.
4. Lifeline to/from which the message was sent/received.

Each state contains the information we need to track, and each field can be used separately.

In PROMELA, we represent each state as a set of flags. For each lifeline, each message, and *send/receive* events a flag is declared. The values of these flags are updated together with each send/receive event. The update is done using a *d_step* statement to make the assignment of all new values as one step at the execution time.

figure 24 shows the PROMELA code of the interaction in figure 19(a) with its flags to represent states.

```
1 proctype a(){
2  atomic{d_step{send=1; receive=0; msg_p=1; msg_q=0; proc1_a=1;
3                proc1_b=0; proc2_a=0; proc2_b=1;}; ab_p!p;};
4  atomic{d_step{send=1; receive=0; msg_p=0; msg_q=1; proc1_a=1;
5                proc1_b=0; proc2_a=0; proc2_b=1;}; ab_q!q;};}
6 proctype b(){
7  atomic{ab_p?p;d_step{send=0; receive=1; msg_p=1; msg_q=0;proc1_a=0;
8                       proc1_b=1; proc2_a=1; proc2_b=0;};};
9  atomic{ab_q?q;d_step{send=0; receive=1; msg_p=0; msg_q=1;proc1_a=0;
10                      proc1_b=1; proc2_a=1; proc2_b=0;};};}
```

Figure 24: PROMELA code of the diagram in figure 19(a)

4.3.2 Using flags to specify LTL properties

After defining a methodology to track the execution state, LTL formulas can be written in terms of boolean expressions over the flags. For example, if one wants to say "b sends p to a", he/she should write the following expression: (proc1_b$\wedge$send$\wedge$msg_p$\wedge$proc2_a). A very useful property of the flag-based state is the ease of expressing sentence over all lifelines, or all messages, or all actions only by omitting the respective element in the expression. For example, if one wants to verify if "no lifeline receives messages from a", the respective expression is: $\neg$(receive$\wedge$proc2_a). This example also shows that the proposed technique gives the flexibility to write properties looking at a particular element in the model (lifeline a in that case). Therefore, the proposed flag-based mechanism to track execution state not only provides developers with the information they need to write properties, but also it allows them to specify properties in a very flexible way. Below, we present two examples of LTL properties written using flags and their respective verifications using SPIN model checker.

- **Example 1**: Suppose one wants to verify in the sequence diagram from figure 19(a) whether "no lifeline will send message q until b receives message p". The LTL formula corresponding to this property is:

$$\neg(\text{send} \wedge \text{msg\_q}) \ \mathbf{U} \ (\text{proc1\_b} \wedge \text{receive} \wedge \text{msg\_p}) \tag{7}$$

After model checking, SPIN reports that the property does not hold. The counterexample that is returned is shown in figure 25(a).

- **Example 2**: In the model shown in figure 21(a), suppose one needs to verify if "always, after b receives p, eventually b receives q from a". The LTL formula expressing this property is:

$$\Box((\text{proc1\_b} \wedge \text{receive} \wedge \text{msg\_p}) \rightarrow \Diamond(\text{proc1\_b} \wedge \text{receive} \wedge \text{msg\_q} \wedge \text{proc2\_a})) \tag{8}$$

After model checking, SPIN shows that this property does not hold. The counterexample is presented in figure 25(b).

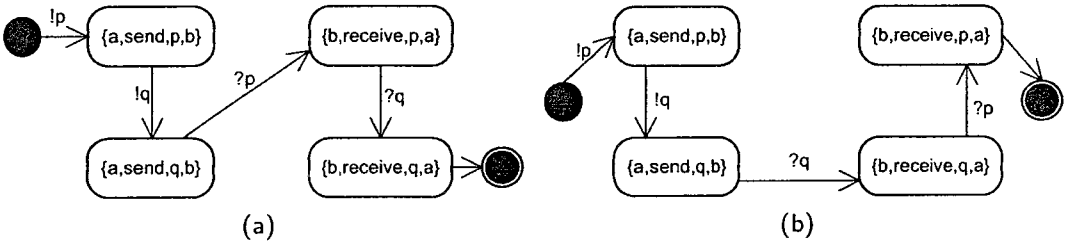


Figure 25: SPIN counterexamples

4.4 Case Study

This section presents part of the system presented in [2]. In that thesis, the authors show the design of an Automated Teller Machine (ATM). The ATM interacts with two other entities: The Customer (User) and the bank. figure 26 describes a use case where the user starts a request by inserting his/her card. The ATM must verify the card and the personal identification number (PIN) to proceed. If the verification fails the card should be ejected. Otherwise, the user has the choice to perform some operations and the card is retained in the machine until the user finishes the transactions. The first and second combined fragments are dealing with the authentication of the card and the PIN, respectively. The third one shows an interaction using "cash in advance" operation.

4.4.1 LTL properties

1. The first property states that the ATM cannot allow the user to request an operation if either the card or the PIN is not valid:

$$\Box(x \rightarrow \neg\Diamond y) \tag{9}$$

where x = (start $\wedge$ ($\neg$cardOK $\vee$ $\neg$PINok)), and y = (proc1_user $\wedge$ receive $\wedge$ msg_waitAccount)

Using the our notation the property is written as following:

ALWAYS ((!cardOK||!PINok) **IMPLY**

!**EVENTUALLY**(User *receives* waitAccount from *any*))

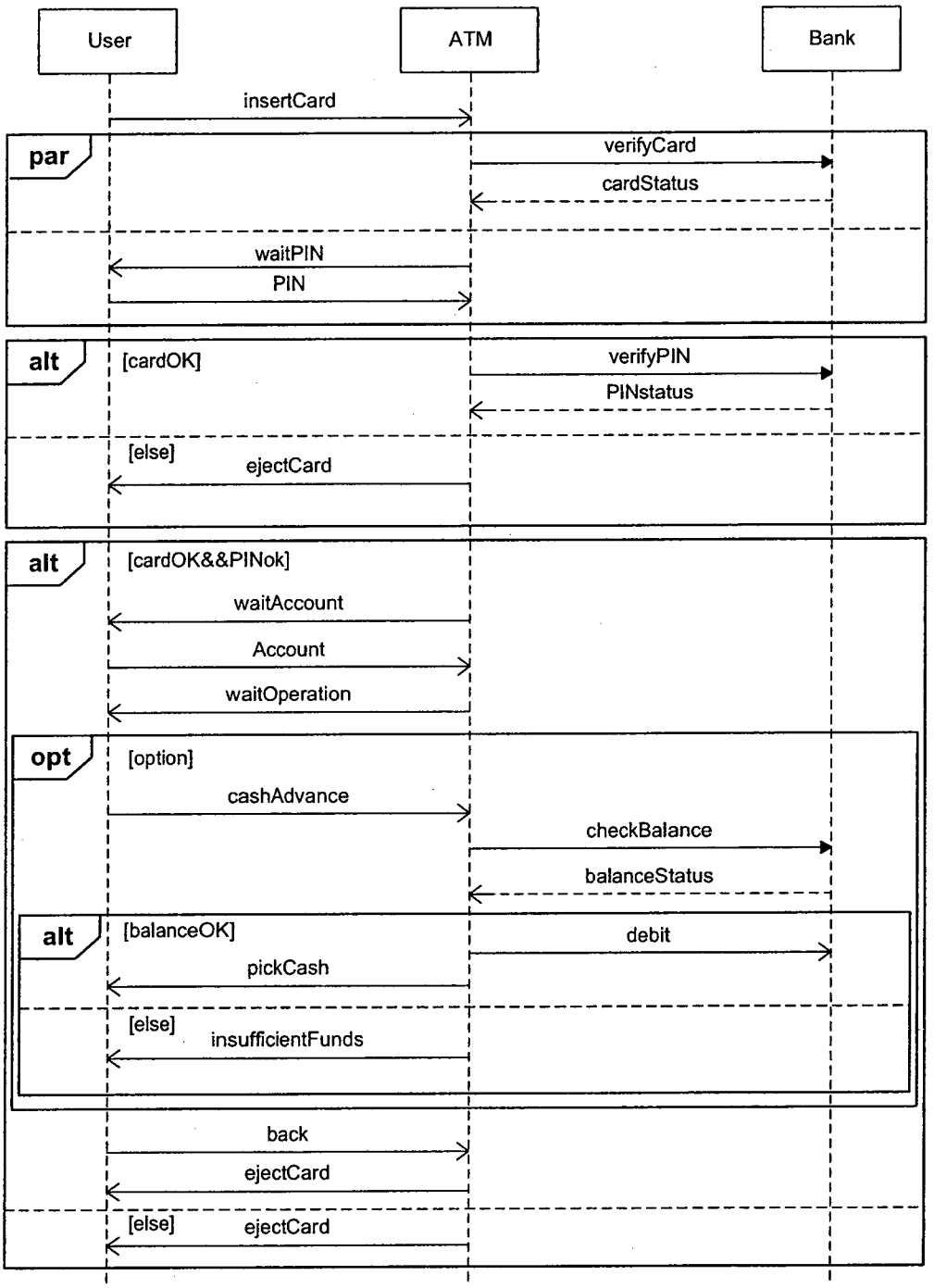


Figure 26: ATM Sequence Diagram

2. The second property is needed to avoid inconsistencies between the money given to the user and the amount debited in the bank. It asserts that the ATM must first debit the amount in the bank, and then give the money to the user. In other words, the user does not receive pickCash until the bank receives debit:

$$\neg x \mathbf{U} y \tag{10}$$

where x = (proc1_user$\wedge$receive$\wedge$msg_pickCash), and y = (proc1_bank$\wedge$receive$\wedge$ msg_debit).

In our notation the property becomes:

!(User *receives* pickCash from *any*) **UNTIL** (Bank *receives* debit from *any*)

3. The third property deals with the usability of the ATM. It states that, if the ATM receives insufficient funds, it should allow the user tochoose other operation before finishing the session:

$$\Box(x \rightarrow (\neg y \ \mathbf{U} \ w)) \tag{11}$$

where x = (proc1_user $\wedge$ receive $\wedge$ msg_insufficientFunds), y = (end), and w = (proc1_atm $\wedge$ send $\wedge$ msg_waitOperation $\wedge$ proc2_User)).

In our notation the property becomes:

ALWAYS ((User *receives* insufficientFunds from *any*) **IMPLY**

!end **UNTIL** (ATM *sends* waitOperation to User))

4.4.2 State Machine property

- The last property is to ensure that, after a failed authentication, the ATM must eject the card and end the session. It says that, if authentication procedure goes to a failed state after ejecting the card, no message should be sent from the ATM to the user until the next authentication starts. This property can be specified using state machine as shown in figure 27.

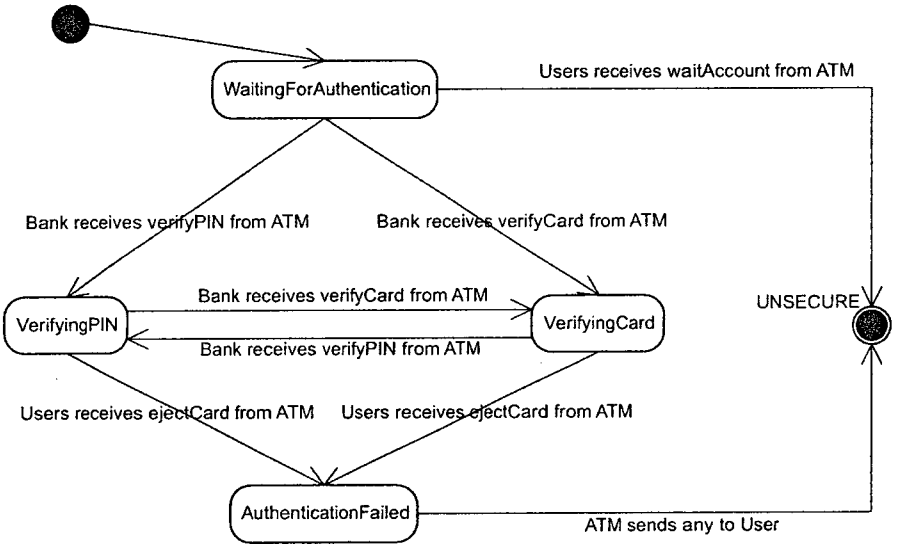


Figure 27: Property specified using state machine

4.4.3 ATM Case Study Results

Using SPIN to verify the properties described previously, we found that only the first property is satisfied. The model checker was able to provide a counterexample for each of the other properties. In the following, we present the failing trace related to the verification of each property.

- **Property 2 Counterexample**: In the trace shown in figure 28(a), it possible

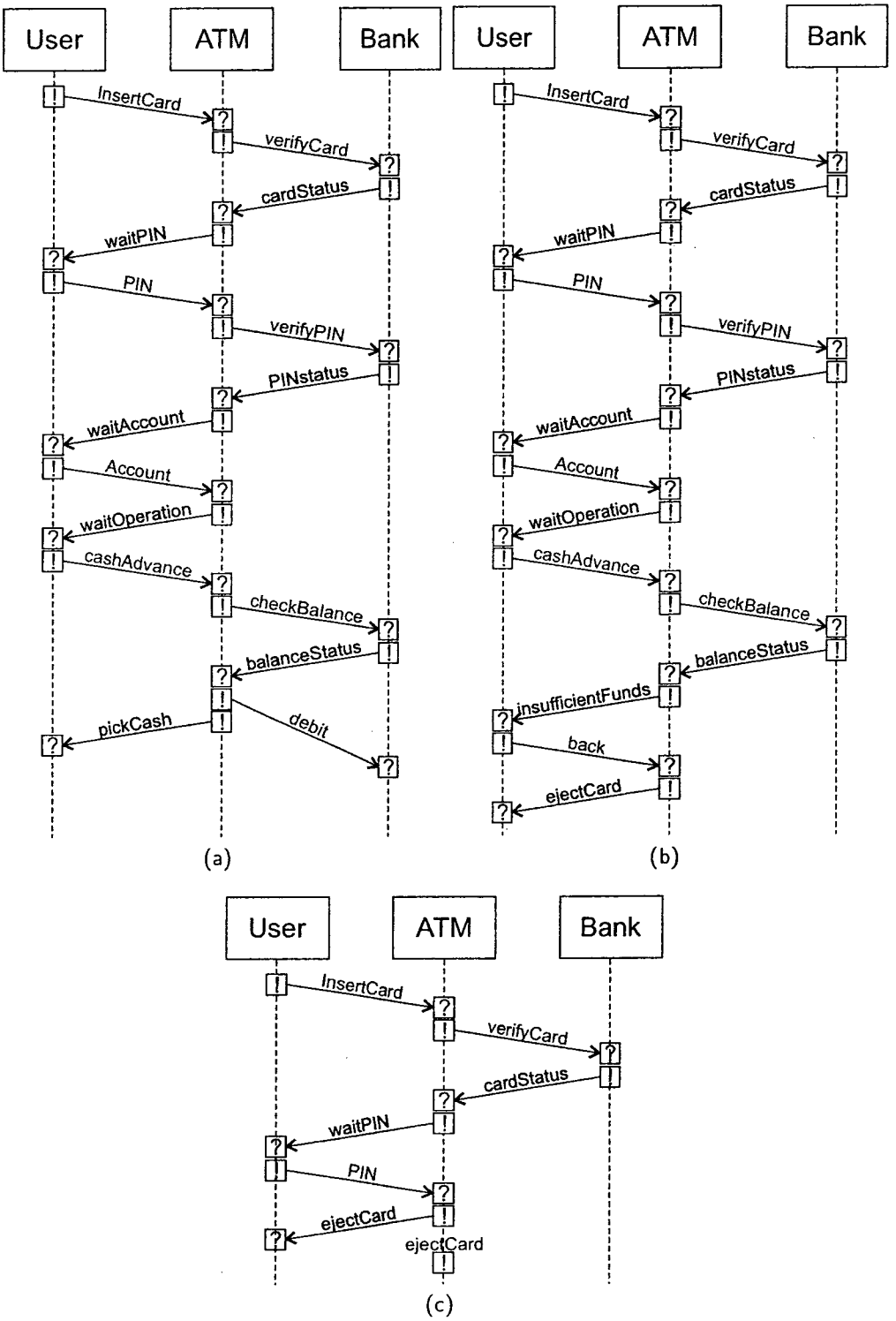


Figure 28: SPIN counterexamples for LTL properties: (a)counterexample of property ii, (b) counterexample of property iii, (c) counterexample of property iv

to see that there is at least one execution path on which the user receives the money before the bank receives the message to debit. If, for some reason, the message to debit is not delivered, the ATM will be not able to ask the user to give the money back.

- **Property 3 Counterexample**: The counterexample of figure 28(b) shows that, after receiving insufficient funds, the user does not have the opportunity to re-enter a different operation. It means that, if the user does a mistake, he/she needs to restart the whole operation from the beginning.

- **State Machine Property Counterexample**: figure 28(c) presents a counterexample where the ATM tries to eject the card twice. This inconsistent behavior should be eliminated from the design.

In order to illustrate the performance of our approach regarding this case study, Table 5 shows a summary of the results along with the number of states, number of transitions, memory and time used by SPIN to perform the verification. It is possible to see that the maximum time spent for verification was 0.132s, which is very reasonable. However, it is well-known that model checkers can have state explosion when verifying bigger models. There are some techniques to optimize the PROMELA specification to avoid state explosion, but those techniques will be considered in a future work.

Property	Result	Stored states	Matched states	Transitions	Memory Usage	Time
i	Passed	10991	11389	22380	3.673MB	0.13s
ii	Failed	88	4	92	2.501MB	0.06s
iii	Failed	986	1211	2197	2.598MB	0.072s
iv	Failed	11448	13729	25177	3.770MB	0.14s

Table 5: Summary of the results

Chapter 5

Implementation of the Tool Support

In this chapter, we provide details about the implementation of the tool to support the verification of UML diagrams. IBM Rational Software Architect has been as the platform of development. It is an advanced model-driven development tool. It leverages model-driven development with the UML for creating well-architected applications and services [20]. It contains a very powerful UML modeler compliant with the UML 2.2 standard. Moreover, as this tool is built on Eclipse, it can be augmented with Eclipse plug-ins, which is a very important capability to make a harmonious incorporation of the verification tool into the software development process.

5.1 Framework Implementation

Figure 29 highlights the components of the framework that have been implemented by our tool. In the following, we detail the implementation of each layer.

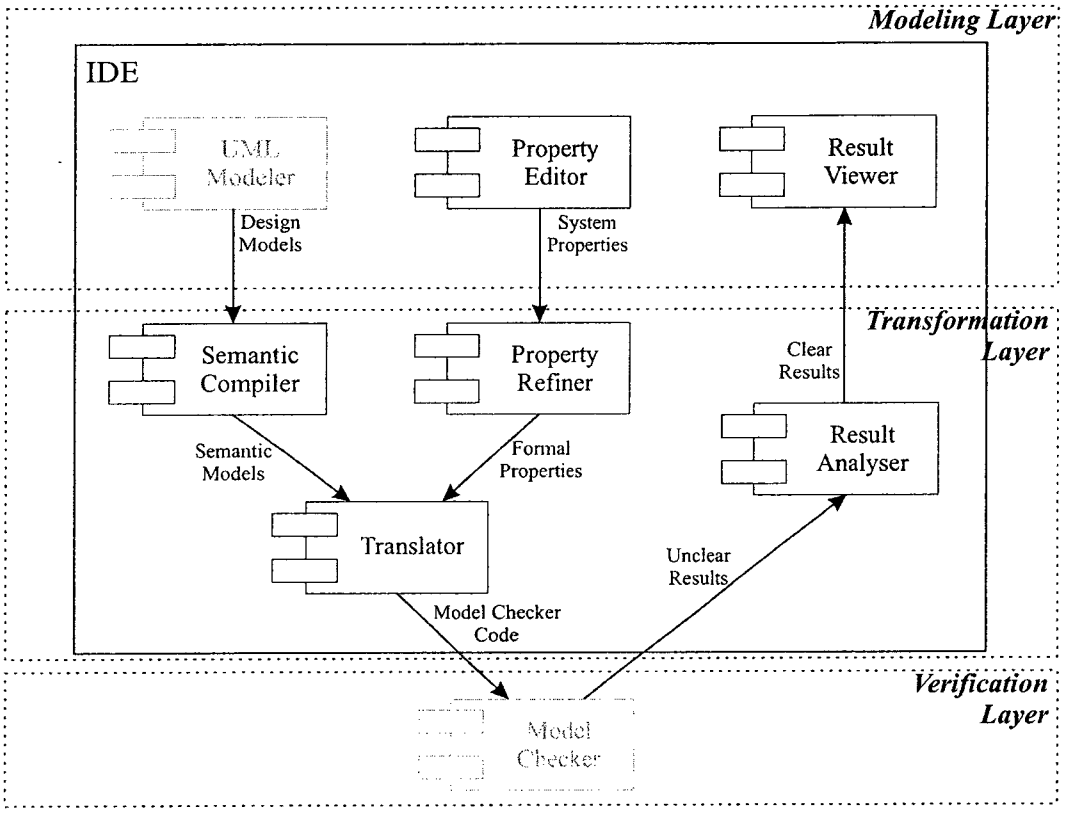


Figure 29: Components of the framework implemented in our tool

5.1.1 Verification Layer

Even though we used existing model checkers for the verification layer, we had to implement the interface between these tools and our plug-in. This interface needs to provide the following features:

1. It has to provide all functionalities of the external tools (i.e., SPIN, NuSMV and

GCC) to the upper layers.

2. Since the verification may take a long time, it needs to create a new thread for each verification request and these threads can be terminated by the user at any time.

3. It has to identify misbehavior and errors from the external tools and handle that or throw an exception to be handled by the upper layers.

4. It has to manipulate (i.e., open, close, create and delete) the files used in the verification procedure.

An example of a service implemented in that layer that appears on the GUI is the multi-thread verification. By using the Eclipse platform, we allow the developer to see the verification progress, terminate the thread at any time, and run the verification in background while doing different tasks. Figure 30 shows the window that appears when a new verification is launched.

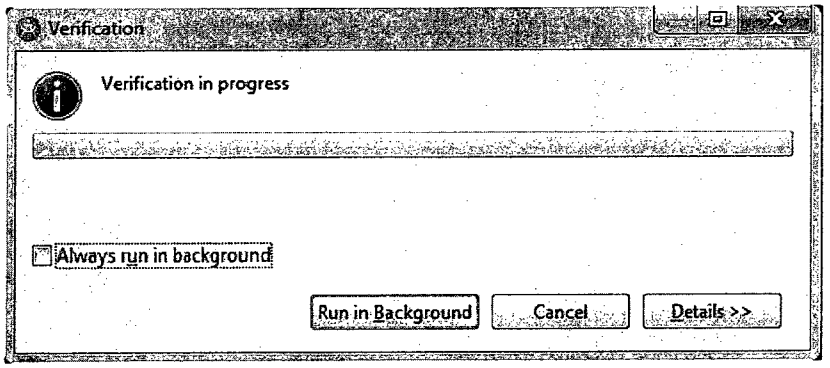


Figure 30: Eclipse window showing the verification progress

5.1.2 Transformation Layer

This layer contains major part of our tool. It provides the main capabilities to make possible the combination of UML and model checking. All components in this layer (i.e., Semantic Compiler, Property Refiner, Translator, and Result Analyser) have been implemented by our research team. Below are the features this layer provides:

1. Compilation of the semantics from UML Interactions, State Machines and Activity diagrams (the implementation of state machine and activities are based on existing approaches [2, 3, 11]).

2. Translation of MOBS2 property specification language and state machines into formal properties, i.e., LTL, CTL and transition systems.

3. Generation of SPIN and NuSMV code (NuSMV is used in state machine and activity verification).

4. Organization of the output of the model checker in order to link the result to the UML model and remove unnecessary information.

5. Handling of the syntax errors from the model checkers.

Among all these features, the compilation UML model is the most critical one. The UML meta-model is very complex and for each UML meta-element it is necessary to do a different processing (we can see this fact in Chapter 4 where each combine fragment generate a different PROMELA code). Moreover, one can compose different elements in a very complex manner. E.g., parallel fragments with if conditions, loops and breaks. To illustrate this complexity, figure 31 shows just a piece of the code

generated for the lifeline *User* in figure 26.

```
proctype iuia(){
xs wocy_36;
atomic{d_step{sends=1;receives=0;msg_sqmu=0;msg_ewsq=0;msg_eyec=0;msg_qmiw=0;
msg_wocy=1;msg_oggw=0;msg_cgws=0;msg_awao=0;msg_smga=0;msg_swao=0;msg_aoyy=0;
msg_cqwk=0;msg_yums=0;msg_sakw=0;msg_wkyu=0;msg_kmsi=0;msg_yiqa=0;msg_qkii=0;
msg_aske=0;msg_yegq=0;proc1_iuia=1;proc1_eyay=0;proc1_cyow=0;proc2_iuia=0;pro
c2_eyay=0;proc2_cyow=1;};wocy_36!wocy};
run sub63b463b4_iuia();
xr qkii_81;
atomic{qkii_81?qkii;d_step{sends=0;receives=1;msg_sqmu=0;msg_ewsq=0;msg_eyec=
0;msg_qmiw=0;msg_wocy=0;msg_oggw=0;msg_cgws=0;msg_awao=0;msg_smga=0;msg_swao=
0;msg_aoyy=0;msg_cqwk=0;msg_yums=0;msg_sakw=0;msg_wkyu=0;msg_kmsi=0;msg_yiqa=
0;msg_qkii=1;msg_aske=0;msg_yegq=0;proc1_iuia=1;proc1_eyay=0;proc1_cyow=0;pro
c2_iuia=0;proc2_eyay=0;proc2_cyow=1;};};
xs aoyy_6;
atomic{d_step{sends=1;receives=0;msg_sqmu=0;msg_ewsq=0;msg_eyec=0;msg_qmiw=0;
msg_wocy=0;msg_oggw=0;msg_cgws=0;msg_awao=0;msg_smga=0;msg_swao=0;msg_aoyy=1;
msg_cqwk=0;msg_yums=0;msg_sakw=0;msg_wkyu=0;msg_kmsi=0;msg_yiqa=0;msg_qkii=0;
msg_aske=0;msg_yegq=0;proc1_iuia=1;proc1_eyay=0;proc1_cyow=0;proc2_iuia=0;pro
c2_eyay=0;proc2_cyow=1;};aoyy_6!aoyy};
iuia_Sub63b463b4_iuia?token;
if
::(cardOK) -> ::else -> xr sqmu_5;
atomic{sqmu_5?sqmu;d_step{sends=0;receives=1;msg_sqmu=1;msg_ewsq=0;msg_eyec=0
;msg_qmiw=0;msg_wocy=0;msg_oggw=0;msg_cgws=0;msg_awao=0;msg_smga=0;msg_swao=0
;msg_aoyy=0;msg_cqwk=0;msg_yums=0;msg_sakw=0;msg_wkyu=0;msg_kmsi=0;msg_yiqa=0
;msg_qkii=0;msg_aske=0;msg_yegq=0;proc1_iuia=1;proc1_eyay=0;proc1_cyow=0;proc
2_iuia=0;proc2_eyay=0;proc2_cyow=1;};};
fi;
if
::(cardOK&&pinOK) -> xr aske_89;
atomic{aske_89?aske;d_step{sends=0;receives=1;msg_sqmu=0;msg_ewsq=0;msg_eyec=
0;msg_qmiw=0;msg_wocy=0;msg_oggw=0;msg_cgws=0;msg_awao=0;msg_smga=0;msg_swao=
0;msg_aoyy=0;msg_cqwk=0;msg_yums=0;msg_sakw=0;msg_wkyu=0;msg_kmsi=0;msg_yiqa=
0;msg_qkii=0;msg_aske=1;msg_yegq=0;proc1_iuia=1;proc1_eyay=0;proc1_cyow=0;pro
c2_iuia=0;proc2_eyay=0;proc2_cyow=1;};};
xs yums_22;
atomic{d_step{sends=1;receives=0;msg_sqmu=0;msg_ewsq=0;msg_eyec=0;msg_qmiw=0;
msg_wocy=0;msg_oggw=0;msg_cgws=0;msg_awao=0;msg_smga=0;msg_swao=0;msg_aoyy=0;
msg_cqwk=0;msg_yums=1;msg_sakw=0;msg_wkyu=0;msg_kmsi=0;msg_yiqa=0;msg_qkii=0;
msg_aske=0;msg_yegq=0;proc1_iuia=1;proc1_eyay=0;proc1_cyow=0;proc2_iuia=0;pro
c2_eyay=0;proc2_cyow=1;};yums_22!yums};
xr oggw_85;
atomic{oggw_85?oggw;d_step{sends=0;receives=1;msg_sqmu=0;msg_ewsq=0;msg_eyec=
0;msg_qmiw=0;msg_wocy=0;msg_oggw=1;msg_cgws=0;msg_awao=0;msg_smga=0;msg_swao=
0;msg_aoyy=0;msg_cqwk=0;msg_yums=0;msg_sakw=0;msg_wkyu=0;msg_kmsi=0;msg_yiqa=
0;msg_qkii=0;msg_aske=0;msg_yegq=0;proc1_iuia=1;proc1_eyay=0;proc1_cyow=0;pro
c2_iuia=0;proc2_eyay=0;proc2_cyow=1;};};
```

Figure 31: Piece of the model checker code from the Lifeline *User* in figure 26

In the current version of our model checker code generation, we associate a unique 4-letters ID to each UML element. It avoids syntax problems due to labels used in the model.

5.1.3 Modeling Layer

Figure 32 shows a screenshot of the IBM RSA workspace. The label 1 and 2 of this figure shows the Project explorer and the UML diagram editor, respectively, provided by the IBM RSA. The label 3 and 4 shows contributions of our plug-in to the workspace, where label 3 shows the summary of the verification result displaying which properties failed or passed, and label 4 shows the graph of the counterexample that is automatically generated by our tool.

Figure 33 depicts the textual property editor. In order to facilitate the task of writing properties, the tool provides the developer with a very friendly user interface which contains the following feature:

1. Syntax highlighter: it differentiates temporal operators, propositional formulas, and key words like *any*, *sends*, *receives*, *to* and *from*. This feature is very useful to detect typos and not well-formed properties.

2. Pre-defined operators: if helps a developer who is not familiar with temporal operator by showing the most common used operator on the top of the property editor. The user just needs to select the which is best suitable to his case.

3. Content assist: It is a very useful feature to help developer to pick the correct element to be added in his property. It also helps to avoid typos when writing

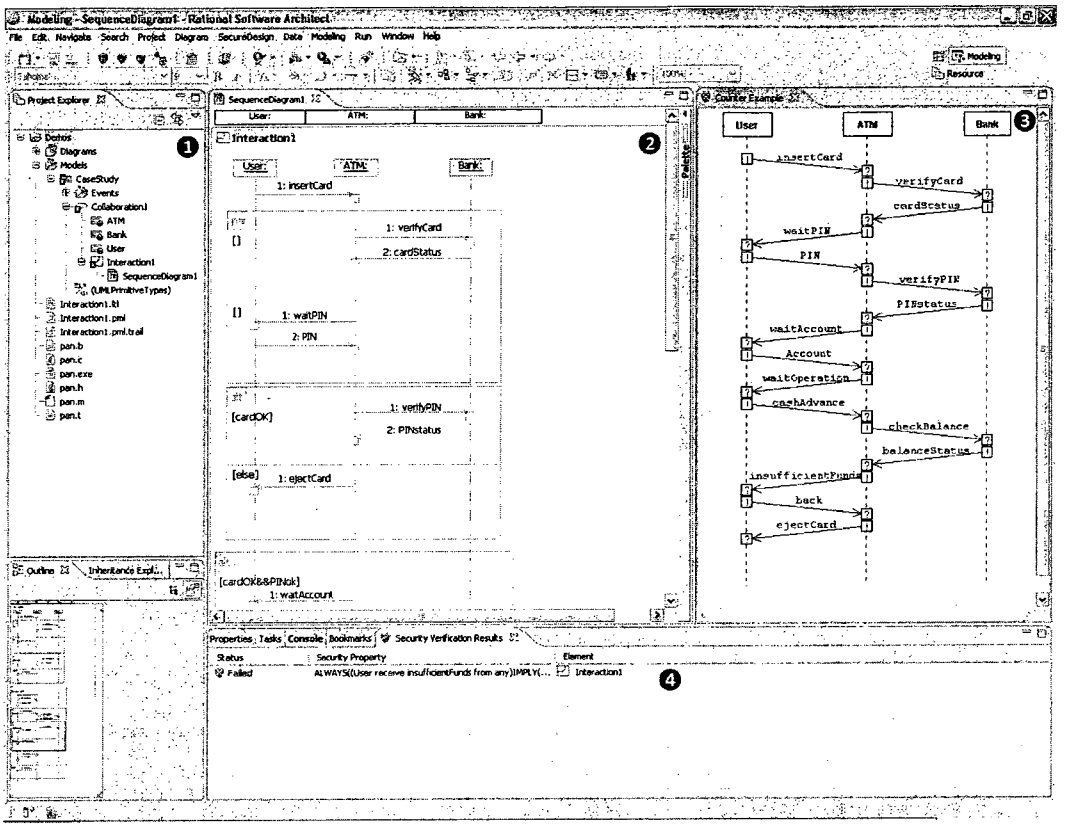


Figure 32: Screenshot of the IBM RSA workspace

the element's name.

Figure 34 shows a screenshot of the state machine property editor. The great advantage of this property editor is that it inheres all capabilities of the regular state machine editor provide by RSA. Therefore, it is expected that any developer, who is familiar with creating UML state machines, can easily use this tool to specify properties. Moreover, a property specify using this tool can be export/imported to a standard open-source format (XMI) and can be edited in any UML 2.0 compliant tool.

It is also important to mention that for the developer point of view, there is no

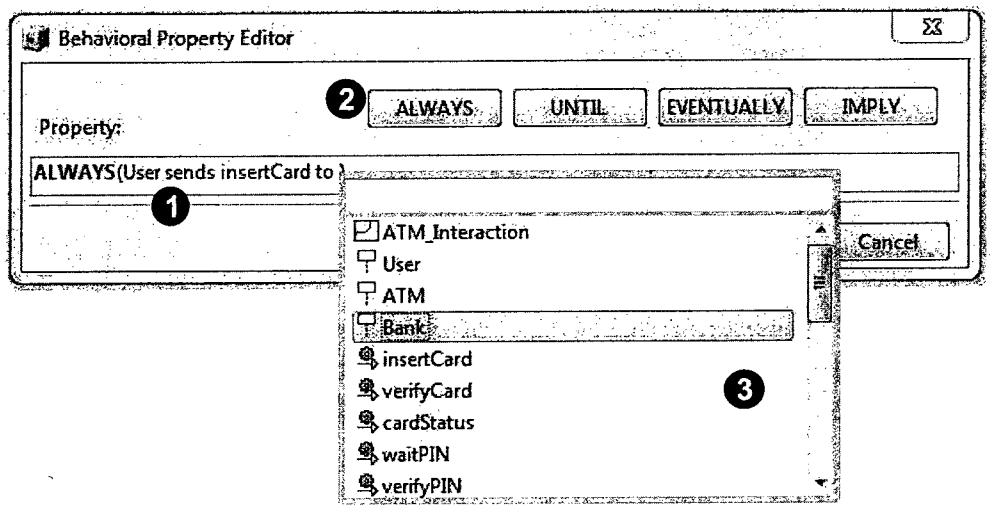


Figure 33: Textual property editor

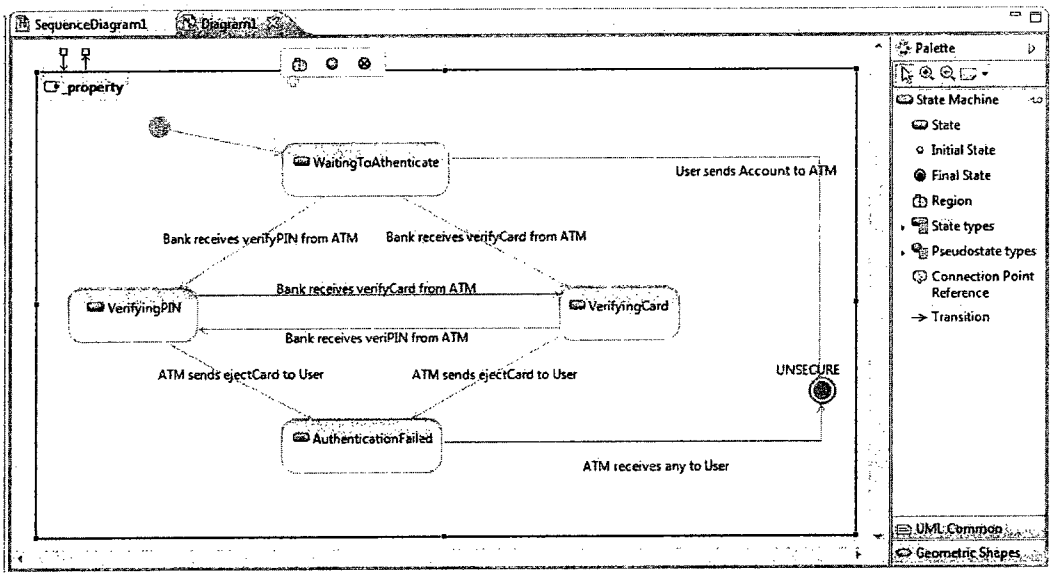


Figure 34: State machine property editor

need to understand the mechanism that performs the verification. The translation to PROMELA, the execution SPIN and the interpretation of the SPIN's output are all done in background. The developer should only assign properties to diagram, start the verification and assess the results.

Chapter 6

Conclusion

In this research work, we provide a framework for automatic verification of system design models expressed in UML based on model checking techniques. The framework provides a fully automatic verification engine with user-friendly mechanism for property specification and easy-to-understand graphical verification results.

In order to solve the problem of the complexity of formal logics, we performed an extensive investigation of the state-of-the-art on property specification in UML. It has been shown that there is a need to provide developer with more alternatives to specify property that can be automatically translated into a formal representation (i.e., LTL, CTL or transition system). Chapter 3 presents the main UML artifacts used for property specification along with a comparative study. It shows a lack of works trying to specify properties using behavior diagrams despite the wide tool support. This fact led us to propose a user-friendly technique that uses UML state machine for system property specification. Moreover, we also suggested a new language, on top of formal logics, that reduces the gap between the formal property and natural

languages. The properties written using this language are understandable even by a non-expert.

From the review of the state-of-the-art on verification of UML models, we observed a limited number of works dealing with UML 2.0 interactions. Chapter 4 addresses this problem. It introduces the semantics of UML 2.0 interactions along with the translation rules to obtain the model checker code. Since it takes into account the most popular UML combined fragments, this approach allows the developer to detect flaw in more completed complex interactions. The mechanism introduced in this work to keep track of the execution state provides the information the developer needs to write properties. Moreover, the way it was implemented gives flexibility to write very expressive properties. An illustrative interaction case study was explored in order to demonstrate the feasibility and effectiveness of the proposed approach.

Furthermore, in the Chapter 5, we provide details about the implemented tool. It shows that the proposed approach has been incorporated into the IBM Rational Software Architect IDE. In addition, it depicts how our approach is used from developer point of view.

The proposed methodology can be smoothly incorporated into the software development process in order to advance the error detection to the design phase. Consequently, it is becomes a powerful tool to reduce significatively the overall cost of an application. Moreover, this verification engine can be also applied to design models of already existing systems, which can help to discover so far unknown undesired behaviors and vulnerabilities.

As future work, an inter-diagram analysis might provide results that are much

more consistent in the assessment of UML models. Since many security aspects are usually not encapsulated in only one type of diagram; normally they are scattered in various diagrams of different kinds. Moreover, we indent to improve our approach in order to deal with very large and distributed system design without facing the state explosion problem. For that goal, we are developing a compositional verification approach where the base model is decomposed into small and easy to verify modules. Then, the result of the verification in each module is combined in such a way that we can obtain the result of the verification in the whole model.

Bibliography

[1] Gail-Joon Ahn and Michael E. Shin. Role-based authorization constraints specification using object constraint language. In *WETICE '01: Proceedings of the 10th IEEE International Workshops on Enabling Technologies*, pages 157–162, Washington, DC, USA, 2001. IEEE Computer Society.

[2] Luay Alawneh. Verification and validation in systems engineering : application to UML 2.0 activity and class diagrams. Master's thesis, Dept. of Electrical and Computer Engineering, Concordia University, November 2006.

[3] Luay Alawneh, Mourad Debbabi, Fawzi Hassane, Yosr Jarraya, and Andrei Soeanu. A unified approach for verification and validation of systems and software engineering models. In *Proc. of the 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems (ECBS'06)*, page 10pp., March 2006.

[4] K. Alghathbar and D. Wijeskera. Consistent and complete access control policies in use cases. In *Proceedings of UML 2003 - The Unified Modeling Language. Model Languages and Applications*, pages 373–387, October 2003.

[5] Rajeev Alur and Mihalis Yannakakis. Model checking of message flow diagrams. *United States Patent*, (US 6,516,306 B1), 2003.

[6] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[7] Andrej Bogdanov, Stephen J. Garland, and Nancy A. Lynch. Mechanical translation of i/o automaton specifications into first-order logic. In *FORTE '02: Proceedings of the 22nd IFIP WG 6.1 International Conference Houston on Formal Techniques for Networked and Distributed Systems*, pages 364–368, London, UK, 2002. Springer-Verlag.

[8] Batrice Brard, Michel Bidoit, Alain Finkel, Franois Laroussinie, Antoine Petit, Laure Petrucci, Philippe Schnoebelen, and Pierre McKenzie. *Systems and software verification: model-checking techniques and tools*. Springer-Verlag New York, Inc., New York, NY, USA, 1999.

[9] G. Brose, M. Koch, and K. P. Lhr. Integrating access control design into the software development process. In *Proceedings of 6th International Conference on Integrated Design and Process Technology (IDPT)*, Pasadena, CA, June 2002.

[10] Honghua Cao, Shi Ying, and Dehui Du. Towards model-based verification of bpel with model checking. In *CIT '06: Proceedings of the Sixth IEEE International Conference on Computer and Information Technology*, page 190, Washington, DC, USA, 2006. IEEE Computer Society.

[11] Andrei Soeanu Caval. Automatic verification of behavioral specification in software intensive systems. Master's thesis, Concordia University, Montreal, Quebec, Canada, May 2007.

[12] M.T. Chan and L.F. Kwok. Integrating security design into the software development process for e-commerce systems. In *Information Management & Computer Security*, volume 9, pages 112–122. MCB UP Ltd, 2001.

[13] Brian Chess and Jacob West. *Secure programming with static analysis.* Addison-Wesley Professional, 2007.

[14] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NUSMV: A new symbolic model verifier. In *CAV '99: Proceedings of the 11th International Conference on Computer Aided Verification*, pages 495–499, London, UK, 1999. Springer-Verlag.

[15] Ernesto Damiani, Claudio Agostino Ardagna, and Nabil El Ioini. *Open Source Systems Security Certification.* Springer Publishing Company, Incorporated, 2008.

[16] Maria del Mar Gallardo, Pedro Merino, and Ernesto Pimentel. Debugging UML designs with model checking. *Journal of Object Technology*, 1:101–117, 2002.

[17] T. Doan, L. D. Michel, and S. A. Dermurjian. A formal framework for secure design and constraint checking in uml. In *Proceedings of the International Symposium on Secure Software Engineering, ISSE'06*, Washington, DC, Mars 2006.

[18] Pete Epstein and Ravi Sandhu. Towards a uml based approach to role engineering. In *RBAC '99: Proceedings of the fourth ACM workshop on Role-based access control*, pages 135–143, New York, NY, USA, 1999. ACM.

[19] Eduardo B. Fernández. A methodology for secure software design. In *Software Engineering Research and Practice*, pages 130–136, 2004.

[20] Galina Bernshteyn/New York/IBM. Ibm - rational software architect. `http://www-01.ibm.com/software/awdtools/architect/swarchitect/`.

[21] Nicolas Guelfi and Amel Mammar. A formal approach for the verification of e-business processes with promela. Technical Report TR-SE2C-04-10, Software Engineering Competence Center, University of Luxembourg, Luxembourg, 2004.

[22] Nicolas Guelfi and Amel Mammar. A formal semantics of timed activity diagrams and its promela translation. *apsec*, 0:283–290, 2005.

[23] Øystein Haugen and Ketil Stølen. STAIRS - steps to analyze interactions with refinement semantics. In *UML 2003 - The Unified Modeling Language. Model Languages and Applications. 6th International Conference, San Francisco, CA, USA, October 2003, Proceedings*, volume 2863 of *LNCS*, pages 388–402. Springer, 2003.

[24] Gerard Holzmann. *Spin model checker, the: primer and reference manual.* Addison-Wesley Professional, 2003.

[25] Gerard J. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.

[26] International Telecommunication Union. ITU Recomendation Z.120: Message Sequence Chart (MSC), April 2004.

[27] Jan Juerjens. *Secure Systems Development with UML.* SpringerVerlag, 2003.

[28] Toni Jussila, Jori Dubrovin, Tommi Junttila, Timo Latvala, Ivan Porres, and Johannes Kepler Universitt Linz. Model checking dynamic and hierarchical UML state machines. In *Proceedings of MoDeV 2 a (2006)*, pages 94–110.

[29] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997.

[30] Alexander Knapp and Jochen Wuttke. Model checking of UML 2.0 interactions. In *Models in Software Engineering*, pages 42–51. Springer Berlin / Heidelberg, 2007.

[31] Philippe Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, 1995.

[32] Philippe Kruchten. *The Rational Unified Process: An Introduction.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[33] Diego Latella, Istvan Majzik, and Mieke Massink. Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker. *Formal Aspects of Computing*, 11(6):637–664, 1999.

[34] Yves Ledru, Régine Laleau, Michel Lemoine, Sylvie Vignes, Didier Bert, Véronique Donzeau-Gouge, Catherine Dubois, and Fabien Peureux. An attempt

to combine uml and formal methods to model airport security. In *CAiSE Forum*, 2006.

[35] Stefan Leue and Peter B. Ladkin. Implementing and verifying MSC specifications using promela/xspin. In *Proc. of of the DIMACS Workshop SPIN96, the 2nd International Workshop on the SPIN Verification System*, pages 65–89, 1997.

[36] Benjamin A. Lieberman. *The Art of Software Modeling.* Auerbach Publications, Boston, MA, USA, 2006.

[37] V. Lima, C. Talhi, D. Mouheb, M. Debbabi, L. Wang, and M. Pourzandi. Formal verification and validation of uml 2.0 sequence diagrams using source and destination of messages. *Electron. Notes Theor. Comput. Sci.*, 254:143–160, 2009.

[38] Torsten Lodderstedt, David A. Basin, and Jürgen Doser. Secureuml: A uml-based modeling language for model-driven security. In *UML '02: Proceedings of the 5th International Conference on The Unified Modeling Language*, pages 426–441, London, UK, 2002. Springer-Verlag.

[39] Erich Mikk, Yassine Lakhnech, Michael Siegel, and Gerard J. Holzmann. Implementing statecharts in promela/spin. In *WIFT '98: Proceedings of the Second IEEE Workshop on Industrial Strength Formal Specification Techniques*, page 90, Washington, DC, USA, 1998. IEEE Computer Society.

[40] Russ Miles and Kim Hamilton. *Learning UML 2.0.* O'Reilly Media, Inc., 2006.

[41] C. Montangero, M. Buchholtz, L. Perrone, and S. Semprini. For-lysa: UML for authentication analysis. In *Global Computing: IST/FET International Workshop, GC 2004*, volume 3267 of *Lecture Notes in Computer Science*, pages 93–106. Springer Verlag, 2005.

[42] Object Management Group. Introduction to OMG UML. `http://www.omg.org/gettingstarted/what_is_uml.htm`.

[43] Object Management Group. OCL 2.0 Specification, 2006.

[44] Object Management Group. UML 2.0 Superstructure Specification, 2007.

[45] Object Management Group. Business Process Model and Notation (BPMN) Version 1.2 Specification, 2009.

[46] Object Management Group. Common Warehouse Metamodel (CWM) Version 1.1 Specification, 2009.

[47] F. Painchaud, D. Azambre, M. Bergeron, J. Mullins, and R. M. Oarga. Socle: Integrated design of software applications and security. In *Proceedings of the 10th International Command and Control Research and Technology Symposium, ICCRTS'2005*, McLean, VA, USA, 2005.

[48] J. Pavlich-Marscal, L. Michel, and S. Demurjian. Enhancing uml to model custom security aspects. In *Proceedings of the 11th International Workshop on Aspect-Oriented Modeling*, 2007.

[49] G. Popp, J. Jürjens, G. Wimmel, and R. Breu. Security-critical system development with extended use cases. In *APSEC '03: Proceedings of the Tenth*

Asia-Pacific Software Engineering Conference Software Engineering Conference, page 478, Washington, DC, USA, 2003. IEEE Computer Society.

[50] I. Ray, N. Li, D. K. Kim, and R. France. Using parameterized uml to specify and compose access control models. In *Proceedings of the 6th IFIP TC-11 WG 11.5 Working Conference on Integrity and Internal Control in Information Systems, IICIS'03*, Lausanne, Switzerland, November 2003.

[51] Alfonso Rodriguez, Eduardo Fernandez-Medina, and Mario Piattini. Security requirement with a uml 2.0 profile. In *ARES '06: Proceedings of the First International Conference on Availability, Reliability and Security*, pages 670–677, Washington, DC, USA, 2006. IEEE Computer Society.

[52] James Rumbaugh, Ivar Jacobson, and Grady Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004.

[53] T. Schäfer, A. Knapp, and S. Merz. Model checking UML state machines and collaborations. *Electronic Notes in Theoretical Computer Science*, 55(3):357–369, 2001.

[54] Johann M. Schumann. *Automated theorem proving in software engineering*. Springer-Verlag New York, Inc., New York, NY, USA, 2001.

[55] Igor Siveroni, Andrea Zisman, and George Spanoudakis. Property specification and static verification of UML models. *ares*, 0:96–103, 2008.

[56] Eunjee Song, Raghu Reddy, Robert France, Indrakshi Ray, Geri Georg, and Roger Alexander. Verifiable composition of access control and application features. In *SACMAT '05: Proceedings of the tenth ACM symposium on Access control models and technologies*, pages 120–129, New York, NY, USA, 2005. ACM.

[57] C. Talhi, D. Mouheb, V. Lima, M. Debbabi, L. Wang, and M. Pourzandi. Usability of security specification approaches for uml design: A survey. In *Journal of Object Technology*, 2009.

[58] Bhuvan Unhelkar. *Verification and Validation for Quality of UML 2.0 Models.* Wiley-Interscience, 2005.

[59] M. F. van Amstel. Design and assessment of analysis techniques for UML sequence diagram. Master's thesis, Technische Universiteit Eindhoven, July 2006.

[60] Marcel F. Van Amstel, Christian F. J. Lange, and Michel R. V. Chaudron. Four automated approaches to analyze the quality of UML sequence diagrams. In *COMPSAC '07: Proceedings of the 31st Annual International Computer Software and Applications Conference - Vol. 2*, pages 415–424, Washington, DC, USA, 2007. IEEE Computer Society.

[61] José Luis Vivas, José A. Montenegro, and Javier Lopez. Towards a business process-driven framework for security engineering with the uml. In *ISC*, pages 381–395, 2003.

[62] Tim Weilkiens and Bernd Oestereich. *UML 2 Certification Guide: Fundamental & Intermediate Exams (The OMG Press).* Morgan Kaufmann Publishers Inc.,

San Francisco, CA, USA, 2006.

[63] Paul Ziemann and Martin Gogolla. OCL extended with temporal logic. In Manfred Broy and Alexandre Zamulin, editors, *5th Int. Conf. Perspectives of System Informatics(PSI'2003)*, volume 2890 of *LNCS*. Springer, 2003.

[64] Andrea Zisman. A static verification framework for secure peer-to-peer applications. In *ICIW '07: Proceedings of the Second International Conference on Internet and Web Applications and Services*, page 8, Washington, DC, USA, 2007. IEEE Computer Society.