Model Transformation For Validation Of Software Design

Shadi Al Dehni

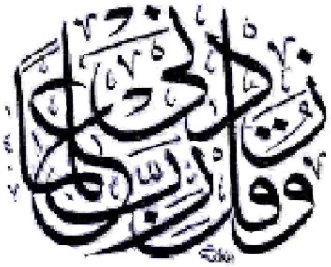
München, 2008

Institut für Informatik
Lehr- und Forschungseinheit für
Programmierung und Softwaretechnik

Ludwig
Maximilians
Universität
München
LMU

Dissertation im Fach Informatik
an der Fakultät für Mathematik, Informatik und Statistik
der Ludwig-Maximilians-Universität München

vorgelegt von
Shadi Al Dehni
aus Damaskus, Syrien



Erstgutachter: Prof. Dr. Martin Wirsing

Zweitgutachter: Dr. rer. nat. habil. Stephan Merz

Prüfer: Prof. Dr. Hans-Peter Kriegel

Prüfer: Prof. Dr. Rolf Hennicker

Tag der mündlichen Prüfung: 08. Juli 2008

Abstract

Model checking is a method for formally verifying finite-state concurrent systems such as circuit designs and communication protocols. System specification is expressed as temporal logic formula, where efficient symbolic algorithms are used to traverse the model defined by the system and check if the specification holds or not. Large state space can often be traversed in minutes. Graphical notation plays an important role in software modeling and designs. The Unified Modeling Language (UML) is a standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems. Nowadays graph grammars enable a high level of abstraction of software architecture and form a basis for various analysis and transformations. Their methods, techniques, and results have already been applied in many fields of computer science.

In this thesis, we propose new techniques for an efficient transformation of UML software designs into a formalization for the model checking software, expressed by the approach of graph grammars and graph transformation systems. We have implemented our techniques in several case studies like ATM designs and security protocols. We demonstrate empirically that our transformation techniques are well-suited to apply them in specific UML software designs. Our transformation techniques run along two lines: The first line is to transform the UML state machines into equivalent simpler state machines called executable state machines, where the model checker HUGO and SPIN are called upon to verify whether certain required properties are indeed realized by the UML state machine designs. The second line is to transform the UML state machines into predicate diagrams, whereas the JML assertions and the Bandera Specification Language (BSL) are used to verify the desired properties. The model checker DIXIT attempts to verify the properties against the created predicate diagrams. Our prototype tool DAMAS is developed to use our transformation strategies to transform and compile the UML software designs into formalization of model checking software and vice versa.

Zusammenfassung

Model Checking ist eine Methode zur formalen Verifikation von nebenläufigen Systemen mit endlichem Zustandsraum. Beispiele solcher Systeme sind elektrische Schaltungen oder Kommunikationsprotokolle. Eine Systemspezifikation ist durch temporallogische Formeln gegeben, für die es effiziente symbolische Algorithmen gibt, um das Systemmodell zu durchlaufen und die Gültigkeit der Spezifikation zu überprüfen. Sogar große Zustandsräume können mit diesen Algorithmen in wenigen Minuten überprüft werden. Graphische Notationen spielen bei der Modellierung und beim Entwurf von Software eine große Rolle. Die Unified Modeling Language (UML) bildet dabei einen Standard für die Spezifikation, Visualisierung, Konstruktion und Dokumentation von Artifakten von Software-Systemen. Heutzutage erlauben Graphgrammatiken einen hohen Grad der Abstraktion von Software-Architekturen und bilden eine Basis für verschiedene Analysen und Transformationen. Die verwendeten Methoden, Techniken und Ergebnisse wurden bereits in vielen Bereichen der Informatik angewandt.

In dieser Arbeit wurden neue Techniken zur effizienten Transformation von UML-Software-Entwurfsmodellen in eine für eine Model Checking Software geeignete Form, die durch Graphgrammatiken und Graphtransformationssysteme erzeugt wird, entwickelt. Diese Techniken werden an verschiedenen Fallstudien, wie u.a. an Entwurfsmodellen von Geldautomaten-Software oder von Sicherheitsprotokollen, erprobt. Die Erfahrung hat gezeigt, dass die entwickelten Transformationstechniken bei bestimmten UML-Entwurfsmodellen besonders gut geeignet sind. Die Transformationstechniken arbeiten auf zwei Wegen: auf dem ersten werden UML-Zustandsmaschinen zunächst in einfachere Zustandsmaschinen, die so genannten ausführbaren Zustandsmaschinen transformiert. Auf diese werden dann die Model Checker HUGO und SPIN angewendet, um zu prüfen, ob bestimmte geforderte Eigenschaften von den UML-Zustandsmaschinen tatsächlich erfüllt werden. Auf dem zweiten Weg werden UML-Zustandsdiagramme in Prädikatendiagramme transformiert, wobei JML-Zusicherungen und die Bandera Specification Language (BSL) zum Nachweis der gewünschten Eigenschaften verwendet werden. Der Model Checker DIXIT kann zur Überprüfung der erzeugten Prädikatendiagramme verwendet werden. Die vorgestellten Strategien zur Transformation und Kompilation von UML-Entwurfsmodellen in eine für eine Model Checking Software geeignete Form, und umgekehrt, wurden in dem Werkzeug-Prototyp DAMAS implementiert.

Acknowledgement

This PhD thesis is the result of almost five years of work at the Ludwig Maximilians University in Munich. The move from Syria to Germany and everything I learned and experienced during this time is impossible to describe in full. Writing a PhD thesis in another continent is a process that requires not only hard work but a lot of support from family, friends, and colleagues. Therefore, it is my pleasure to thank the many people who made this thesis possible.

I am especially grateful to my supervisor, Professor Martin Wirsing, for his patient guidance into the theoretical, conceptual, and methodological areas of my PhD research. Prof. Wirsing has supported me not only by providing a research assistantship over the five years, but also he has provided me with inspiration and motivation to complete this thesis. Herrn Wirsing, Vielen herzlichen Dank an Sie!

I would like also to thank Professor Fred Kröger, his recommendations and suggestions have been invaluable to my research project.

My sincere thanks go out to Prof. Alexander Knapp for all his help and support especially in the initial period of my research studies. Danke Alex!

Special thanks also go to Dr. Jan Jürjens for the collaborative work in the security systems, and many thanks for the time we spent together in the workshop in UK.

It is an honor to have a thesis committee devised of scholarly and dedicated educators. I would like to thank Dr. Stephan Merz for his invaluable suggestions to improve my thesis. I am also highly grateful to prof. Rolf Hennicker and prof. Hans-Peter Kriegel for the great discussions.

My profound gratitude to the PST team in the Institute of Informatics in LMU for providing me invaluable scientific assistance and an excellent and very stimulating working atmosphere. I am very grateful to Matthias Ludwig for his continuous help in Latex during writing my dissertation and many thanks for Dr. Hubert Baumeister for our great discussions.

The financial support of DAAD (Deutscher Akademischer Austauschdienst) allowed the realization of my PhD at the LMU München. Many thanks to DAAD for the help and care during my research study at LMU.

A special word of thanks goes to my parents, my brothers Fadi and Nour. Mum and dad, thank you for teaching me to have the patient to achieve this scientific work. Many thanks to my twin brother Dr. Fadi for his constant support and encouragement of my academic pursuits, and for Nour for all his help. Above all, thanks to my friends for their warm friendship.

Shadi Al-Dehni,

München, 05.06.2008

Contents

1 Introduction

Object-oriented methods are widely accepted for software development in the business application domain and have also been advertised for the design of embedded and real-time systems [SS99]. Software architecture and design are usually modeled and represented by informal diagram languages, such as architecture diagrams and UML diagrams, while these graphical notations are easy to understand and are convenient to use, they are not amenable to automatic verification and transformation. Validation methods make it possible to check both the correctness of the specification, and to establish that the known requirements of the specified system are clearly and unambiguously expressed within the standard. The main validation techniques implemented in automatic tools are interactive simulation and various types of state space exploration, but the kind of methods is still not well-developed for analysing and validation design.

Verification techniques for reactive systems are traditionally classified as either deductive or algorithmic. Whereas deductive verification can in principle establish properties of arbitrary complex systems, algorithmic verification such as model checking is usually restricted to finite-state systems [CMMag]. Model checking is the most successful approach that has emerged for verifying requirements. Model checking is a method for formally verifying finite-state concurrent systems such as circuit designs and communication protocols. It explores all possible paths through a design which implies that the number of paths is finite. It uses a specification language based on some kind of temporal logic for expressing properties.

In applying model checking to software design, in particular of UML, we find the software design usually involves infinite state spaces. This is not directly suited for model checking, since model checkers accept only designs where the state space is finite. On the other hand, the semantic definitions of software model checking are also very different from the semantic definition of the software design.

Applying model checking to software design, confronts us with two problems:

1. Infinite state spaces: Software design usually infinite state spaces, but model checkers accept only those cases where the state space is finite.

2. Bridging the semantic gap: Bridging together, the semantics of software design with model checking semantics.

In the last few years, several techniques have been developed for abstracting the infinite state spaces to finite ones. The construction of abstraction is essential for reducing large or infinite state space systems to small or finite systems to be explored. That is, to solve the first problem, we use the concept of executable state machines trying to reduce state space of UML software design to be explored and allow model checking of more complex system to be run. For the second problem, we use the concept of graph grammar and graph transformation for representing and implementing the software design to be well-suited for the software model checking.

In my thesis we illustrate the application of graph grammar and graph transformation systems of UML state machines for software model checking. We compile UML state machines to new graph representation (graph models) and transform the graph models into new models suitable for model checking softwares, where the model checkers HUGO and DIXIT are used.

In the following sections we briefly discuss some basic approaches of our work that underline the thesis. In section 1.1 we describe the Unified Modeling Language (UML) which provides the ability to capture the characteristics of a system by using notations for documenting systems based on the object-oriented design principles. In section 1.2 we outline the technique of software model checking and discuss model checking tools like SPIN and HUGO. Section 1.3 gives a short overview on graph grammar and graph transformation system and the tool that we use to implement our case studies. In section 1.4 we present two important abstraction techniques; executable state machines and the predicate diagram to refine the UML designs. Section 1.5 illustrates the transformation of security model using graph grammars approach into software model checker DIXIT to implement the approach of predicate diagram. We introduce our case studies and our prototype tool DAMAS for automatic transformation in section 1.6. Finally, the last section 1.7 describes the structure of my thesis.

1.1 UML Software Design

Graphical notations are widely used in software design and development. These notations can greatly help with modeling and representing of software architecture and design [SG95]

Notations like UML [BRJ99] are very good for communicating designs. UML is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system. UML consists of two parts: a notation, used to describe a set of diagrams (also called the syntax of the language) and a metamodel (also called the semantics of the language) that specifies the abstract integrated semantics of UML modeling concepts. The UML defines nine diagram types, which allow different aspects (static, behavioral, interaction, and implementation) and properties of a system design to be expressed.

The diagram which is relevant to my work is the UML state machine diagram. A state machine diagram is a specification that describes all possible behaviors of some dynamic model element. Behavior is modeled as a traversal of a graph of state nodes interconnected by one or more joined transition arcs that are triggered by the dispatching of series of event instances. During this traversal, the state machine executes series of actions associated with various elements of it.

UML 2.0 incorporates an action semantics, which adds to UML the syntax and semantics of executable actions and procedures [Gro03b]. Action semantics refers to the ability to formally describe actions that can be analyses by a computer and executed. Formal actions make models executable. Action semantics is a partial metamodel integrated in the global UML metamodel. It allows the specification of many kinds of actions, such as computational algorithms to be applied to data, as well as reactive and concurrent behavior with asynchronous and synchronous communication. Therefore, action correspond to a manipulation of the object model; it can modify it or just read it. There are numerous actions like creating and deleting an object, getting an object attribute value, setting an object attribute value, calling an object operation, creating, deleting and traversing an association linking two objects, sending a signal to an object or receiving a signal of an object.

There are several modeling tools for designing UML diagrams [Qua98,RVR+99,Poe04,Mag04]. ArgoUML is an open source Java-based UML tool [RVR+99]. It supports most of the nine standard UML diagrams, it has also the ability of reverse engineering compiled Java code and generating UML diagrams for the code. Commercial tools are e.g. Rose [Qua98], Together [Tog04], Poseidon [Poe04] and MagicDraw [Mag04]. Among them, MagicDraw is a visual UML modeling and CASE tool with team work support. MagicDraw contains a handy UML editor, a powerful code engineering tool, UML model reporting facilities, a custom OO model generator, a team modeling tool, and a database modeling tool.

In our study we use MagicDraw to create the UML state machines diagrams of our case studies. MagicDraw stores the UML state machines diagrams as standard XMI files in a ZIP archive. Most of our case studies and examples are represented using MagicDraw.

1.2 Software Model Checking

Verifying a program is providing, in a formal mathematical way, that the program satisfies a specification written in a logical language. Therefore verification has often been claimed to be a promising approach for ensuring the correctness of software. Formal verification, where a system is verified with respect to desired behavior, has now become popular in industry, especially in mission and safety critical applications. Specifically model checking methods, which can be

fully automated, are being used extensively to verify that a finite state system meets a desired behavior. The desired behavior is often specified by a temporal logic formula. Software model checking is typically applied to system whose intricacy resides more in the control than in the data; this includes for instance hardware, concurrent protocols, process control systems, and more generally as reactive systems [MQR95].

A model checking tool accepts a design (called system model) and a property (called specification) that the final system is expected to satisfy. The model checker explores all paths through the state space in order to check whether the specification holds for the model. The tool then outputs yes if the given model satisfies the given specification and generates a counter example otherwise (see figure 1.1) . A consequence of this procedure is that the state spaces has to be finite. The counter example details why the model doesn't satisfy the specification. By studying

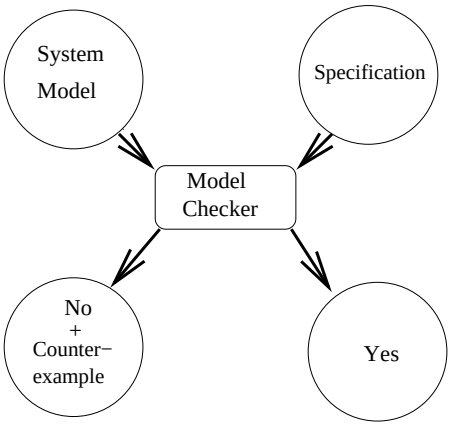


Figure 1.1: Model Checker Tool Mechanism

the counter example, we can pinpoint the source of the error in the model, correct the model, and try again. The idea of this iterative process is to ensure that the model satisfies enough system properties.

There are several model checking tools such as SPIN [Hol04b], FDR [(Lt], UPPAAL [BLPY97], and NuSMV [Nus02]; SPIN is one if the most popular software tool that can be used for the formal verification of distributed software systems [Hol04b]. SPIN uses a high-level language to specify system descriptions, called PROMELA. SPIN has been used to trace logical design errors in distributed systems design, such as operating systems, data communication protocols etc. The tool checks the logical consistency of a specification. It reports on deadlocks, unspecified receptions, flags incompleteness, race conditions, and unwarranted assumptions about the relative speeds of processes, SPIN and the model checkers mentioned above do not accept software design directly, but use logic and automata formalisms for describing models and specifications. A tool for UML software model checking is the HUGO system [KM11]. HUGO

validates UML software designs by translating UML models to different model checkers and theorem provers and by reflecting the results of theses systems back to UML. More specifically, HUGO connects UML state diagrams and OCL with the model checkers SPIN, UPPAAL and the system language of the interactive theorem prover KIV [KIV86].

We use HUGO to check the desired properties against the UML state machines. HUGO takes as input standard XMI files that can be produced by MagicDraw and allows both the model and the properties to be specified in terms of UML diagrams. We use HUGO to verify the properties against the diagrams.

1.3 Graph Grammars Approach

The research area of graph grammar and graph transformations dates back to 1970 [Roz97]. It combines ideas from graph theory, algebra, logic, and category theory. Its methods, techniques, and results have already been applied in many fields of computer science. This wide applicability is due to the fact that graphs are a very natural way to explain complex situations on an intuitive level. A graph grammar can be used to define the set of syntactically correct diagrams in an application area, whereas graph transformation bring the evolution of structures. It has become an attractive means to model and to study very different structures in a uniform way.

The field of graph grammars applies formal language theory to the specification of graphs. A graph grammar consists of a set of productions that can be used to construct valid sentences in a graph (network) language. The production are analogous to macros that can be applied to edit the network or graph. For example, vehicle routine systems commonly provide a mechanism to reroute a customer. figure 1.2 shows a production of a graph grammar for reroutings. The left

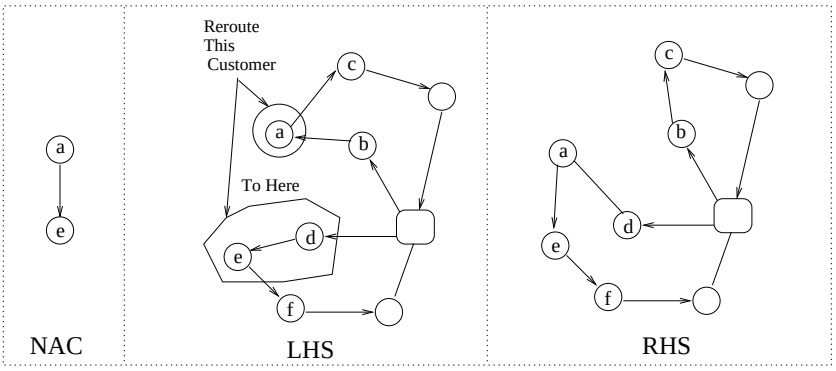


Figure 1.2: Graph Grammar Production

side of this figure shows that we want to reroute the customer a which is placed between b and

c to another route between d and e. On the right hand side, the result of reroutings a is depicted by the graph. Applying the production rule to a network (matching the left hand side of the rule) will perform this reroutings of a.

Graph transformations associated with graph grammars are well-suited for modeling the dynamic behavior of systems [Roz97]. Therefore graph grammars and graph transformations become attractive as a programming paradigm for software and graphical interfaces.

Tools for graph transformation systems are VIATRA [VVP02], Fujaba [Hom02], AGG [TER99]. VIATRA "Visual Automated model Transformation" is a prototype tool which provides a general and automated framework for specifying transformations between arbitrary models conforming to their metamodel. Fujaba, AGG are visual tools environments which support a hybrid programming style based on graph transformation and Java. They consist of editors, interpreters and debuggers for attributed graph transformation and attribute computation by Java.

We use AGG transformation tool to represent our case studies as graph models. More precisely we define attributed type graphs and the rules leading to transformation of the host graph into new graph models which are suitable for software model checking.

1.4 Abstraction Techniques

For applying software model checking to software design we face the problem, that infinite state spaces are not suited for software model checking. In the last few years, several techniques have been developed for abstracting from infinite state spaces to finite ones. In general, abstraction concerns the transformation of the formal description of a system into a simpler form. We use two techniques to abstract our UML software design:

1. Abstracting the UML state machines in equivalent simpler designs called executable state machines, which are an UML state machines consists of just simple states with actions and guards [SM05].

2. The second technique is to refine the UML software design using predicate diagrams for the design of reactive system [CMMag]. In this case we have to define the system as TLA specification and a model as predicate diagram, and then to verify if the model satisfies the system specification. A specification $Spec$ satisfies a property F if and only if the implication $Spec \Rightarrow F$ is valid [Lam94]. Predicate diagram can be used to refine this implication into two conditions: first, all behaviors allowed by $Spec$ must also be traces through the diagram (in other words, the diagram is a correct abstraction of $Spec$), and second, every trace through the diagram must satisfy F.

In my approach, we use the previous abstraction techniques to refine the UML software design. We transform the UML software design using graph transformation engine (usually AGG) to one of the previous abstraction techniques and then we check the validity of the desired properties using the software model checker HUGO and DIXIT.

1.5 Security Model Transformation

A security protocol (cryptographic protocol or encryption protocol) is an abstract or concrete protocol that performs a security-related function and applies cryptographic methods. Secure Sockets Layer (SSL), is a cryptographic protocol that provide secure communications on the Internet for such things as web browsing, e-mail, and other data transfers. The SSL protocol exchanges a series of messages between an SSL-enabled server and an SSL-enabled client when they first establish an SSL connection. This exchange of messages is designed to facilitate the following actions:

- Authenticate the server to the client.
- Allow the client and server to select the cryptographic algorithms, or ciphers, that they both support.
- Optionally authenticate the client to the server.
- Use public-key encryption techniques to generate shared secrets.
- Establish an encrypted SSL connection.

JESSIE is a free, clean-room implementation of the Java Secure Sockets Extension, the JSSE. It provides the core API for programming network sockets with the Secure Socket Layer (SSL), which creates an authenticated, unforgeable, and protected layer around network communications. Its goal is to be a drop-in package for free Java class libraries such as Classpath and its derivatives, and is being written to depend only on free software, and only with the API specification and the public protocol specifications [JES].

We propose an approach for verfying implementations of security protocols. In particular, we use the JML (Java Modeling Language) and BSL (Bandera Specification Language) assertions to verify whether the cryptographic connection is correctly implemented. As an example for our approach we present the verification process of JESSIE.

Graph transformation techniques are used to transform the security protocols into new designs that are well-suited for software model checking.

1.6 Case Studies and Tool Design

Case studies and examples are important for illustrating the research results, and for giving feedback whether the results are usable in practice. In my PhD research, I developed some examples and case studies for validating the research results. Case studies like ATM state machines and the state machines of 2-phase commit protocol are carried out in my thesis. We check the validation of the state machine model using the model checker and we use the concept of graph grammar and graph transformation to transform the state machine models into new simpler abstracted models. The case study SSL-Handshake protocol is produced during our researching on security transformation approach. We transform the protocol using graph transformation approach into new design that is well-suited for the model checkers DIXIT and HUGO.

In order to make our theoretical results applicable to larger examples and practical case studies, we developed a tool called DAMAS to support our model transformation techniques. DAMAS provides the user the ability to transform the UML state machines into simpler one (executable state machines) automatically and to check if the required properties using the model checker HUGO and DIXIT are valid in the design or not. DAMAS implements also an appropriate graph translator for HUGO and DIXIT.

1.7 Thesis Structure

The thesis is organized as follows: In the second chapter we shall review briefly the temporal logic approach and model checking software techniques. In this chapter we present the case study state machines of ATM (Automatic Teller Machine) and check using HUGO model checker whether certain specified collaborations are indeed feasible for the required ATM state machines. Technically, HUGO compiles state machines into a PROMELA model, and collaborations into sets of Büchi automata (" never claim"). The model checker SPIN is then called to verify the model against the automata.

In the third chapter we discuss the approach of graph grammar and graph transformation systems. We introduce in this chapter the formal definition of graph grammar and graph transformation rules. We pursue strategy to create the graph model from the observed behavior of the scenario. We illustrate the approach of attributed graph transformations systems and the graph transformation mechanisms like SPO (single pushout) and DPO (double pushout). We use in this chapter the theoretical definitions of graph grammars and graph transformation systems to represent some practical examples like PacMan game.

In chapter 4 we develop a new transformation techniques to transform the UML state machine

designs into simpler form. That is, we transform the state machine into simpler executable state machine using the graph transformation approach. In this chapter, we transform the case study (state machine of ATM) into simpler state machines whereas the model checker HUGO and SPIN are called upon to see if our transformation strategies are useful in reducing the state space of the model.

In chapter 5 we introduce the approach of security protocols. We research in this chapter the SSL security protocols and we introduce the implementation of these protocols in the security software JESSIE. In order to ensure that the security protocols are indeed well-implemented in JESSIE, we use the JML and Bandera assertions. The next step is to transform the protocol into new graph models that are well-suited for software model checking. We use the approach of predicate diagram to automatically verify whether the required security properties are indeed realized by the protocols or not.

Finally, chapter 6 introduces our tool DAMAS and illustrates verifying some protocols automatically using DAMAS.

Appendix A illustrates the UTE specification language of HUGO model checker for the state machine models of ATM, whereas appendix B illustrates how we automatically verify SSL-Handshake protocol using DAMAS. Appendix C shows the Bandera abstraction tool to verify using the Bandera Specification Language (BSL) if the Java implementation of the SSL-Handshake protocol in JESSIE is indeed verify the disered features of the encrypted connections.

2 Logical Foundations and Tools

2.1 Introduction

The serious study of logic as an independent discipline began with the work of Aristotle (384-322 BCE). Generally, however, Aristotle's sophisticated writings on logic dealt with the logic of categories and quantifiers such as "all", and "some", which are not treated in propositional logic. However, in his metaphysical writings, Aristotle espoused two principles of great importance in propositional logic, which have since come to be called the Law of Excluded Middle and the Law of Contradiction. Interpreted in propositional logic, the first is the principle that every statement is either true or false, the second is the principle that no statement is both true and false. These are, of course, cornerstones of classical propositional logic. There is some evidence that Aristotle, or at least his successor at the Lyceum, Theophrastus (d. 287 BCE), did recognize a need for the development of a doctrine of "complex" or "hypothetical" propositions, i.e., those involving conjunctions (statements joined by "and"), disjunctions (statements joined by "or") and conditionals (statements joined by "if... then..."), but their investigations into this branch of logic seem to have been very minor.

Statement A statement can be defined as a declarative sentence, or part of a sentence, that is capable of having a truth-value, such as being true or false. So, for example, the following are statements:

Angela Merkel is the Chancellor of Germany

Berlin is the capital of Germany.

Proposition The term *proposition* is sometimes used synonymously with statement. However, it is sometimes used to name something abstract that two different statements with the same meaning are both said to "express". In this usage, the German sentence, "Es regnet", and the French sentence "Il pleut", would be considered to express the same proposition. However, the nature or existence of propositions as abstract meanings is still a matter of philosophical controversy.

Propositional logic is a branch of logic that studies ways of combining or altering statements or propositions to form more complicated statements or propositions. Joining two simpler propositions with the word ”and” is one common way of combining statements. When two statements are joined together with ”and”, the complex statement formed by them is true if and only if both the component statements are true. Because of this, an argument of the following form is logically valid:

Berlin is the capital of Germany **and** Berlin has a population of over four million.
Therefore, Berlin has a population of over four million.

Propositional logic largely involves studying logical connectives such as the words ”and” and ”or” and the rules determining the truth-values of the propositions they are used to join, as well as what these rules mean for the validity of arguments, and such logical relationships between statements as being consistent or inconsistent with one another.
Propositional logic also studies way of modifying statements, such as the addition of the word ”not” that is used to change an affirmative statement into a negative statement. Here, the fundamental logical principle involved is that if a given affirmative statement is true, the negation of that statement is false, and if a given affirmative statement is false, the negation of that statement is true [ABKS]. On the other hand, Propositional logic can also be thought as the study of logical operators. A logical operator is any word or phrase used either to modify one statement to make a different statement, or join multiple statements together to form a more complicated statement. In English, words such as ”and”, ”or”, ”not”, ”if ... then...”, ”because”, and ”necessarily”, are all operators. we will use the abbreviation PL to refer to the propositional logic in the next sections.

A Statement Letter of PL is defined as any uppercase letter written with or without a numerical subscript. According to this definition, ’A’, ’B’, ’B2’, ’C3’, and ’P14’ are examples of statement letters. The numerical subscripts are used just in case we need to deal with more than 26 simple statements: in that case, we can use ’P1’ to mean something different than ’P2’, and so forth.

A Connective or operator of PL is any of the signs $\neg$ (negation), $\wedge$ (conjunction), $\vee$ (disjunction), $\rightarrow$ (implication), and $\leftrightarrow$ (equivalence).

A well-formed formula of PL is defined recursively as follows:

1. Any statement letter is a well-formed formula.

2. α is a well-formed formula, then so is $\neg\alpha$.
3. α and β are well-formed formulas, then so is $\alpha \wedge \beta$.
4. α and β are well-formed formulas, then so is $\alpha \vee \beta$.
5. α and β are well-formed formulas, then so is $\alpha \rightarrow \beta$.
6. α and β are well-formed formulas, then so is $\alpha \leftrightarrow \beta$.
7. Nothing that cannot be constructed by successive steps of (1)-(6) is a well-formed formula.

The notion of a well-formed formula should be understood as corresponding to the notion of a grammatically correct or properly constructed statement of language PL. This definition tells us, for example, that "$\neg(Q \vee \neg R)$" is grammatical for PL because it is a well-formed formula, whereas the string of symbols, "$)\neg Q\neg v(\leftrightarrow P\&$", while consisting entirely of symbols used in PL, is not grammatical because it is not well-formed.
In proposition logic a fact such as 'Alison likes Falafel' would be represented as a simple atomic proposition, let's call it AP. we can use now the logical connectives $\wedge$, $\vee$, $\neg$, $\rightarrow$ to build more complex expressions. So if we had the proposition Q representing the fact 'Alison eats Falafel' we could have the facts:

$P \vee Q$: Alison likes Falafel or Alison eats Falafel.
$P \wedge Q$: Alison likes Falafel and Alison eats Falafel.
$\neg Q$: Alison doesn't eat Falafel.
$P \rightarrow Q$: If Alison likes Falafel then Alison eats Falafel.

A truth table is a complete list of the possible truth values of a sentence. We can determine the truth value of arbitrary sentences using truth tables which define the truth values of sentences with logical connectives. The truth tables provide a simple semantics for expressions in propositional logic. As sentences can only be true or false, truth tables are in this case very simple, for example:

X	Y	$X \rightarrow Y$
T	T	T
T	F	F
F	T	F
F	F	F

Formal Definition of PL Let's assume a set $\mathcal{P} = p, q, p_1, ...$ of (atomic) propositions which can be either true or false. For example the propositions *stack-is-empty* denotes the fact that the stack is really empty. We can now write the syntax of the propositional logic PL as following:

$$PL ::= \mathcal{P} | \perp | (PL \rightarrow PL) \qquad \text{that is,}$$

1. every $p \in \mathcal{P}$ is a well-formed formula of propositional logic,
2. $\perp$ is a well formed formula 'the falsum',
3. if φ and ψ are well-formed formula, then so is $(\varphi \rightarrow \psi)$, and
4. nothing else is a formula.

Interpretation An Interpretation $\mathcal{I}$ for the propositions is a function assigning a truth value from $\{true, false\}$ to every proposition. For example, the propositions *stack-is-empty* is interpreted differently on a farm, in a library, or in front of a computer terminal.

Propositional Model $\mathcal{M} \triangleq (\mathcal{U}, \mathcal{I})$ consists of the fixed binary domain $\mathcal{U} \triangleq \{true, false\}$ and an interpretation for $\mathcal{P}$.

Validation Relation $\models$ between a model $\mathcal{M}$ and a formula φ is defined by the following clauses.

1. $\mathcal{M} \models p$ iff $\mathcal{I}(p) = true$,
2. $\mathcal{M} \not\models \perp$, and
3. $\mathcal{M} \models (\varphi \rightarrow \psi)$ iff $\mathcal{M} \models \varphi$ implies $\mathcal{M} \models \psi$

That is, $\mathcal{M} \models (\varphi \rightarrow \psi)$ iff $\mathcal{M} \not\models \varphi$ or $\mathcal{M} \models \psi$, if $\mathcal{M} \models \varphi$. Then we say that $\mathcal{M}$ validates φ, or, equivalently, φ is valid in $\mathcal{M}$.

2.2 Temporal Logic

2.2.1 Introduction

Temporal logic is rooted in the field of exact philosophy and is a variant of modal logic. Modal logic deals with properties which are interpreted with respect to a set of possible worlds. The truth value of propositions depends on the respective world and basically two operators ”necessarily” and ”possibly” exist which denote that a proposition is true in all or some possible worlds. Even the ancient Greek philosophy schools of the Megarians, Stoics, and Peripatetic as well as Aristotle used some temporalized form of these modal operators. During the Middle Ages Arabian and European logicians resumed and refined the ancient approaches in order to discern different types of necessity and possibility. In modern times, the interest in symbolic logic grew during the first half of the 20th century, and - with some delay- new modal and temporal logic approaches occurred. First publications date back to the 1940's.

In particular, the logicians Prior, Rescher, Kripke, and Scott contributed to the development of modern temporal logic. Kripke presented a formal possible world semantics for modal logics. Prior proposed a temporal interpretation. An ordered set of possible worlds can correspond to a temporal sequence of states. In result, the two basic modal operators ”necessarily” and ”possibly” become the temporal quantifiers ”always” and ”eventually”. Based on the linearity of time additional operators like ”next” and ”until” as well as past operators were introduced. Rescher and Urquhart outlined the history and introduced several major systems of temporal logic in [RU71]. In 1974, Burstall proposed the application of temporal logic in computer science for the first time. Pnueli improved this approach in [Pnu77, MP92], which is regarded as the classic source of temporal logic based program specification and verification.

In temporal logic one can specify and verify how components, protocols, objects, modules, procedures and functions behave as time progresses. The specification is done with temporal logic statements that make assertions about properties and relationships in the past, present, and the future. In other words, temporal logic is a formalism for describing sequences of transition between states in a *reactive systems*; such systems whose role is to maintain an ongoing interaction with their environment rather than produce some final value upon termination. Typical examples are Air traffic control system, Programs controlling mechanical devices such as a train, a plane, or ongoing processes such as a nuclear reactor. In this section we introduce some formal definitions of temporal logic, compositional tree logic and linear tree logic. Some examples are also presented here to explain the concepts of temporal logic.

2.2.2 Concurrent Systems

A concurrent system consists of a set of components that execute together. Normally the components have some means of communicating with each other. The mode of execution and the mode of communication may differ from one system to another. We will consider two modes of execution: *Asynchronous execution*, in which only one component makes a step at a time, and *synchronous execution*, in which all of the components make a step at the same time.

We use interpreted first order formula to describe concurrent systems. Thus the predicate and function symbols that occur in such formulas will have a predefined meaning. Usually, this meaning will be clear from the context. Let $V = \{v_1, ..., v_n\}$ be the set of system variables. We assume that the variables in V range over a finite set D (sometimes called the *domain* or *universe* of the interpretation). A *valuation* for V is a function that associates a value in D with each variable v in V.

State of a concurrent system can be described by giving values for all of the elements in V. In other words, a state is just a valuation $s : V \rightarrow D$ for the set of variables in V. Given a valuation, we can write a formula that is true for exactly that valuation. For example, given $V = \{v_1, v_2, v_3\}$ and the valuation $< v \leftarrow 2, v_2 \leftarrow 3, v_3 \leftarrow 5 >$, we derive the formula $(v_1 = 2) \wedge (v_2 = 3) \wedge (v_3 = 5)$. In addition to representing sets of states, we must be able to represent sets of transitions between states. To do this, we extend the idea used above. This time we use a formula to represent a set of ordered pairs of states. We cannot do this using just a single copy of the system variables V, so we create a second set of variables V'. We think of the variables in V as *present state* variables and the variables in V' as *next state* variables. Each variable v in V has a corresponding next state variables in V', which are denote by v'. A valuation for the variables in V and V' can be viewed as designating an ordered pair of states or a transition, and we can represent sets of these valuations using formulas as above. We refer to a set of pairs of states as a *transition relation*. If R is a transition relation, then we write $R(V, V')$ to denote a formula that represents it. In this case, the AP will typically have the form $v = d$ where $v \in V$ and $d \in D$. A proposition $v = d$ will be true in a state s if $s(v) = d$. When v is a variable over the boolean domain {True, False}, it is not necessary to include both $v = True$ and $v = False$ in AP. We will write v to indicate that $s(v) = True$ and $\neg v$ to indicate that $s(v) = False$.

2.2.3 Kripke Structures

Let AP be a set of atomic propositions. A Kripke structure M over AP is a four tuple $M = (S, S_0, R, L)$ where

- S is a finite set of states.

- $S_0 \subseteq S$ is the set of initial states.
- $R \subseteq S \times S$ is a transition relation that must be total, that is, for every state $s \in S$ there is a state $s' \in S$ such that $R(s, s')$.
- $L : S \rightarrow 2^{AP}$ is a function that labels each state with the set of atomic propositions true in that state.

For example, consider a simplified model of ATM[1] of figure 2.1.

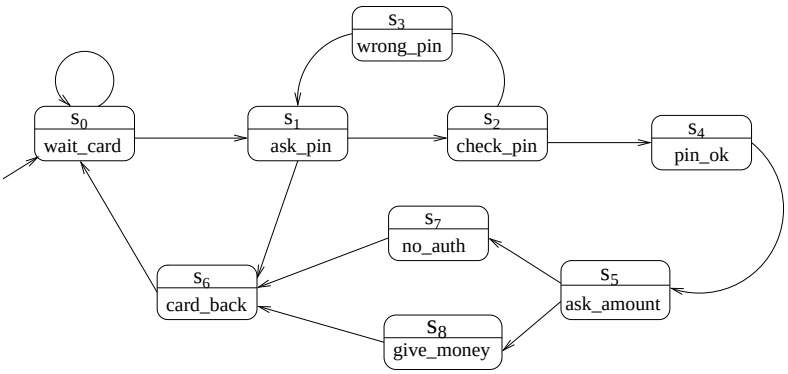


Figure 2.1: ATM model using Kripke Structure

Figure 2.1 represents the ATM as following:

- S is a finite set of states, where $S = \{s_0, s_1, s_2, ..., s_8\}$.
- $S_0 \subseteq S$ is the set of initial states, where $S_0 = \{s_0\}$.
- $R \subseteq S \times S$ is the transition relation, where:
  $R = \{(s_0, s_0),\ (s_0, s_1),\ (s_1, s_2),\ (s_2, s_3),\ (s_3, s_1),\ (s_2, s_4),\ (s_4, s_5),\ (s_5, s_7),\ (s_5, s_8),$
  $(s_7, s_6),\ (s_8, s_6),\ (s_6, s_0)\}$ and.
- $L : S \rightarrow 2^{AP}$, is a label function where $L(s_0) = \{wait\_card\}$, $L(s_1) = \{ask\_pin\}$, $L(s_2) = \{check\_pin\}$, $L(s_3) = \{wrong\_pin\}$, $L(s_4) = \{pin\_ok\}$, $L(s_5) = \{ask\_amout\}$, $L(s_6) = \{card\_back\}$, $L(s_7) = \{no\_auth\}$ and $L(s_8) = \{give\_money\}$.

The remainder of this section contains a precise description of the syntax and semantics of temporal logic. We define the semantics of temporal logic with respect to a Kripke structure. Recall that a Kripke structure M is a triple (S, R, L), where S is the set of states; $R \subseteq S \times S$

[1] An Automatic Teller Machine "ATM" is a computerized telecommunications device that provides a financial institution's customers a method of financial transactions in a public space without the need for a human support.

is a transition relation, which must be total (i.e., for all states $s \in S$ there exists a state $s' \in S$ such that $(s, s') \in R$); and $L : S \rightarrow 2^{AP}$ is a function that labels each state with a set of atomic proposition true in that state.

A **path** in M is an infinite sequence of states $\sigma = s_0, s_1, ...$ such that for every $i \geq 0$, (s_i, s_{i+1}) $\in R$. (Alternative, we can think of a path as an infinite branch in the computation tree that corresponds to the Kripke structure.). We use σ_i to denote the *suffix* of σ starting at s_i.

There are two types of formulas in temporal logic: **State Formulas** (which are true in a specific state) and **Path Formulas** (which are true along a specific path). If f is a state formula, the notation $M, s \vDash f$ means that f holds at state s in the Kripke structure M. Similarly, if f is a path formula, $M, \sigma \vDash f$ means that f holds along path σ in the Kripke structure M

Boolean Operators For a negation $\neg p$, we define

$$(\sigma, j) \vDash \neg p \text{ iff } (\sigma, j) \nvDash p$$

Thus, $\neg p$ holds at position j iff p does not.
For a disjunction $p \vee q$ we define

$$(\sigma, j) \vDash p \vee q \text{ iff } (\sigma, j) \vDash p \vee (\sigma, j) \vDash q$$

Thus, $p \vee q$ holds at position j iff either p or q does.
For illustration, we display the evaluation of some boolean combinations of formulas. For example, in computer networking, the two-phase commit protocol is a distributed algorithm that lets all nodes in a distributed system agree to commit a transaction. The protocol results in either all nodes committing the transaction or aborting it.

σ	s_0	s_1	s_2	s_3
co	A	C	A	A
pa	C	A	C	A
pa = co	F	F	F	F
$\neg\ (pa = co)$	T	T	T	T
$\neg\ (pa = co) \vee (pa = A)$	T	T	T	T

We mean by the symbol 'co' the coordinator, whose sends and receives the transaction from the participant and 'pa' the participant whose also sends or receives the transactions from the coordinator. We mean by 'A' the Abort result and 'C' is the Commit result of the protocol, where 'T' and 'F' are denote to the true and false values.

2.2.4 Linear Temporal Logic LTL

Linear temporal logic is a temporal logic with modalities referring to time. In LTL, one can encode formulae about the future of paths such as that a condition will eventually be true, that a condition will be true until another fact becomes true, etc. Linear temporal logic built up from a set of proposition $p_1, p_2, ...$, the usual logic connectives $\neg$,$\vee$,$\wedge$,$\rightarrow$ and the following temporal operators:

- **The Next Operator $\bigcirc$ or X** If p is a temporal formula, then so is $X\ p$, read as *next* p. Its semantic defined by:

$$(\sigma, j) \vDash Xp \text{ iff } (\sigma, j+1) \vDash p$$

  Thus, $X\ p$ holds at position j iff p holds at the next position $j+1$. The following table illustrate the evaluation of the formula $(pa = A) \wedge X\ (pa = C)$, which holds for all positions s_j such that $pa = A$ at s_j and $pa = C$ at s_{j+1}.

σ	s_0	s_1	s_2	s_3	s_4	s_5	s_6
pa	A	C	A	A	C	A	C
$pa = A$	T	F	T	T	F	T	F
$pa = C$	F	T	F	F	T	F	T
$X\ (pa = C)$	T	F	F	T	F	T	F
$(pa = A) \wedge X\ (pa = C)$	T	F	F	T	F	T	F

- **The Always Operator $\square$ or G** If p is a temporal formula, then so is Gp , read *always* p. Its semantics is defined by:

$$(\sigma, j) \vDash Gp \text{ iff } (\sigma, k) \vDash p \text{ for all } k \geq j$$

  Thus, $G\ p$ holds at position j iff p holds at position j and all following positions (from now on). For example, this table illustrates the formula $G(pa = A)$.

σ	s_0	s_1	s_2	s_3	s_4	s_5	s_6
pa	A	C	A	A	C	A	A
$pa = A$	T	F	T	T	F	T	T
$G\ (pa = A)$	F	F	F	F	F	T	T

  if $G\ p$ holds at j, then it also holds at any $k \geq j$.

- **The Eventually Operator $\Diamond$ or F** If p is a temporal formula, then so is $F\ p$, read as *eventually* p. Its semantics is defined by

$$(\sigma, j) \vDash Fp \text{ iff } (\sigma, k) \vDash p \text{ for some } k \geq j$$

Thus, $F\ p$ holds at position j iff p holds at some position $k \geq j$. As an example we illustrate the evaluation of the formula $F\ (co = C)$.

σ	s_0	s_1	s_2	s_3	s_4	s_5	s_6
co	A	C	A	A	C	A	C
$co = C$	F	T	F	F	T	F	T
$F\ (co = C)$	F	T	F	F	T	F	T

The eventually operator is dual to the always operator. This means that $F\ p$ holds at a position j iff $G\ \neg p$ does not hold there.
if $F\ p$ holds at position j, then it also holds at any k, $0 \leq k \leq j$.

- **The Until Operator** U The formula $F\ q$ predicts the eventual occurrence of q and the formula $G\ p$ states that p will hold continuously from now on. The *until* formula $p\ U\ q$ (read p until q) combines these statements by predicting the eventual occurrence of q and stating that p holds continuously at least until the (first) occurrence of q. Formally, we define

$$(\sigma, j) \vDash p\ U\ q \text{ iff there exists a } k \geq j, \text{ such that } (\sigma, k) \vDash q, \text{ and for every } i, j \leq i < k, (\sigma, i) \vDash p$$

The following table evaluate the formula $(pa = A)\ \mathcal{U}\ (pa = C)$.

σ	s_0	s_1	s_2	s_3	s_4	s_5	s_6
pa	A	C	A	A	C	A	C
$pa = A$	T	F	T	T	F	T	F
$pa = C$	F	T	F	F	T	F	T
$(pa = A)\ U\ (pa = C)$	T	T	T	T	T	T	T

Note that if position j satisfies formula q, it also satisfies $p\ U\ q$ for any p With $k = j$ in the definition, the requirement that p holds at all positions i, such that $j \leq i < k = j$, is fulfilled vacuously.
Note also that if $p\ U\ q$ holds at position j, then $F\ q$ also holds there.

- **The Unless (or Waiting-for) Operator** R The until formula $p\,\mathcal{U}\,q$ guarantees that q will eventually occur. In some cases we need a weaker property, which states that p holds continuously either until the next occurrence of q or throughout the sequence.
  This is expressed by the formula $p\,\mathcal{W}\,q$, read p *unless* q (also p waiting for q). Using the previously defined operators, its formal definition is given by

  $$(\sigma, j) \vDash p\ R\ q \text{ iff } (\sigma, j) \vDash p\ U\ q \text{ or } (\sigma, j) \vDash Gp$$

  We illustrate the evaluation of the formula $[(pa = A) \vee (pa = C)]\ R\ (co = C)$ in the following table:

σ	s_0	s_1	s_2	s_3	s_4	s_5	s_6
pa	A	C	A	A	C	A	C
$pa = A$	T	F	T	T	F	T	F
$pa = C$	F	T	F	F	T	F	T
$(pa = A) \vee (pa = C)$	T	T	T	T	T	T	T
$[(pa = A) \vee (pa = C)\ R\ (co = C)]$	T	T	T	T	T	T	T

LTL Formula An LTL formula can be evaluated over a sequence of truth evaluations and a position on the given path. An LTL formula is satisfied by a path if and only if it is satisfied for some position on that path. The semantics for the above operators is given in figure 2.2.

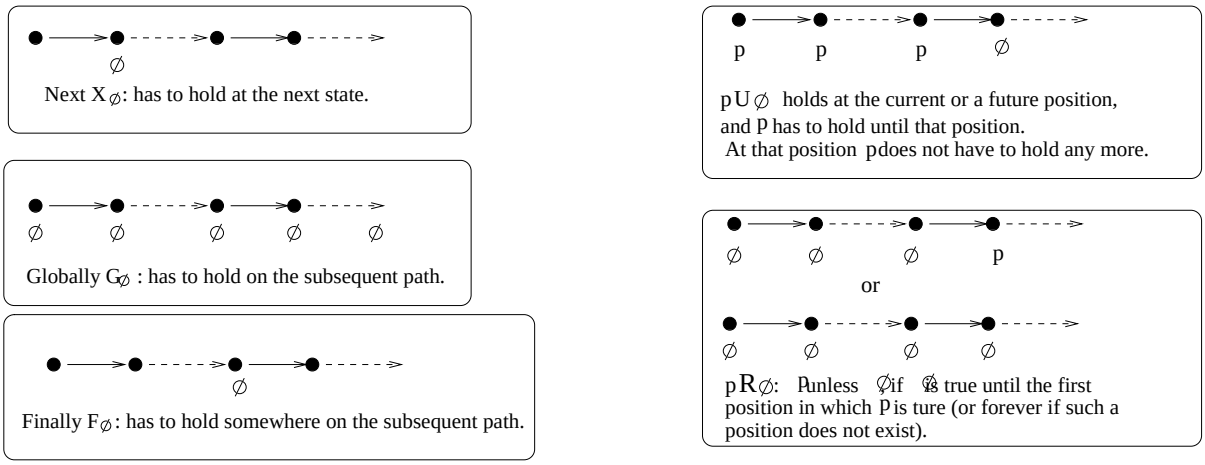


Figure 2.2: Linear Temporal Logic Semantics

Some Examples

Following are several frequently used formulas and their interpretations. We mean by $\sigma \vDash \phi$ For the considered formula ϕ, that ϕ holds at position 0 of σ. We assume that the subformulas pa and co appearing below are state formulas:

1. $co = A \rightarrow Fpa = A$

   This formula states that if the model σ satisfies that the coordinator has the Abort value in the position s_0, in this case the participant also satisfies the subformula $Fpa = A$ which means that the participant has the value Abort for some position $j \geq 0$, pa holds at s_j. In other words; initially co then eventually pa.

2. $G(co = A \rightarrow Fpa = A)$

   This formula states the property; if $co = A$ holds at position j then $pa = A$ holds at some position not smaller than j for $j = 0$. Adding the always operator in front of the formula states that this property holds for all positions $j \geq 0$. In other words; every co position coincides with or is followed by a pa position.

3. $GFco = C$

   This Formula obtain a property that every position in the sequence coincides with or is followed by a later position satisfying $co = C$. In other words; The sequence σ contains infinitely many $co = C$ positions.

4. $FGco = C$

   There exists a position $j \geq 0$ that satisfies $Gco = C$, i.e., there exists a position such that $co = C$ holds at all later position. In other words; eventually permanently $co = C$, or equivalently; the sequence σ contains only finitely many $\neg\, co = C$ positions.

5. $(\neg co = C)\ R\ pa = A$

   The formula states that either $\neg co = C$ holds forever or that it holds until an occurrence of $pa = A$. This means; the first co position must coincide with or be preceded by a pa position.

6. $G(pa = A \rightarrow Xpa = A)$

   The formula states that the subformula $pa = A \rightarrow Xpa = A$ holds at all positions. The subformula holds at position i either if pa is false there or if pa is true both at i and at $i+1$. In other words The successor of every $pa = A$ state is another $pa = A$ state.

7. $G(pa = A \rightarrow Gpa = A)$

   The formula states that if $pa = A$ holds at position i, then it also holds at every position $j \geq i$. Therefore, it expresses the following property Once $pa = A$, always $pa = A$.

8. $G\ \exists u : ((x = u) \wedge X(x = u + 1))$

The formula refers to a rigid variable u and a flexible variable x. It states that at every position j, there exists a value of u such that, at position j, x equals u and, at the next position $j+1$, x equals $u+1$. It follows that $s_{j+1}\{x\} = s_j\{x\} + 1$. This is a way to specify a sequence in which x increase by 1 from each state to the next.

2.2.5 Computation Tree Logic CTL

Computational tree logic [BAMP83,SKM01,CE81] (CTL) is a branching-time logic, meaning that its model of time is a tree-like structure in which the future is not determined; there are different paths in the future, any one of which might be 'actual' path that is realised. In CTL the temporal operators X, F, G, U, and R must be immediately preceded by a path quantifier;

If f and g are state formulas, then Xf, Ff, Gf, fUg, and fRg are path formulas.

Conceptually, CTL formulas describe properties of *computation trees*. The tree is formed by designating a state in Kripke structure as the *initial state* and then unwinding the structure into an infinite tree with the designated state at the root, as shown in figure 2.3, the computation tree shows all of the possible executions starting from the initial state.

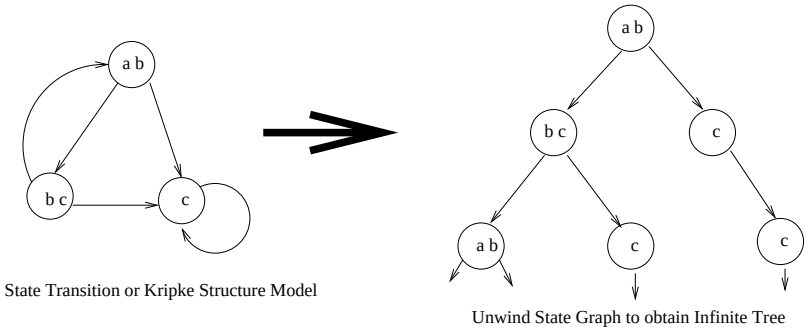


Figure 2.3: Computation Trees

CTL Syntax Each CTL operator is a pair of symbols. The first one is either A ("for All paths"), or E ("there Exists a path"). The second one is one of X ("neXt state"), F ("in a Future state"), G ("Globally in the future") or U ("Until"), where U is a binary operator, it could be written $EU(p,q)$ or $AU(p,q)$.

Example: $AG(p \rightarrow (EFq))$ is read as "It is Globally the case that, if p is true, then there *Exists* a path such that at some point in the *Future* q is true".

CTL Semantics Let $M = (S, R, L)$ be a kripke structure. Let φ be a CTL formula and $s \in S$. $M, s \vDash \varphi$ is defined inductively on the structure of φ, as follows:

$M, s \vDash p$	iff	$p \in L(s)$.
$M, s \vDash \neg\varphi$	iff	$M, s \nvDash \varphi$.
$M, s \vDash \varphi \wedge \psi$	iff	$M, s \vDash \psi$ and $M, s \vDash \psi$.
$M, s \vDash \varphi \vee \psi$	iff	$M, s \vDash \varphi$ or $M, s \vDash \psi$.
$M, s \vDash AX\varphi$	iff	$\forall \acute{s} \in S$ and $sR\acute{s}$ then $M, \acute{s} \vDash \varphi$.
$M, s \vDash EX\varphi$	iff	$\exists \acute{s} \in S$ where $sR\acute{s}$ then $M, \acute{s} \vDash \varphi$.
$M, s \vDash AG\varphi$	iff	for all paths $(s_1, s_2, s_3, s_4, ...)$ where $s_i \in S$ and $s_i R s_{i+1}$ for all i, it is the case that $M, s_i \vDash \varphi$.
$M, s \vDash EG\varphi$	iff	there is a path $(s_1, s_2, s_3, s_4, ...)$ where $s_i \in S,\ s_i R s_{i+1}$and for all i, it is the case that $M, s_i \vDash \varphi$.
$M, s \vDash AF\varphi$	iff	for all paths $(s_1, s_2, s_3, s_4, ...)$ $s_i \in S,\ s_i R s_{i+1}$, there is a state s_i where $s \in S, M, s_i \vDash \varphi$.
$M, s \vDash EF\varphi$	iff	there is a path $(s_1, s_2, s_3, s_4, ...)$ where $s_i \in S,\ s_i R s_{i+1}$, and there is a state $s_i \in S$, and $M, s_i \vDash \varphi$.
$M, s \vDash A[\varphi U \psi]$	iff	for all paths $(s_1, s_2, s_3, s_4, ...)$ where $s_i R s_{i+1}$, $s_i \in S$ and $s_i \vDash \psi$, there is a state $s_j \in S$ where $s_j \vDash \varphi$, for all $i > j$.
$M, s \vDash E[\varphi U \psi]$	iff	there exists a path $(s_1, s_2, s_3, s_4, ...)$ where $s_i R s_{i+1}$, $s_i \in S$ and $s_i \vDash \psi$, there is a state $s_j \in S$ where $s_j \vDash \varphi$, for all $i > j$.

Figure 2.4 illustrates some of these operators.

Some Examples

- $AG(EF(pa = C \wedge co = C))$ This formula means that from any state it is possible to get to the state where the participant and the coordinator are in the commit situation.

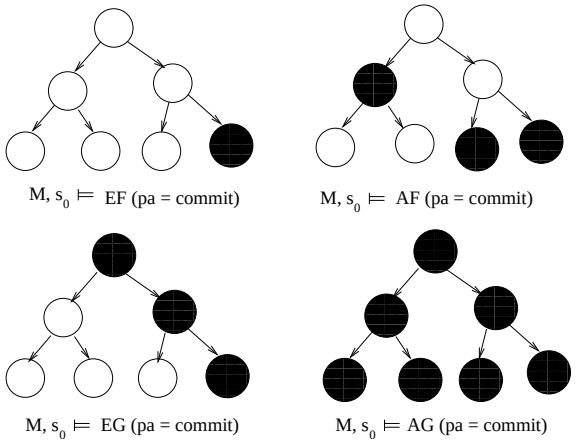


Figure 2.4: Some CTL operators

- $EF(Start \wedge \neg Ready)$: It is possible to get to a state where *Start* holds but *Ready* does not hold.
- $AG(Reg \rightarrow AFAck)$: If a request occurs, then it will be eventually acknowledged.
- $AG(AFcard\_back)$: The proposition that the card will be back in the ATM machine holds infinitely often on every computation path.
- $AG(EFRestart)$: From any state it is possible to get to the *Restart* state.

2.2.6 The Computation Tree Logic CTL^*

CTL^* is an extension of CTL as it allows path quantifiers E and A to be arbitrarily nested with linear temporal operators (such as X and U).

In other words, CTL^* formulas are composed of *path quantifiers* and *temporal operators*. The path quantifiers are used to describe the branching structure in the computation tree. There are two such quantifiers A (for all computation paths) and E (for some computation path). These quantifiers are used in particular state to specify that all of the paths or some of the paths starting at that state have some property. We can say that the CTL^* formula is a logic that combines the expressive power of LTL and CTL.

The difference between CTL and CTL^* is that the CTL^* allows more complex formulas inside path quantifiers such as Boolean combinations and nesting of temporal operators. For example, the following are CTL^* formulas, but not CTL formulas: $E(F_p \wedge G_q)$, $A(FG_p)$, $EA(G_q)$.

Also LTL can be considered as a sublanguage of CTL^* by transforming every LTL formula ϕ to $A\phi$, there are things we might want to say using LTL that we cannot say in CTL:

FG_p: Along every path from the initial state S there is a state from which p will hold forever.

GFp: The fairness constraint in LTL.

On the other hand, there are some formulas can't be expressed in LTL:

$AG(EF_p)$: The reset formulas.

There are also formulas cannot be expressed neither in CTL nor in LTL:

$E(GF_p)$: There is a path with infinitely many p.

2.2.7 Fairness

It is often necessary to provide the reactive systems with some of fairness constraints. For example, if the system allocates a shared resource among several users, only those paths along which no user keeps the resource forever should be considered. Also in our case study "two-phase commit protocol", if we want to consider the property that no message is transmitted from the coordinator but never received to the participants, we have to add the fairness constraint.

Two very commonly used forms of fairness are weak fairness and strong fairness:

- Weak fairness can captured by using the LTL formula:

$$\bigwedge_{1 \leq i \leq n} (FG\, p_i \rightarrow GFq_i)$$

  It means, if an event is continuously enabled it will occur infinitely often. For example in the ATM protocol, if the ATM $card\_Entry$ is true then it is always eventually $card\_back$ is ture.

- The strong fair is characterized by the LTL formula:

$$\bigwedge_{1 \leq i \leq n} (GF\, p_i \rightarrow GFq_i)$$

  It means, if an event is infinitely often enabled it will occur infinitely often. For example in the two-phase commit protocol, if all participants are commit infinitely often, coordinator will eventually commit, and hence commit infinitely often.

Strong fairness is less often used and is stronger than (implies) weak fairness.

2.3 Model Checking

Model checking is a method for formally verifying finite-state concurrent systems such as circuit designs and communication protocols. System specifications are expressed as temporal logic formulas, and efficient symbolic algorithms are used to traverse the model defined by the system and check if the specification holds or not. Large state space can often be traversed in minutes. In this section we will illustrate the basic concepts of software model checking and it's useful algorithms in detecting the error like CTL and Binary Decision Diagram BDD. Finally some tools are represented to implements our case study and see that the required properties are satisfied or not.

2.3.1 Software Model Checking

Verifying a program is proving, in a formal mathematical way, that the program satisfies a specification written in a logical language. Therefore verification has often been claimed to be a promising approach for ensuring the correctness of software. Formal verification, where a system is verified with respect to desired behavior, has nowadays become popular in industry, especially in mission and safety critical applications. Specifically model checking methods, which can be fully automated, are being used extensively to verify that a finite state system meets a desired behavior. The desired behavior is often specified by a temporal logic formula. Software model checking is typically applied to hardware systems, concurrent protocols, process control systems, and more generally as reactive systems [MQR95].

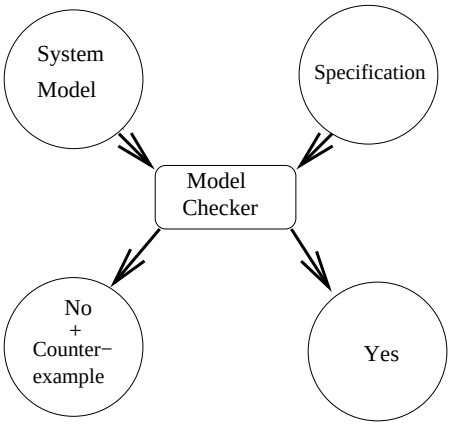


Figure 2.5: Model Checking Strategy

A model-checking tool accepts a design (called system model) and a property (called specification) that the final system is expected to satisfy. The model checker explores all paths through

the state space in order to check whether the specification holds for the model. The tool then output yes if the given model satisfies the given specification and generates a counter example otherwise (see figure 2.5).

The counter example details why the model doesn't satisfy the specification. By studying the counter example, we can pinpoint the source error in the model, correct the model, and try again. The idea of this iterative process is to ensure that the model satisfies enough system properties. There are several model checking tools such as SPIN [Hol04b], FDR [(Lt], UPPAAL [BLPY97], and NuSMV [Nus02]. SPIN is one of the most popular software tool that can be used for the formal verification of distributed software systems [Hol04b]. SPIN uses a high-level language to specify system descriptions, called PROMELA. SPIN has also been used to trace logical design errors in distributed systems design, such as operating systems, data communications protocols etc. The tool checks the logical consistency of a specification. It reports on deadlocks, unspecified receptions, flags incompleteness, race conditions, and unwarranted assumptions about the relative speeds of processes. SPIN and the model checkers mentioned above do not accept software design directly, but use logic and automata formalisms for describing models and specifications.

HUGO is another tool for software model checking [KM11]. HUGO is based on ArgoUML [arg], it validates software designs by translating UML model to different model checkers and theorem provers and by reflecting the results of these systems back to UML. More specifically, HUGO connects UML state diagrams and OCL with the model checkers SPIN, UPPAAL and the system language of the interactive theorem prover KIV.

The model checking problem is formally easy to describe. Suppose we have a Kripke structure M = (S, R, L) that represents a finite-state concurrent system and a temporal logic formula f expressing some desired specification, the model checking means to make all states in S satisfy the formula f:

$$\{\, s \in S \mid M,\ s \vDash f \,\}.$$

Normally, some states of the concurrent system are designed as initial states. The system satisfies the specification provided that all of the initial states are in the set S also.

The first algorithms for solving the model checking problem used an explicit representation of the Kripke structure as a labeled, directed graph with arcs given by pointers. In this case, the nodes represent the states in S, the arcs in the graph give the transition relation R, and the labels associated with the nodes describe the function $L : S \to 2^{AP}$.

2.3.2 CTL Model Checking

Let $M = (S, R, L)$ be a Kripke structure. Assume that we want to determine which states in S satisfy the CTL formula f. The algorithm will operate by labeling each state s with the set $label(s)$ of subformulas of f which are true in s. Initially, $label(s)$ is just $L(s)$. Recall that any CTL formula can be expressed in terms of $\neg, \wedge, EX, EU$ and EG. Thus, it is sufficient to be able to handle six cases, depending on whether g is atomic or has one of the following forms: $\neg f_1$, $f_1 \wedge f_2$, EXf_1, $E[f_1 U f_2]$, or EGf_1.

We could write the following algorithm for formula has the form $(\neg f)$:

procedure *CheckNegation* (f)
for all $s \in \{s | f \notin label(s)\}$
do label(s) := label(s) $\cup \{\neg f\}$
end procedure

For $f_1 \vee f_2$, we do as following:

procedure *CheckConjunction* (f_1, f_2)
for all $s \in \{s | f_1 \in label(s) \ \wedge f_2 \in label(s)\}$
do label(s) := label(s) $\cup \{f_1 \wedge f_2\}$
end procedure

To handle formula of the form $g = E[f_1 U f_2]$ we first find all states that are labeled with f_2. We then work backwards using the converse of the transition relation R and find all states that can be reached by a path in which each state is labeled with f_1. All such states should be labeled with g.

In the following Algorithm, we give a procedure CheckEU that add $E[f_1 U f_2]$ to $label(s)$ for every s that satisfies $E[f_1 U f_2]$, assuming that f_1 and f_2 have already been processed correctly, that is, for every state s, $f_1 \in label(s)$ iff $s \vDash f_1$ and $f_2 \in label(s)$ iff $s \vDash f_2$. This procedure requires time $O(|S| + |R|)$.

Procedure *CheckEU* (f_1, f_2)
$T := \{s | f_2 \in label(s)\}$;
for all $s \in T$ do $label(s) := label(s) \cup \{E[f_1 U f_2]\}$;
While $T \neq \phi$ do
choose $s \in T$;
$T := T \setminus \{s\}$;
for all t such that $R(t, s)$ do
if $E[f_1 U f_2] \notin label(t)$ and $f_1 \in label(t)$ then
$label(t) := label(t) \cup \{E[f_1 U f_2]\}$;
$T := T \cup \{t\}$;
end if;
end for all;
end while;
end procedure

The case in which $g = EGf_1$ is slightly more complicated. It is based on the decomposition of the graph into nontrivial strongly connected components. *A strongly connected component (SCC)* C is maximal subgraph such that every node in C is reachable from every other node in C along a directed path entirely contained within C. C is nontrivial iff either it has more than one node or it contains one node with a self-loop.

Let M' be obtained from M by deleting from S all of those states at which f_1 does not hold and restricting R and L accordingly. Thus $M' = (S', R', L')$ where $S' = s \in S | M, s \vDash f_1$, $R' = R\|_{S' \times S'}$, and $L' = L\|'_S$. Note that R' may not be total in this case. The states with no outgoing transitions may be eliminated, but this is not essential for the correctness of our algorithm. The algorithm depends on the following observation.

LEMMA 1 $M, s \vDash EGf_1$ iff the following two conditions are satisfied:

1. $s \in S'$
2. There exists a path M' that leads from s to some node t in nontrivial strongly connected component C of the graph (S', R').

Proof Assume that $M, s \vDash EGf_1$. Clearly $s \in S'$. Let σ be an infinite path starting at s such that f_1 holds at each state on σ. Since M is finite, it must be possible to write σ as $\sigma = \sigma_0 \sigma_1$ where σ_0 is a finite initial segment and σ_1 is an infinite suffix of σ with the property that each state on σ_1 occurs infinitely often. Then, σ_0 is contained in S'. Let C be the set of states in σ_1. Clearly, C is contained in S'. We now show that there is a path within C between any pair of

states in C. Let s_1 and s_2 be states in C. Pick some instance of s_1 on σ_1. By the way in which σ_1 was selected, we know that there is an instance of s_2 further along σ_1. The segment from s_1 to s_2 lies entirely within C. This segment is a finite path from s_1 to s_2 in C. Thus, either C is strongly connected component or it is contained within one. In either case, both conditions (1) and (2) are satisfied.

Procedure *CheckEG* (f_1)
$S' := \{s | f_1 \in label(s)\}$;
$SCC := \{C | C$ is a nontrivial SCC of $S'\}$;
$T := \bigcup_{c \in SCC} \{s | s \in C\}$;
for all $s \in T$ do $label(s) := label(s) \cup \{EGf_1\}$;
while $T \neq \phi$ do
choose $s \in T$;
$T := T \setminus \{s\}$;
for all t such that $t \in S'$ and $R(t, s)$ do
if $EGf_1 \notin label(t)$ then
$label(t) := label(t) \cup EGf_1$;
$T := T \cup t$;
end if;
end for all;
end while;
end procedure

We will illustrate the model checking algorithm for CTL on a small example that describes the behavior of a microwave oven. Figure 2.6 gives the Kripke structure for the oven. For clarity, each state is labeled with both the atomic propositions that are true in the state and the negations of the propositions that are false in the state. The labels on the arcs indicate the actions that cause transitions and are not part of the Kripke structure.
We check the CTL formula $AG(Start \rightarrow AF\ Heat)$ which is equivalent to the formula $\neg EF(Start \wedge EG\ \neg Heat)$ (here, we use EFf as an abbreviation for $E[true\ U\ f]$). We start by computing the set of states that satisfy the atomic formulas and proceed to more complicated subformulas. Let $S(g)$ denote the set of all states labeled by the subformula g. Note that, with a suitable data structure, the computation of $S(p)$ for all $p \in AP$ requires time $O(|S| + |R|)$.
S(Start) = { 2, 5, 6, 7 }.

$S(\neg Heat)$ = { 1, 2, 3, 5, 6 }.

In order to compute $S(EG\neg Heat)$ we first find the set of nontrivial strongly connected component in $S' = S(\neg Heat)$. SCC = $\{\{1, 2, 3, 5\}\}$. we proceed by setting T, the set of all states

that should be labeled by $EG\neg$ Heat to be the union over the elements of SCC, that is, initially $T = \{1, 2, 3, 5\}$. Note other state in S' can reach a state in T along a path in S'. Thus, the computation terminates with

$S(EG\neg Heat)$ = $\{1, 2, 3, 5\}$

Next we compute

$S(Start \wedge EG\ \neg Heat)$ = $\{2, 5\}$.

When computing $S(EF(Start \wedge EG\ \neg Heat))$, we start by setting $T = S(Start \wedge EG\neg Heat)$. Next, we use the converse of the transition relation to label all states in which the formula holds. We get:

$S(EF(Start \wedge EG\neg Heat))$ = $\{1, 2, 3, 4, 5, 6, 7\}$.

Finally, we compute that

$S(\neg EF(Start \wedge EG\neg Heat)) = \phi$

Since the initial state 1 is not contained in this set, we conclude that the system described by the Kripke structure does not satisfy the given specification.

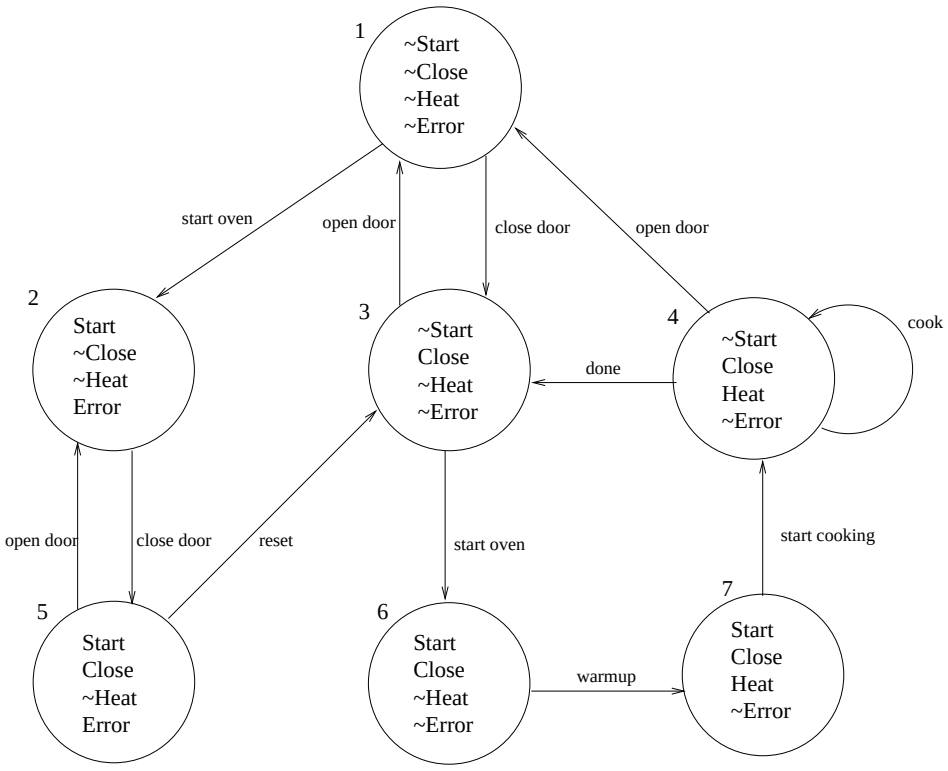


Figure 2.6: Microwave oven example

2.3.3 Symbolic Model Checking

In this section we describe how to represent finite state systems symbolically using binary decision diagrams. first of all we discuss how binary decision diagrams can be used to represent boolean functions. The boolean functions are defined over 0 and 1 where 0 represents *False* and 1 represents *True*. We show that the size of the binary decision diagrams depends strongly on the ordering that is selected for the variables and briefly discuss some heuristics that can be used for selecting good orderings. We also describe how various logical operations can be efficiently implemented using this representation.

2.3.4 Binary Decision Diagram

The Binary Decision Diagrams are widely used for a variety of applications in computer aided design, including symbolic simulation and verification of finite-state concurrent systems. Let's discuss the binary decision tree. A binary decision tree is rooted, directed tree that consists of two types of vertices, *terminal vertices* and *nonterminal vertices*. Each nonterminal vertex v is labeled by a variable var(v) and has two successors: $low(v)$ corresponding to the case where the variable v is assigned 0, and $high(v)$ corresponding to the case where v is assigned 1. Each terminal vertex v is labeled by $value(v)$ which is either 0 or 1.

A binary decision tree for the two-bit comparator, given by the formula $f(co, pa_1, pa_2, pa_3)$ = $(co \leftrightarrow pa_1) \wedge (co \leftrightarrow pa_2) \wedge (co \leftrightarrow pa_3)$, is shown in figure 2.7. One can decide whether a particular truth assignment to the variables make the formula true or not by traversing the tree from the root to a terminal vertex. If the variable v is assigned 0, then the next vertex on the path from the root to a terminal vertex will be $low(v)$. If v is assigned 1 then the next vertex in the path will be $high(v)$. The value that labels the terminal vertex will be the true of the function for this assignment. For example, the assignment $< co := 1, pa_1 := 0, pa_2 := 1, pa_3 := 1 >$ leads to a leaf vertex labeled 0; hence, the formula is false for this assignment.

Binary decision trees are essentially the same size as truth tables. Fortunately, there is usually a lot of redundancy in such trees. For example, in the tree of figure 2.7 there are eight subtrees with roots labeled by participant 3 (pa_3), but only two are distinct. Thus, we can obtain a more concise representation for the boolean function by merging isomorphic subtrees. This results in a directed acyclic graph (DAG) called a *binary decision diagram*: More precisely, a binary decision diagram is a rooted, directed acyclic graph with two types of vertices, terminal vertices and nonterminal vertices. As in the case of binary decision trees, each nonterminal vertex v is labeled by a variable var(v) and has two successors, low(v) and high(v). Each terminal vertex is labeled by either 0 or 1. Every binary decision B with root v determines a boolean function $f_v(x_1, ..., x_n)$ in the following manner:

co	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1
pa1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1
pa2	0	0	1	1	0	0	1	1	0	1	1	1	0	0	1
pa3	0	1	0	1	0	1	0	1	0	0	0	1	0	1	1
f	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1

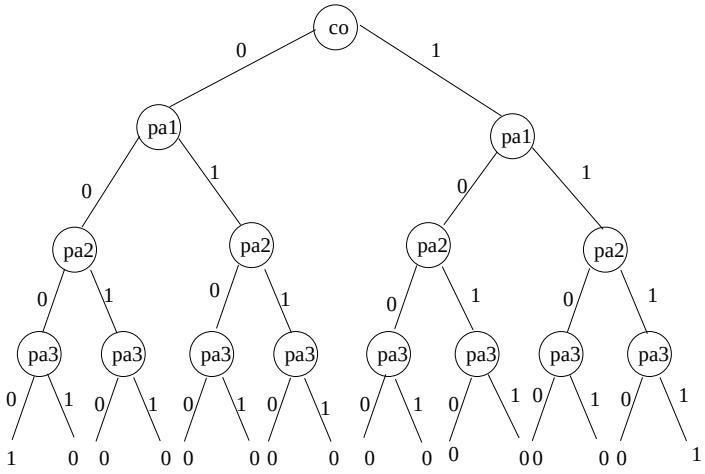


Figure 2.7: Truth Table and Binary Decision Tree for the Formula f

1. if v is a terminal vertex:

   If $value(v) = 1$ then $f_v(x_1, ..., x_n) = 1$.

   If $value(v) = 0$ then $f_v(x_1, ..., x_n) = 0$.

2. If v is a nonterminal vertex with $var(v) = x_i$ then f_v is the function

   $f_v(x_1, ..., x_n) = (\neg x_i \wedge f_{low(v)}(x_1, ..., x_n)) \vee (x_i \wedge f_{high(v)}(x_1, ..., x_n))$

Two binary decision diagrams are morphic if there exists a one-to-one and onto function h that maps terminals of one to terminals of the other and nonterminals of one to nonterminals of the other, such that for every terminal vertex v, $value(v) = value(h(v))$ and for every nonterminal vertex v, $var(v) = var(h(v))$, $h(low(v)) = low(h(v))$, and $h(high(v)) = high(h(v))$.

Bryant [Bry] showed how to obtain a canonical representation for boolean functions by placing two restrictions on binary diagrams. First, the variables should appear in the same order along each path from the root to a terminal. Second, there should be no isomorphic subtrees or redundant vertices in the diagram. The first requirement is archived by imposing a total ordering $<$ on the variables that label the vertices in the binary decision diagram and requiring that for any vertex u in the diagram, if u has a nonterminal successor v, then $var(u) < var(v)$. The second requirement is achieved by repeatedly applying three transformation rules that do not alter the function represented by the diagram:

1. Remove duplicate terminals Eliminate all but one terminal vertex with a given label and redirect all arcs to the eliminated vertices to the remaining one.

2. Remove duplicate nonterminals If two nonterminals u and v have var(u) = var(v), low(u) = low(v) and high(u) = high(v), then eliminate u or v and redirect all incoming arcs to the other vertex.

3. Remove redundant tests If nonterminal v has low(v) = high(v), then eliminate v and redirect all incoming arcs to low(v).

Starting with a binary decision diagram satisfying the ordering property, the canonical form is obtained by applying the transformation rules until the size of the diagram can no longer be reduced. Bryant shows how this can be done in a bottom-up manner by a procedure called *Reduce* in time which is linear in the size of the original binary decision diagram [Bry]. The term *ordered binary decision diagram* (OBDD) will be used to refer to the graph obtained in this manner. For example, if we use the ordering $co < pa_1 < pa_2 < pa_2$ for the two-bit comparator function, we obtain from the OBDD shown in figure 2.8.

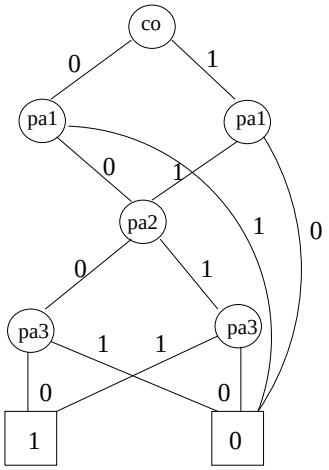


Figure 2.8: OBDD for two-bit comparator

We next explain how to implement various important logical operations using $OBDD_s$. We begin with the function that *restricts* some argument x_i of the boolean function f to a constant value b. This function is denoted by $f|_{x_i \to b}$ and satisfies the identity

$$f|_{x_i \to b}(x_1, ..., x_n) = f(x_1, ..., x_{i-1}, b, x_{i+1}, ..., x_n).$$

If f is represented as an OBDD, then the OBDD for the restriction $f|_{x_i \to b}$ can be easily computed by a depth-first traversal of the OBDD. For any vertex v which has a pointer to a vertex w such that $var(w) = x_i$, we replace the pointer by $low(w)$ if b is 0 and by $high(w)$ if b is 1. The boolean operations can be implemented uniformly by using the **Shannon expansion**

$$f = (\neg x \wedge f|_{x\to 0}) \vee (x \wedge f|_x \to 1).$$

Bryant [Bry] gives a uniform algorithm called *Apply* for computing all boolean operations. Below we briefly explain how *Apply* works. Let * be an arbitrary two-argument boolean operation, e.g.,$*(x, y) = x \wedge y$, or $*(x, y) = x \to y$, or $*(x, y) = x$ and let f and f' be two boolean functions represented by OBDD nodes v and v', where $x = var(v)$, $x' = var(v')$. To simplify the explanation of the algorithm we introduce the following cases:

1. If v and v' are terminal, then $f * f' = value(v) * value(v')$.
2. If $x = x'$, then
   $f * f' = (\neg x \wedge (f|_{x\to 0} * f'|_{x\to 0})) \vee (x \wedge (f|_{x\to 1} * f'|_{x\to 1}))$
3. If $x < x'$ in the ordering of variables, then $f'|_{x\to 0} = f'|_{x\to 1} = f'$ since f' does not depend on x. thus $f * f' = (\neg x \wedge (f|_{x\to 0} * f')) \vee (x \wedge (f|_{x\to 1} * f'))$
4. If $x > x'$, dualize the previous case.

2.4 Model Checking Tools

There are a wide variety of Model checking tools available nowadays, most of them accepts system requirements or design (called models) and a property (called specification) that the final system is expected to satisfy. The tool then outputs yes if the given model satisfies the required specifications and generates a counterexample otherwise. The counterexample details why the model doesn't satisfy the specification. By studying the counterexample, we can pinpoint the source of the error in the model, correct the model, and try again. The idea is that by ensuring that the model satisfies enough system properties, we increase our confidence in the correctness of the model. SPIN is a popular open-source software tool, used worldwide for the formal verification of distributed software systems [SP]. It was written by Gerard J. Holzmann [Hol04a] and others, and has evolved for more than 15 years. SPIN is an automata-based model checker. Systems to be verified are described in PROMELA (Process Meta Language), which supports modeling of asynchronous distributed algorithms as non-deterministic automata. Properties to be verified are expressed as Linear Temporal Logic (LTL) formulas, which are negated and then converted into Büchi automata as part of the model-checking algorithm. In addition to model-checking, SPIN can also operate as a simulator, following one possible execution path through the system and presenting the resulting execution trace to the user.

NuSMV is a symbolic model checker developed as a joint project between the Formal Methods group in the Automated Reasoning System division at ITC-IRST [Itl], the Model Checking

group at Carnegie Mellon University , the Mechanized Reasoning Group at University of Genova and the Mechanized Reasoning Group at University of Trento. NuSMV is a reimplementation and extension of SMV [Itl], the first model checker based on BDDs. NuSMV has been designed to be an open architecture for model checking, which can be reliably used for the verification of industrial designs, as a core for custom verification tools, as a testbed for formal verification techniques, and applied to other research areas.

The Unified Modeling Language provides two complementary notations, state machines and collaborations, for the specification of dynamic system behavior. HUGO is designed to automatically verify whether the interactions expressed by a collaboration can indeed be realized by a set of state machines [SKM01]. HUGO compile state machines into a PROMELA model and collaborations into set of Büchi automata (”never claims”). The model checker SPIN is called upon to verify the model against the automata. For the purpose of my research, we focused on the model checkers SPIN and HUGO, our case studies are represented using SPIN and HUGO. The next subsections give more details about the tow model checkers.

2.4.1 SPIN Model Checker

The tool SPIN supports a high level language to specify systems descriptions, called PROMELA (a PROcess MEta LAnguage), which is the specification language to model finite-state systems allows the dynamic creation of concurrent processes, where Communication via message channels can be defined to be synchronous or asynchronous. SPIN has been used to trace logical design errors in distributed systems design, such as operating systems, data communications protocols, switching systems, concurrent algorithms, railway signaling protocols, etc. The tool checks the logical consistency of a specification. It reports on deadlocks, unspecified receptions, flags incompleteness, race conditions, and unwarranted assumptions about the relative speeds of processes. SPIN works on-the-fly, which means that it avoids the need to pre-construct a global state graph, or Kripke structure, as a prerequisite for the verification of system properties. SPIN can be used as a full LTL model checking system, supporting all correctness requirements expressible in linear time temporal logic, but it can also be used as an efficient on-the-fly verifier for more basic safety and liveness properties. Correctness properties can be specified as system or process invariants (using assertions), as linear temporal logic requirements (LTL), as formal Büchi Automata, or more broadly as general omega-regular properties in the syntax of never claims.

2.4.1.1 Specification Language PROMELA

PROMELA model consists of **type** declarations, **channel** declarations, **variable** declarations, **process** declarations and **init** process. A PROMELA model corresponds with usually very large, but a finite transition system, so no unbounded data, no unbounded channels, no unbounded processes and no unbounded process creation. Figure 2.9 explain the PROMELA model body. The process type (proctype) consist of a name, a list of formal parameters, local variable declarations and the body which consist of a sequence of statements. A process is defined by a proctype

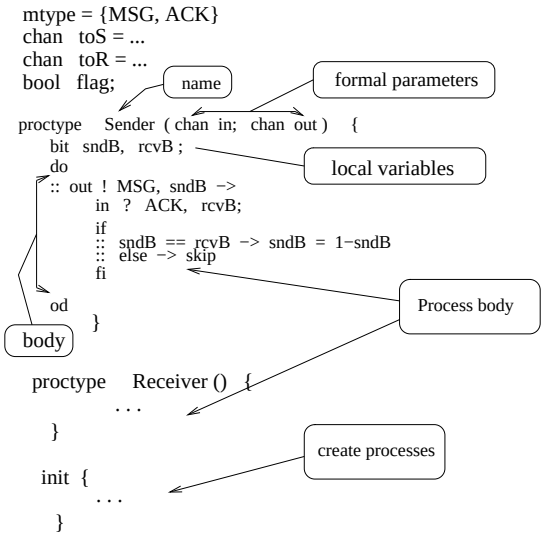


Figure 2.9: PPROMELA body

definition, executes concurrently with all other processes, independent of speed of behavior, communicate with other processes using global (shared) variables and channels. There may be several processes of the same type. Each process has its own local state; process counter (location within the proctype), contents of the local variables. Processes are created using the run statement (which returns the process id). Processes can be created at any point in the execution (within any process) and usually start their executing after the run statement. Processes can also be created by adding active in front of the proctype declaration.

The following example presents a sample PROMELA program which calculate the factorial value of a number n at the process **fact**, returning the result through the channel p [Hol].

```
proctype  fact (int n; chan p) {
    int result;
    if
    :: (n <= 1 ) -> p!1
```

```
    :: (n >= 2 ) ->
            chan child = [1] of {int};
            run  fact(n-1, child);
            child?result;
            p!n*result;
    fi
     }
 init {
      int result;
      chan child = [1] of {int};
      run  fact(5, child);
      child?result
      printf("result: $\%dn"$, result)
      }
}
```

The statement $p!1$ means the sending of the constant one through the channel p. To receive a value or a message from the head of a channel, a receive statement expressed in the symbol $?$ is used such as child?result. The channel in the example can store up to one integer value because the size of each channel is declared as one. For synchronous communication, the channel size has zero. There are three kinds of control flow constructs in PROMELA namely, selection, repetition, and unconditional jumps. The if selection contains several execution sequence, each preceded by a double colon. A sequence can be selected only if its first statement is executable. The first statement is therefore called a guard. If none of the guards of the statement is executable, the construct blocks. In the above example, the if construct has two sequences, but only one sequence can be selected among the sequences. The repetition construct do conducts the same mechanism as if. But it repeats the construct until it meets the $break$. Another way to terminate the repetition is to jump to a label outside the statement $goto$. A label identifies a unique control state and can appear before a statement.

By prefixing a sequence of statements enclosed in parentheses with the keyword $atomic$ a user can indicate that the sequence is to be executed as one indivisible unit, that is noninterleaved with any other processes. Meanwhile, SPIN is able to express the correctness properties in the PROMELA statements such as $assert$, end label, $progress$ label, $accept$ label, and $never\ claims$ [Hol93].
The following shows a result of validation of the factorial program using SPIN:

```
(Spin Version 4.1.3 -- 24 April 2004)
        + Partial Order Reduction
```

```
Full statespace search for:
        never claim             - (none specified)
        assertion violations    +
        acceptance   cycles     - (not selected)
        invalid end states      +

State-vector 124 byte, depth reached 27, errors: 0
      28 states, stored
       0 states, matched
      28 transitions (= stored+matched)
       0 atomic steps
 hash conflicts: 0 (resolved)
 (max size 2^18 states)

 unreached in proctype fact
        (0 of 9 states)
 unreached in proctype :init:
        (0 of 4 states)
```

Internally, SPIN maintains three key data structure [HDR02]: statevector, depthfirst stack, and seen set. The state-vector shows the size of a state which is composed of the value of local and global variables, control flow location of each process, and the contents of message channels. In the example, each state occupies 124 bytes.

The depth reached field represents the deepest stack depth reached during the depth first search of the state space. The seen set holds the state already explored during the search. Thus, the **stored** means the number of states stored in the seen set. The **matched** is the number of states that were already found in the seen set.

2.4.1.2 PROMELA Specification of Two-Phase Commit Protocol

Two-phase commit is a transaction protocol designed for the complications that arise with distributed resource managers. With a two-phase commit protocol, the distributed transaction manager employs a coordinator to manage the individual resource managers. One can specify the protocol in two phases as following:

- Phase 1

  - Each participating resource manager coordinates local operations and forces all log records out.
  - If successful, respond ”OK”.
  - If unsuccessful, either allow a time-out or respond ”OOPS”.

- Phase 2
  - If all participants respond ”OK”.
    * Coordinator instructs participating resource managers to ”COMMIT”.
    * Participants complete operation writing the log record for the commit.
  - Otherwise:
    * Coordinator instructs participating resource managers to ”ROLLBACK”
    * Participants complete their respective local undos

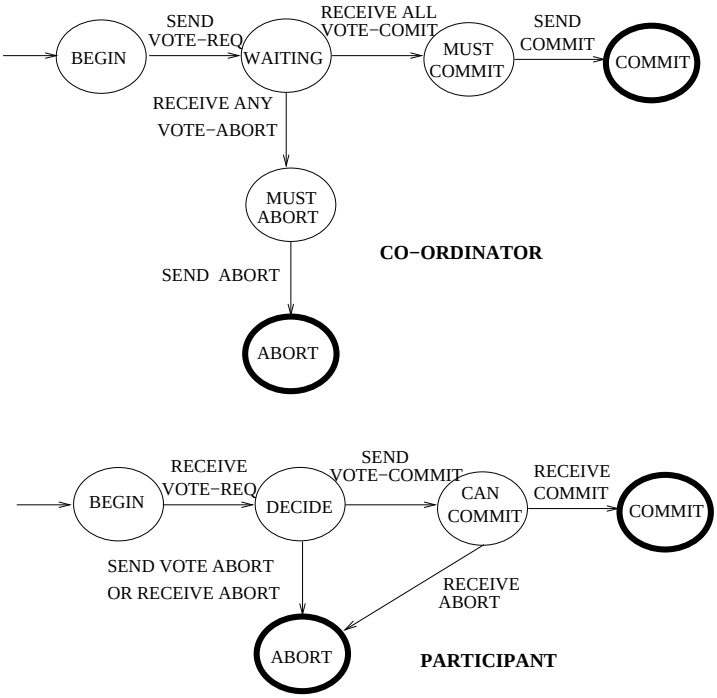


Figure 2.10: Two-Phase Commit Protocol

Figure 2.10 explains the two-phase commit protocol as a finite state machine. The specification of two-phase commit protocol in PROMELA is given as follows:

```
#define  PARTICIPANTS 4
mtype = {START, COMMIT, ABORT, NODECISION}
```

```
chan pchannel[PARTICIPANTS] = [0] of {mtype},
      cchannel = [0] of {mtype}
int  globaldecision = NODECISION;
proctype   participant (int id)
{
  mtype  decision = NODECISION;
  pchannel [id] ? START ->
      if
        :: cchannel ! ABORT; decision = ABORT
        :: cchannel ! COMMIT;  pchannel[id] ? decision
      fi;  assert (decision == globaldecision)
}
init
{
int   count = 0;
do
  :: (count < PARTICIPANTS) -> run  participant(count);
       count++
  :: (count == PARTICIPANTS) -> break
od;  run  coordinator();
}
proctype   coordinator ()
{
   int  count1 = 0, count2 = 0, count3 = 0; mtype  vote;
   do
    ::  (count1 < PARTICIPANTS) ->
                pchannel[count1] ! START;
          count1++
    ::  (count1 == PARTICIPNATS) ->
                break
   od;
   globaldecision = COMMIT;
   do
    :: (count2 < PARTICIPANT) ->
       cchannel ? vote ;  count2++ ;
       if
       :: (vote  ==  ABORT) ->
```

```
                globaldecision = ABORT
        :: (vote  ==  COMMIT) ->
                skip
        fi
    :: (count2 == PARTICIPANTS)  ->
                break
    od;
    do
    ::  (count3 < PARTICIPANTS) ->
        pchannel[count3] ! globaldecision;
         count3 ++
    ::  (count3 == PARTICIPANTS) ->
        break
    od
}
```

2.4.2 HUGO Model Checker

HUGO is a prototype tool designed to automatically verify whether the interactions expressed by a collaboration diagram can indeed be realized by a set of state machines. Technically, this is achieved by compiling state machines into a $PROMELA$ model, and collaborations into sets of Büchi automata [2] (" never claim"). The model checker SPIN is then called to verify the model against the automata. The idea to analyze UML state machines and other variants of Statecharts using model checking has been suggested before [Kwo00, LMM99, LP99, MLS97, MLSH99], but HUGO is based on dynamic computation of Statechart behavior rather than a predetermined, static calculation of possible state transitions in response to input events. HUGO has the advantage of being more modular, more flexible, and easier to adapt to variants of Statechart semantics, including possible changes to the semantics of UML state machines.

Besides model checking, HUGO also supports animation and the generation of Java code from UML state machine models, based on the same structure of implementation. HUGO provides us also the correctness of the generated code with respect to the properties verified from the PROMELA model. (see figure 2.11)

[2] A Büchi automata is the extension of a finite state machine to infinite inputs. It accepts an infinite input sequence, iff there exists a run of the automaton (in case of a deterministic automaton, there is exactly one possible run) which has infinitely many states in the set of final states. It is named after the Swiss mathematician Julius Richard Büchi. Finite state machine (FSM) is a model of behavior composed of a finite number of states, transitions between those states, and actions.

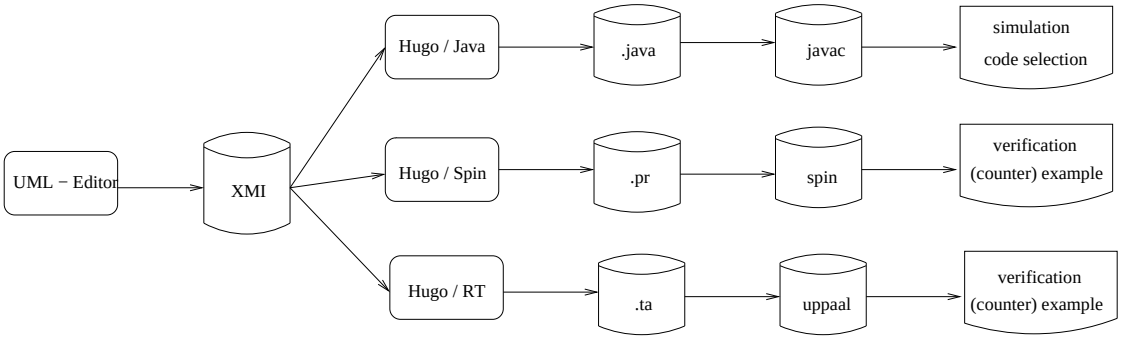


Figure 2.11: HUGO model checker

Verifying ATM State Machines using HUGO We design our case study the state machines of ATM using UML notations with MagicDraw software as shown in fig 2.12, The class diagram in this figure specifies two (active) classes ATM and Bank connected by an association such that instances of Bank can refer to an instance of ATM via atm. Classes define $attributes$, i.e., local variables of its instances, and $operation$ and $signals$ that may be invoked on instances by call or send actions, respectively.

The state machine for class $Bank$ is shown in figure 2.13, consisting of states and transitions

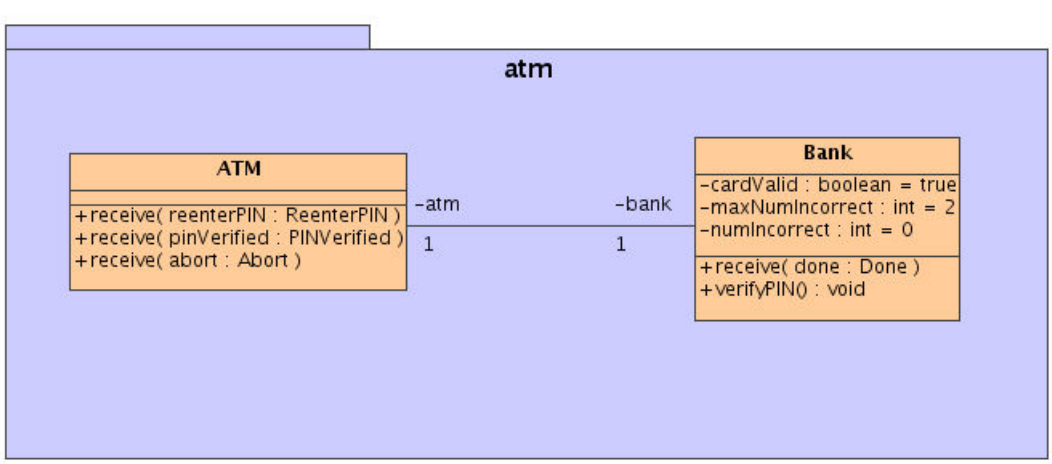


Figure 2.12: Two-Phase Commit Protocol

between states. States can be simple (such as $Idle$ and $PINCorrect$) or composite (such as $Verifying$); a concurrent composite state contains several orthogonal regions, separated by dashed lines. Moreover, fork and join ($pseudo-$) states, shown as bars, synchronize several transitions to and from orthogonal regions; junction (pseudo-)states, represented as filled circles, chain together multiple transitions. Transitions between states are triggered by $events$. Transitions may also be guarded by $conditions$ and specify $actions$ to be executed or $events$

to be emitted when the transition is fired. For example, the transition leading from state $Idle$ to the $fork$ pseudostate requires signal $verifyPIN$ to be present; the transition branch from $VerifyingCard$ to $CardValid$ requires the guard $cardValid$ to be true; the transition branches from $CardValid$ to $Idle$ set the Bank attribute $cardValid$.

Events may also be emitted by $entry$ and $exit$ actions that are executed when a state is activated or deactivated. Transitions without an explicit $trigger$ (e.g. the transition leaving $DispenseMoney$), are called completion transitions and are triggered by completion events which are emitted when a state completes all its internal activities.

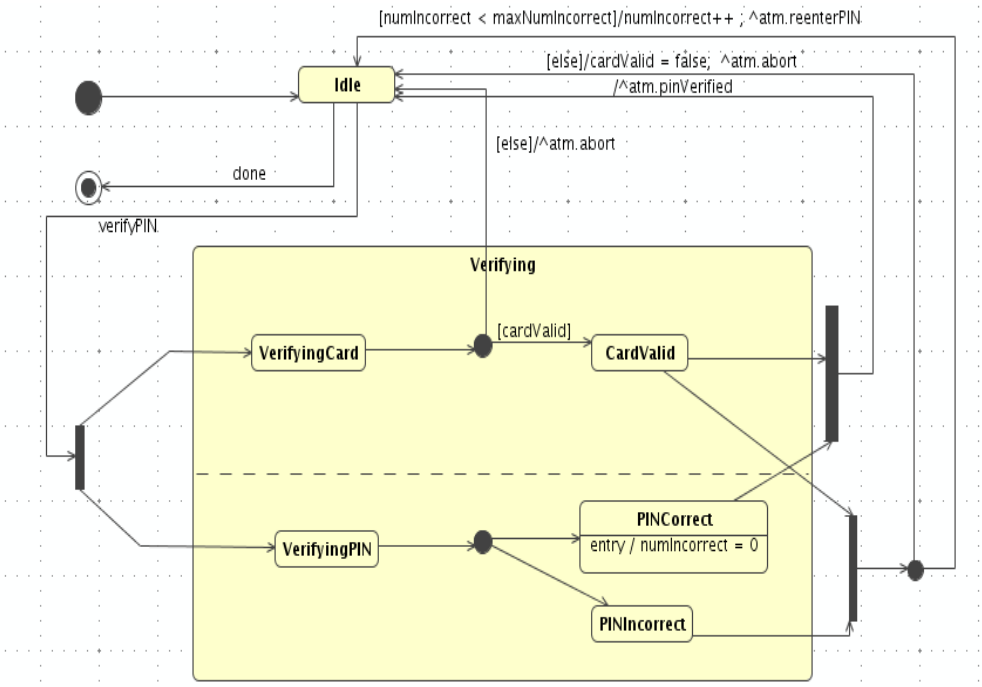


Figure 2.13: State Machine of the Bank in ATM

The state machine for class ATM is shown in figure 2.14, it consists of some simple states (like $CardEntry$ and a composite state ($GivingMoney$). Transitions in this state machines have also triggers like ($reenterPIN$) or events like ($bank.verifyPIN()$) or ($bank.done$). This example of ATM simulate the interaction of an ATM with a single hypothetical user and a bank computer. The simulation focuses on card and PIN validation, after the user has entered his bank card, the ATM requests a PIN to be entered and then asks the bank to verify the entry, waiting for a reply. If both card and PIN are valid, the ATM may proceed to dispense money; if the PIN is invalid the ATM will have the user reenter the PIN; if the card is invalid the ATM will be requested to abort the transaction and return the card immediately (this ATM does not keep invalid cards). After having retrieved his card, the user may reenter the same card as many times as he wishes or end

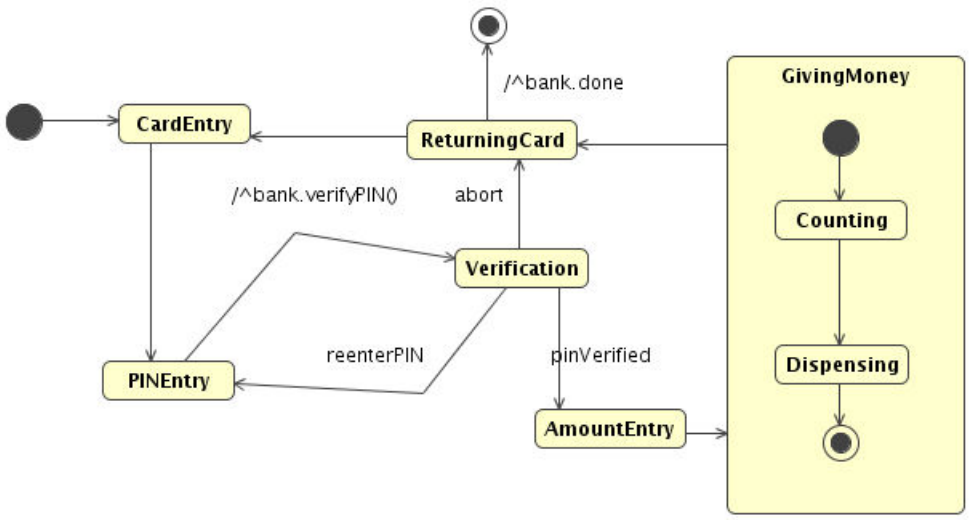


Figure 2.14: State Machine of the atm in ATM

the interaction. As shown in Fig. 2.13, the bank computer validates the bank card concurrently to the PIN code. If the card is not valid, the concurrent validation is exited immediately and the ATM is requested to abort the transaction. The completion transition leaving VerifyingPIN simulates any possible PIN entry by branching non-deterministically into the states PINCorrect and PINIncorrect. The two join transitions evaluate the results of the concurrent validations. If an incorrect PIN has been entered and the card is valid, the counter of invalid PIN entries is incremented; however, if the counter $numIncorrect$ exceeded a maximum value, the card is invalidated and the transaction aborted. In contrast, if a correct PIN has been entered, the counter is reset to zero [SKM01].

Verifying Collaborations HUGO is mainly intended to verify whether certain specified collaborations are indeed feasible for the required state machines. To do so, it generates Büchi automata that accept all executions that conform to the collaboration, and calls on SPIN to verify that no execution of the model is accepted by these ”never claims”. If the collaboration is possible, SPIN will produce a ”counter example” that allows the successful execution to be replayed. HUGO/RT reads UML models in either XMI or UTE [3] format. We discuss in Appendix A the representation of our case study UML state machine of ATM as UTE textual format.

We can add our own collaboration and the desired properties at the end of the UTE specification to show if the properties are indeed verified in our case study or not. For example we can add the following text inside the UTE file.

[3]The UTE is a proprietary UML text format reflects all UML features that are handled by HUGO/RT.

```
collaboration test {
    object bank : Bank {
      atm = atm;
    }
    object atm : ATM {
      bank = bank;
    }
          interaction success {
             atm -> bank : verifyPIN();
             bank -> atm : reenterPIN();
             atm -> bank : verifyPIN();
             bank -> atm : pinVerified();
          }
    assertion deadlockFree {
      AG not deadlock;
    }
  }
```

The property success in the previous text check if the bank sends the message `pinVerified` after successfully verifying the pin number of the card.

HUGO first compiles the UML model into PROMELA code. Given a configuration of instances, each with its corresponding state machine, it will use SPIN to check whether the model contains any deadlocks. As we add also our collaboration to be satisfied, HUGO generates never claims and calls on SPIN to generate an analyzer and run the verification. If SPIN finds a way to satisfy the collaboration, it will generate a trail, and HUGO causes that trail to be executed. The following text is the result of SPIN for the exhaustive search proving that the interaction (success) specified in the collaboration is indeed impossible.

```
(Spin Version 4.1.3 -- 24 April 2004)
        + Partial Order Reduction
Full statespace search for:
        never claim                - (not selected)
        assertion violations       - (disabled by -A flag)
        cycle checks               - (disabled by -DSAFETY)
        invalid end states        +

State-vector 60 byte, depth reached 2, errors: 0
       3 states, stored
```

```
       0 states, matched
       3 transitions (= stored+matched)
       0 atomic steps
hash conflicts: 0 (resolved)
(max size 2^19 states)
Stats on memory usage (in Megabytes):
0.000         equivalent memory usage for states
0.266         actual memory usage for states
        (unsuccessful compression: 130545.10%)
        State-vector as stored =
        88763 byte + 8 byte overhead
2.097         memory used for hash table (-w19)
0.320         memory used for DFS stack (-m10000)
2.622         total actual memory usage
```

The result shows that the added interaction ”success” is valid in the ATM state machine model. The assertion deadlock is also valid in the case study.

let’s try to write a new assertion that is not valid in our model:

```
interaction  failing  {
   atm  -> bank  :  verifyPIN();
   bank -> atm   :  abort();
   atm  -> bank  :  verifyPIN();
   bank -> atm   :  pinVerified();
                       }
```

Now if we want to use HUGO to see if this property is valid in the atm model or not, we have to write the following command:

hugort spin -i = failing models/models/atm.ute

It means, satisfy the property failing in the given model atm using SPIN model checker. The result of HUGO is: **Property is not satisfied**

2.5 Result and Discussion

We introduced in this chapter the basic concepts of propositional logic, temporal logic and model checking theories. Two case studies are also presented; automatic teller machine and two-phase commit protocol. Model checker tools like HUGO and SPIN are used to verify if the desired properties in our case studies are valid or not.

The UML is widely used for the description of object-oriented software designs and provides an excellent environment for implementation the software. Therefore we represented our case studies as UML state machines and we used HUGO model checker to verify the desired properties of the case studies. HUGO takes the standard XMI files as input and translates them into an intermediate textual specification language called UTE. The desired properties are also specified in the UTE language. HUGO uses the SPIN model checker for verifying the required behavior.

3 Graph Language

3.1 Introduction

Software architecture and design are usually modeled and represented by informal diagrams such as Unified Modeling Languages (UML) diagrams. While these graphic notations are easy to understand and are convenient to use, they are not amenable to automated verification and transformation. Graphic notations are widely used in software design and development. These notations can greatly help on the modeling and representing of software architecture [CES86] and design [Lam94]. Nowadays graph grammars enable a high level of abstraction of software architectures and form a basis for various analysis and transformations. Furthermore, software verifications are also performed through these new concepts of graph grammar and graph transformation system [KZDS].
The research area of graph grammars and graph transformations dates back to 1970 [Roz97]. It combines ideas from graph theory, algebra, logic, and category theory. Its methods, techniques, and results have already been applied in many fields of computer science [EEPT06]. This wide applicability is due to the fact that graphs are very natural way to explain complex situations on an application area, whereas graph transformations bring theses situations to be built and interpreted. The field of graph grammars applies formal languages theory to the specification of graphs. A graph grammar consists of a set of productions that can be used to construct valid sentences in a graph (network) language. Graph transformations associated with graph grammars are well-suited for modeling the dynamic behavior of Systems [Roz97]. Therefore graph grammars and graph transformations become attractive as a programming paradigm for software and graphical interfaces.

Graph transformation has originally evolved in reaction to shortcomings in the expressiveness of classical approaches to rewriting, like Chomsky grammars [1] [Cho56] and term rewriting, to deal with non-linear structures. The first proposal, appearing in the late sixties and early seventies [PA69, Pra71] are concerned with rule-based image re-cognation, translation of diagram languages, etc. In Particular, graphs provide a simple and a powerful approach to a variety of

[1] This hierarchy of grammars was described by Noam Chomsky in 1956. It is also named after Marcel-Paul Schützenberger who played a crucial role in the development of the theory of formal languages.

problems that are typical to software engineering [GJM91a].

A graph transformation rule describe the evolution of models in a visual language in a general way. Simply, a graph transformation rule $r = (LHS,\ RHS,\ NAC)$ contains a left-hand side graph LHS, a right-hand side graph RHS, and negative application condition NAC. The application of a rule r to a host model (instance graph) M replaces a matching of the LHS in M by an image of the RHS. This is performed in two phases:

- **Pattern matching :** Find a match of LHS in the model M (by graph pattern matching), then check the negative application condition NAC [2] [EEHP04].
- **Updating:** Remove a part of the model M that can be mapped to LHS but not to RHS yielding to the context model, then glue the context model with an image of the RHS by adding new objects and links (that can be mapped to the RHS but not to the LHS) to obtain the derived model M'

Example: Vehicle routine systems commonly provide a mechanism to reroute a customer. Figure 3.1 shows a production of a graph grammar for rerouting. As we see in the left side of this figure, we can reroute the customer a which is placed between b and c to another route between d and e. On the right hand side, the result of rerouting a is depicted by the graph. Applying the production rule to a network (matching the left hand side of the rule) performs this rerouting of a.

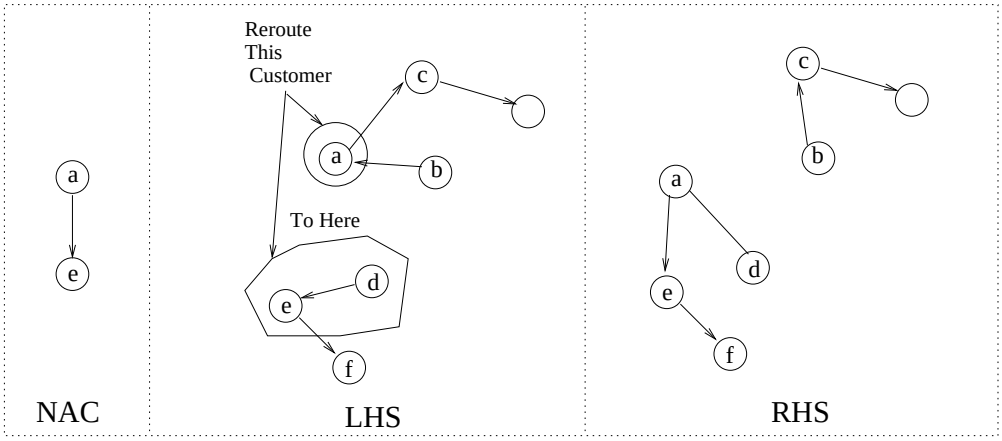


Figure 3.1: The graph grammar production

This production has a negative application condition NAC, which expresses that the process cannot be done if there is already route between the node a and e.

[2] We discuss the negative application condition in the next sections

This production can be applied on the model as we presented in figure 3.2. Suppose that in the pattern matching phase, a, b and c of the production are mapped to a, b and c of the network, respectively; thus the idea of pattern matching has been terminated successfully. Since the selected production does not have any associated requests and the negative application condition does not prohibit the execution of the rule. In the updating stage, the edges between b and a and between a and c are removed from the model, and new edges between d and a and also between a and e are created.

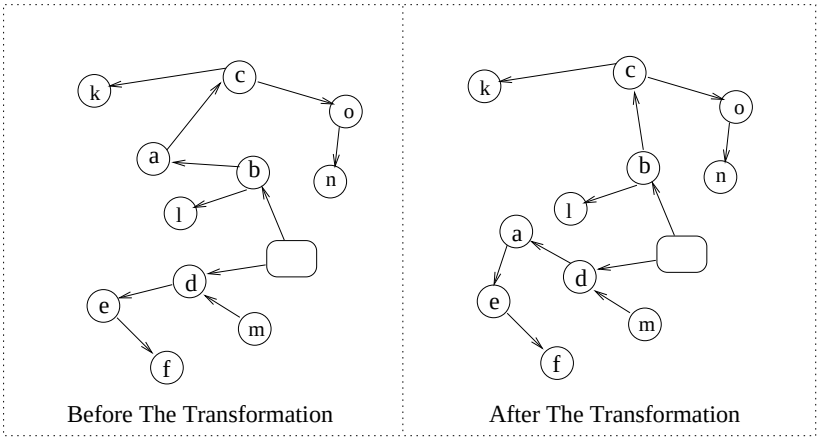


Figure 3.2: Implementing the production at the network graph

Definition (Graph) A graph $G = (V, E, source, target)$ consists of a set V of nodes (also called vertices), E of edges and two functions source, target : $E \rightarrow V$, the source and target functions.

$$E \underset{t}{\overset{s}{\rightrightarrows}} V$$

Example (Graph) The graph $G_S = (V_S, E_S, s_S, t_S)$, with nodes set $V_S = \{u, v, x, y\}$, edge set $E_S = \{a, b\}$, source function $s_S : E_S \rightarrow V_S : a, b \mapsto u$ and target function $t_S : E_S \rightarrow V_S : a, b \mapsto v$, is represented as shown in figure 3.3:

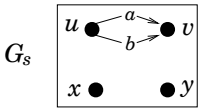
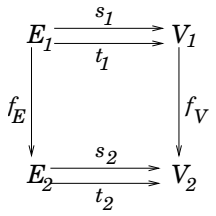


Figure 3.3: Example of graph with edges and vertices

Definition (graph morphism) graph morphism Given graphs G_1, G_2 with $G_i = (V_i, E_i, s_i, t_i)$ for $i = 1, 2$, a graph morphism $f : G_1 \rightarrow G_2, f = (f_V, f_E)$ consists of tow functions $f_V : V_1 \rightarrow V_2$ and $f_E : E_1 \rightarrow E_2$ that preserve the source and target functions, i.e. $f_V \circ s_1 = s_2 \circ f_E$ and $f_V \circ t_1 = t_2 \; o \; f_E$:



In the next section we discuss the process of creating the graph models from the observed scenario. Section 3.3 provides more advance concepts to create the rules for the transformation processes from the observed scenario. We illustrate the mechanism of transformations in section 3.4, and section 3.5 provides some formal definitions of graph constraints. Finally section 3.6 discusses the tools that are useful to represent our examples and case studies. Some examples are also presented in this chapter to give a better understanding of graph transformation approaches.

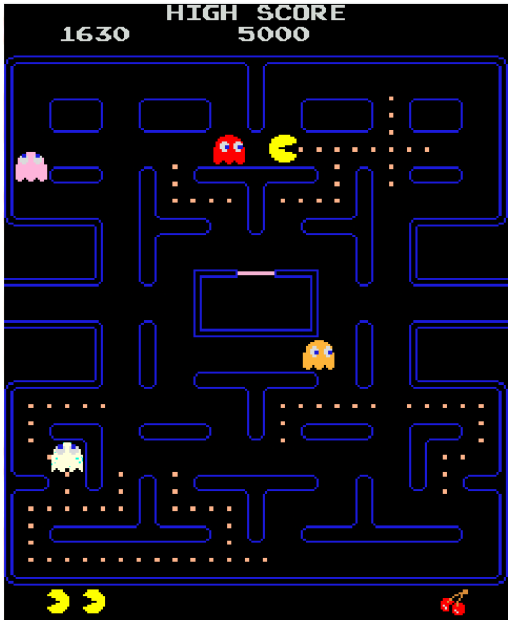
3.2 From Scenario to Graph Language

Business applications support business processes, processes which can be described using a variety of process models (hence the term). It's important in the modeling stage to extract the concepts from concrete objects, or rules from observed behavior. We can create graph models after our well-understanding to the objects of the system and the way that make these objects active. In this section we illustrate in details how we define our graph models from the observed scenarios.

3.2.1 The Scenario

Actions, Events, Entries, Exists ... etc, are concepts of scenario. The scenario is a method that some organizations use to make flexible long-term plans or is a synthetic description of an event or series of actions. Scenario should represent a situation that can be encountered in the real world. Avoiding this discussion, we illustrate the forms of scenario by means of a simple video game of **PacMan**. The insights gained from this example is to show how we build our graph model from the given scenario and to explain the main concepts of graph grammar and graph transformation systems.

Figure 3.4 exemplifies the concepts and behaviors of PacMan game. Imagine that you are PacMan, and you must eat all the small dots to get to the next level. You must also keep away from the ghosts, if they take you, you lose one life, unless you have eaten a large dot, then you can (for a limited amount of time) chase and eat the ghosts. Our observation of the game is represented



PacMan is allowed to move right, left, up and down.

The Ghosts are allowed also to move right, left, up and down.

Four Ghosts like monsters also wander the maze in an attempt to eat the PacMan.

A level, or board, is finished when all dots are eaten.

Figure 3.4: The scenario of PacMan video game

by the scenario as shown in figure 3.5 in three successive parts. **Conceptual** where we have to notice that there are three types of characters. **PacMan**, **Ghost**, and **Marble**, each of which has several instances in the scenario. This conceptual part and the corresponding relation between a concept (type) and it's instances is the first main basic idea in the graph transformation systems.

Rules can be derived systematically by determining their scope in the movement and cutting off (abstracting from) the irrelevant context (see figure 3.5). For the rules **collect** and **kill** in this figure, their scope is given when the PacMan eat the marble and when the Ghost kill the PacMan, respectively.

The idea of extracting rules as general behavior descriptions from sample state transformation action is called *programming by example*. Programming by example provides a perfect example of the second basic idea of graph transformation: *the definition of rules as specifications of state transformations*.

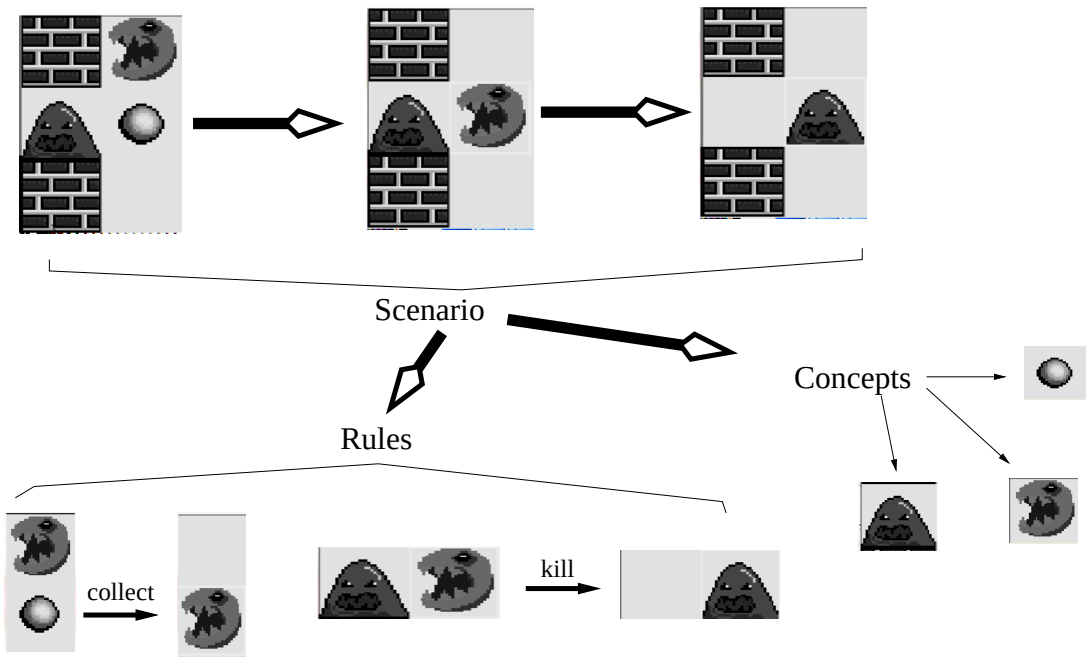


Figure 3.5: The scenario of PacMan video game

3.2.2 Type and Instance Graph

Graphs are used to specify software by distinguishing individual components and their relationships. A graph consists of a set of vertices V and a set of edges E such that each edge e in E has a source and a target vertex $s(e)$ and $t(e)$ in V, respectively. Graphs can represent States by modeling concrete entities as vertices and relations between these entities as edges. In our model, The type of vertices; $P : PacMan$, $G : Ghost$, $M : Marble$ represent the corresponding characters in the scenario. In other words P is a vertex from type PacMan, G is a also a vertex from type Ghost and M is a vertex from type Marble. We need also another type of vertex to represent fields, i.e., the open spaces in the scenario where characters can be located. Edges represent the current location of characters as well as the neighborhood relation of fields.

In modeling this scenario we have implicitly assumed that vertices have a type, like F_1 to F_4 of figure 5.22 having type of **Field**. The type of a vertex (or an edge) represents the conceptual part of the corresponding real-world entity. The collection of these concepts may be represented as a type graph. At the top side of figure 5.22 we shows an example of a type graph representing the conceptual part of the PacMan game, whereas at the bottom side is an instance graph of this game. The relation between concepts and their occurrences in the scenario is formally captured by the notion of **typed graphs:** TG which represents the type (concept) part of the scenario and it's **instance graphs** the individual states in the scenario.

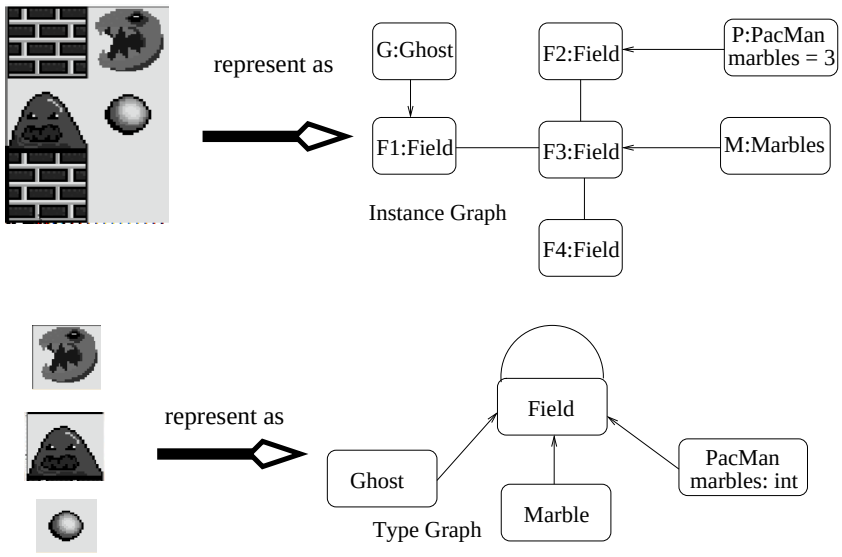


Figure 3.6: Type and instance graphs from the scenario

We get the concepts of type and instance graphs and their relationship from the notation of class and object diagrams in the Unified Modeling Language (UML), i.e., $o : C$ represents a vertex o (like an object) of type C (like the class). In addition to vertices and edges, graphs may contain attributes to store values of pre-defined data types. In our example, this notion is used to represents the number of marbles where PacMan has collected before his movement to the next state. Also attributes have a type-level declaration $a : T$, where a is the name of the attribute and T is the data type of this attribute, and an instance-level occurrence $a = v$ where attribute a is assigned to value v

The relation between type and instance level is determined as following:

- For each vertex $o : C$ in the instance graph there must a vertex type C in the type graph;
- For each edge between objects $o_1 : C_1$ and $o_2 : C_2$ there must be a corresponding edge type in the type graph between vertex types C_1 and C_2;
- For each attribute value $a = v$ associated with a vertex $o : C$ in an instance graph, there must be a corresponding declaration $a : T$ in vertex type C such that v is of data type T;

Definition (typed graph) A type graph is a distinguished graph $TG = (V_{TG},\ E_{TG},\ s_{TG},\ t_{TG})$. V_{TG} and E_{TG} are called the vertex and the edge type alphabets, respectively. A tuple $(G, type)$ of a graph G together with a graph morphism type : $G \rightarrow TG$ is the called a typed graph

Definition (typed graph morphism) Given typed graphs $G_1^T = (G_1, type_1)$ and $G_2^T = (G_2,\ type_2)$, a typed graph morphism $f : G_1^T \rightarrow G_2^T$ is a graph morphism $f : G_1 \rightarrow G_2$ such that $type_2 \circ f = type_1$ as explained in figure 3.7.

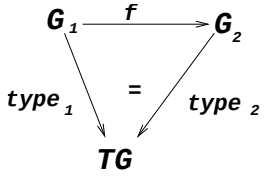


Figure 3.7: Type graph morphism

In order to model attributed graphs with attributes for nodes and edges, we have to extend the classical notion of graphs to $E-$graphs. An $E-$graph has two different kinds of nodes, representing the graph and data nodes, and three kind of edges, the usual graph edges and special edges used for the node and edge attribution. The differences between $E-$graphs, graphs, and labeled graphs are discussed below.

Definition ($E-$graph) An $E-$graph G with $G = (V_G,\ V_D,\ E_G,\ E_{NA},\ E_{EA},\ (source_j,\ target_j)_{j\in\{G,NA,EA\}})$ consists of the sets

- V_G and V_D, called the graph and data nodes (or vertices), respectively;
- E_G, E_{NA} and E_{EA} called the graph, node attribute, and edge attribute edges, respectively;

and the source and target functions

- $source_G : E_G \rightarrow V_G, target_G : E_G \rightarrow V_G$ for graph edges;
- $source_{NA} : E_{NA} \rightarrow V_G,\ target_{NA} : E_{NA} \rightarrow V_D$ for node attribute edges; and
- $source_{EA} : E_{EA} \rightarrow E_G, target_{EA} : E_{EA} \rightarrow V_D$ for edge attribute edges:

Figure 3.8 illustrate the $E-$graph model.

Definition ($E-$graph morphism) Consider the $E-$graphs G^1 and G^2 with $G^k = (V_{G^k},\ V_{D^k},\ E_{G^k},\ E_{NA^k},\ E_{EA^k},\ (source_{j^k}, target_{j^k})_{j\in\{G,NA,EA\}})$ for $k = 1,\ 2$. An $E-$graph morphism $f : G^1 \rightarrow G^2$ is a tuple $(f_{V_G}, f_{V_D}, f_{E_G}, f_{E_{NA}}, f_{E_{EA}})$ with $f_{V_i} : V_{i^1} \rightarrow V_{i^2}$ and $f_{E_j} : E_{j^1} \rightarrow E_{j^2}$ for $i \in \{G, D\}$, $j \in \{G, NA, EA\}$ such that f commutes with all source and target functions, for example $f_{V_G} \circ source_G^1 = source_G^2 \circ f_{E_G}$.

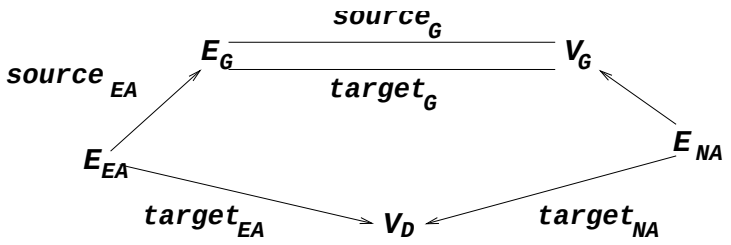
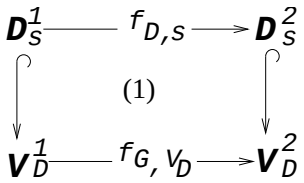


Figure 3.8: The E graph

An attributed graph is an $E-$graph combined with an algebra over a data signature $DSIG$ [3]. In the signature, we distinguish a set of attribute value sorts. The corresponding carrier sets in the algebra can be used for the attribution.

Definition (attributed graph and attributed graph morphism) Let $DSIG = (S_D, OP_D)$ be a *data signature* with attribute value sorts $S'_D \subseteq S_D$. An attributed graph $AG = (G, D)$ consists of an $E-$graph G together with a $DSIG - algebra\ D$ such that $\cup_{s \in S'_D} D_S = V_D$.

For two attributed graphs $AG^1 = (G^1, D^1)$ and $AG^2 = (G^2, D^2)$, an attributed graph morphism $f : AG^1 \rightarrow AG^2$ is a pair $f = (f_G, f_D)$ with an $E-$ graph morphism $f_G : G^1 \rightarrow G^2$ and an algebra homomorphism $f_D : D^1 \rightarrow D^2$ such that (1) commutes for all $s \in S'_D$, where the vertical arrows below are inclusions:



Attributed graph morphisms are used later for transformations but they are also the basis for defining a type graph, which restricts the structure of graphs in a system.

Example (attributed graph) Let's try to represent the states PINEntry, Verification and transition between these two states of the case study ATM state machines as attributed graph. In figure 3.9 we represent the state PINEntry and Verification as nodes in the graph. The transitions between PINEntry and Verification is also represented as nodes in the attributed graph. Figure 3.9 shows that every node has a special attribute refers to the name of this node. For example, the state PINEntry in the ATM state machine is represented as a node from type State in attributed graph, and this node has an attribute called $St\_name ='' PINEntry''$ from type $string$. The same strategy is also implemented for other states and transitions in ATM state machines.

[3] Consider a data signature $DSIG =< S,\ OP >$ with attribute value sorts S and a graph $G =< V,\ E >$.

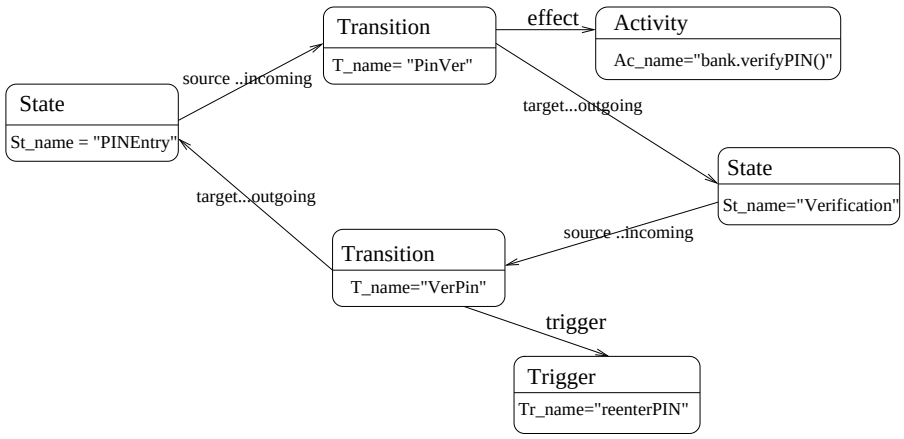
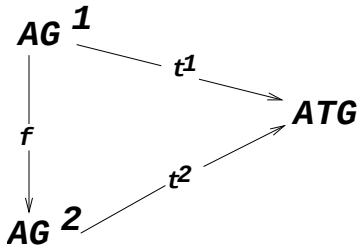


Figure 3.9: Part of ATM state machines as attributed graph

Definition (Typed Attributed Graph and Typed Attributed Graph Morphism) Given a data signature $DSIG$, an attributed type graph is an attributed graph $ATG = (TG, Z)$, where Z is the final $DSIG-$algebra. A typed attributed graph (AG, t) over ATG consists of an attributed graph AG together with an attributed graph morphism $t : AG \rightarrow ATG$. A typed attributed graph morphism $f : (AG^1, t^1) \rightarrow (AG^2, t^2)$ is an attributed graph morphism $f : AG^1 \rightarrow AG^2$ such that $t^2 \circ f = t^1$. The following examples of attributed graphs are shown



in a compact notation according to UML class diagrams, where the attributes are written inside the corresponding nodes or edges, respectively.

This notation is also used in the graph transformation tool **AGG**[4] for attributed graph grammars, which is used in our case studies as described later.

Example This example of attributed type graph (figure 3.10) shows the representation of states and transitions of the case study ATM state machines as nodes and edges in the type graph, respectively. Every node has a type (type of the node) and a special attribute refers to the name of the node.

[4]AGG: is abbreviation for Attributed Graph Grammar, which is a tool for graph grammar implementations, AGG is discussed in the last section.

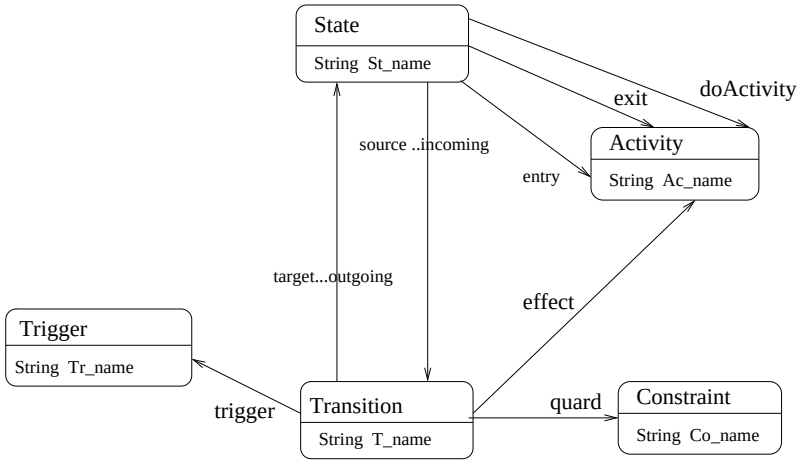


Figure 3.10: Part of ATM state machines as attributed type graph

3.3 From Scenario to Rules and Transformations

As we discussed in the previous section, we could represent the states of scenario as instance graph over a type graph. Let's try to represent the actions between the states in the scenario. If we can determine an instance graph and transform it to another instance graph, we extract actually the rule which represent the transformation of this step of movement. Following the idea of extracting rules from transformation scenario, figure 3.11 shows a graph representation of the behavioral part of the scenario. The rule is achieved by focusing on the relevant subgraph

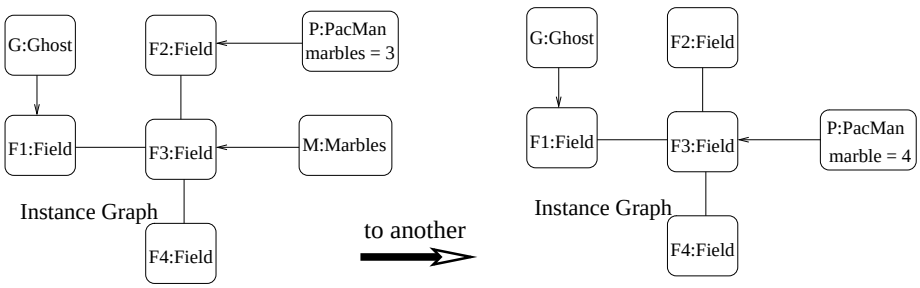


Figure 3.11: Representing the behavioral part as graph transformation

in the source state and observing its changes in the target state. But besides cutting off context, we also abstract from the concrete attribute values replacing, e.g., $marble = 4$. The marble number is increased by one as we show in figure 3.11. The resulting rules are shown in the right hand side of this figure, with the rule for moving PacMan one step to the bottom.

Formally, fixing a type graph TG, a *graph transformation rule* $p : L \rightarrow R$ consists of a name p

and a pair of instance graphs over TG whose structure is compatible [5]. The left-hand side L represents the pre-conditions of the rule while the right-hand side R describes the post-conditions. Rules do also have a constructive meaning, besides being generalizations of transformation, they *generate* transformations by replacing in a given graph an occurrence of the left-hand side with a copy of the right-hand side. Thus, a *graph transformation* from a pre-state G to a post-state H, as shown in figure 3.12 denoted by $G \Longrightarrow^{p(o)} H$, is performed in three main steps:

- Find an occurrence o_L of the left-hand side L in the given graph G.
- Delete from G all vertices and edges matched by $L \setminus R$.
- Paste to the result a copy of $R \setminus L$, yielding the derived graph H.

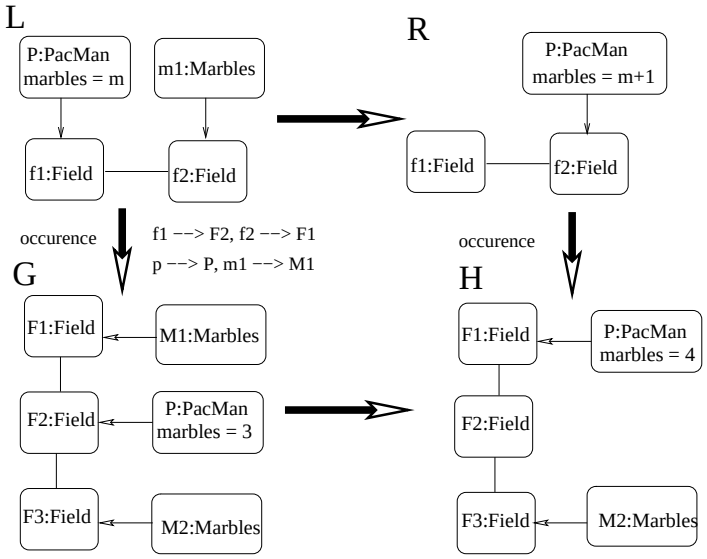


Figure 3.12: Creating graph transformation from behavioral scenario

In figure 3.12 the occurrence o_L of the rule's left-hand side is indicated next to the left downward arrow. The variable $marble$ representing the value of the marble attribute before the step is assigned value 3. The transformation deletes the edges from PacMan P to Field F_2, because it is matched by the edge from F_1 to p in L which does not occur in R. The same applies to the value 3 of the marbles attributes of vertex P. To the graph obtained after deletion, we paste a copy of the edge from p to f_2 in R. The occurrence o_L tells us where this edge must be added, i.e., to the images of p and f_2, P and F_1, respectively. At the same time, the new attribute value $marbles = 3 + 1 = 4$ is computed from the memorized old value $m = 3$.

[5] Vertices with the same identity in L and R have the same type and attributes, and edges with the same identity have the same types, source, and target

However, this is not the only possibility for applying this rule. Another option would be to map $f_1 \rightarrow F_2$, $f_2 \rightarrow F_3$, $p \rightarrow P$, $m \rightarrow M_2$, collecting the lower marble instead. Also, we could have chosen to apply the $movePM$ rule. That means, there are two causes of non-determinism: choosing the rule and the occurrence at which it is applied.

The total behavior of our PacMan game is given by the set of all sequences of consecutive transformation steps $G_0 \rightarrow ... \rightarrow G_n$ using the rule of the game and starting from a valid instance graph G_0. As a simple example, we recall the two-step sequence in figure 3.13 which is re-generated by application of the two previously extracted rules. Note that all graphs of a sequence must be valid instances of the fixed type graph TG.

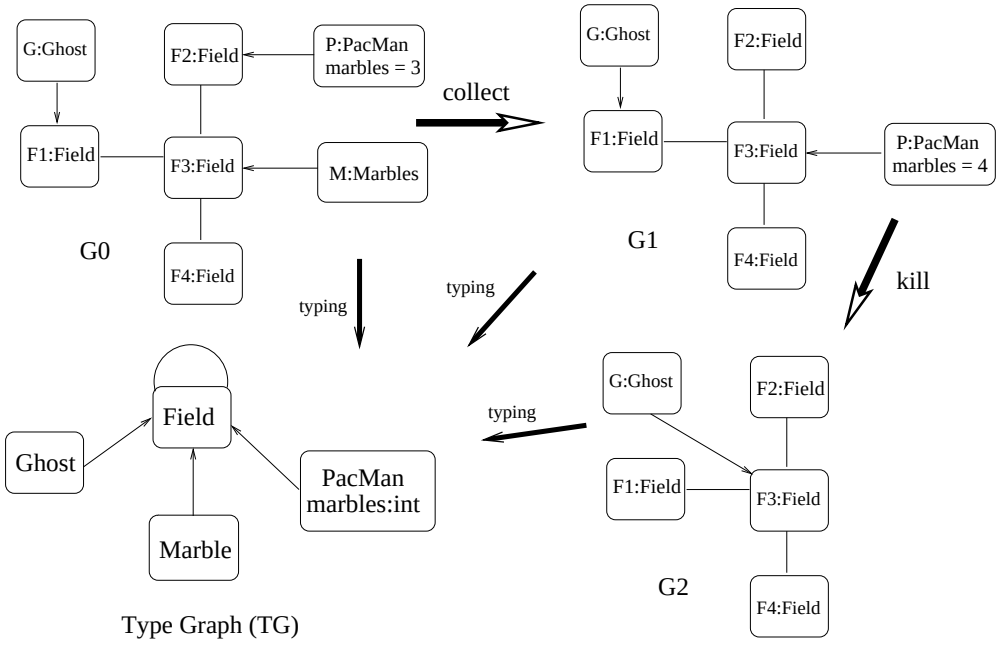


Figure 3.13: The total behavior of PacMan game

Definitions (Graph Production) A graph productions $p = (L \leftarrow^l K \rightarrow^r R)$ consists of graphs L, K, and R, called the left-hand side, the interface, and the right-hand side of P respectively, and two injective graph morphisms $l : K \rightarrow L$ and $r : K \rightarrow R$.

Given a graph production p, the inverse production is defined by $p^{-1} = (R \leftarrow^r K \rightarrow^l L)$

Definitions (Graph Grammar) A graph grammar GG is a pair $GG =< (p : L \leftarrow^l K \rightarrow^r R)_{p \in P}, G_0 >$ where the first component is a family of productions indexed by production names in P, and G_0 is the start graph.

Example Our example specifies the dining philosophers problem [oP]. Five philosophers sit around a circular table. Each philosopher spends his time alternatively thinking and eating. In the center of the table is a large plate of noodles. A philosopher needs two forks to eat from the noodles. Unfortunately, the philosophers can only afford five forks. One fork is placed between each pair of philosophers and they agree that each will only use the fork to his immediate right and left. The graph grammar which models Dinning Philosophers problem represented in figure 3.14 as Attributed graph includes five productions, named *getHungry*, *getLeftFork*, *getRightFork*, *ReleaseFork*, *FinishEating*, respectively. Every philosopher could changed to status hungry using the production *getHungry*. Each philosopher may claim a fork next to her using the productions *getLeftFork* and *getRightFork*. Once a hungry philosopher has two forks in her possession, she may start eating. Finally, an eating philosopher can release her forks at any time and return to thinking using productions *ReleaseFork* and *FinishEating*. If each of the philoso-

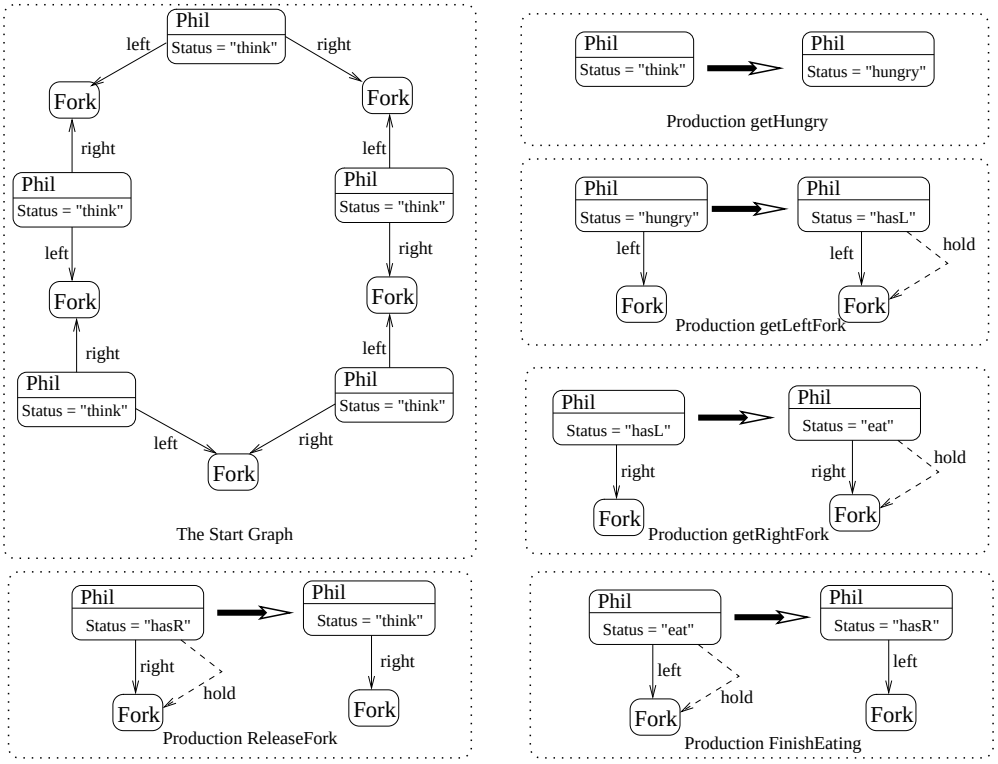


Figure 3.14: Productions and graph grammars of dinning philosopher problem

phers picks up the fork to her left then no further productions are possible and the philosophers starve to death. In order to avoid deadlock, one specifies that each of the initial actions can be reversed [6].

[6]In this case, we have to implement the reversing of the initial actions as new grammar in the philosopher example.

3.4 Graph Transformation

For the application of graph transformation rule to a graph, we need a technique to glue graphs together along a common subgraph, Intuitively, we use this common subgraph and add all other nodes and edges from both graphs. The idea of a pushout generalizes the gluing construction in the sense of category theory, i.e. a pushout object emerges from gluing two objects along a common subobject.

3.4.1 Gluing Condition

Within the algebraic approach and the application of graph transformation, we will only have a valid graph transformation if the match of the left graph L of the rule of transformation in the given graph G satisfies the gluing condition. The gluing condition is verified if and only if the two sub conditions are valid:

- The identification condition, this condition will be satisfied if two different elements x and y of the left graph L either are mapped injectively (none two different elements of the definition quantity are mapped to the same element of the target quantity) or may only not be mapped injectively if these two elements are not deleted by the transformation rule. Formally $O(x) = O(y)$ only if $x = y$ or $x, y \in L \bigcap R$
- The dangling condition, this condition will be satisfied if an edge e of $M - g(L)$ neither has its source node nor its target node in $g(L) - g(K)$. That means that, if we delete one node, we have to delete all edges that are adjacent to this node.

The addition of these two sub conditions forms the gluing condition [Kwo00]

3.4.2 Double-Pushout Approach DPO

The double−pushout approach, shortly called DPO, is a sub approach of the algebraic approach and is the frequently used approach for graph transformations. The DPO adopts a specific rule for the graph transformation which answer two important questions in graph transformations:

- which parts are replaced by which other?
- which kinds of transformations are allowed?

From the perspective of the DPO a graph rewriting rule is a pair of morphisms in the category of graphs with total graph morphisms as arrows, specified by the formal rule $r = (L \leftarrow K \rightarrow R)$.

The graphs L and R are respectively called as we already also mentioned, the left-hand side and the right-hand side of the rule. The graph K is often called gluing graph or interface graph. A rewriting step with the application of the DPO−production is defined as a pair $(L \leftarrow K \rightarrow R)$ or $(L \supseteq K \subseteq R)$ of two graph morphisms as arrows in the category of graphs with an interface graph K, where $K \rightarrow L$ is injective. Because of that the interface graph K is a real subgraph of L as well as of R [CER79, EKRR91]. A rule application $r = (L \supseteq K \subseteq R)$ can be depicted

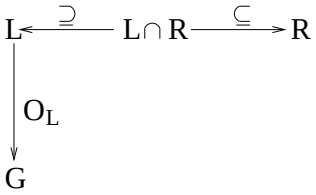


Figure 3.15: Mechanism to find a match

by the following diagrams. They will illustrate the formal step of a graph rewriting and will describe why this approach is called **double−pushout** approach. The gluing graph K is rightly described in the diagrams as $L \cap R$ (see Fig. 3.15). The graph morphism O_L in the shown

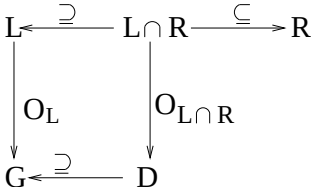


Figure 3.16: building the temporary graph

diagram models an occurrence of L in G and is called the match. Practical understanding of this is that L is a subgraph that is matched from G and after a match is found, the left side of the rule (L) is replaced by the right side of the rule (R) in the host graph G where K as $L \cap R$ serves as some kind of interface.

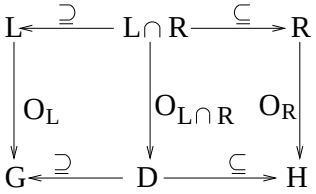


Figure 3.17: construction of the final graph

If a match is found (see Fig. 3.15) there are two steps to achieve the graph rewriting. First you have to build a temporary graph (D) as a subgraph of the host graph G by deleting the matching

elements [7] (figure 3.16). Finally we have to build the final transformed graph H by adding the elements of $R - L \bigcap R$ to the built temporary graph D (see Fig. 3.17).

Formally:

- The single graph morphisms $O_{L \bigcap R} : L \bigcap R \rightarrow D$ and O_R: $R \rightarrow H$ are given by $O_{L \bigcap R}(v) = O_L(v)$ for all $v \in V_{L \bigcap R}$
- $O_{L \bigcap R}(e) = O_L(e)$ for all $e \in E_{L \bigcap R}$
- $O_R(v) = O_L \bigcap R(v)$ if $v \in V_L \bigcap R$.
- $O_R(v) = v$ if $v \in V_R - V_{L \bigcap R}$
- $O_R(e) = O_{L \bigcap R}(e)$ if $e \in E_{L \bigcap R}$
- $O_R(e) = e$ if $e \in E_R - E_{L \bigcap R}$ [KKHK].

3.4.3 Single-Pushout Approach SPO

In contrast to the recently mentioned DPO, a graph rewriting rule of the SPO approach is only a single morphism and therefore only a single derivation of the host graph G with context again to the category of graphs.

The SPO is often used in cases where the interface graph K as in the DPO is only a set of nodes but without any adjacent edges. Then we do not have to look at the edges for the graph rewriting step. We can use the graphical rule representation without an interface graph by depicting only the graphs L and R. Thus a rewriting step is only defined by a single pushout diagram with a single graph morphism as arrows as the formal rule (production) $r : L \rightarrow R$ [AGG].

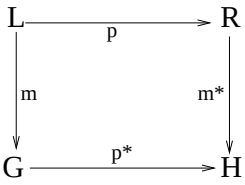
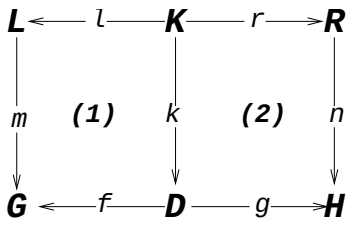


Figure 3.18: building the temporary graph

Figure 3.18 illustrate the practical understanding of the SPO. We perform the rewriting step only a single derivation (morphism) as a single−pushout from the host graph L to the target graph R.

[7] The dangling condition within the gluing condition ensures that D is graph

Definition (Graph Transformation) Given a (typed) graph production $p = (L \leftarrow^{l} K \rightarrow^{r} R)$ and a (typed) graph G with a (typed) graph morphism $m : L \leftarrow G$, called the match, a direct (typed) graph transformation $G \Rightarrow^{p,m} H$ from G to a (typed) graph H is given by the following double-pushout (DPO) diagram, where (1) and (2) are pushout in the category $Graphs$ or $Graphs_{TG}$, respectively :



A sequence $G_0 \Rightarrow ... \Rightarrow G_n$ of direct (type) graph transformations is called a (typed) graph transformation and is denoted by $G_0 \Rightarrow^* G_n$. For $n = 0$, we have the identical (typed) graph transformation $G_0 \cong G_0^{'}$, because pushout and hence also direct graph transformations are only unique up to isomorphism.

The application of a production to a graph G can be reversed by it's inverse production. The result is equal or at least isomorphic to the original graph G

Definition (Graph Language) A graph transformation system $GTS = (P)$ consists of a set of graph productions P.

A typed graph transformation system $GRS = (TG, P)$ consists of a type graph TG and a set of typed graph productions P.

A (typed) graph grammar $GG = (GTS, S)$ consists of a (typed) graph transformation system GTS and a (typed) start graph S.

The (typed) graph language L of GG is defined by $L = \{G | \exists$ (typed) graph transformation $S \rightarrow^* G\}$ We shall use the abbreviation "GT system" for "graph and typed graph transformations system".

3.5 Constraint

Usually we implement the constraints at the host graphs, However type graphs are not expressive enough to define such restrictions. For example, in order to model the PacMan gameboards, it makes sense to require that each Ghost, PacMan, or Marble vertex is linked to exactly one *Filed* vertex. Such constraints need to be expressed by additional cardinality annotations as shown in the type graph of Fig.3.19.

More complex constraints could mean with the (non-) existence of certain patterns, including paths, cycles, .. etc. They can be expressed in terms of logic formulae or as graphical constraints. As we show in figure 9, the constraint expresses by means of a *forbidden subgraph* that there must not be a Ghost and a PacMan situated at the same Field. In order to satisfy the constraint,

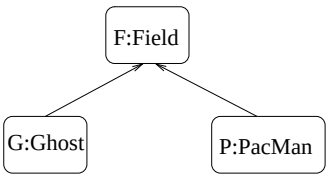
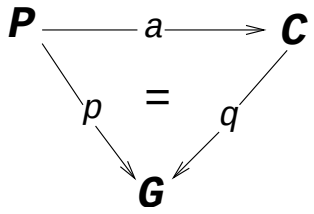


Figure 3.19: The constraint as forbidden subgraph

a graph G must not contain a subgraph isomorphic to it. In first order logic, the same property could read $\neg\exists g : Ghost; p : PacMan; f : Filed.at(g, f) \wedge at(p, f)$.

Definition (Graph Constraint) An atomic (typed) graph constraint is of the form $PC(a)$, where $a : P \rightarrow C$ is a (typed) graph morphism.

A (typed) graph constraint is a boolean formula over atomic (typed) graph constraints. This means that true and every atomic (typed) graph constraint are (typed) graph constraints, and for (typed) graph constraints c and c_i with $i \in I$ for some index set I, $\neg c_i$, $\wedge_{i\in I} c_i$ are (typed) graph constraints: A (typed) graph G satisfies a (typed) graph constraint c, written $G \models c$, if

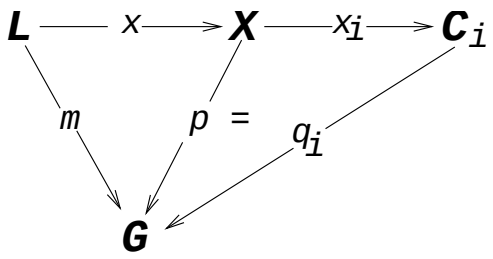


- $c = true$;
- $c = PC(a)$ and, for every injective (typed) graph morphism $p : P \rightarrow G$, there exists an injective (typed) graph morphism $q : C \rightarrow G$ such that $q \circ a = p$;
- $c = \neg c'$ and G does not satisfy c';
- $c = \wedge_{i\in I} c_i$ and G satisfies all c_i with $i \in I$;
- $c = \vee_{i\in I} c_i$ and G satisfies some c_i with $i \in I$.

Two (typed) graph constraints c and $c^{'}$ are equivalent, denoted by $c \equiv c^{'}$, if for all (typed) graphs G, $G \models c$ if and only if $G \models c^{'}$.

Definition (Application Condition) An atomic application condition over a (typed) graph L is of the form $P(x, V_{i\in I}\ x_i)$, where $x : L \to X$ and $x_i : X \to C_i$ with $i \in I$ for some index set I are (typed) graph morphisms.

An application condition over L is a boolean formula over atomic application over L. This means that true every atomic application condition are application conditions, and, for application conditions acc and acc_i with $i \in I, \neg acc_i, \wedge_{i\in I}\ acc_i$, and $\vee_{i\in I}\ acc_i$ are application conditions: A



typed attributed graph morphism $m : L \to G$ satisfies an application condition acc, written $m \models acc$, if

- $acc = true$;
- $acc = P(x, V_{i\in I}\ x_i)$ and for all injective typed attributed graph morphisms $p : X \to G$ with $p \circ x = m$, there exists an $i \in I$ and an injective (typed) graph morphism $q_i : C_i \to G$ with $q_i \circ x_i = p$;
- $acc = \neg acc'$ and m does not satisfy acc';
- $acc = \wedge_{i\in I}\ acc_i$ and m satisfies all acc_i with $i \in I$;
- $acc = \vee_{i\in I}\ acc_i$ and m satisfies some acc_i with $i \in I$

Two application conditions acc and acc' over a (type) graph L are equivalent, denoted by $acc \equiv acc'$, if for all (typed) graph morphisms $m : L \to G$ for some G, $m \models acc$ if and only if $m \models acc'$.

The application condition $\neg true$ is abbreviated as false.

Definition (Negative Application Condition NAC) A simple negative application condition is of the form $NAC(x)$, where $x : L \to X$ is a typed attributed graph morphism. A typed attributed graph morphism $m : L \to G$ satisfies $NAC(x)$ if there does not exist an injective typed attributed graph morphism $p : X \to G$ with $p \circ x = m$. A positive application condition $PAC(x)$ for a rule demands a pattern in a graph for its application and is just the negation of a negative application condition.

3.6 Graph Transformation Tools

Model transformation techniques and tools have become very common nowadays. Any model design problems can be formulated as graph transformation problems, thus, a variety of tools choose this new technique as underlying mechanism for the transformation engine. We explain in this section some of these tools to model transformation that apply graph grammar and graph transformations techniques. The most powerful tools are AGG [AGG], AToM3 [dLV02], VIATRA2 [VP04] and VMTS [VMT, LLMC04].

The underlying concepts of these four tools are all based on graph transformation and model transformations. We illustrate the basic task of these tools. Most of our case studies and examples are represented using AGG.

Attributed Graph Grammar (AGG) is a development environment for attributed graph transformation systems supporting an algebraic approach to graph transformation. It aims at specifying and rapid prototyping applications with complex, graph structured data. Since graph transformation can be applied on very different levels of abstraction, it can be non−attributed, attributed by simple computations or by complex processes, depending on the abstraction level. AGG supports typed graph transformations including type inheritance and multiplicities. Rule application can contain non−deterministic choice of rules which may be controlled by rule layers. Due to its formal foundation, AGG offers validation support being consistency checking of graphs [8] and in graph transformation systems according to graph constraints, critical pair [9] analysis to find conflicts between rules and checking of termination criteria for graph transformation systems.

AToM3 (A Tool for Multi-formalism and Meta-Modeling) is a tool for the design of Domain Specific Visual Languages [dLV02]. It allows defining the abstract and concrete syntax of the Visual Language by means of meta-modeling and expressing model manipulation by means of graph transformation [EEKR99]. With the meta-model information, AToM3 generates

[8] A graphical consistency constraint is a total injective morphism $c : P- > C$, the left graph P called premise and the right graph C conclusion. A graphical consistency constraint is satisfied by a graph G, if for all total injective morphisms $p : P- > G$ there is a total injective morphisms $q : C- > G$ such that $q \circ c = p$. If CC is a set of graphical consistency constraints, we say that G satisfies CC, if G satisfies all constraints in CC.

[9] A critical pair is a pair of transformations $(p_1,\ p_2)$, where $p_1(m_1) : G \Rightarrow H_1$ and $p_2(m_2) : G \Rightarrow H_2$ which are in conflict, and such that graph G is minimal, i.e., G is gluing of the left−hand sides of the rules p_1 and p_2. It can be computed by overlapping L_1 and L_2 in all possible ways, such that the intersection of L_1 and L_2 contains at least one item that is deleted or changed by one of the rules and both rules are applicable to G at their respective occurrences.

a customized modeling environment for the described language. Recently, AToM3 has been extended with functionalities to generate environments for Multi-View Visual Languages (such as UML) and triple graph grammars [Sch].

VIsual Automated model TRAnsformation (VIATRA2) is an Eclipse-based general-purpose model transformation engineering framework that will support the entire life−cycle for the specification, design, execution, validation and maintenance of transformations within and between various modeling languages and domains. Using efficient importers and exporters. VIATRA2 is able to cooperate with an arbitrary external system, and execute the transformation with a native transformation model (plug-in), which is generated by VIATRA2. Its rule specification language combines the graph transformation and abstract state machines into a single paradigm. Essentially, elementary transformation steps are captured by graph transformation rules, while complex transformations are assembled from these basic steps by using abstract state machine rules as control flow specification. Furthermore, model constraints are also captured by the same graph pattern concept.

The Visual Modeling and Transformation System (VMTS) is a general purpose meta-modeling and transformation environment. VMTS is a highly configurable environment offering capabilities for specifying visual languages applying meta-modeling techniques. VMTS uses the instantiation relationship residing between the $M0$ and $M1$ layers in the MOF standard [Gro03a], namely, the one between the UML class diagram and object diagram. The VMTS Presentation Framework (VPF) facilitates a means of rapid development for plug-ins as a customized presentation of the concrete syntax of the models. VMTS defines the model constraints in terms of OCL constraints placed in the meta-model. Since the rules in VMTS are specified by meta-model elements of the input and the output models, the transformation constraints are also expressed in OCL. VMTS has an automated support for preserving, guaranteeing, and validating constraints. The crosscutting concerns are handled with aspect−oriented techniques.

3.6.1 Attributed Graph Grammar (AGG)

AGG graph transformation rules consist of a left−hand and right−hand side graph, a mapping morphism between nodes (and edges) on both sides, and a set of ”negative application conditions” (NAC). A Screenshot of the AGG system shows the working graph and the rules which are present. In the upper right, the selected rule can be found, and in the lower right, the actual working graph is shown. Rules having a NAC are displayed by three graphs (NAC, left−hand side, right−hand side), rules without a NAC are displayed by two graphs. Numbers in front of

node labels represent the morphism of the rule. The working graph in Fig. 3.20 corresponds to our PacMan example in Fig. 3.4.

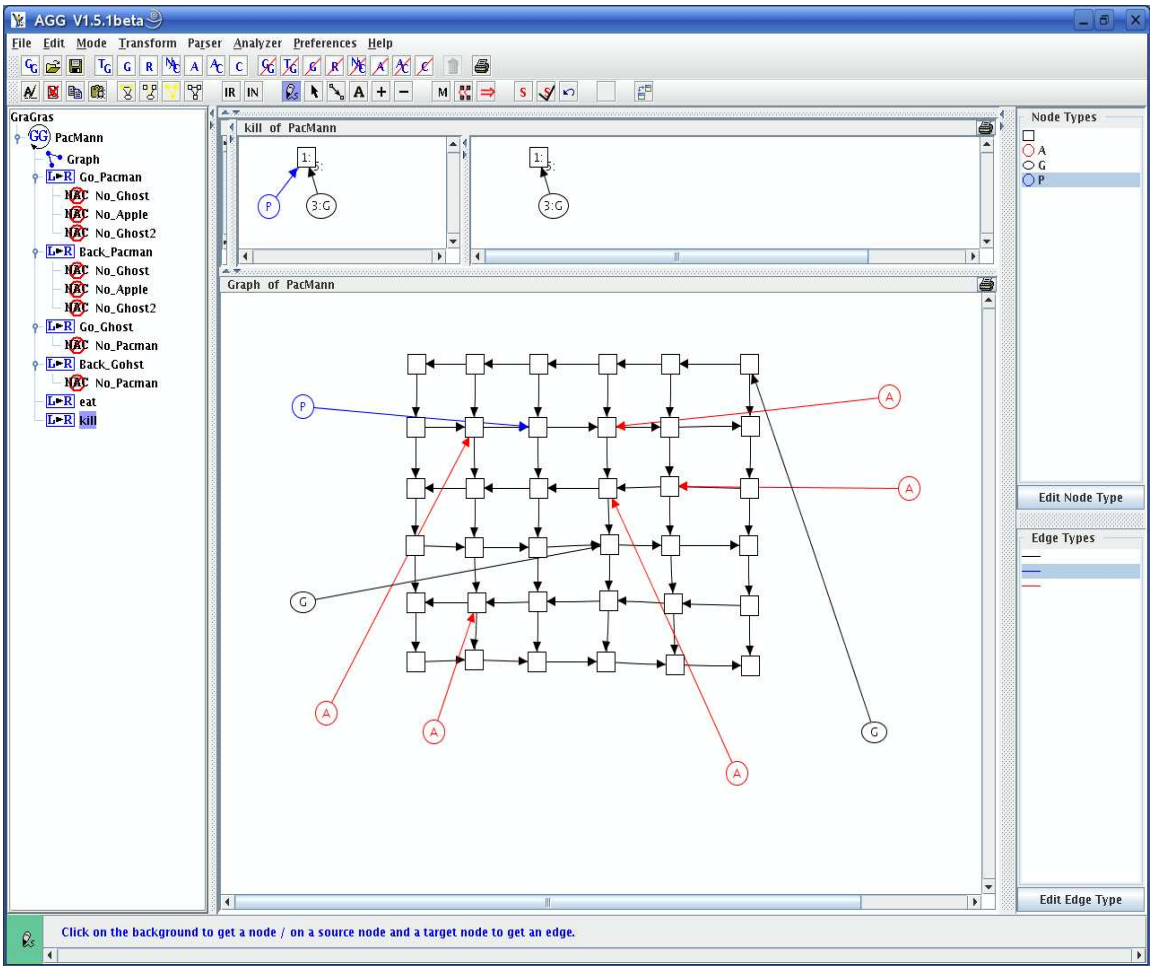


Figure 3.20: AGG Tool Interface

The transformation rules as shown in Fig. 3.20 are; Go_PacMan, to move PacMan continuously forward, $Back\_PacMan$, to move PacMan continuously backward, Go_Ghost, to move also the ghost continuously one step, $Back\_Ghost$ to move the ghost continuously one step backward, eat to eat the apple which suited in the same position where the PacMan is suited and. Finally the $kill$ rule which responsible to delete the PacMan from the Map, if the ghost is at the same position where the PacMan arrived.

The rules in AGG are actually executed by clicking the " $\Rightarrow$ " button in the Toolbar.

As we see also in Fig. 3.20 the tool environment provides graphical editors for graphs and rules and an integrated textual editor for Java expressions. Moreover, visual interpretation and validation is supported. Using AGG we could design complex data structures as graphs which

may be typed by a type graph [AGG]. The system's behavior is specified by graph rules using an $if - then$ description style. Application of a graph rule in AGG transforms the structure graph (host graph), whereas application of several rules sequentially shows the implementations of the required scenario. AGG graphs could be attributed by Java objects and types. Basic data types as well as object classes already available in Java class libraries may be used. Moreover, new Java classes could be included. The graph rules may be attributed by Java expressions which are evaluated during rule applications and the rules may have attribute conditions as boolean Java expressions.

3.7 Result and Discussion

In this chapter, we have shown how graph grammars and graph transformation rules can be defined from the observed behavior of a given scenario. It was interesting to note that (typed) attributed graph transformation systems can be used to represent the behavior of the system (PacMan game). Moreover, we introduced in this chapter some of the formal definitions of graph grammars. Graph transformation mechanisms like SPO or DPO are also illustrated in this chapter. Our case studies (state machines of ATM or state machines of two-phase commit protocol) are also presented here to show that such case studies can be modeled by typed attributed graph transformations. As pointed out in this chapter, the theory of typed attributed graph transformation described here provides a good basis for defining and implementing the model transformations. For a more detailed discussion of attributed graph transformation, we refer to [EE05].

4 Graph Transformation for UML Software Design

4.1 Introduction

UML diagrams are a modeling technique to define the dynamic behavior of a system [GJM91b]. They provide us the following two important parts:

1. The structure part provides diagram techniques to define the static structure of a system including the well known and widely used class diagrams.

2. Dynamic aspects of a system can be specified with diagrams of the behavior part. It allows defining e.g. Activities, Interactions, and state machines by the common diagram types Activity Diagrams, Sequence Diagrams, and state machine Diagrams.

The division of static and dynamic behavior diagram techniques is fundamental in computer science and especially in formal specification techniques. As Sequence Diagrams belong to the second property of the UML they specify parts of the behavior.

The visual language of Sequence Diagrams was defined by the Object Management Group and their contributors. The redefinition of this language using the approach of graph grammar and graph transformation systems is nowadays strongly required and leads to some advantages. It provides on the one hand a strict formal definition for the UML diagrams, on the other hand possibilities for transformation in means of model checking system to verify the required properties of UML diagrams.

Section 4.2 focuses on the UML (Unified Modeling Language) diagrams with an overview. What is UML and how is it used? Section 4.3 illustrates the state machine diagrams and the notations of the UML state machine (state, transition, initial state ...etc). Section 4.4 concentrates on the fundamental issue of executable state machines and the transformation requirements to transform the UML state machines to the executable state machines. A detailed presentation of creating the graph models of UML state machines and the application areas of our case studies is given in section 4.5, 4.6 and 4.7. The transformation rules are discussed in section 4.8.

Finally section 4.9 discusses using the model checker HUGO and SPIN to verify some desired properties. Some Discussion and the result of this chapter are provided in section 4.10.

4.2 Unified Modeling Language

Graphical notations are widely used in software design and development. These notations can greatly help with modeling and representation of software architecture and design [SG95]. Notations like UML [BRJ99] are very good for communicating designs. UML is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software−intensive system. UML consists of two parts: a notation, used to describe a set of diagrams (also called the syntax of the language) and a metamodel (also called the semantics of the language) that specifies the abstract integrated semantics of UML modeling concepts. The UML defines nine diagram types, which allow different aspects (static, behavioral, interaction, and implementation) and properties of a system design to be expressed.

In its current form UML is comprised of two major components: a Meta-model and a notation.

- The Meta-model UML is unique in that it has a standard data representation. This representation is called the metamodel. The meta-model is a description of UML in UML. It describes the objects, attributes, and relationships necessary to represent the concepts of UML within a software application. Interested readers can learn more about the Meta-model by downloading the UML documents from the rational web site [Rat]

- The UML notation is very rich and specify all the required design. It is comprised of two major subdivisions. There is a notation for modeling the static elements of a design such as classes, attributes, and relationships. There is also a notation for modeling the dynamic elements of a design such as objects, messages, and finite state machines.

UML defines nine types of diagrams: class, object, use case, sequence, collaboration, statechart, activity, component, and deployment. we will give the reader a short introduction for every type of UML diagrams:

- **Class Diagrams** The purpose of a class diagram is to depict the classes within a model. In an object oriented application, classes have attributes (member variables), operations (member functions) and relation-ships with other classes. The UML class diagram can depict all these things quite easily. The fundamental element of the class diagram is an icon which represents a class. This icon is shown in figure 4.1.

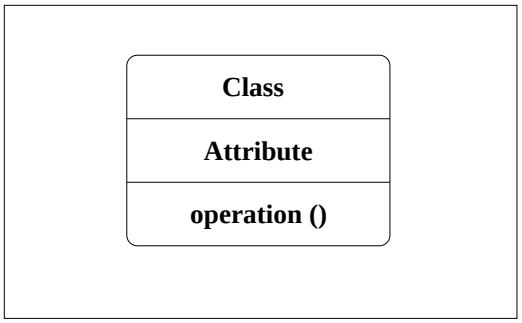


Figure 4.1: The Class Icon

A class icon is simply a rectangle divided into three compartments. The topmost compartment contains the name of the class. The middle compartment contains a list of attributes (member variables), and the bottom compartment contains a list of operations (member functions).

- Package diagrams are a subset of class diagrams, but developers sometimes treat them as a separate technique. Package diagrams organize elements of a system into related groups to minimize dependencies [1] between packages (see Fig. 4.2).

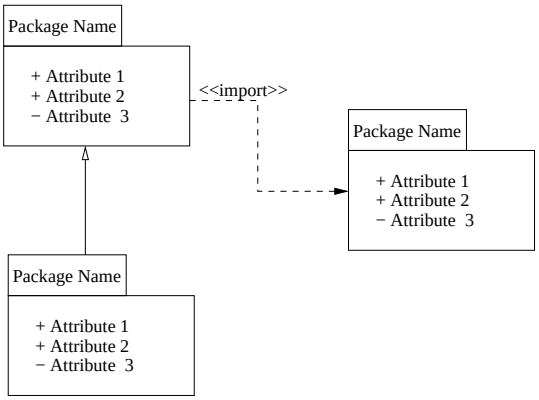


Figure 4.2: Package Diagrams

- Object diagrams describe the static structure of a system at a particular time. They can be used to test class diagrams for accuracy as shown in figure 4.3.

  As with classes, we can list object attributes in a separate compartment. However, unlike classes, object attributes must have values assigned to them.

[1] Dependency defines a relationship in which changes to one package will affect another package. The figure shows importing dependency which is a type of dependency that grants one package access to the contents of another package.

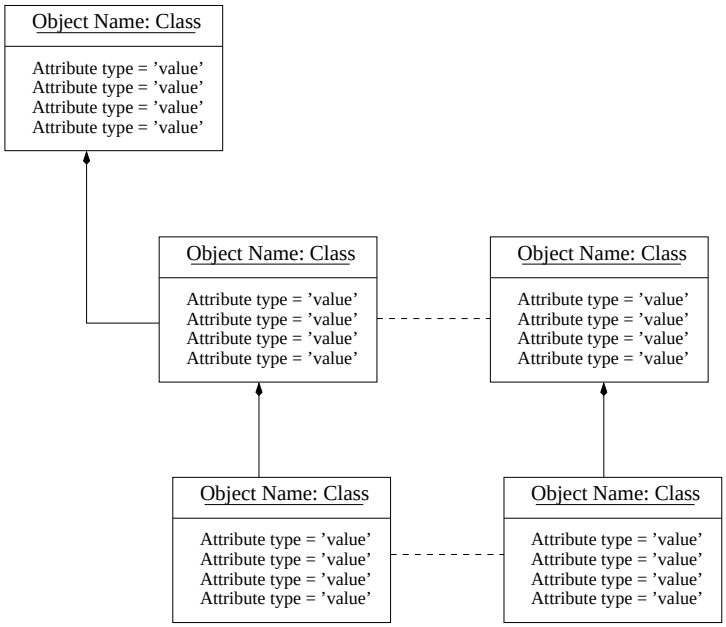


Figure 4.3: Object Diagram

- Use case diagrams model the functionality of system using actors and use cases. We draw use cases using ovals. Label with ovals with verbs represents the system's functions. Actors are the users of a system. When one system is the actor of another system, label the actor system with the actor stereotype.

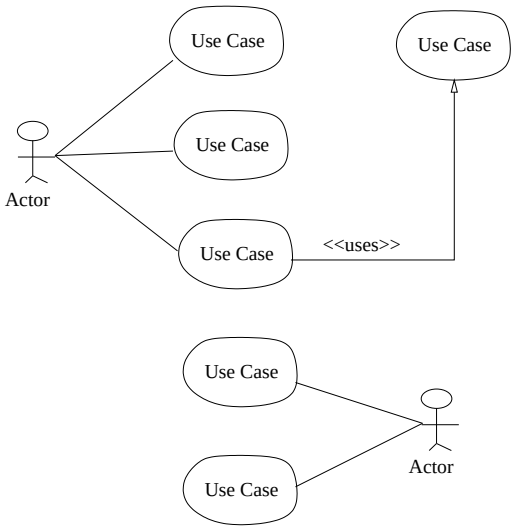


Figure 4.4: Use Case Diagram

- Sequence diagrams describe interactions among classes in terms of an exchange of mes-

sages over time. Messages are arrows that represent communication between objects. We use half-arrowed lines to represent asynchronous messages (see figure 4.5). Asynchronous messages are sent from an object that will not wait for a response from the receiver before continuing its tasks.

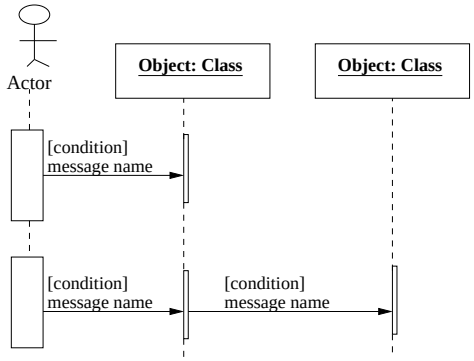


Figure 4.5: Sequence Diagram

- Collaboration diagrams represent interactions between objects as a series of sequenced messages. Collaboration diagrams describe both the static structure and the dynamic behavior of a system. Unlike sequence diagrams, collaboration diagrams do not have an

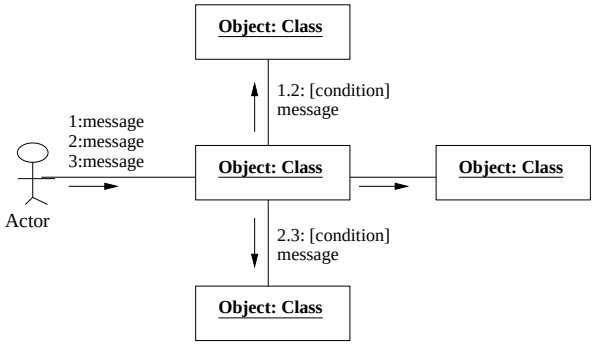


Figure 4.6: collaboration Diagram

explicit way to denote time and instead number messages in order of execution as shown in figure 4.6. Sequence numbering can become nested using the Dewey decimal system. For example, nested messages under the first message are labeled 1.1, 1.2, 1.3, and so on.

- Statechart or state machine diagrams describe the dynamic behavior of a system in response to external stimuli. State machine diagrams are especially useful in modeling reactive objects whose states are triggered by specific events. States represent situations

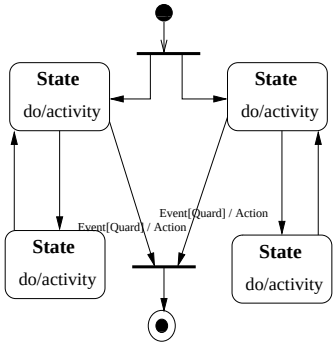


Figure 4.7: State Machine Diagram

during the life of an object. We can easily illustrate a state in MagicDraw by using a rectangle with rounded corners. Transition shown as solid arrow represents the path between different states of an object. The transition is labeled with the event that triggered it and the action that results from it as shown in figure 4.7.

- Component diagrams describe the organization of physical software components, including source code, run-time (binary) code, and executables.

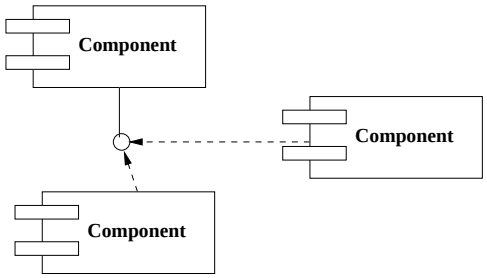


Figure 4.8: Component Diagram

We draw dependencies among components using dashed arrows (see figure 4.8).

- Deployment diagrams depict the physical resources in a system, including nodes, components, and connections. A node is a physical resource that executes code components. Association refers to a physical connection between nodes, such as Ethernet.

  As shown in figure 4.9 we place components inside the node that deploys them.

For the purpose of this research we concentrate our work on the UML state machine diagrams, which is a specification that describe all possible behaviors of some dynamic model element. Behavior is modeled as a traversal of a graph of state nodes interconnected by one or more

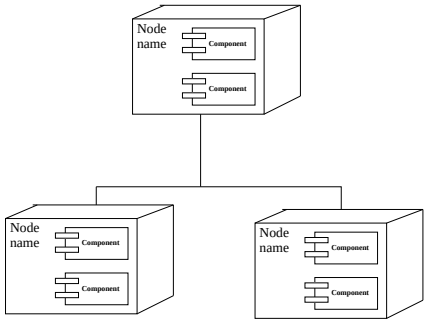


Figure 4.9: Deployment Diagram

joined transition arcs that are triggered by the dispatching of series of event instances. During this traversal, the state machine executes series of actions associated with various elements of it.

UML 2.0 incorporates an action semantics, which adds to UML the syntax and semantics of executable actions and procedures [Gro03b]. Action semantics refers to the ability to formally describe actions that can be analyzed by a computer and executed. Formal actions make models executable, also referred to as *model simulation*. Action semantics is a partial metamodel integrated in the global UML metamodel. It allows the specification of many kinds of actions, such as computational algorithms to be applied to data, as well as reactive and concurrent behavior with asynchronous and synchronous communication. Therefor, action semantics are applicable to both information and embedded systems. The key concept is *action*. An action corresponds to a manipulation of the object model; it can modify it or just read it.

There are several modeling tools for designing UML diagrams [Qua98, RVR+99, Poe04, Mag04]. ArgoUML is an open source Java-based UML tool [RVR+99]. It supports most of the nine standard UML diagrams, it has also the ability of reverse engineering compiled Java code and generating UML diagrams for the code. Commercial tools are e.g. Rose [Qua98], Together [Tog04], Poseidon [Poe04] and MagicDraw [Mag04]. Among them, MagicDraw is a visual UML modeling and CASE tool with teamwork support. MagicDraw contains a handy UML editor, a powerful code engineering tool, UML model reporting facilities, a custom OO model generator, a team modeling tool, and a database modeling tool.

4.3 UML State Machines

UML state machine diagrams describe the behavior of a class over time through illustrations of the states and transitions of a single object progressing through its lifetime. State machine diagrams are a traditional object-oriented way to show behavior and to document how an object

responds to events, including internal and external stimuli.

In this section we discuss the two important parts of UML state machine; states and the transitions.

4.3.1 States

A state is a condition or situation during the life of an object, which it satisfies some condition, performs some activity, or waits for some event. An object remains in a state for a finite amount of time. For example, the ATM machine in the bank might be in any of six states **CardEnry** (waiting for somebody to enter his Bank Card), **PINEntry** (Waiting for customer to enter his pin number), **Verification** (the ATM determine where the card and the pin number are satisfied) **AmountEntry** (Waiting for the customer to enter the required amount), **GivingMoney** (the ATM machine will give the required money to the customer) and the **ReturningCard** state (Returning the card to the customer).

When an object's state machine is in a given state, the object is said to be in that state. For example, an instance of ATM might be CardEntry or perhaps ReturningCard. A state has several properties Fig 4.10. The **Name** of the state which is a textual string that distinguishes the state from other states; a state may be anonymous, meaning that it has no name. **Entry/Exit** actions which are executed on entering and exiting the state, respectively. **Internal transitions** are transitions that are handled without causing a change in state. The **Substates** is the nested structure of a state, involving disjoint (sequentially active) or concurrent (concurrently active) substates. and finally the **Deferred events** which are a list of events that are not handled in that state but, rather, are postponed and queued for handling by the object in another state.

Figure 4.10: Properties of the State in UML State Machine

Initial and Final States There are two special states that may be defined for an object's state machine. First, the initial state, which indicates the default starting place for the same machine or substate. An initial state is represented as a filled black circle. Second, the final state, which indicates that the execution of the state machine or the enclosing state has been completed. A final state is represented as a filled black circle surrounded by an unfilled circle as shown in figure 4.11.

Figure 4.11: Initial and final States in UML State Machine

4.3.2 Transitions

A transition is a relationship between two states indicating that an object in the first state will perform certain actions and enter the second state when a specified event occurs and specified conditions are satisfied. On such a change of state, the transition is said to fire, it is said to be in the target state. For example, in our case study the state machines of ATM the transition from the *Verification* state to *ReturningCard* state will be done when an event such as abort occurs.

The properties of transitions are as follows; the **source state** which mean if an object is in the source state, an outgoing transition may fire when the object receives the trigger event of the transition and if the guard condition, if any, is satisfied. The **Event trigger** is an event whose reception by the object in the source state makes the transition eligible to fire, providing its guard condition is satisfied. The **Guard condition** is a boolean expression evaluated when the transition is triggered by the reception of the event trigger; if the expression evaluates True, the transition is eligible to fire; if the expression evaluates False, the transition does not fire and if there is no other transition that could be triggered by that same event, the event is lost. the **Action** is an executable atomic computation that may directly act on the object that owns the same machine, and indirectly on other objects that are visible to the object. Finally the **Target state** which is a state that is active after the completion of the transition. As figure 4.12 shows, a

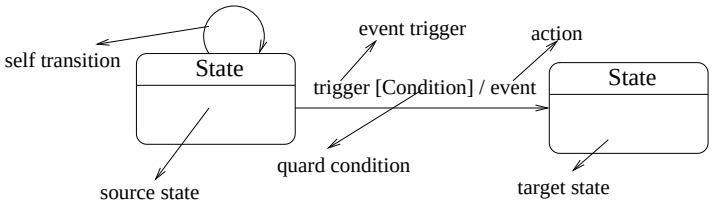


Figure 4.12: The Transition in UML State Machine

transition is rendered as a solid directed line from the source to the target state. A self−transition is a transition whose source and target states are the same.

Note: A transition may have multiple sources (in which case, it represents a join from multiple concurrent states) as well as multiple targets (in which case, it represents a fork to multiple concurrent states).

4.3.3 State Machines of 2PC-Protocol

We have already discussed in the first chapter section 2.2.6, the two-phase commit protocol which is a distributed algorithm which lets all sites in a distributed system agree to commit a transaction. The protocol results in either all nodes committing the transaction or aborting, even in the case of site failures and message losses. The two phases of the algorithm are broken into the COMMIT−REQUEST phase, where the **coordinator** attempts to prepare all the **participants**, and the COMMIT phase, where the **coordinator** completes the transactions at all **participants**.

In this section we introduce the two-phase protocol as UML state machines diagrams as follows:

Coordinator State Machine At the beginning of the interaction between the client and the server, the coordinator is in the preparing transaction state (figure 4.13 at the left), it waits for responses from each of the participant. If any participant responds vote_abort() then the transaction must be aborted and proceed to the Abort state. In this case the coordinator sends the *Participant.Abort()* message to each participant. If all participants respond *vote_commit()* then the

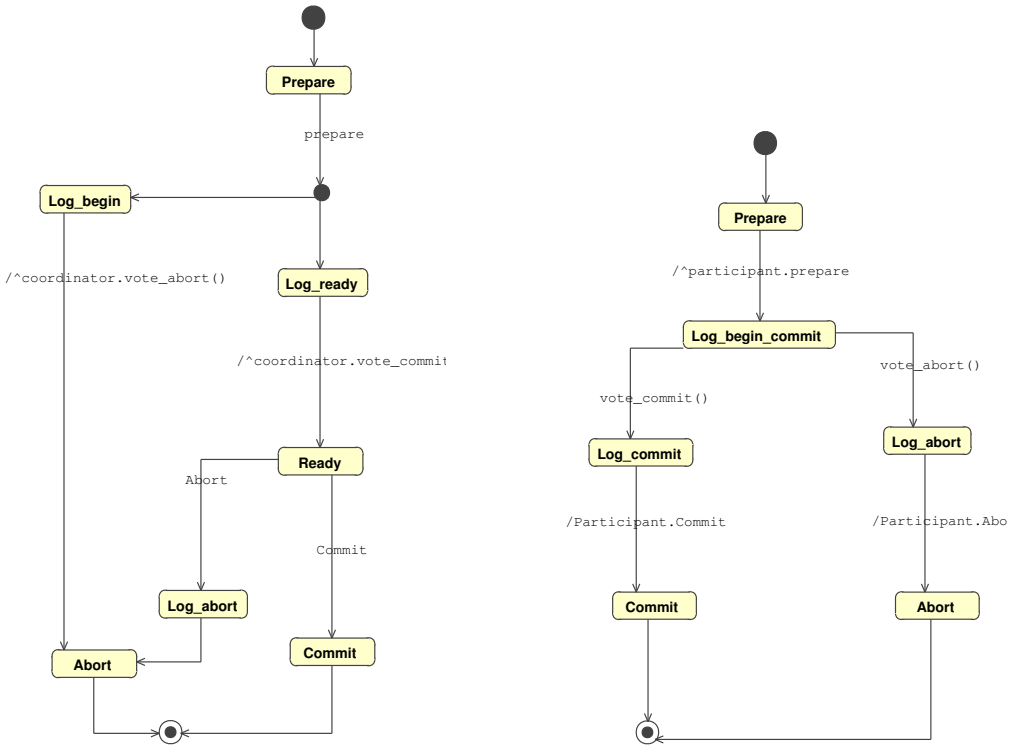


Figure 4.13: UML State Machines for Two-Phase Commit Protocol

transaction may be committed, and proceed to state *Commit*, then sends the *participant_Commit* message to all participants.

Participant State Machine If a prepare message is received from the coordinator state machine (figure 4.13 at the right), the participant must be ready to make his decision, either he responds with *yes* and go to the next state *Log_Ready*, or it decides with *no* and go to the state *log_begin*. If the participant is in the *Log_Ready* state, it sends the message *Coordinator.vote_commit* to the coordinator state machine and be in the state Ready waiting for a message from the coordinator state machine. If the participant is in the *Log_begin* state, it sends the signal *Coordinator.vote_abort* to the coordinator state machine, and it removes to the state *Abort*.

In the *Ready* state if an *abort* message is received then it finishes the transition and go to the *Abort* state. If a *commit* message is received then the transaction is prepared for committal and go to the *Commit* state.

4.4 Executable State Machine

Executable UML is a major innovation in the field of software development. They use it to produce a comprehensive and understandable model of a solution independent of the organization of the software implementation. Executable UML models are nowadays gaining interest in embedded systems design. This domain is strongly devoted to the modeling of reactive behavior using stateChart variants. In this context, the direct execution of UML state machines is an interesting alternative to native code generation approaches since it significantly increases portability. In this section we will describe, how the UML state machines can be represented using a small set of states enables an efficient execution called executable state machines [SM05].

In this section we illustrate the representation of the UML state machines as graph models and the transformation techniques that we used to transform the UML state machines into Executable state machines.

4.4.1 Executable state machines

The Executable State Machines (ESMs) are equivalent to simpler state machines, which are more suitable for efficient execution [SM05]. We can consider that ESMs are a subset of UML state machines. Figure 4.14 shows the model of ESMs. The major changes are the removal of composite states and the limitation to only the doActivity. After removing the hierarchy, the entry and exit activities become meaningless anyway.

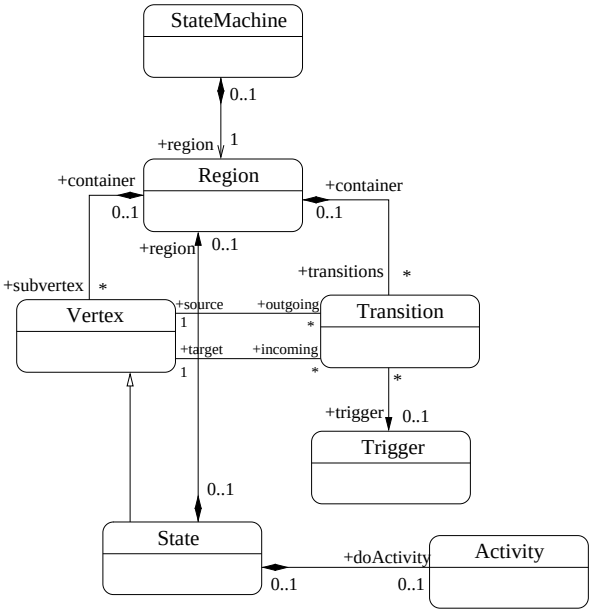


Figure 4.14: Executable State Machine (ESM) Model

This approach of executable state machines is based on using a fully featured of UML state machine to describe the behavior of an operation [SM05]. In this section we will illustrate the mechanism to get the executable state machine from the UML state machines. The essential different between UML state machines and executable state machines are eliminating composite states, removing entry/exit activities from states, and effect and guard from transition

Initial Model We assume that the initial model is valid and contains complete sets of initial and final states for at all levels. Composite states must contain at least one substate. The strategy for eliminating composite states is to replace all transitions on the composite state with equivalent transitions to its substates. Ones the transitions are replaced, we move the substates from the lowest level one level up and push the according entry/exit behavior into the transition effect. To remove incoming transitions on a composite state, we replace those with direct transitions to the state marked by the initial pseudostate and remove that pseudostate (see figure 4.15). Since it is the only pseudostate in this approach, this also ensures that the region afterwards only contains states and final states.

Super State Transition The step in 4.16 transfer all transitions from a super state to the contained states. Before doing this step, all outgoing transitions of each state are marked for copy and each final state is replaced by a normal state and has all completion transitions (these may have different guards) from the super state marked to be copied to that state. It is important

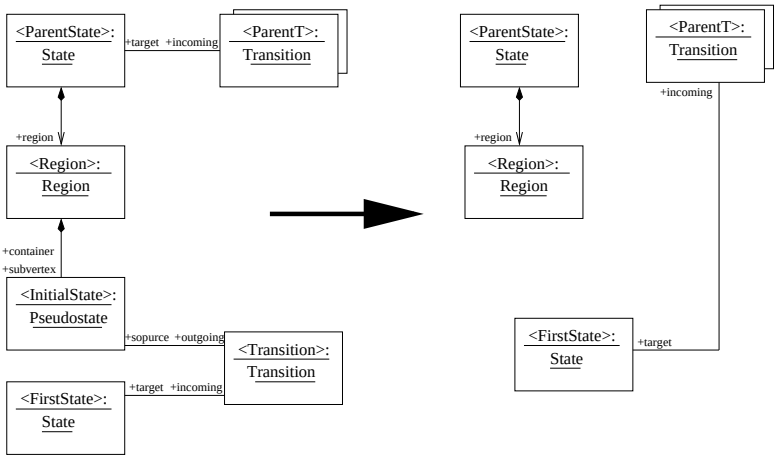


Figure 4.15: Initial Substate

to notice, that at this point all Vertices in the Region are exactly states. The second step in 4.16 replicate such a marked transition for a state and marks it as copied. The third step deletes transitions that have been copied to all relevant substates.

Creating New States As ESMs do not support transition effects, we have to move these effects from the transitions into states. Thus, we have to encode the effect into the target state of each transition. As that target state is likely to have several incoming transitions, it is necessary to replicate the state for each incoming transition. An additional state will be introduced to apply these mechanisms to transition to a final state. The rule in figure 4.17 is for that purpose. A copy of the target state including deferred triggers and outgoing transitions is made using the copy mechanism for transition. A new doActivity is created from the transition effect and the state doActivity. Again, if the state has none, a similar simpler strategy can be used, but has been omitted here.

After exhaustive application of this strategy considered so far, we yield a flat state machine where states have only a doActivity and transitions have no effect. However, a state still may have deferred triggers.

We finally have to eliminate the guards by replacing all outgoing transition to a new state evaluating the respective guards in the doActivity. This doActivity then may directly trigger the according transition. If none of the guards evaluates to true, we go directly to a new state embedding the same set of outgoing transitions we installed on the original state.

Example In this example, we will create from a default UML state machine a new executable state machines. Figure 4.18 shows the original UML state machine, which has simple states with

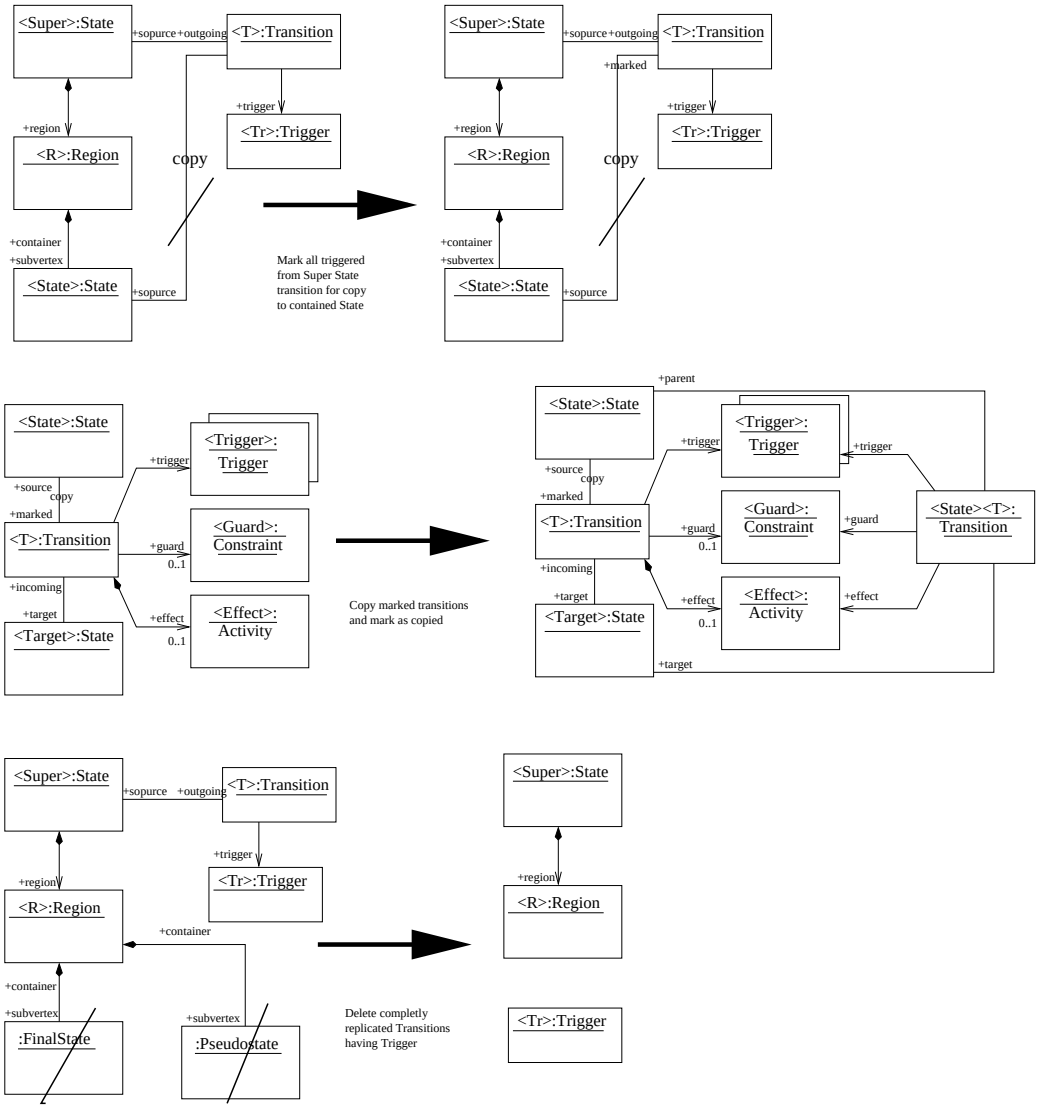


Figure 4.16: Move outgoing Superstate Transitions to Substates

activities and a composition state that include simple, initial and final states. we will explain the creating of executable state machines in three parts:

- The first part shows the state machine (figure 4.19) after removing the initial and final state and all the transition from the composite state
- The next part shows the state machine after the removal of the composite state where the entry and exit activities have been moved to the transition effects for all inter-level transitions from the composite state as shown in figure 4.20. Furthermore, trigger conflicts have been resolved resulting in the removal of the composite state transition from C to A and the extended guard condition of the transition from D to A.

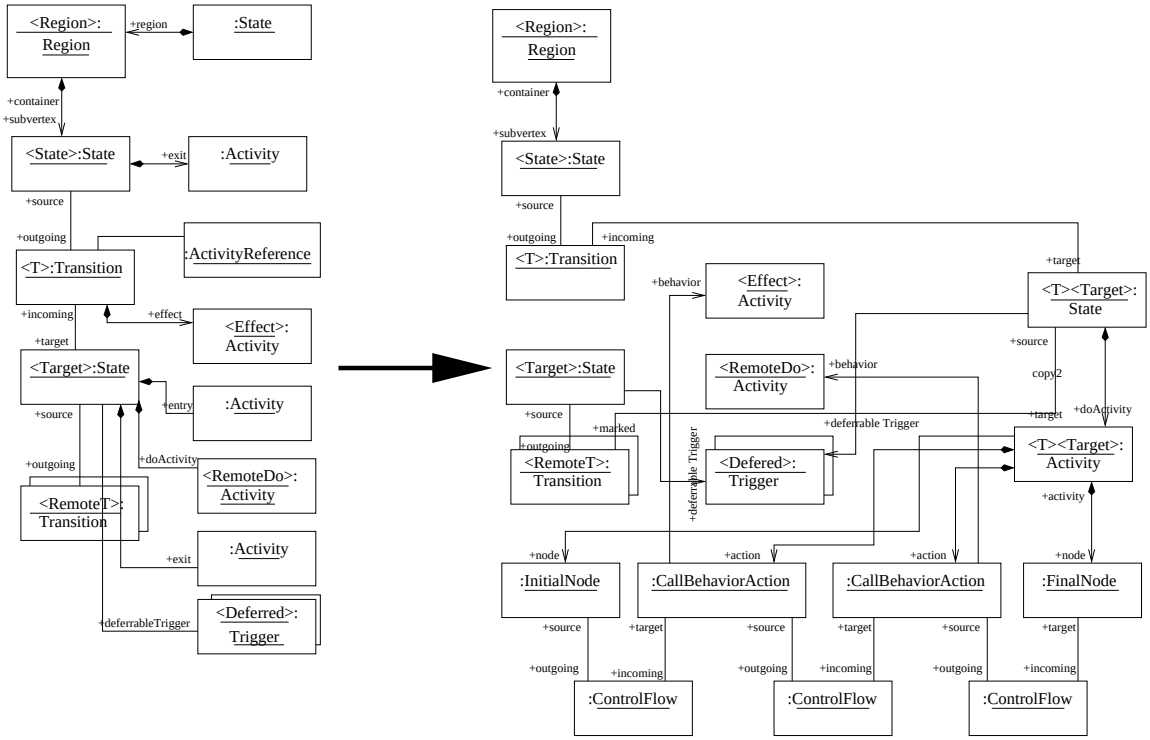


Figure 4.17: Move Transition Effect to new State

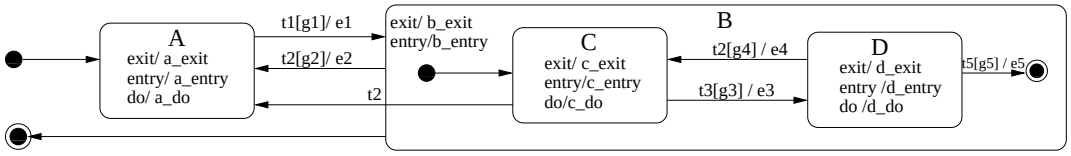


Figure 4.18: The Original Form of UML State Machine

- The last part (figure 4.21) shows the state machine after moving the transitions effect into states by introducing new states for every incoming transition.

4.5 Graph Models of UML State Machines

UML state machines are popular and very useful to specify the dynamic components of software design. Graph transformation constitutes a well-studied area with many theoretical results and practical application domains. In this section, the representation of UML state machines as graph models based on graph grammars and graph transformation systems is proposed. In more detail, we represent the UML state machine as a graph model in such a way that the properties of UML state machines are satisfied and the transitions between the states in the state machines must

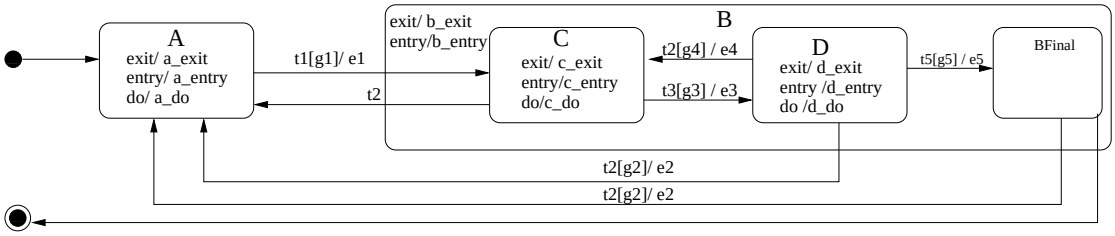


Figure 4.19: Removing Initial, Final, and Transition from Composite State

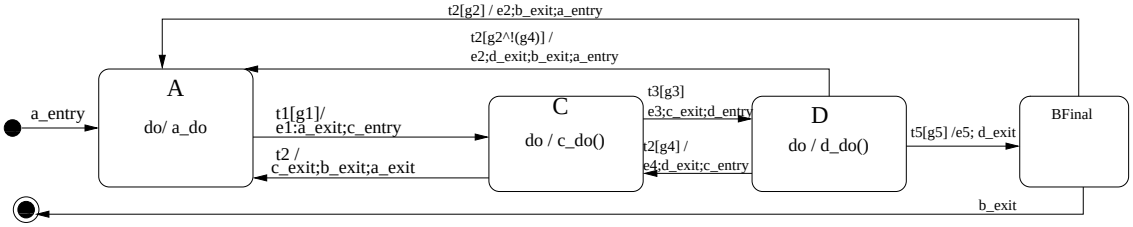


Figure 4.20: Removing Initial, Final, and Transition from Composite State

correspond to the application of the graph models. We illustrate in this section the definition of type graph and graph model of UML state machines. Our case studies state machines of ATM and state machines of two-phase commit protocol are also represented as a graph models based on graph transformation system.

4.5.1 Type Graph of UML State Machine

In the second chapter we have already described how we create from the given scenario the type graph and the graph models. If we consider UML state machine as a scenario to be represented as a graph models, we can use the same strategy as we defined the graph models of PacMan scenario. In order to define the type graph of UML state machines, we must researched the concepts of UML state machines. The UML state machines consist of the following concepts: transitions, states, triggers, events, constraints, entries, exists, doActivities ...etc. We represent the states in UML state machines as nodes in the type graph, and each activity of the state as also node in the type graph (entry activity as node in the type graph, exit activity as node in the type graph, and doActivity as also node in the type graph). We can define the activities of the states in UML state machine as nodes in the type graph and the edges between these nodes represent the relationships between the states and their activities as shown in figure 4.22.

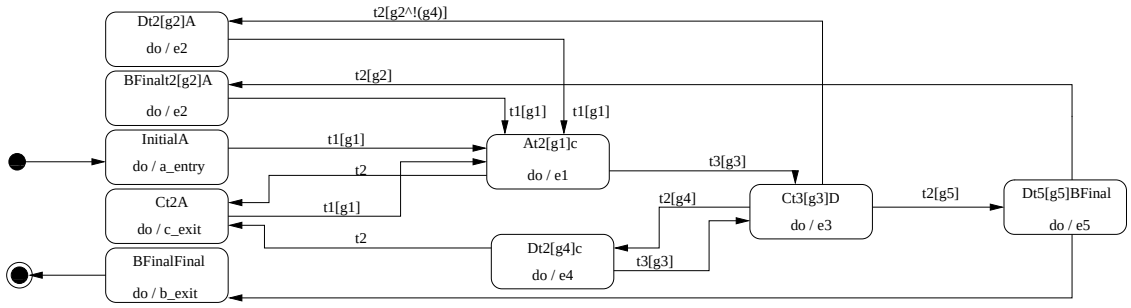


Figure 4.21: Creating New State for every Incoming Transition

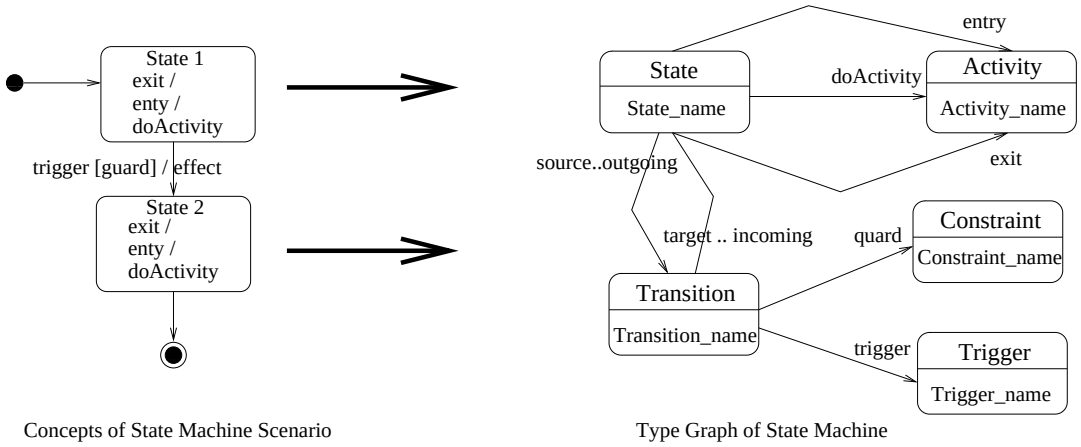


Figure 4.22: Type Graph of the State Machine

4.5.2 Graph Model of UML State Machine

We have already defined the type graph of UML state machine. Figure 4.23 shows the graph model of two states and one transition between them. The first state *state* 1 in this figure is represented as four nodes in the graph model, three of them are represented for the activities of the state and the fourth ones is for the state that consists of attribute denotes to the name of the state. The second state *state* 2 is also represented in the same manner. The transition between *state* 1 and *state* 2 is represented as four nodes in the graph model, three of them represent the actions of the transition (trigger, guard and effect) and the fourth one denotes to the transition name. That is, in this way we can represent every state machine as graph model. The number of the nodes and the edges depend on the size UML state machines.

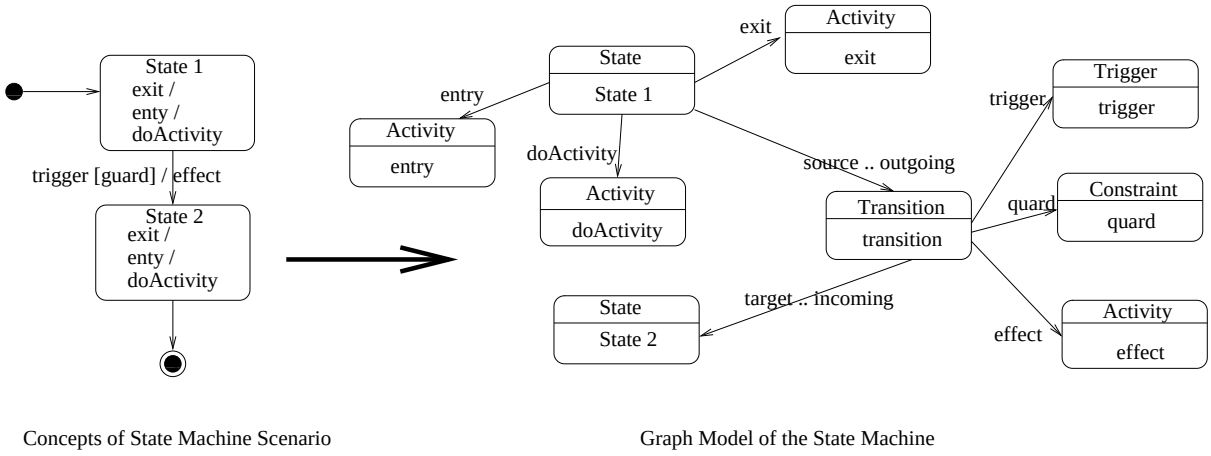


Figure 4.23: Type Graph of the State Machine

4.6 Graph Model of ATM state machines

We represent the state machines of the ATM case study in the same manner as illustrated in the previous section.

Suppose we have two state machines of ATM as we already illustrated in chapter 2 (see figure 2.13 and figure 2.14). We discussed in the scenario of UML state machine that every state is represented as four nodes and every transition is represented also as four nodes. In this case study, we have also to represent the initial state, the final state and the composite state. If the state is initial state, we add new special node to denote to the initial state. If the state is final one, we also add new special node to denote to the final state. If our case study has any composite state, we should refer to every state in the composite state with a special edge between the composite state and it's internal states. The ATM graph model is shown in figure 4.24

In this graph model we represent the initial state in ATM state machine as new node called *Pseudostate* with two string attributes. One of them represents the name of the Pseudostate *Ps_name* and the other is for the type of this Pseudostate *Ps_kind*. The same strategy is implemented for the final state in ATM state machines.

The edge which labeled with "source .. outgoing" denotes to the outgoing transition in the UML state machine, whereas the edge that labeled with "target .. incoming" denote to the incoming transition in the UML state machine.

There is a special edge for the composite state labeled with "container .. subvertex", this edge is represented to denote to the composite state and it's internal states.

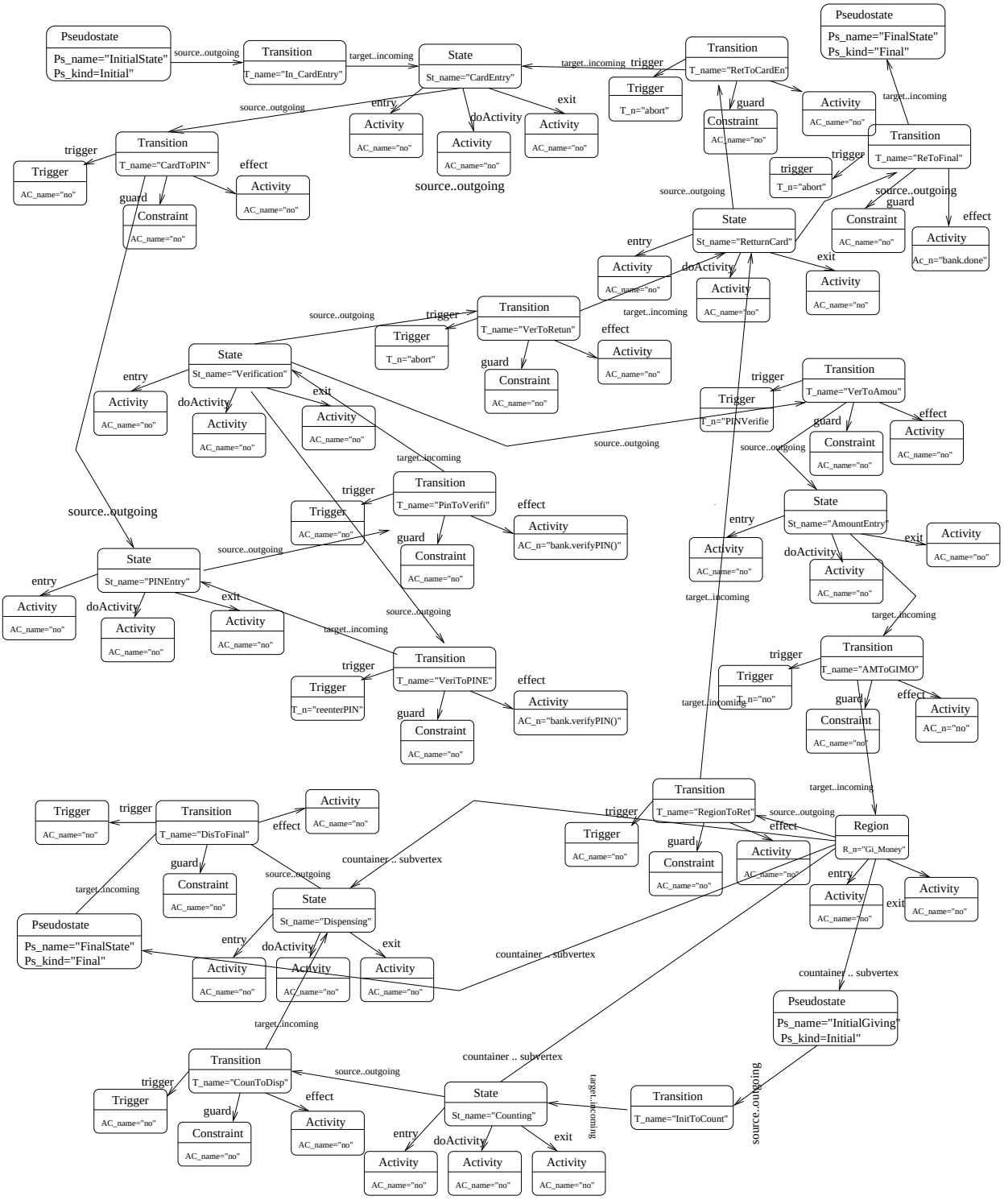


Figure 4.24: The Graph Model of ATM State Machine

4.7 Graph Transformation of Executable State Machines

In this section we illustrate the transformation from UML state machine to an executable state machine using the strategy that declared in chapter 3 (see section 3.3) and give an overview about the tool AGG (Attributed Graph Grammar).

First, we have to define the state machine as graph model, as a running example for this section we choose our case study ATM state machines. We have already defined the graph model for the ATM state machines as illustrated in section 4.6. The transformation modifies the static structure of the graph model; One transformation for example for removing the composite state in the graph model of ATM state machine, the other transformation is for removing the initial or final state.

In order to implement this transformation as a graph transformation, we need to implement our graph models using the AGG system. AGG graph transformation rules consist of a left-hand and a right-hand side graph, a mapping morphism between nodes (and edges) on both sides, and a set of ”negative application condition ”(NAC). A Screenshot of the project shows the working graph and the rules which are present. In the upper right, the selected rule can be found, and in the lower right, the actual working graph is shown. Rules having a NAC are displayed by three graphs (NAC, left-hand side, right-hand side), rules without a NAC are displayed by two graphs. Number in front of node labels represent the morphism of the rule. The working graph in figure 4.25 corresponds to the graph model of ATM as explained in figure 4.24.

4.7.1 Executable State Machine of ATM

We represented the ATM state machines as graph model, so we can transform the state machines of ATM to an executable form using the concepts of executable state machine. The new form of ATM is shown in figure 4.26. This figure shows us that every state has just one activity, and every transition has also one trigger. This new model of executable state machine of ATM is correspond to the UML state machine model of ATM.

4.7.2 Transformation Rules

In order to transform the graph model of ATM state machine into new graph model represents the executable state machine of ATM, we need about 30 rules as shown in figure 4.25 . In this section we illustrate some of them. The first transformation rule which called ”Trans-

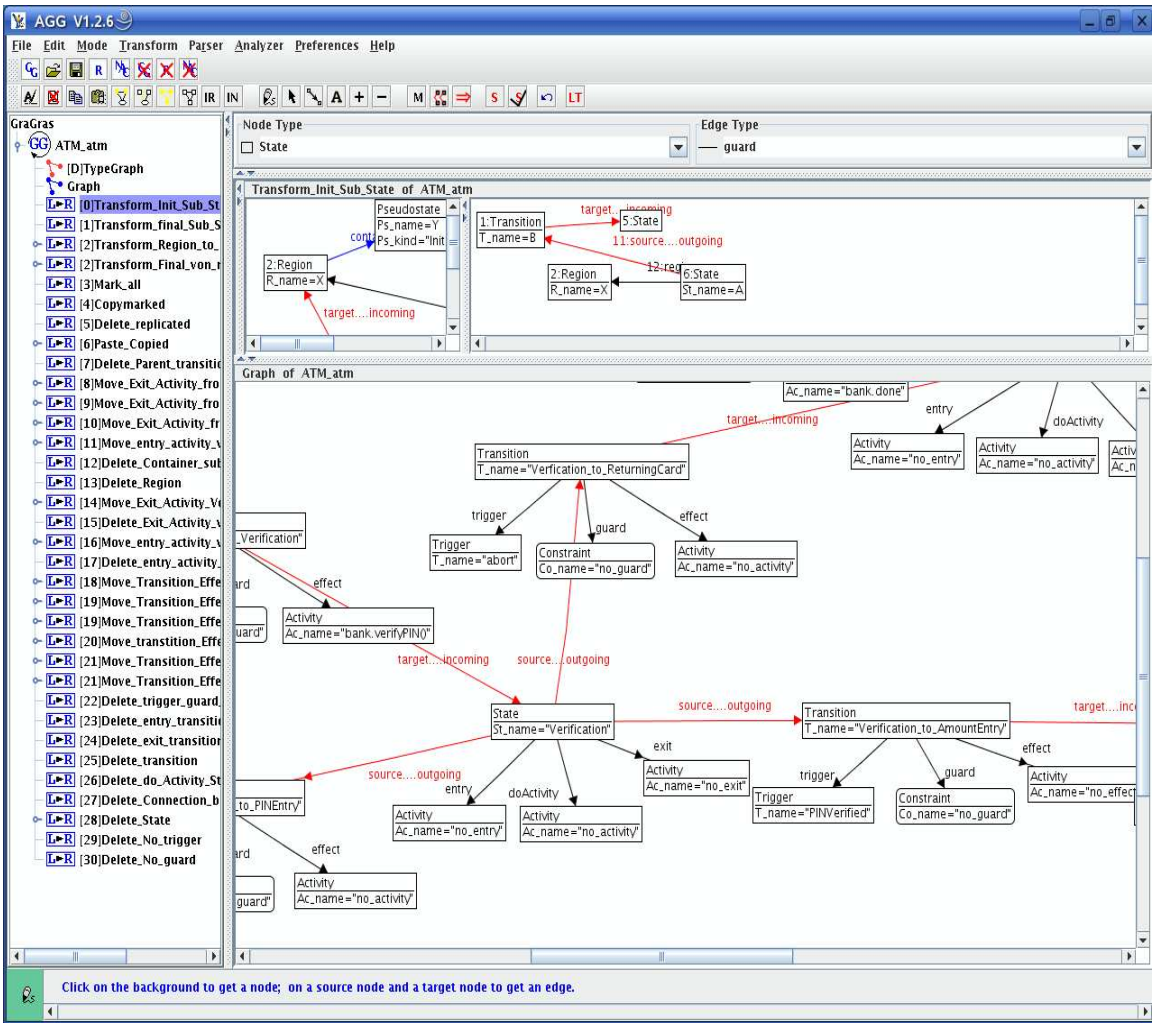


Figure 4.25: Executable Graph Model of ATM State Machine

form_Init_Sub_State" as shown in figure 4.27 deletes the initial state in the composite state and create a new transition from the super state to every internal states in composite states. The second rule "Transform_Final_Sub_State" is to remove the final state from the Composite state.

To apply the rule, the rule name from left-hand side must be matched in the working graph. This can be done through AGGs 'map' mode, available through the context menu. The rule is actually executed by clicking the "⇒" button in the Toolbar.

The rule number 14 is an interesting one (see Fig. 4.28), it is responsible for moving the exit activity from it's state to the outgoing transition. This node has a negative application condition (NAC) "No_exit_befor", the NAC is for checking if the state has already move the exit activity or not.

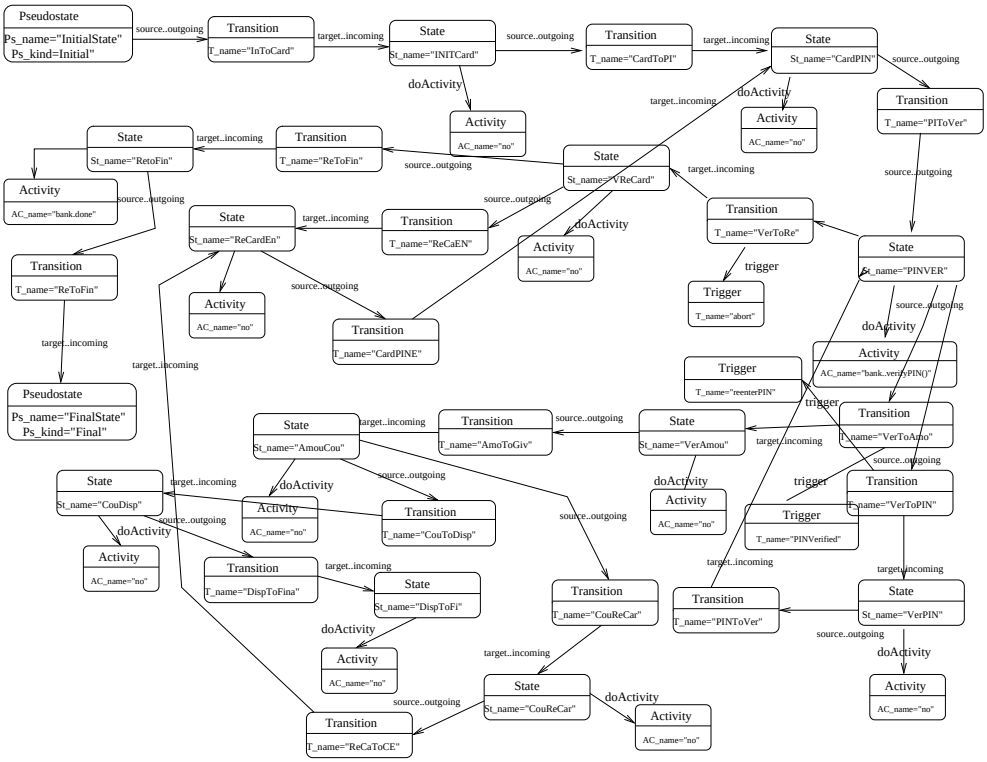


Figure 4.26: Executable Graph Model of ATM State Machine

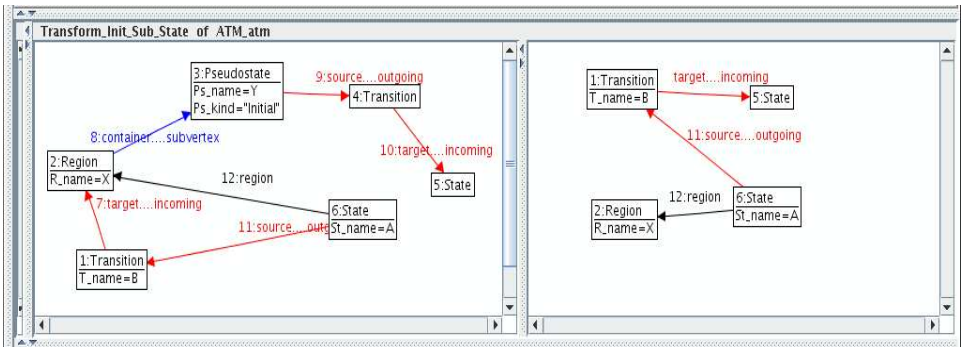


Figure 4.27: Removing Initial State from Composite State

After transforming the corresponding nodes to their new places, we begin to remove the transition and the states that are meaningless. Figure 4.29 shows a rule to delete the transition between two states. We are actually allowed to apply this rule after 25 executions of the previous rules.

If we apply all the 30 transformation rules shown in Fig 4.25, we will have the graph model of executable state machine. Figure 4.30 shows the final graph after transforming the UML state machine of ATM in graph model into new graph models represent the ESMs approaches.

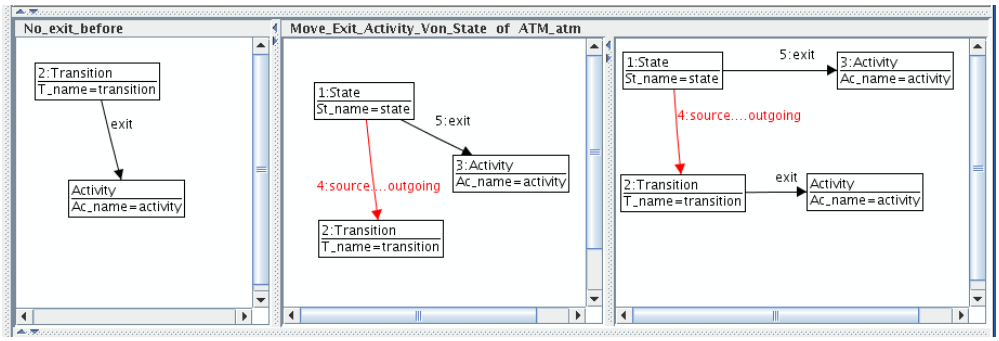


Figure 4.28: Removing the Exit Activity from State

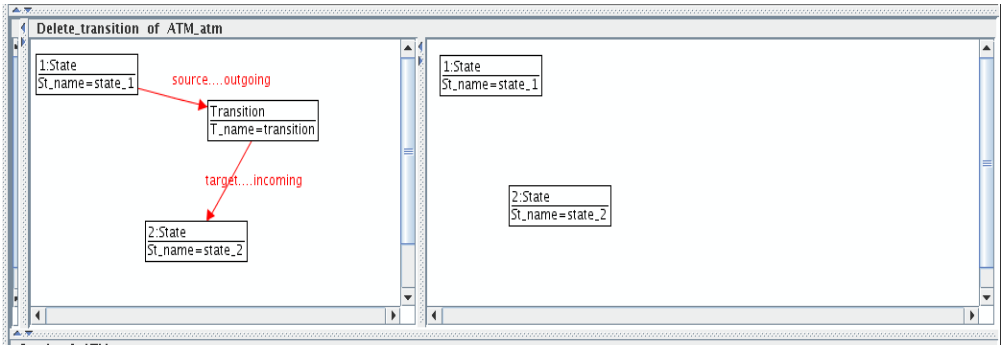


Figure 4.29: Removing Transition between States

4.8 Two-Phase Commit Protocol (2PC)

The state machines of two-phase commit protocol are shown in figure 4.13. We design the second case study as a state machine graph model in three steps:

- Creating the graph model of two-phase commit state machine using our experience in defining the graph model of state machine.

- We use the pre-defines rules to transform the two-phase commit graph model to new executable state machines graph model. We have already the pre-defined rules (graph grammars) in AGG transformation tool.

- We translate the graph model of executable state machine (Manuel or Automatic) to UML state machine. We usually use the MagicDraw tool to see the new executable Form of UML state machine.

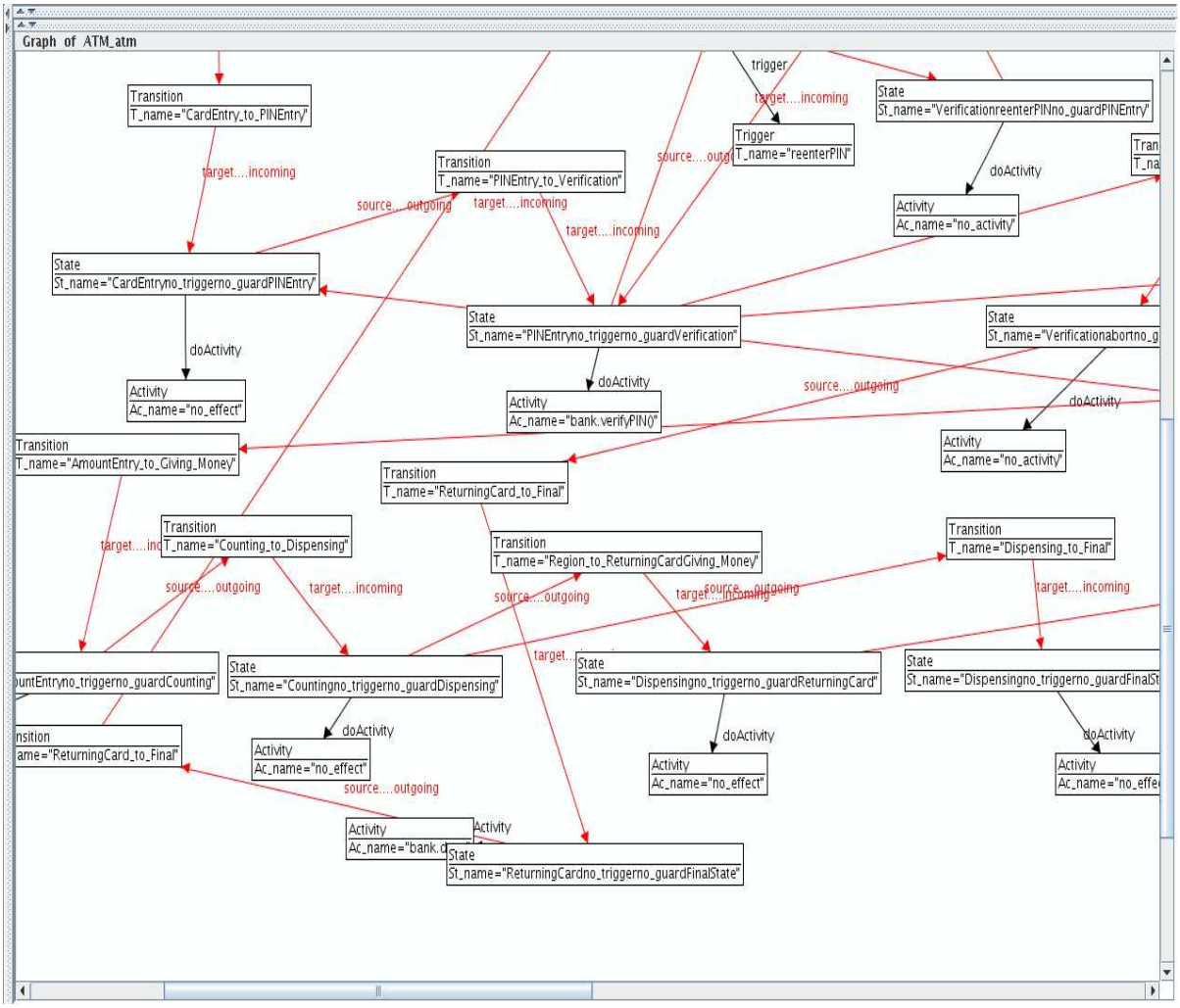


Figure 4.30: Model Graph of ESMs in AGG

4.8.1 Graph Model of 2PC

We create the graph model of two-phase commit protocol using the same strategy which declared in chapter 3 section 3.2 to represent the state machine as a graph model. The graph model of two-phase commit protocol is shown in figure 4.31 consisting of nodes and edges between the nodes. Node can be state node like the state node of ”prepare” or pseudo state node like ”InitialNode”. The state nodes are connected with other kind of nodes represent the entry, exit and doActivity of the state node. Nodes that represent transition between state nodes are connected also with three nodes that represent trigger, guard and effect nodes.

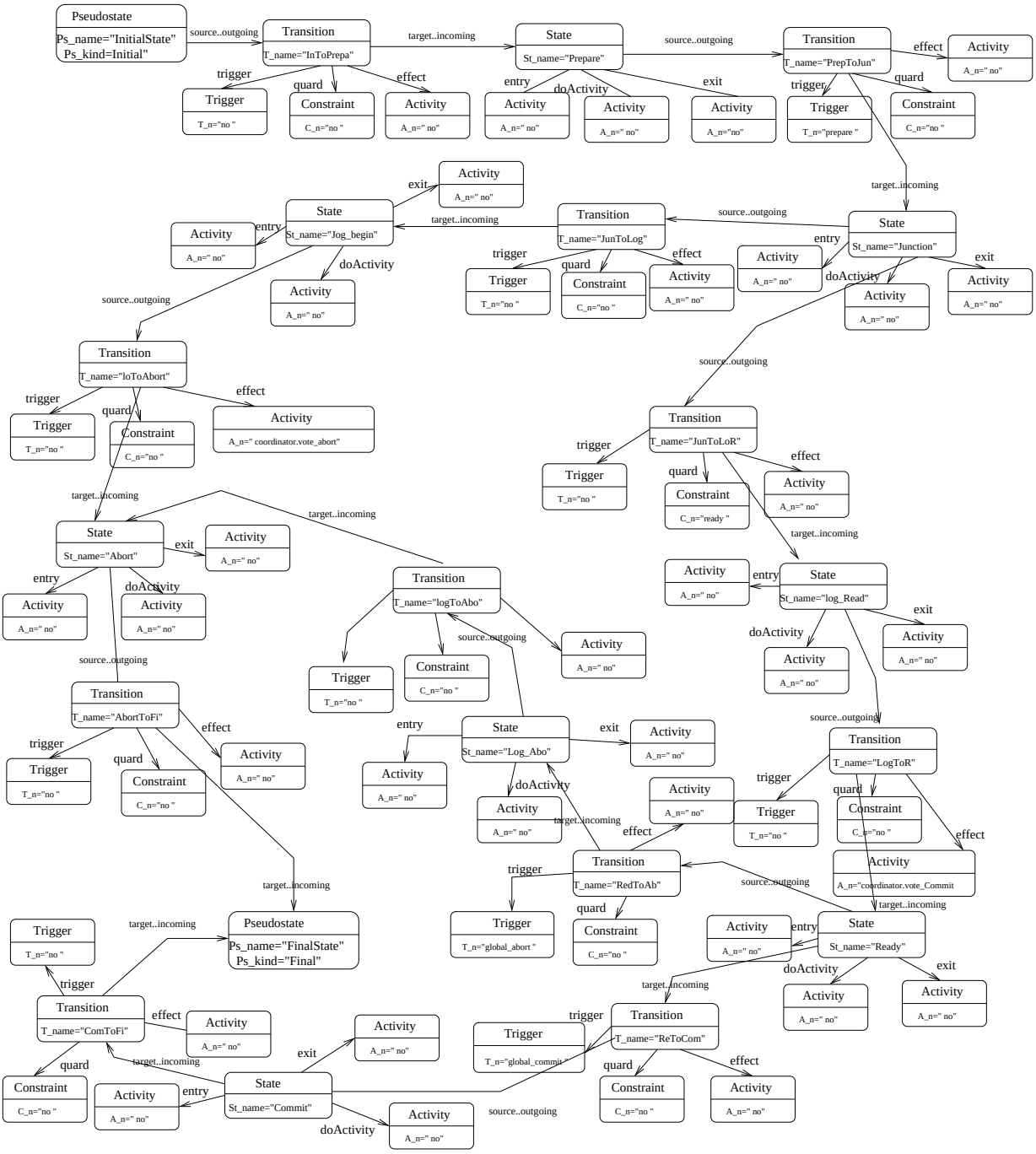


Figure 4.31: Graph Model of 2PC Protocol

4.8.2 Executable State Machines of 2PC

In this section we describe our new graph model of 2PC. The graph which shown in 4.32 is the graph model of executable form for the two-phase commit protocol. Every state node in this graph model is connected with one activity node, whereas the transition nodes are connected with two nodes represent the trigger and the guard node. This graph has no representation for the pseudo, final and composite states. The graph model is created after running the rules using the graph transformation tool AGG.

4.9 Verifying Results using HUGO

To verify the graph model using HUGO, we first need to compile the graph model into UML state machine, then we use HUGO to verify the required properties. Figure 4.33 shows the executable UML state machine diagram after compiling the graph model of ATM state machine. We notice in this figure that the diagram contains one Initial Pseudostate and one final Pseudostate. The simple states in this diagram have one doActivitis, whereas the transition between simple states have triggers and guards.

HUGO supports us the correctness of the UML state machines diagrams (e.g no deadlock in the diagrams), the model checker SPIN is called upon to verify the model against the desired behavior. We use HUGO to verify the state machines of ATM and the state machines of two-phase commit protocol before and after running the transformation rules to create the executable state machines diagrams.

As we see below in SPIN's statistics for the exhaustive search proving that the given properties (usually specified as PROMELA model, or UTE file) is indeed satisfied. This result is mentioned for UML state machine of ATM case study before the transformation step.

```
Spin Version 4.1.1 -- 24 April 2006)
+ Partial Order Reduction
Full statespace search for:
never claim -(not selected)
assertion violations -(disabled by -A flag)
cycle checks -(disabled by -DSAFETY)
invalid end states +

State-vector  60 byte, depth reached 2, errors: 0
```

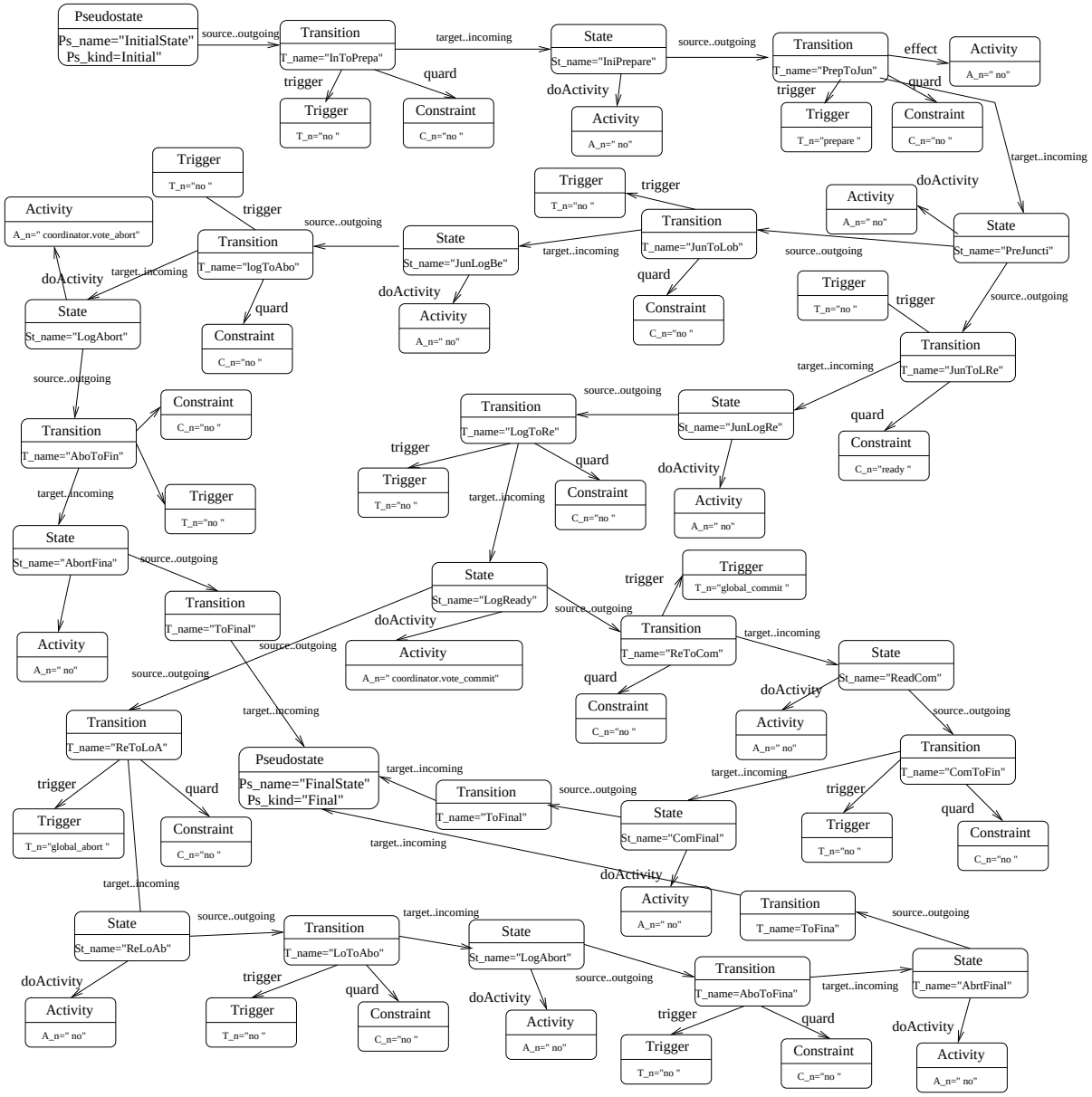


Figure 4.32: Graph Model of Executable 2PC Protocol

```
3 states, stored
0 states, matched
3 transitions (= stored+matched)
0 atomic steps
hash conflicts: 0 (resolved)
(max size 2^19 states)
```

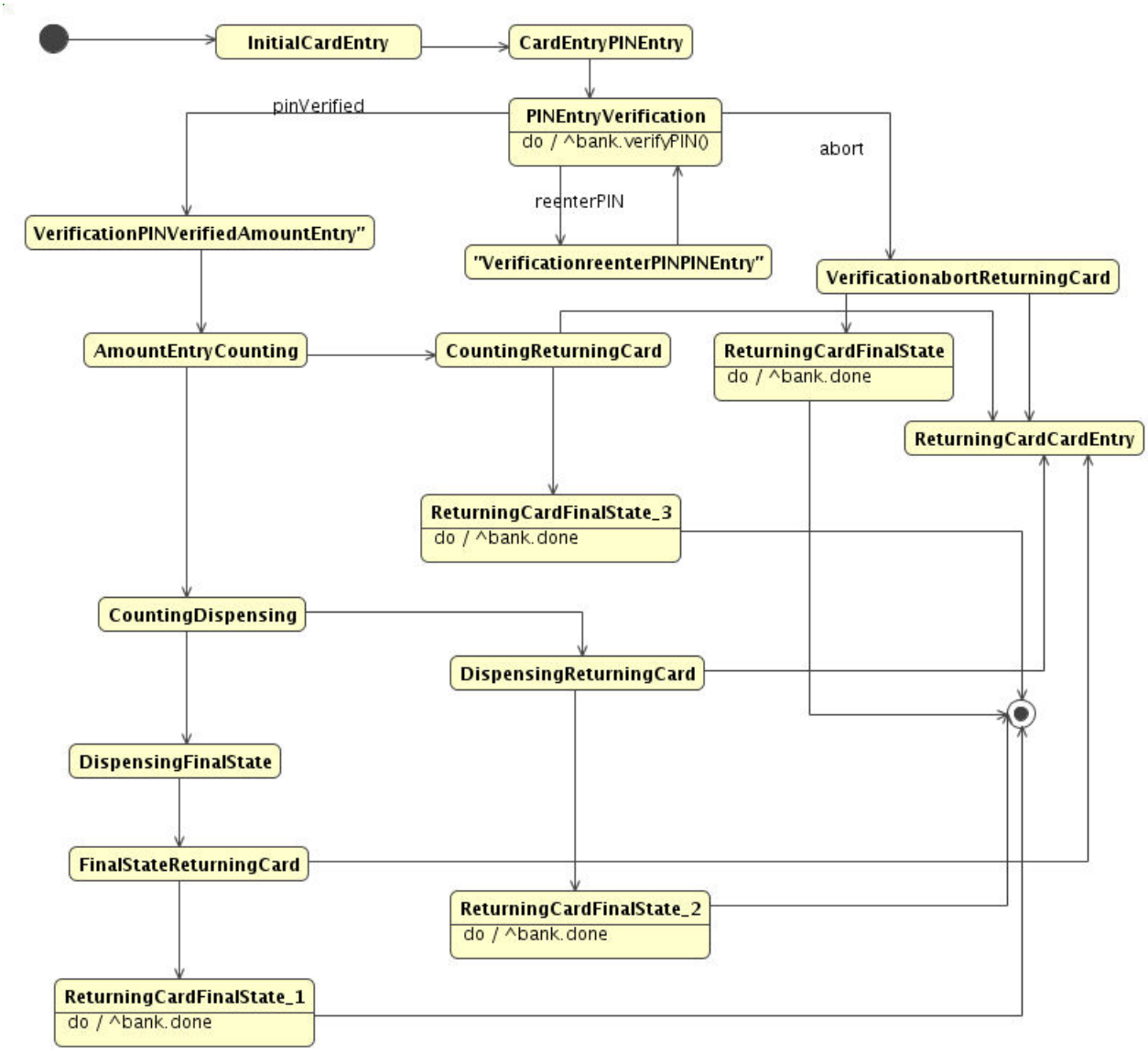


Figure 4.33: Graph Model of Executable 2PC Protocol

```
stats on memory usage (in Megabytes):
0.000 equivalent memory usage for states (stored*(state-vector
 + overhead))
0.266 actual memory usage for states (unsuccessful compression:
  130545.10%)State-vector as stored = 88763 byte +
8 byte overhead
2.097 memory used for hash table (-w19)
```

```
0.320 memory used for DFS stack (-m10000)
2.622 total actual memory usage
```

Again we use HUGO to check the state machines of ATM case study in the executable form. We translate the graph model of executable ATM state machine to UML state machine, then we use HUGO to verify this executable form of ATM, we get the following result:

```
Spin Version 4.1.1 -- 24 April 2006)
+ Partial Order Reduction
Full statespace search for:
never claim -(not selected)
assertion violations -(disabled by -A flag)
cycle checks -(disabled by -DSAFETY)
invalid end states +

State-vector  60 byte, depth reached 2, errors: 0
3 states, stored
0 states, matched
3 transitions (= stored+matched)
0 atomic steps
hash conflicts: 0 (resolved)
(max size 2^19 states)
stats on memory usage (in Megabytes):
0.000 equivalent memory usage for states (stored*(state-vector
  overhead))
0.244 actual memory usage for states (unsuccessful compression:
  119384.31%)State-vector as stored = 81173 byte +
8 byte overhead
2.097 memory used for hash table (-w19)
0.320 memory used for DFS stack (-m10000)
2.622 total actual memory usage
```

Comparing model results

The application of graph transformation system to UML state machine, to create an executable state machines, give us a very useful result; Using the described transformation rules reduce the state space in the model checker memory about 10 percent (see fig. 4.1). We implement the case study state machines of ATM, the state machines of two-phase commit protocol, and we

ATM Model checking in UML Form	*ATM Model checking in ESMs Form*
0.000 equivalent memory for states	0.000 equivalent memory for states
0.266 actual memory for states	**0.244 actual memory for states**
2.097 memory used for hash table	2.097 memory used for hash table
0.320 memory used for DFS stack	0.320 memory used for DFS stack

Table 4.1: Reducing State Space about 10 Percent

extended the two-phase commit protocol to manipulate more than two participants (maximal five participants) and we get always the same result as shown in the table 4.1.

4.10 Result and Discussion

In this chapter, we proposed a new graph transformation strategy based on graph grammars and graph transformation systems to transform UML state machines into executable state machines. After sketching the main concepts of our approach, we carried out several test cases to evaluate our implementation using the transformation engine of AGG tool and HUGO model checker.

The main conclusion that can be drawn from our experiments is that executable state machines help us to reduce the state space of UML software model checking systems. We noticed after implementing the case studies using our transformation strategy that the state space of the model are reduced about 10 percent. We call attention to the fact that our transformation strategy is indeed realistic to implement it in UML software design, especially, for model checking software with a large state space.

5 Secure System Transformations

5.1 Introduction

Modern business techniques depend on networked information systems, even the smallest company uses the Internet and deploys at least one or two security software packages. Nowadays more and more attack tools are appearing, Attacks against the software can threats the economical or even physical well-being of people and organizations. There is widespread interconnection of information systems via the Internet, while is becoming the world's largest public electronic marketplace and being accessible to untrusted users. Attacks can be waged anonymously and from a safe distance. If the Internet is to provide the platform for commercial transactions, it is vital that sensitive information (like credit card numbers or cryptographic keys) is stored and transmitted securely.

When security software is deployed on an application server alongside other day-to-day productivity applications, and when that server runs a standard, commercial operating system, their security software is vulnerable to the same attacks that target the other applications. The security software therefore must not only be able to protect the network and the other applications, it must also protect itself from attack. The security system is the first thing the attacker want to see, and if it is vulnerable to the same attacks as the rest of the network. In this case, it is useless. For example, as part of a 1997 exercise, an NSA hacker team demonstrated how break into US Department of Defense computers and the US electric power grid system, among other things simulating a series of rolling power outages and 911 emergency telephone overloads in Washington, DC, and other cities [NSN99]. While there are of course many more recent examples of security breaches, this particular example also shows that there is more to be concerned about than website defacement and credit card misuse.
Indeed, the fact that in different applications fundamentally different security protocols is difficult and error prone. Part of the difficulty of secure systems development is that the goal of correctness is often in conflict with that of low development cost. Where the methods of system design are high cost through personnel training and use, they are all too often avoided.

Developing secure software systems correctly is difficult and error-prone. Many flaws and pos-

sible sources of misunderstanding have been found in protocol or system specification [Jue04]. In this chapter we propose a flexible new technique to easily verify the implementation of the software designs (like the security software JESSIE).

Our treatment in this chapter depends on two parts:

The first one is to verify the software at the implementation level, we describe an efficient technique for software analysis, that enables an automatic verification of the source code. JML annotations and Bandera specification language (BSL) are used to verify the desired behavior of the software. As an example for the verification task of the source code, we present the verification of the SSL-Handshake protocol in JESSIE.

The second part concentrates on the specification level, we specify the software as graph models and we use the approach of graph grammars and graph transformation systems to transform the graph models into well-suited designs for model checking software. Different model checkers (like DIXIT and HUGO) are used to verify whether the required properties are indeed realized in the software. As an example for the verification task at the specification level, we create the graph models of SSL-Handshake protocol and transform them into predicate diagrams where the model checker DIXIT is called upon to verify the diagrams.

5.2 JAVA Secure Sockets Extension (JESSIE)

JESSIE is a free, clean−room implementation of the Java Secure Sockets Extension, the JSSE. It provides the core API for programming network sockets with the Secure Socket Layer (SSL), which creates an authenticated, unforgeable, and protected layer around network communications. Its goal is to be a drop-in package for free Java class libraries such as Classpath and its derivatives, and is being written to depend only on free software, and only with the API specification and the public protocol specifications.

The SSL protocol is a security protocol based on Web applications. It specifies the security mechanism for data exchanges between application protocols, (for example HTTP, Telnet and FTP) and the TCP/IP protocol, and provides TCP/IP connections with data encryption, server authentication, and optional client authentication.

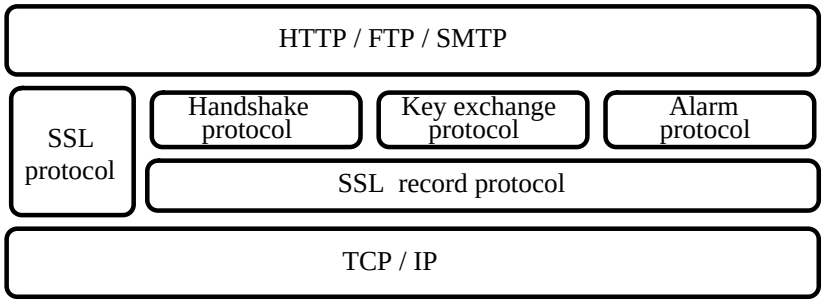


Figure 5.1: SSL Protocols

The SSL protocol comprises the SSL record protocol, handshake protocol, key exchange protocol, and alarm protocol (as shown in Fig. 5.1). All these protocols jointly provide authentication, encryption and anti-distortion functions to application access connections. The SSL handshake protocol is mainly used for the mutual authentication between the server and the client. The negotiation encryption algorithm and the message authentication code (MAC) [1] algorithm are used to generate encryption keys in SSL records. The SSL record protocol provides basic security services to higher-layer protocols. Its working mechanism is as the follows: Each application program message is divided into several manageable data blocks, with the data to be zipped, and a MAC message is generated; the data blocks and the MAC message are encrypted and a new file head is added in; all the data is transmitted via the transfer control protocol (TCP) . It receives end decapsulates the received data, authenticate, unzip and regroup the data before the

[1] A cryptographic message authentication code (MAC) is a short piece of information used to authenticate a message. A MAC algorithm accepts as input a secret key and an arbitrary-length message to be authenticated, and outputs a MAC

data is finally submitted to higher-layer applications. The SSL key exchange protocol comprises a message; it is used to copy the uncertain status as the current status, and update a key group used for the current connection. The SSL alarm protocol is to transmit SSL−related information to peer bodies; alarms transmitted are classified into three levels: warnings, major alarms and critical alarms.

JESSIE depends on the cryptographic algorithms from GNU Crypto [CRY99], GNU Crypto is part of the GNU project aims at providing free, high-quality, and provably correct implementations of cryptographic primitives and tools in the Java programming language for use by programmers and end−users. JESSIE is meant to be run on Java platforms that use GNU Classpath as their class libraries, including GCJ [ORG] and Kaffe [KAF]. JESSIE also uses the GNU Crypto package for its cryptography algorithms.

The whole JESSIE project currently consists of about 5 MB of code, but the part which relevant to my work (Implementation of SSL Handshake Protocol) consists of less than 700 KB in about 70 classes.

5.3 SSL-Handshake Protocol

An SSL session always begins with an exchange of messages called the SSL handshake. The handshake allows the server to authenticate itself to the client using public-key techniques, then allows the client and the server to cooperate in the creation of symmetric keys used for rapid encryption, decryption, and tamper detection during the session that follows. Optionally, the handshake also allows the client to authenticate itself to the server.

The exact programmatic details of the messages exchanged during the SSL handshake [BJ08] are shown in figure 5.2 and explained in the next steps:

1. The client sends the server the client's SSL version number $Pver$, cipher settings $Ciph$, randomly generated data R_c, and other information the server needs to communicate with the client using SSL $Comp$.

2. The server sends the client the server's SSL version number, cipher settings, randomly generated data, and other information the client needs to communicate with the server over SSL. The server also sends its own certificate and, if the client is requesting a server resource that requires client authentication, requests the client's certificate($X509\ Cert_s$).

3. The client uses some of the information sent by the server to authenticate the server (there are a special authentication methods but we don't need them in our running example). If the server cannot be authenticated, the user is warned of the problem and informed that

an encrypted and authenticated connection cannot be established. If the server can be successfully authenticated, the client goes on to Step 4 [$ver(Cert_s)$].

4. Using all data generated in the handshake so far, the client (with the cooperation of the server, depending on the cipher being used) creates the premaster secret for the session PMS, encrypts it with the server's public key (obtained from the server's certificate, sent in Step 2) enc_K_5, and sends the encrypted premaster secret to the server $ClientKeyExchange$.

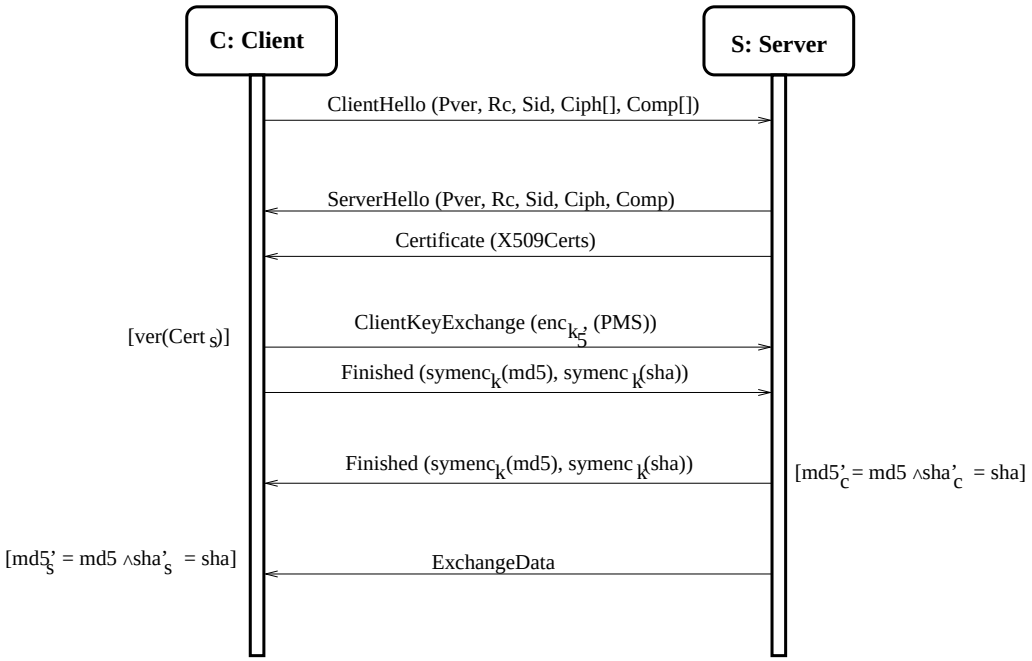


Figure 5.2: SSL−Handshake Protocol

5. If the server has requested client authentication (an optional step in the handshake), the client also signs another piece of data that is unique to this handshake and known by both the client and server. In this case the client sends both the signed data and the client's own certificate to the server along with the encrypted premaster secret.

6. If the server has requested client authentication, the server attempts to authenticate the client (There is also a special method for client authentication but we don't need it in our running example). If the client cannot be authenticated, the session is terminated. If the client can be successfully authenticated, the server uses its private key to decrypt the premaster secret, then performs a series of steps (which the client also performs, starting from the same premaster secret) to generate the master secret.

7. Both the client and the server use the master secret to generate the session keys, which are symmetric keys used to encrypt and decrypt information exchanged during the SSL session $Symenc$ and to verify its integrity. that is, to detect any changes in the data between the time it was sent and the time it is received over the SSL connection.

8. The client sends a message to the server informing it that future messages from the client will be encrypted with the session key. It then sends a separate (encrypted) message indicating that the client portion of the handshake is finished $Finished$.

9. The server sends a message to the client informing it that future messages from the server will be encrypted with the session key. It then sends a separate (encrypted) message indicating that the server portion of the handshake is finished $Finished$.

10. The SSL handshake is now complete, and the SSL session has begun. The client and the server use the session keys to encrypt and decrypt the data they send to each other and to validate its integrity$ExchangeData$.

We use the SSL−handshake protocol given in figure 5.2 together and the open-source Java implementation JESSIE (http://www.nongnu.org/jessie) of the Java Secure Socket Extension **(JSSE)** as a running case study in the remainder of this chapter.

5.3.1 Send and Receive Data in JESSIE

First of all, we want to see the implementation of sending and receiving the data in the JESSIE software, se we reseach the mechnisms that implement send and receive the data. We notice that each message is represented by a message class, it stores the data to be written in the communication buffer [Jue07]. At the same time, this class can also read messages from the communication buffer (as visualized in figure 5.3). We found that this mechanism is implemented using the methods *write()* for sending messages, and *read()* for receiving messages. As explained above,

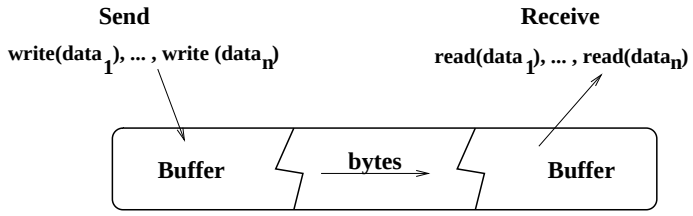


Figure 5.3: Sending and Receiving Data in JESSIE

communication is implemented as follows: With the method call $msg.write(dout, version)$,

the message msg is written into the output buffer $dout$. Each occurrence of such a method call can be identified and associated with the abstract function $send(msg)$ in the specification model. The method call $dout.flush$ later flushes the buffer. The assignment $msg = Handshake.read$ reads a message from the buffer during the handshake part of the protocol. As an example, the code fragment for initializing and sending the ClientHello message is given in figure 5.4.

```
ClientHello clientHello = new ClientHello(session.protocol, clientRandom, sessionId,
                                          session.enabledSuites, comp, extensions);
Handshake msg = new Handshake (Handshake.Type.CLIENT_HELLO, clientHello) ;
msg.write (dout, version) ;
```

Figure 5.4: Initializing and sending the ClientHello message

In our case study (SSL-Handshake protocol), setting up the connection is done by two methods: **doClientHandshake()** on the client side and **doServerHandshake()** on the server side, which are part of the SSLsocket class in JESSIE. After some initializations and parameter checking, both methods perform the interaction between client and server that is specified in the protocol. Each of the messages is implemented by a class, whose main methods are called by the **doClientHandshake()** rp. **doServerHandshake()** methods. The associated data is given in table 5.1.

5.4 Specification Language JML

JML is a behavioral interface specification language for Java modules, Specifications can be written as annotations in Java program files, or stored in separate specification files. Various tools are available that make use of the extra behavioral information that JML specifications provide, and, because JML annotations take the form of Java comments, whether embedded in Java code or in separate files, Java modules with JML specifications can be compiled unchanged with any Java compiler. The behavior of the program inside it's class is specified in JML by writing pre-

Message name in protocol	Class of message type in Jessie	Message type in Jessie
ClientHello	ClientHello	CLIENT_HELLO
ServerHello	ServerHello	SERVER_HELLO
Certificate *	Certificate	CERTIFICATE
ClientKeyExchange	ClientKeyExchange	CLIENT_KEY_EXCHANGE
Finished	Finished	FINISHED

Table 5.1: Data for Handshake message

and postconditions of the methods exported by the module. Using JML specifications must be guarantee that before calling a methods exported by the module, some of preconditions must holds, and after such a call other postconditions must hold also. The use of such pre- and postconditions is dates back to Hoare's 1969 paper on formal verification [Hoa69]. The pre- and post conditions depend on the approach of design by contract [Mey92].

The following sections describe some abstract syntax (Informal and formal specification), and the most useful JML annotations that are used in our case study (SSL-Handshake protocol)

5.4.1 Informal Specifications

Comment usually is a programming language construct used to embed information in the source code of a program to describes method's behavior. JML supports this without requiring that these comments be formalized by allowing informal descriptions in specifications. An informal description looks like this one:

(* Text to describe the properties of the method*)

JML treats an informal description as a boolean expression. This allows informal descriptions to be combined with formal statements, and is convenient when the formal statement is not easier to write. For example, the following JML specification describes the behavior of the method sqrt using informal descriptions.

```
//@ requires (*x is positive *);
/*@ ensures (*\result is an
  @    approximation to
  @    the square root of x *)
  @  && \result >= 0;
  @*/
  public static double sqrt(double x) {
  return Math.sqrt(x);  }
```

Informal specifications are convenient for organizing informal documentation. Informal specifications can also be very useful when there's not enough time to develop a formal description of some aspect of the program. For example, currently JML does not have a formal specification for input and output. Thus, methods that write to and read from files typically have to use informal descriptions to describe parts of their behavior. However, there are several drawbacks to using informal descriptions. A major drawback is that informal descriptions are often ambiguous or incomplete. Another problem is that informal descriptions cannot be manipulated by tools. For

example, JML's runtime assertion checker has no way of evaluating informal descriptions, so these cannot be checked at runtime. Thus, whenever time permits, one should try to use formal notation instead of informal descriptions.

5.4.2 JML Annotations

JML specifications are added to Java code in the form of annotations in comments. Java comments are interpreted as JML annotations when they begin with an @ sign [JML]. That is, comments of the form

`//@ <JML specification>}` or `/*@ <JML specification> @*/`

Overview of JML Syntax The syntax of JML allows one to write specifications that consist of individual clauses, so that one can say just what is desired. Some of JML Syntax are given in the following table 5.2. JML supports several kinds of quantifiers in assertions. Table 5.3

Syntax	Meaning
requires	Defines a precondition on the method that follows
ensures	Defines a postcondition on the method that follows
signals	Defines a condition on when a given Exception can be thrown by the method that follows
assignable	Defines which fields are allowed to be assigned by the method that follows
pure	Declares a method to be side effect free[2] (this shorthand for assignable `\nothing`)
invariant	Defines an invariant property of the class
also	Declares that a method should inherit conditions from its super types
assert	Defines a JML assertion

Table 5.2: JML Syntax

illustrates some of these quantifiers.

The main restriction in JML is that expressions used in JML's assertions cannot have side effects. Thus Java's assignment expressions $(=, +=, etc.)$ and its increment $(++)$ and decrement $(--)$ operators are not allowed. In addition, only pure methods can be called in assertions. Some authors call such methods "query" methods, because they can be used to ask about the state of an object without changing it. One must tell JML that a method to be pure by using the pure modifier in the method's declaration.

Quantifier	Meaning
\result	An identifier for the return value of the method that follows
\old(<name>)	A modifier to refer to the value of variable $< name >$ at the time of entry into a method
\ for all	The universal quantifier
\ exists	The existential quantifier
$a \Longrightarrow b$	The logical construct a implies b
$a \Longleftrightarrow b$	The logical construct a if and only if b

Table 5.3: JML Quantifiers

non-null annotations In JML, there are two ways to make such an assertion. If we want to make sure that a variable is never null (for example, we would call its method in a moment and it could produce a $Null$ Pointer Exception), we add the $/*@$ non_null $@*/$ annotation like this one:

```
/*@ non_null @*/ String s = "Hi there!";
```

A more interesting example is the method dentition. If we want a method argument to be non_null, we could write something like this:

```
public void checkLength(/*@ non_null @*/ String s);
```

or, we could add something like:

```
// @$ requires s $!=$ null
public void checkLength(String s);
```

Notice the subtle difference between those examples. In the first one, if the method body would contain the line:

```
s = null;
```

we would get an error. In the second example, as long as at entry point the non_null assertion is fulfilled, the statement won't generate an error.

Invariants An invariant is a property that should hold in all client−visible states. It must be true when control is not inside the object's methods. That is, an invariant must hold at the end of each constructor's execution, and at the beginning and end of all methods. In JML, a public invariant clause allows one to define the acceptable states of an object that are client-visible; such invariants are sometimes called type invariants. In JML one can also specify invariants with more restrictive visibility; such invariants, which are not visible to clients, are sometimes called

representation invariants. Representation invariants can be used to define acceptable internal states of an object; for example, that a linked list is circular, or other similar design decisions. Public invariants about spec public, private fields, such as this one in Person, have the flavor of both type and representation invariants.

5.4.3 Example ATM

As an example about JML assertions we choose our case study the Java classes of ATM. We wrote some JML annotations inside the Java code of ATM to verify a special feature as followings:

```
public class ATM {
private /*@ spec_public @*/ BankCard  insertedCard = null;
private /*@ spec_public @*/
boolean CustomerAutonticated = false;
/*@ public normal_behavior
    @  requires insertedCard != null;
    @  requires !customerAuthenticated;
    @  requires pin == insertedCard.correctPIN;
    @  assignable customerAuthenticated;
    @ ensures customerAuthenticated;
@*/
public void enterPIN (int pin) {
// the rest of the implementation.
```

another specification:

```
/*@   public normal_behavior
     @  requires insertedCard != null;
     @  requires !customerAuthenticated;
     @  requires pin != insertedCard.correctPIN;
     @  requires wrongPINCounter < 2;
     @  assignable wrongPINCounter;
     @  ensures wrongPINCounter
     @   == \old(wrongPINCounter) + 1;
     @  ensures !customerAuthenticated;
@*/
public void enterPIN (int pin) {
```

```
// the rest of the implementation.
```

`\old(wrongPINCounter)` refers to the value of the field `wrongPINCounter` before method invocation.

example for static invariant

```
public class BankCard {
 /*@   public static invariant
 @     (\forall BankCard p1, p2 ;
 @     \created (p1) && \created (p2);
 @     p1 != p2  ==> p1.cardNumber != p2.cardNumber )
 @*/
private /@ spec_public @/ int cardNumber;
// the rest of the class follows
```

5.4.4 JML Checker

An annotation language like JML would be quite useless without a tool that can extract information from the annotations and use it to verify some, if not all, of its required features. In general, we divide the checkers into two categories:

- Run-time checking tools, like JMLrac [CL02] annotations are converted into assertions that are verified when the code they describe is executed
- Static checking tools, like ESC/Java and ESC/Java2 [CL] do not require running the program; instead they try to prove that annotations are fulfill by statically analyzing possible execution paths.

Advantages and disadvantages of each method can be clearly seen. Run-time checkers can check any assertion, no matter how complicated, but if a method is never run, its assertions will not be executed and verified. Besides, the execution time is longer due to additional instructions in the code. Static checkers, on the other hand, are limited by their reasoning capabilities. Hence they can sometimes show nonexistent errors (false positives) or fail to find some existing ones (false negatives).

5.5 SSL Protocol in JESSIE

In JESSIE, setting up the connection is done by two methods: **doClientHandshake()** on the client side and **doServerHandshake()** on the server side, which are part of the SSL socket class in JESSIE [3]. After some initializations and parameter checking, both methods perform the interaction between client and server that is specified in figure 5.2. Each of the messages is implemented by a class, whose main methods are called by the doClientHandshake() rp. doServerHandshake() methods. First of all, we research the state machines that represents the

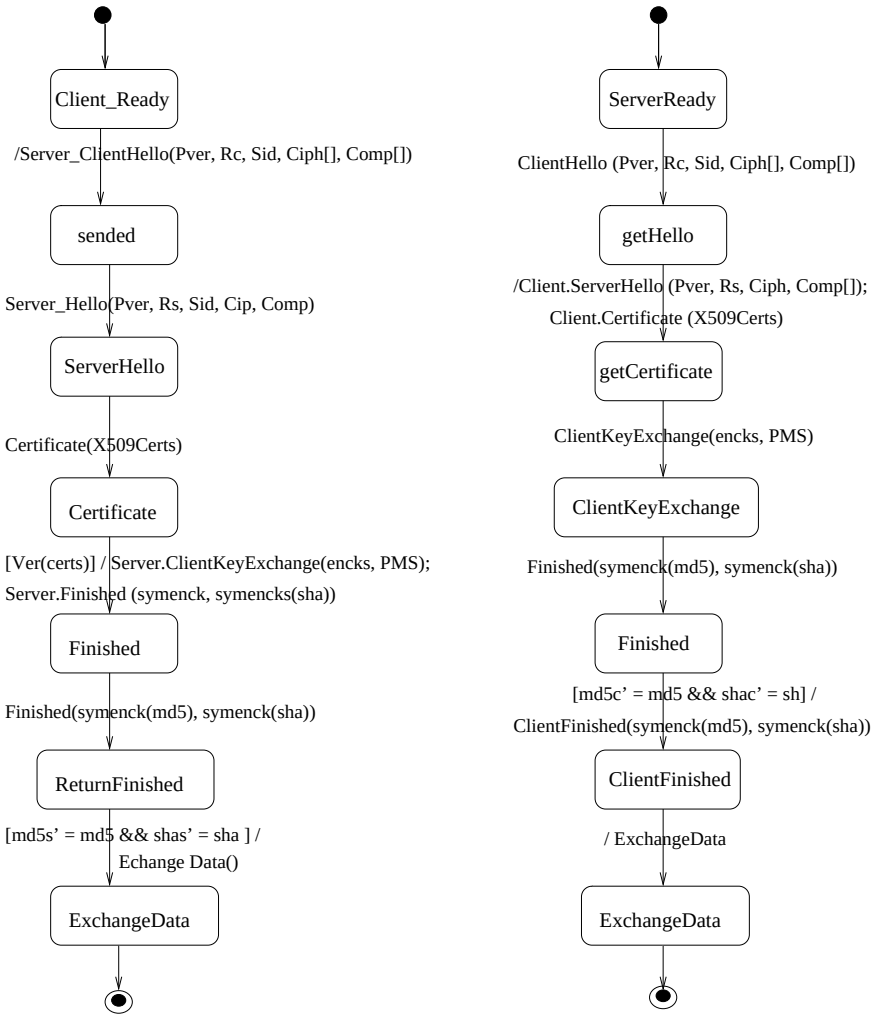


Figure 5.5: SSL State Machines

interaction of the client and the server as specified in SSL-Handshake protocol. We examine the Java code of both methods (doClientHandshake() rp. doServerHandshake()) to specify our

[3] /org/metatstatic/jessie/provider

UML state machines of the protocol that shown in figure 5.5.

At the left hand side of figure 5.5 is the state machine of the client and at the right hand side is the state machine of the server. First of all we illustrate the client's state machine then we will discuss the server state's machine. The implementation of SSL-Handshalke protocol in Java is also in [Kir06] declared.

5.5.1 Client State Machine in JESSIE

The following describes the interaction of the client's state machine as explained also in [Kir06]:

1. **ClientHello:** The client sends the message **ClientHello** to the sever. It has client_version, the version of the SSL protocol by which the client wishes to communicate during this session, generated client's random structure R_c, session$-ID$ which is the ID of a session the client wishes to use for this connection Sid, a list of the cryptographic options supported by the client, sorted with the client's first preference first $Ciph[]$, and finally a list of the compression methods supported by the client, sorted by client preference $Comp[]$.

   $$C \Rightarrow S : ClientHello(Pver, R\_c, Sid, Ciph[], Comp[])$$

   We examine the Java code of the JESSIE software to find that figure 5.6 implements sending the message *ClientHello*.

```
ClientHello clientHello = new ClientHello (session protocol, clientRandom, sessionId,
                                           session.enabledSuites, comp, extensions);
Hanshake msg = new Handshake (Handshake.Type.CLIENT_HELLO, clientHello) ;
msg.write (dout, version) ;
```

   Figure 5.6: Sending the Message ClientHello to the Server

2. **ServerHello:** The answer of the server which consists of Server_version $Pver$, R_s which is random number generated by the server and must be different from (and independent of) ClientHello.random. Sid The single cipher suite selected by the server from the list in ClientHello.cipher_suites. $Comp$ is a single compression algorithm selected by the server from the list in ClientHello.compression_methods. The answer message from the server is implemented in JESSIE as shown in figure 5.7

   $$S \Rightarrow C : ServerHello(Pver, R\_s, Sid, Ciph, Comp)$$

3. **Certificate*** If the server is to be authenticated (which is generally the case), the server sends its certificate immediately following the server hello message. The certificate type

```
msg = Handshake.read(din);

ServerHello serverHello = (ServerHello) msg.getBody();
```

Figure 5.7: Receiving ServerHello message from the Server in JESSIE

must be appropriate for the selected cipher suite's key exchange algorithm, and is generally an X.509.v3 certificate. The same message type will be used for the client's response to a certificate request message. The Certificate message is implemented also in JESSIE as shown in figure 5.8.

$$S \Rightarrow C : Certificate * (X509Cert\_s)$$

```
msg = Handshake.read (din, certType) ;

Certificate serverCertificate = (Certificate) msg.getBody();
```

Figure 5.8: The Server's Certificate which sent to the Client

4. **ClientKeyExchange** If RSA is being used for key agreement and authentication, the client generates a 48-byte pre-master secret PMS, encrypts it under the public key from the server's certificate or temporary RSA key from a server key exchange message, and sends the result in an encrypted premaster secret message to the server.

$$C \Rightarrow S : ClientKeyExchange(enc\_K\_S(PMS))$$

Figure 5.9 shows the implementation of sending the message ClienKeyExchange.

```
ClientKeyExchange ckex = null ;
ckex = new ClientKeExchange (Util.trim(bi)) ;
msg = new Handshake (Handshake.Type.CLIENT_KEY_EXCHANGE, ckex) ;
msg.write (dout, version) ;
```

Figure 5.9: SendClientKeyExchange

5. **Finished:** At the end of Client's side, the client sends the message Finished to the server. This message contains two hashes, which are created using the hash algorithms $HashMD5$ and $HashSHA$. Figure 5.17 shows the implementation of sending the message Finished in JESSIE.

$$C \Rightarrow S : Finished(symenc\_K(md5, sha))$$

```
Finished finis = null ;
finis = generateFinished (version, (IMessageDigest) md5.clone() ,
                                   (IMessageDigest) sha.clone(), true) ;
msg = new Handshake ( Handshake.Type.FINISHED, finis) ;
msg.write (dout, version) ;
```

Figure 5.10: Sending the message Finished

6. **Finished:** The client receives the message Finished from the server which contains two hashes, which are created using the hash algorithms $HashMD5$ and $HashSHA$. Figure 5.16 shows the implementation of receiving the message Finished in JESSIE.

$$C \Leftarrow S : Finished(symenc(md5, sha)K$$

```
msg = Handshake.read (din, suite, null ) ;
finis = (Finished) msg.getBody( ) ;
```

Figure 5.11: Receiving the message Finished

5.5.2 Server State Machine in JESSIE

The following steps are states of the server's state machine as explained also in [Kir06]:

1. **ClientHello:** The server receives the message ClientHello from the Client. It contains the client_version, which is the version of the SSL protocol by which the client wishes to communicate during this session, generated client's random structure R_c, session$-ID$ which is the ID of a session the client wishes to use for this connection Sid, a list of the cryptographic options supported by the client, sorted with the client's first preference first $Ciph[]$, and finally a list of the compression methods supported by the client, sorted by client preference $Comp[]$. We reseach in JESSIE for the Java code which implement the message ClientHello, we find that the figure 5.12 implements receiving the message *ClientHello* to the server.

$$S \Leftarrow C : ClientHello(Pver, R\_c, Sid, Ciph[], Comp[])$$

2. **ServerHello:** The server processes the client hello message and responds with either a handshake_failure alert or server hello message. The server Hello message contains the Protocol_version $Pver$, R_s a generated structure random by the server and must be different from (and independent of) ClientHello.random. Sid the session_id which is the

```
Handshake msg = Handshake.read (din) ;
ClientHello clientHello = (ClientHello) msg.getBody ( ) ;
```

Figure 5.12: Receiving the ClientHello message from the Client

identity of the session corresponding to this connection. $Ciph$ The single cipher suite selected by the server from the list in ClientHello.cipher_suites. $Comp$ The single compression algorithm selected by the server from the list in ClientHello.compression_methods [FKK96]. We reseach in JESSIE for the Java code which implement the message ClientHello, we find that the figure 5.13 implements sending the message *ServerHello* in JESSIE.

$$S \Rightarrow C : ServerHello(Pver, R\_c, Sid, Ciph, Comp)$$

```
ServerHello serverHello = new ServerHello (version, serverRandom,
                                session.getId(), suite, comp, extensions ) ;
msg = new Handshake.Type.SERVER_HELLO, serverHello) ;
msg.write (dout, version) ;
```

Figure 5.13: Sending the message ServerHello to the Client

3. **Certificate** If the server is to be authenticated (which is generally the case), the server sends its certificate immediately following the server hello message. The certificate type must be appropriate for the selected cipher suite's key exchange algorithm, and is generally an $X.509.v3$ certificate. The same message type will be used for the client's response to a certificate request message. Figure 5.14 shows the implementation of *Certificate* message in JESSIE.

$$S \Rightarrow C : Certificate * (X509Cert\_S)$$

```
Certificate serverCert = new Certificate (certs) ;
          msg = new Handshake (Handshake.Type.CERTIFICATE, serverCert) ;
          if (DEBUG_HANDSHAKE_LAYER)  debug.println (msg) ;
          msg.write (dout, version) ;
```

Figure 5.14: Sending the Certificate Message

4. **ClientKeyExchange** The server received generates 48-byte pre-master secret message by the client, which encrypted under the public key from the server's certificate or temporary RSA key from a server key exchange message. Figure 5.15 shows this step of receiving the message *ClientKeyExchange*.

```
msg = Handshake.read (din, suite, kexPair.getpublic( ) ) ;
ClientKeyExchange ckex = (ClientKeyExchange) msg.getBody ( ) ;
```

Figure 5.15: Receiving ClientKeyExchnge message

$$C \Rightarrow S : ClientKeyExchange(enc\_K\_S(PMS))$$

5. **Finished** The server receives the message *Finished* from the client. This message contains two hashes, which are created using the hash algorithms $HashMD5$ and $HashSHA$. Figure 5.16 shows the implementation of the message *Finished* in JESSIE.

$$S \Leftarrow C : Finished(symenc\_K(md5, sha))$$

```
Finished finis = null ;
msg = Handshake.read (din, suite, null ) ;
finis = (Finished) msg.getBody ( ) ;
```

Figure 5.16: Receiving the message Finished

6. **Finished** Finally, the server sends the message *Finished* to the client. This message contains two hashes, which are created using the hash algorithms $HashMD5$ and $HashSHA$. Figure 5.17 shows the implementation of the message Finished in JESSIE.

$$S \Rightarrow C : Finished(symenc\_K(md5, sha))$$

```
finis = generateFinished (version, md5, sha, false) ;
msg = new Handshake (Handshake.Type.FINISHED, finis) ;
msg.write (dout, version) ;
```

Figure 5.17: Sending the message Finished to the client

The ExchangData message in the both state machines (client and server's state machines) mean that the application data messages are carried by the Record Layer and are fragmented, compressed and encrypted based on the current connection state. The messages are treated as transparent data to the record layer.

5.6 JML Assertions in JESSIE

Determining the security properties satisfied by software using cryptography is difficult: Security requirements such as secrecy, integrity and authenticity of data are notoriously hard to

establish, especially in the context of cryptographic interactions [Jue06].

In order to be sure that the implemented protocol in JESSIE indeed verify the SSL-Handshake protocol, we write our JML assertions inside the Java code of JESSIE. For example let's explain this strategy for the Java code which implements sending the message *HelloServer* to the sever in the procedure *doClientHandshake*(), by inspecting the location where the assertions must be written. We write our JML assertion as shown in figure 5.18 whereas the keyword **ensures** checks if the properties that follows are verified or not. As written in the Protocol, the server

```
//@ ensures  (DEBUG_HANDSHAKE_LAYER  == true && clientHello ==
new ClientHello (session.protocol, clientRandom, sessionId, session.enabledSuites,
                                comp, extension) &&
  msg == new Handshake(Handshake.  Type.CLIENTHELLO, clientHello));
```

Figure 5.18: JML specifications for Message ClientHello

now should receive the message from the client. This part of protocol is implemented with Java in **doServerHandshake** procedure in JESSIE as shows in figure 5.12. In order to verify if the server indeed receive the message from the client, we add our JML specification which ensures that the message is delivered to the server. The JML specifications are shown in figure 5.19.

```
//@ ensures (Handshake.Type == Handshake.Type.SERVER_HELLO
    && serverHello != null
```

Figure 5.19: JML specifications for Message ClientHello

5.6.1 Verifying Client State Machine in JESSIE

To verify the client state machine in JESSIE, we write JML assertions before the procedure's name inside Handshake class. The assertions must verify the states in the client's state machine. Figure 5.20 shows the name of the states in the client's state machine and the appropriate JML assertions. The ensure clause in this figure satisfy the pre-conditions for the procedure **doClientHandshake**. For example the first ensure clause means that the object **clientHello** is created and assigned to the variable **msg**, whereas the condition **serverCertificate != null** means that the variable is assigned to the appropriate certificate code.

```
// State Client_Ready in client' state machine
// Transition Server.ClinetHello (Pver, Rc, Sid, Cip, Comp) in client's state machine
@ensure (DEBUG_HANDSHAKE_LAYER == true && clientHello == new ClientHello (session.protocol,
clientRandom, sessionId, session.enabledSuites, comp, extensions)  &&  msg == new
Handshake (Handshake.Type.Client_HELLO, clientHello)) ;

// state sended in client's state machine
// Transition serverHello (Pver, Rc, Sid, Cip, Comp) in client's state machine
@ ensure  (Handshake.Type == Handshake.Type.SERVER_HELLO && serverHello ! = null) ;

// State serverHello in client' state machine
// Transition Certificate (X509Certs) in Client's state machine
@ ensures  (Handshake.Type == Handshake.Type.CERTIFICATE &&  serverCertificate != null);

// State Certificate in Client's state machine
// Transition Server.ClientKexExchange in client's state machine
@ ensures (Handshake.Type == Handshake.Type.CLIENT_KEY_EXCHANGE && ckex != null);

// State Certificate in client' state machine
// Transition Server.Finished in client's state machine
@ ensures (Handshake.Type == Handshake.Type.FINISHED && finis != null) ;

// State Finished in client's state machine
// Transition Finished () in client's state machine
@ ensures (Handshake.Type == Handshake.Type.FINISHED && Returnfinis != null);

// State ReturnFinished in client's state machine
// Transition ExchangeData() in client's state machine
@ assignable  msg, finis, Returnfinis;
@*/

/*@ spec_public @*/ private void doClientHandshake ()  throws IOException
```

Figure 5.20: JML assertions for client's state machine

5.6.2 Verifying Server State Machine in JESSIE

In order to verify the server state machine in JESSIE, we write JML assertions before the procedure's name inside the *Handshake* class. The assertions must verify the states in the server state machine. Figure 5.21 shows the name of the states in the client state machine and the appropriate JML assertions. The assertions are written before the procedure name. For example the assignment **ServerClientHello != null** denotes to assign a new valid value to the variable **ServerClientHello**, when the variable **ClientFinished** has no null value any more, it means that the client sends the request messages to the server and finished the protocol.

We have identified JML as a good candidate for verifying the SSL-Handshale protocol in Jessie. The most basic tool support for JML is parsing and typechecking. The parsing and typechecking JML tools will catch any type incompatibilities, references to names that no longer exist, etc. The JML checker **(jml)** developed at Iowa State University performs parsing and typechecking of Java programs and their JML annotations. We use the **(jml)** checker to check our JML assertions, first we get some typechecking errors. Howerver, The errors have already

```
// State ServerReady in server' state machine
// Transition ClientHello () in server state machine
@ ensures  ServerClientHello  != null  &&  ServerMsg ! = null ;

// State getHello in server' state machine
// Transition Client.ServerHello in server's state machine
@ ensures (Handshake.Type == Handshake.Type.SERVER_HELLO &&
Server_ServerHello ! = null) ;

// state getHello in server's state machine
// Transition Client.Certificate (X509Certs)
@ ensures (Handshake.Type == Handshake.Type.CERTIFICATE && serverCert != null) ;

// State getCertificate in server's state machine
// Transition ClientKeyExchange (encks, PMS) in server's state machine
@ ensures (Handshake.Type == Handshake.Type.CLIENT_KEY_EXCHANGE &&
Server_ckex != null );

// state ClientKeyExchange in server's state machine
// Transition Finished in server's state machine
@ ensures (Handshake.Type == Handshake.Type.FINISHED && Server_finis != null);

// State Finished in server's state machine
// Transition ClientFinished () in server's state machine
@ ensures (Handshake.Type == Handshake.Type.FINISHED && ClientFinished != null);
@* /


/*@ spec_public @*/ private void doServerHandshake() throws IOException
```

Figure 5.21: JML assertions for the server's state machine

been corrected and we improved also the JML assertions to be as shown in 5.20 and 5.21.

One way of checking the correctness of JML specifications is by runtime assertion checking, i.e., simply running the Java code and testing for violations of JML assertions. Such runtime assertion checks are accomplished by using the JML compiler **jmlc**. Unfortunately, we could not use the **jmlc** to compile the Java code of SSL-Handshake protocol in JESSIE because the **jmlc** can not compile a Java code that references a class outside of the current file. Thus if any variable calls a method with a parameter outside the allowed bounds, the **jmlc** shows an error and stop the process of compiling the Java code with the JML assertions.

We decided to write again a new assertions in BSL (Bandera Specification Language) to use the Bandera tool for verifying the desired behavior of the protocol. The next section illustrates the using of BSL assertions for verifying the SSL-Handshake protocol.

5.7 Verifying SSL-Handshake via Bandera

The Bandera Tool Set is an integrated collection of program analysis, transformation, and visualization components designed to facilitate experimentation with model-checking Java source code. Bandera takes as input Java source code and a software requirement formalized in Bandera's temporal specification language, and it generates a program model and specification in the input language of one of several existing model-checking tools (including Spin, dSpin, SMV, and JPF)

Property specification Source code properties to be checked are written in the Bandera Specification Language (BSL). BSL is based on a collection of field-tested temporal specification pattern [DAC99b] that allow users to write specifications in a stylized English format. These patterns essentially are parameterized macros that can be instantiated to one or more temporal logics such as LTL and CTL.

To verify our SSL-Handshake protocol via Bandera we write the BSL (Bandera Specification Language) assertions inside the Java code of the SSL-Handshake protocols. For example to verify if the client indeed sends the `ClientHello` message to the server, first of all, we have to define a new boolean variable indside the Java code which indicates sending the `HelloServer` message as the following:

```
private boolean SendClientHello = false;
```

if the variable `SendClientHello` has the value true, it means that the client sends the message `HalloServer` to the server. Secondly we assign to the added varibale the appropriate value inside the part which implements sending the `HelloServer` message as following:

```
:
:
//create and send Client Hello
//set ProtocolVersion
gotProtocolVersion=1;
//create random ...
gotClientRandom=1;
//create session ID
gotSessionID=1;
//available CipherSuites
gotCipherSuitesList=1;
```

```
writeToChannel(Channel.CLIENTHELLO);
SendClientHello = true;
:
:
```

finally, we write the pre- and postconditions that verify the process of sending `ClientHello` message to the server. The pre- and postconditions must be added before the procedure header of the `ServerHelloRequest()` procedures as shown in the specification below:

```
/**
* @assert
*  PRE  SendClientHello: (SendClientHello == false);
*  POST SendClientHelloTrue: (SendClientHello==true);
*/
private void ServerHelloRequest()
{
//create and send Client Hello message
//set ProtocolVersion
gotProtocolVersion=1;
//create random ...
gotClientRandom=1;
//create session ID
gotSessionID=1;
//available CipherSuites
gotCipherSuitesList=1;
writeToChannel(Channel.CLIENTHELLO);
SendClientHello = true;
}
```

The role of the tag `@assert` is to indicate that an assertion is being defined. The label `PRE` indicates that the following is the precondition assertion that needs to hold right befor entering the constructor code. The `SendClientHello` refers to the name of the assertion. This assertions means that before entering the constructor of the code, the variable `SendClientHello` is assigned to the value `false`.

The second post-condition which indicates by the name `SendClientHelloTrue` checks if after executing the constructor code the inserted boolean variable `SendClientHello` is assigned to the value `true`. If this varibale is assigned to the `true` value during the execution

of the Java code, it means that the client did his process and sent the `ClientHello` message to the server.

5.7.1 Verifying BSL via Spin

Invoking Bandera brinds up the main window of the Bandera User Interface (BUI). The Session button invokes the Session Manager View which is used to load, configure, and save sessions and session files. The Checker button is provided to enable and disable model checking in the currently active session. first, we create a new session by the Session Manager and load the appropriate Java files with the BSL assertions to be checked. In the Session Manager we choosed the Spin model checker to be active in this session. Secondly, we activate the Checker Button and we execute the session by clicking the run button in the Toolbar, to see if the desired properties which specified as BSL are verified in the Java code of the SSL-Handshake protocol. The result of the SPIN model checker is as following:

```
pan.exe -n -m1000000 -w18 -e
(Spin Version 4.1.3 -- 24 April 2004)

Full statespace search for:
        never claim             - (not selected)
        assertion violations    +
        cycle checks            - (disabled by -DSAFETY)
        invalid end states      +

State-vector 888 byte, depth reached 21963, errors: 0
   75524 states, stored
  139406 states, matched
  214930 transitions (= stored+matched)
  548201 atomic steps
hash conflicts: 19397 (resolved)
(max size 2^18 states)

97.145  memory usage (Mbyte)

*** END ***
```

This result of SPIN shows that the BSL assertions in the Java code which specify the properties of SSL-Handshake protocol are verified. That is, the implementation of the SSL-Handshake in

Java Code is well-specified in JESSIE and satisfies the properties of SSL-Handshake protocol.

In appendix C we discuss all of the BSL assertions that we need to verify the SSL-Handshake protocol and We summarize results of using Bandera to verify properties of SSL-Handshake protocol via SPIN model checker. The actual time that we need to verify the SSL-Handshake protocol with BSL assertions vis SPIN model checker is about 32 seconds.

5.8 Graph Transformation of Handshake Protocol

In this section we explain how we design the SSL Handshake protocol as graph model. we define for every state machine in the protocol the appropriate graph model. Type and Host graphs are defined also for using them to any transformation rules.

5.8.1 Designing Graph Models

Modeling can be described as an abstraction processes. The most important process in the modeling is how to build the models as representations of reality. Another process in the modeling is to extract the concepts from concrete objects, or rules from observed behavior.

Send, Receive, Check, Encrypt ... etc, are concepts of the scenario of Handshake protocol. So we illustrate the forms of Scenario by means of Handshake Protocol. We remember that we present the protocol as UML state machines in section 5.5, we notice that this figure represents the scenario of the protocol. In this case, we can use our experience in creating the graph model of UML state machine, so it is now easy to create the graph models of SSL Handshake protocol which represented as UML state machines.

5.8.2 Type and Instance Graph

Graphs can represent States by modeling concrete entities as vertices and relations between theses entities as edges. In our model, The type of vertices; $c : Client$, $s : Server$ represent the corresponding concepts in the Scenario. In other words c is a vertex from type Client, s is a also a vertex from type Server.

The relation between concepts and their occurrences in the Scenario is formally captured by the notion of *typed graphs*: TG which represents the type (concept) part of the Scenario and its instance graphs the individual states in the Scenario. Figure 5.22 illustrates this relation between types and concepts.

5.8.3 Graph Model of Client State Machine

We have already illustrated in our case study the state machines of ATM how we represent every state as a node in the graph model. The transitions between the state in the client's state machine are represented as nodes in the graph model. Every node in the graph model is connected with their attributes, it means that the state node is connected with three nodes (represent doActivity, exit activity and entry activity). The transition node is also connected with three nodes (represent

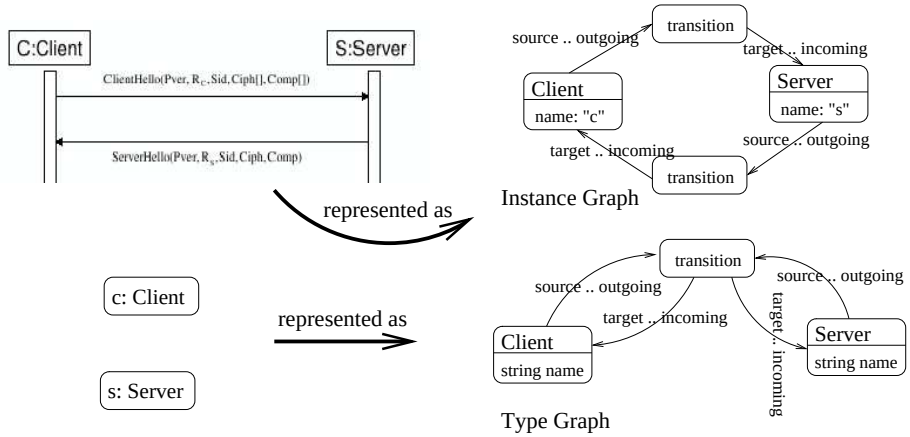


Figure 5.22: Type and Instance Graph from Scenario

trigger, event and guard). In this case, we can represent the client's state machine as shown in figure 5.23.

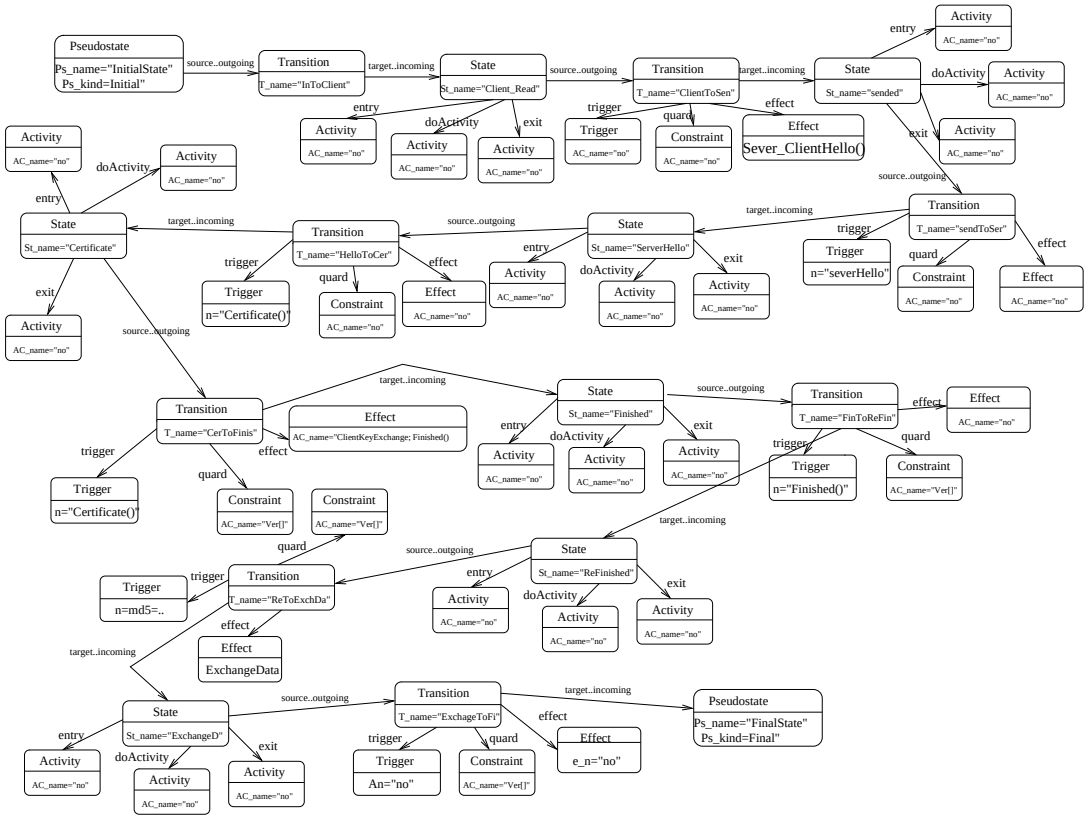


Figure 5.23: Graph Model of Client State Machine

5.8.4 Graph Model of Server State Machine

We use the same strategy to create the graph model of the server state machine. Figure 5.23 illustrates the host graph model of server state machine. Usually every state in the state machine is represented as four nodes (the node itself and the three connections node, they represent the entry, do and exit activities) in the host graph, and every transition between two states is represented also as four nodes (the transition itself as a new node, and three other node for trigger and quard and effect properties) in the host graph.

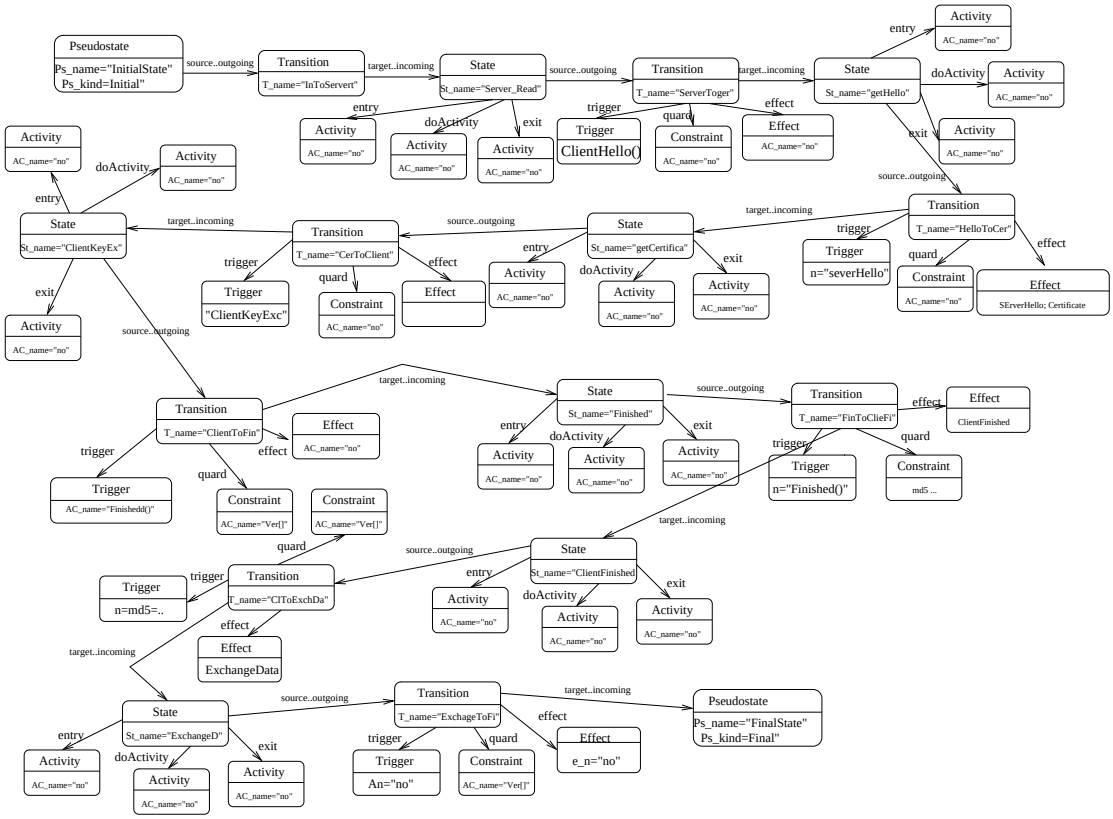


Figure 5.24: Graph Model of Server State Machine

5.9 Predicate Diagrams

A predicate diagram is a finite graph whose nodes are labeled with sets of (possibly negated) predicates, and whose edges are labeled with action names as well as optional annotations that assert certain expressions to decrease with respect to an ordering in $O^{=}$ [CMMag]. Intuitively, a node of a predicate diagram represents the set of system states that satisfy the formulas contained in the node. (We indifferently write n for the set and the conjunction of its elements.) An edge (n, m) is labeled with action A if the action may cause a transition from a state represented by n to a state represented by m. An action A may have an associated fairness condition; it applies to all transitions labeled by A rather than to individual edges. We let edges be labeled with action names instead of action formulas because, in a top-down development, the precise action formula that defines an action is not known until the final specification has been derived.

Formally, the definition of predicate diagrams is relative to finite sets P and A that contain the state predicates and the (names of) actions. We write $\overline{P}$ to denote the set containing the predicates in P and their negations.

Definition A predicate diagram $G = (N, I, \delta, o, \zeta)$ over P and A consists of

- a finite set $N \subseteq 2^{\overline{P}}$ of nodes,
- a finite set $I \subseteq N$ of initial nodes,
- a family $\delta = (\delta_A)_{A \in \mathcal{A}}$ of relations $\delta_A \subseteq N \times N$ (by $\delta_=$ we denote the reflexive closure of the union of these relations),
- an edge labeling o that associates finite sets $f(t_1, \prec_1), ..., (t_k, \prec_k)$ of terms t_i paired with a relation $\prec_i \in O^{=}$ with the edges $(n, m) \in \delta$, and
- a mapping $\zeta : \mathcal{A} \rightarrow \{NF, WF, SF\}$ that associates a fairness condition with every action in A; the possible values represent no fairness, weak fairness, and strong fairness.

We say that the action $A \in \mathcal{A}$ can be taken at node $n \in N$ iff $(n, m) \in \delta$ holds for some $m \in N$, and denote by $En(A) \subseteq N$ the set of nodes where A can be taken.

5.9.1 Dining Philosophers Example

We specify the TLA+ specification of the 'Dining Philosophers' problem which is introduced in [oP] as following:

$\texttt{Init} \equiv n \in Nat \wedge n \neq 0 \wedge c_0 = \texttt{"t"} \wedge c_1 = \texttt{"t"}$
$Eat_0 \equiv c_0 = \texttt{"t"} \wedge even(n) \wedge \acute{c_0} = \texttt{"e"} \wedge \acute{c_1} = c_1 \wedge \acute{n} = n$
$Thk_0 \equiv c_0 = \texttt{"e"} \wedge \acute{c_0} = \texttt{"t"} \wedge \acute{n} = n/2 \wedge \acute{c_1} = c_1$
$Eat_1 \equiv c_1 = \texttt{"t"} \wedge \neg even(n) \wedge \acute{c_1} = \texttt{"e"} \wedge \acute{c_0} = c_0 \wedge \acute{n} = n$
$Thk_1 \equiv c_1 = \texttt{"e"} \wedge \acute{c_1} = \texttt{"t"} \wedge \acute{n} = 3 * n + 1 \wedge \acute{c_0} = c_0$
$Next = Eat_0 \vee Thk_0 \vee Eat_1 \vee Thk_1$
$v \equiv < v_0,\ c_1,\ n >$
$DM \equiv Init \wedge \Box[Next]_v \wedge WF_v(Next)$

The TLA+ specification consists of two processes (whose control states are represented by the variables c_0 and c_1) that communicate via a shared integer variable n. Each process can be in either thinking ($'t'$) or eating ($'e'$) state. The variable n, initialized to some positive integer, controls access to the eating states: process 0 may eat when n is even and divides n by 2 when returning to state $'t'$. Conversely, process 1 may eat when n is odd and assigns $3n + 1$ to n when it stops eating. The purpose of the protocol is to ensure mutual exclusion of the $'e'$ states without introducing starvation for either process. A predicate diagram for this system appears in Fig.

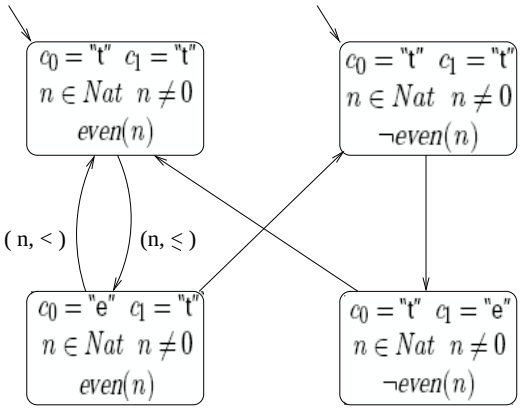


Figure 5.25: Predicate Diagram of Dining Philosophers Problem

5.25. It consists of four nodes, each labeled with a set of literals. The two top nodes are initial; in fact, while the control states are fixed and n is known to be a non-zero natural number, it can be even or odd. Considering the top left-hand node, it is easy to see that only possible successor state is represented by the lower left-hand node, corresponding to the occurrence of action Eat_0, which sets c_0 to $'e'$ without changing c_1 or n. In particular, n is still positive and even. From there, only the action Thk_0 is possible, and will lead back to a state where both processes are in their $'t'$ states. Moreover, since n must be at least 2 in the source state, $n = 2$ is at least 1, so n is still a positive integer. However, it could be even or odd, as represented by the two abstract transitions of the diagram. The justification of the remaining transitions is similar.

A predicate diagram represents every possible behavior of the system, and properties (over the predicates represented in the abstraction) can therefore be verified by model checking. For example, it is easy to prove mutual exclusion ($\Box\neg(c_0 =' e' \wedge c_1 =' e')$) just by looking at the states of the diagram. Similarly, weak fairness of the next state relation $Next$ ensures that process 0 will eat infinitely often ($\Box\Diamond(c_0 =' e')$), since no trace through the diagram can forever avoid visiting the lower left-hand node.

5.9.2 Predicate Diagram of SSL−Handshake Protocol

In this section we illustrate the representations of SSL−Handshake Protocol as a predicate diagram to prove some secure system properties which must hold in the protocol.

As we can see in figure 5.2), we define five attributes. Client_Hello which is positive when the Client sends his message to the server. Server_Hello is also an attribute refers to the server when it sends his message to the client. Certificate is an attribute that hold when the server sends his certificate to the client. ClientKeyExchange is a an attribute, it is active when the client sends his message to the server. Finished is an attribute refers to the end of the connection between the server and the client. Finally, the attribute ExchangeData refers to beginning an safety connection between the client and the server. Figure 5.26 shows TLA specification of SSL−Handshake protocol. We mean by `ClientHello = "no"` that the client did not send Hello message until now. When the `ClientHello = "yes"`, it means that the client has already sent his message to the sever. The property $Prep_1$ means that there is a probably for changing Data when the client sent first his message. The predicate Diagram for SSL-Handshake protocol appears in Fig. 5.27. It consists of five nodes, the middle node is initial; it consists of six attributes. The Client_Hello attribute is positive, it's means that the client sends now the message to the server. The other attributes in this initial node are negative and they mean; they did not receive or sent any message. On the right top node the Server_Hello and Certificate attributes are positive whereas the other attributes are negative. It means that the server received the message from the client and sent his Certificate and message Server_Hello to the client. The third node on the left top side, the attributes finished and ClientKeyExchage are positive whereas the other are negative, it means that the client sent the message ClientKeyExchange and the message $Finished$ to refers to finishing the connection with the server. The lower two nodes are to represent the finish process of sending the protocol's message between the client and the server, and begin with the next task to transform the data between the client and the server safety.

```
VARIABLE  ClientHello, ServerHello, Certificate, ClientKeyExchange,
Finished, ExchangeData
----------------------------------------------------------------------------------------------
Client == ClientHello = “no“ ^ ClientHello' = “yes“ ^ UNCHENGED <<ServerHello,
Certificate, ClientKeyExchange, Finished, ExchangeData>>

Server == ServerHello = “no“ ^ ServerHello' = “yes“ ^
Certificate = “no“ ^ Certificate' = “yes“ ^ UNCHANGED <<ClientHello,
ClientKeyExchange, Finished, ExchangeData>>

Client_1 == ClientKeyExchnge = “no“ ^ ClientKeyExchange' = “yes“ ^
Finished = “no“ ^ Finished' =“yes“ ^ UNCHANGED << ClientHello,
ServerHello, Certificate, ExchangeData >>

Server_1 == ExchangeData = “no“ ^ ExchangeData' = “yes“ ^
UNCHANGED << ClientHello, ServerHello, Certificate, Finished,
ClientKeyExchange >>
----------------------------------------------------------------------------------------------
Init == ClientHello = “no“ ^ ServerHello = “no“ ^ Certificate =
“no“ ^ ClientKeyExchange = “no“ ^ Finished = “no“ ^ ExchangeData = “no“

Next == Client V Server  V Client_1  V Server_1
v == << ClientHello, ServerHello, Certificate,
ClientKeyExchange, Finished, ExchangeData >>
----------------------------------------------------------------------------------------------
Spec == Init ^ [][Next]v  ^ WF_v(Next)
----------------------------------------------------------------------------------------------
\* Proerties
Prop_1 == [](ClientHello = “yes“ ) -> <> (ExchangeData = “yes“))
==============================================================
```

Figure 5.26: TLA Specification of Handshake Protocol

5.10 Rules Transformations of SSL−Handshake Protocol

In this section we propose to use graph transformation rules to describe our strategy for graph transformation of our case study. These rules provide precise specifications needed for an automated transformation for graph model of UML state machine to a graph model of predicate diagrams. To transform the state machine into predicate diagram, we have to follow the following tasks:

1. First of all, we define the type graph of client and server state machines, we can do this step easily because we have already defined the type graph of the state machine in chapter 4.

2. We represent the client or server state machine as Host-graph models. We use the trans-

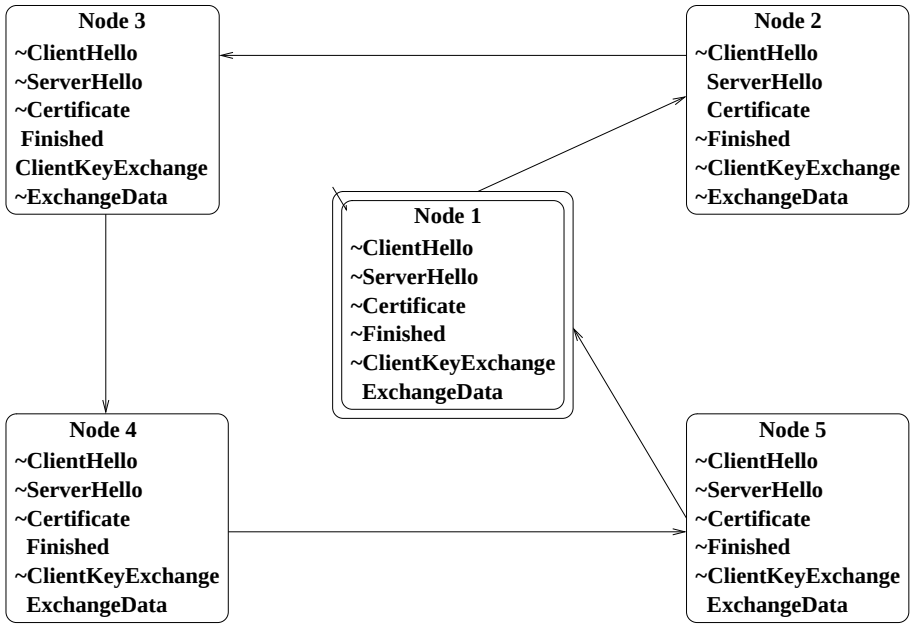


Figure 5.27: Predicate Diagram of SSL−Handshake Protocol

formation tool AGG to represent the host graph of the client state machine. Figure 5.28 shows a part of the host graph model of the client state machine using AGG.

3. We define the transformation rules. They are about fourteen grammars as shown in the left side of figure 5.28. We will not discuss here all the rules that we defined in AGG. For example, figure 5.29 shows how we create the initial node from the given host graph.

   As shown in this figure, at the right hand side of this rule we create a new node called "Initial" represents the initial node of the predicate diagram.

   Another rule is shown in figure 5.30 that defines the variable of the node in predicate diagram. As we can see in this figure we have in this rule the NAC to guarantee not to create the variable one more time if it is already created.

4. We run the rules that we define to transform the graph model of client state machine to a graph model of predicate diagram, after the transformation is done we get the final graph which represents the predicate diagram of the Handshake Protocol [4]

Figure 5.31 shows a part of graph model of SSL-Handshake protocol. Every node has six attributes, When the attribute dosn't has any negative symbol, it means that this attribute sends his message to the client or to the server. For example the attribute serverHello in `Node1` hasn't

[4]We do not have to worry about the server state machine, just one of them in the graph model format is enough to define the predicate diagram of the SSL-Handshake protocol.

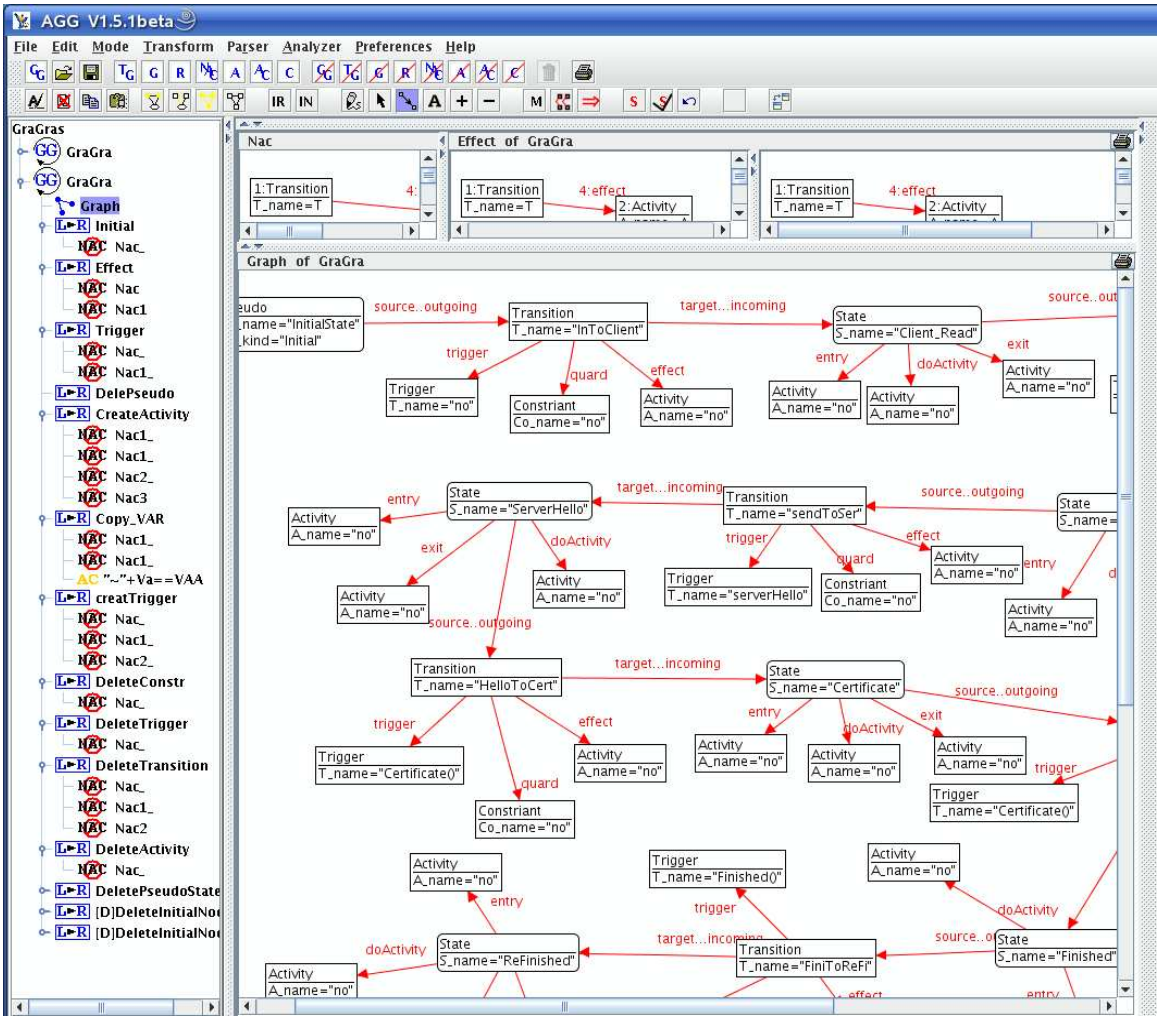


Figure 5.28: Part of Host-graph of SSL-Handshake in AGG

any negative symbol, it means in this case that the client sent the message `HalloServer` to the server.

5.11 Properties verification via DIXIT

The DIXIT toolkit is intended to assist a user in performing the kind of defining the predicate diagrams. Predicate diagrams can be drawn in a graphical editor, either from scratch or as a structural refinement of an existing diagram. In the latter case, the node mapping is defined implicitly, as no new nodes may be added, but existing nodes may be split [FMM05]. DIXIT can also generate proof obligations that ensure that a diagram conforms to a TLA+ system specification associated with the diagram. Verification steps can be initiated from a hierarchical

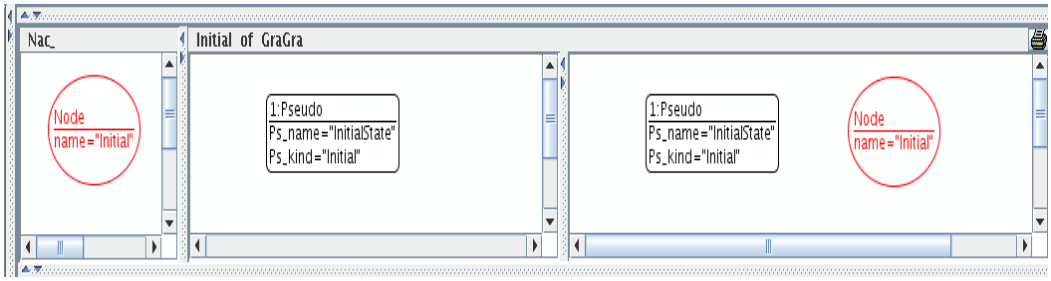


Figure 5.29: Graph Grammar to define the Initial Node

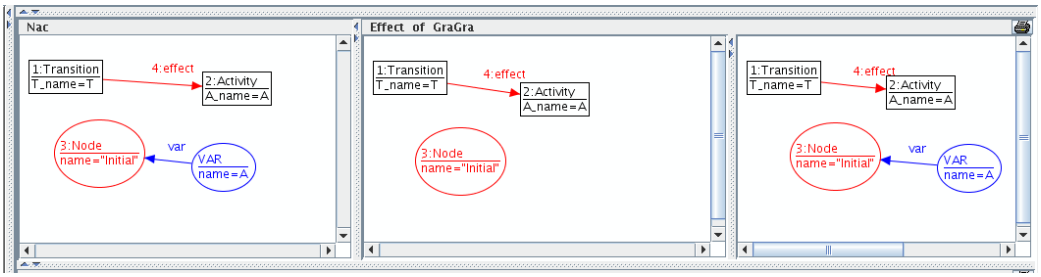


Figure 5.30: Graph Grammar to define the Attribute

project view in a separate window. The kernel interacts with external verification tools through well-defined interfaces. It generates proof obligations for theorem provers, model checkers, as well as structural conditions that can be verified at the diagram level itself. Currently, DIXIT is oriented towards the analysis of TLA+ models, and therefore it interacts with the TLA+ parser TLASANY. Externally, a DIXIT project is stored in XML format; it may also include (pointers to) files that are not processed by the kernel, such as TLA+ modules. Diagrams can be exported in Postscript, GIF, and SVG formats.

In the next section we discuss how we represent our case study SSL-Handshake protocol in the DIXIT model checker.

5.11.1 SSL-Handshake in DIXIT

First of all we have to translate the final graph Fig. 5.31 of SSL-Handshake protocol into predicate diagram [5] where the nodes in the graph model are represented as states with six attributes in the predicate diagram. The edges in the graph model are represented as actions between the states in the predicate diagram. Figure 5.32 shows the predicate diagram of SSL-Handshake protocol using DIXIT interface.

[5] We do this step either manually or automatically using DAMAS. DAMAS is declared in chapter 6.

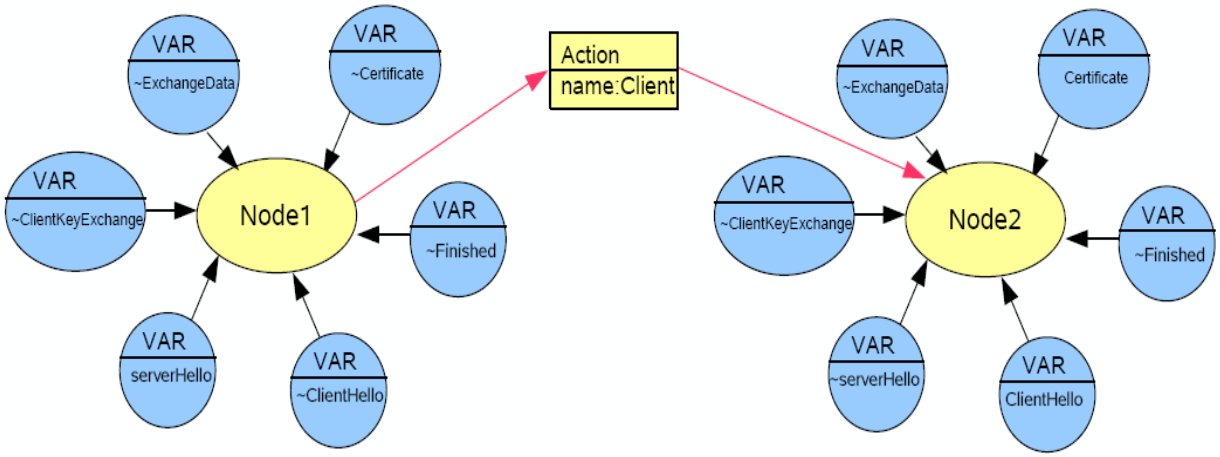


Figure 5.31: Predicate Diagram as Graph Grammar

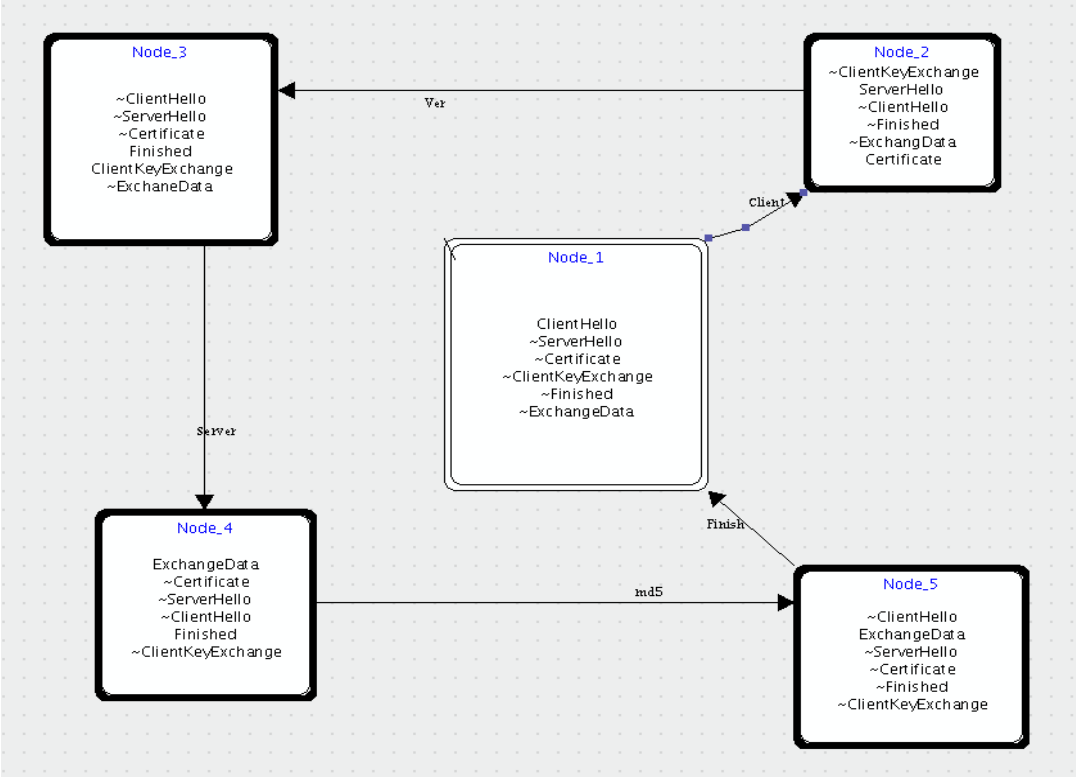


Figure 5.32: Predicate Diagram of Handshake in DIXIT

The nodes of the diagram in Fig. 5.32 reflects the different phases of the SSL-Handshake protocol, and the edges represent the possible transitions. For example, the actions is enabled from the initial node `Node_1` to the node `Node_2` when the client sends the message `ServerHello` to the server.

Satisfying Properties We check whether the desired properties are valid in the predicate diagram or not. The verification can be done by using the model checker SPIN or TLA model checker. To verify the properties using DIXIT we have to write them as a temporal logic properties. For example, to verify that the protocol satisfy the property:

`If the client sends the Message HelloServer,`

`it must be a data exchange between the client and the server.`

We can write such this property in temporal logic as following:

$$\Box(Client \Rightarrow <> ExchangeData)$$

This property is valid in the predicate diagram which shown in Fig. 5.32

5.12 Result and Discussion

In Chapter 5 we indicated a strategy to verify security protocols using the approach of graph grammars and graph transformation systems. First we introduced basic concepts about the SSL security protocols and investigated the mechanism of the SSL-Handshake protocol. Second, we determined in the JESSIE software the Java code which implements the SSL-Handshake protocol. JML assertions are used to verify the SSL-Handshake protocol in JESSIE, whereas the Bandera model checker is also used to verify the desired security properties in JESSIE software. In the third step we compiled the SSL-Handshake protocol into a graph model to use the approach of graph grammars and graph transformation systems. In this case, we could transform the protocol into new graph model that is compilable into predicate diagram form.

Finally, we verified the created predicate diagram using the DIXIT model checker and checked whether the security properties in our case study are indeed valid.

6 DAMAS

In order to make our theoretical results applicable to larger examples and practical case studies, we design our prototyping tool DAMAS . DAMAS supports an automatic translation of UML state machines into graph models and to create the required files for the software model checking systems. Figure 6.1 illustrates a simple view of DAMAS. DAMAS is designed to bridge the gab between UML software design and model checking system. In applying model checking to software design, in particular of UML, we find that software design usually involves infinite state spaces, this is not directly suited for model checking, since model checker accept only designs where the sate space is finite. On the other hand, the semantic definitions of model checking systems are different from UML software semantic. Using DAMAS provides us to reduce the state space of the required model using our pre-defined rules about 10 percent, and to define a new semantic of the models using graph transformation techniques. As we can see in the left side of figure 6.1, DAMAS reads the model as UML state machines, then it compiles the model into graph language and use the transformation engine for transforming the compiled model using pre-defined rules into final graph. The second aim of DAMAS is to compile the final graph model into an specification language that is suitable for model checker tools like HUGO and SPIN.

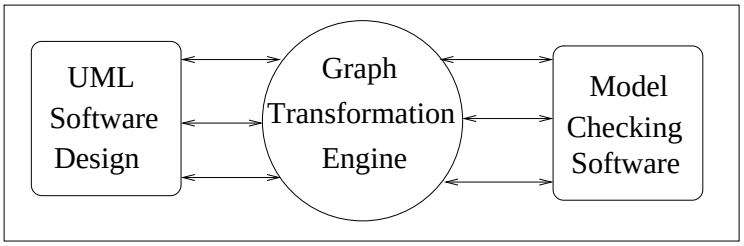


Figure 6.1: Prototype DAMAS Tool

We describe in this section our prototype tool DAMAS. Several tools provide verification support for the state machine view as UML model via transition into the input languages of model checkers [LMM99, LP99]. Our tool is different because it is completely depend on graph transformations approach to create new graph models. We will discuss our tool and implement some

case studies in the next four sections as following:

The first section illustrates using DAMAS to now how to compile UML state machine designs into Host-graph models. The second section discusses how we use our prototype DAMAS to translate the host graph into new graph model via our pre-defined rules. Section 3 illustrates compiling the final graph into output language for model checker HUGO and DIXIT. Finally, section 4 verifies some insurance properties using our tool DAMAS. We intend to carry out our case study ATM as running example in this chapter. The implementation of SSL-Handshake protocol via DAMAS is illustrated in Appendix B.

6.1 DAMAS and UML Software Design

DAMAS accepts the software design as UML state machine diagrams, so first of all we implement our case study as UML state machine diagrams, let's consider again our case study ATM in chapter 2 (see figure 2.13 and 2.14), we use usually MagicDraw which is a visual UML modeling and CASE tool with teamwork support to model our examples. MagicDraw provides full support for UML 2.0 metamodel, including class, use case, communication, sequence, state, activity, implementation, package, component, composite structure, and deployment diagrams. In addition, MagicDraw provides explicit support for UML profiles and custom diagrams. Figure 6.4 shows the screenshot of MagicDraw.

We design the case study ATM using MagicDraw as shown in figure 6.3, which describes the interaction of an automatic teller machine (ATM), a bank computer, and a single user. The interaction focuses on the validation of the Card and the PIN-Code. These two state machines specify the dynamic behavior of our case study ATM. As an input file to DAMAS we take the XMI file that which produces from the UML-Editor MagicDraw and compiles it into new file represents the host-Graph of ATM case study.

To compile the UML state machine file into Host-graph file using DAMAS, we have to write the following command:

```
DAMAS Graph "Inputfile" "Outputfile"
```

the `inputfile` is an XMI file which produces using any UML-Editor, whereas the `ouputfile` is an gxx file which created to use the graph transformation engine AGG.

For example to translate the ATM state machine into graph models we have to write the following command:

```
DAMAS Graph ATM.xmi ATM.gxx
```

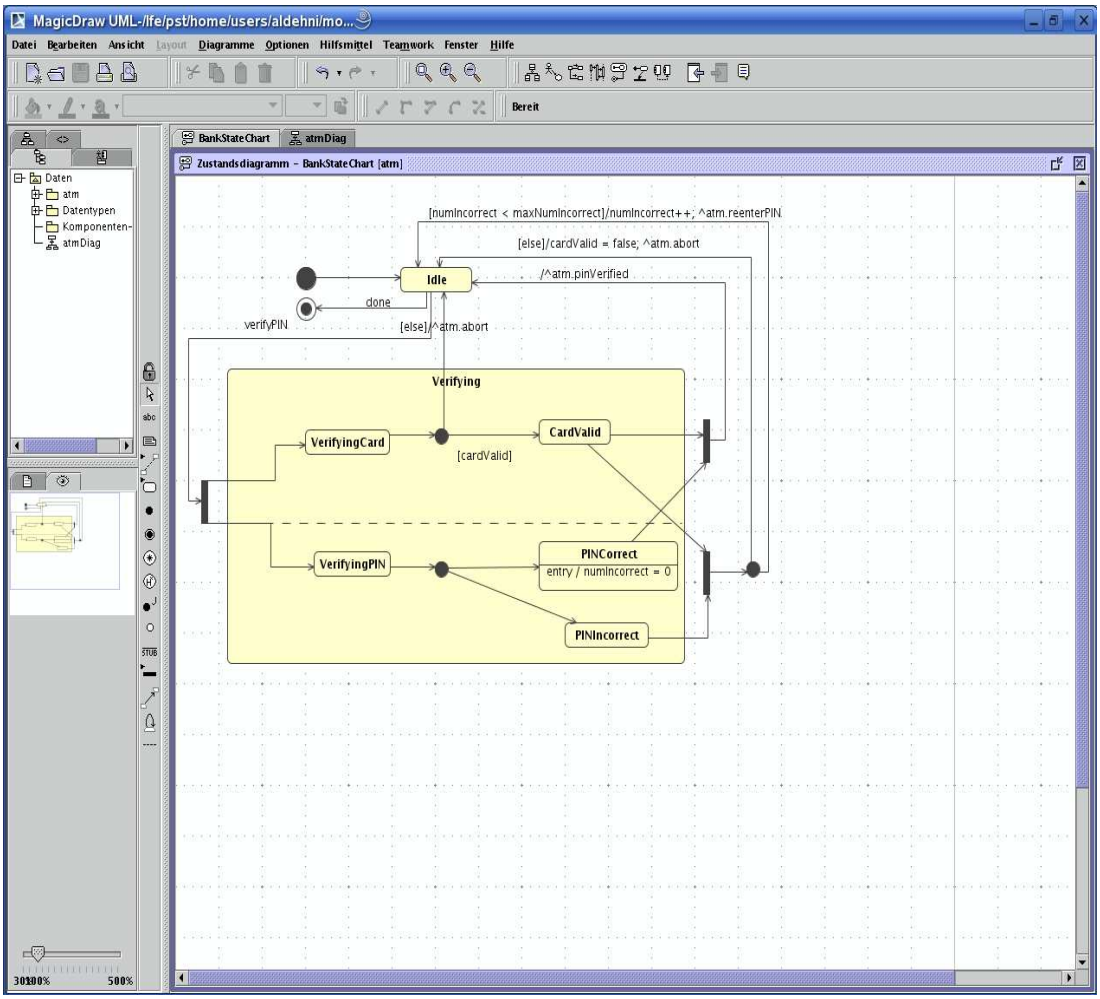
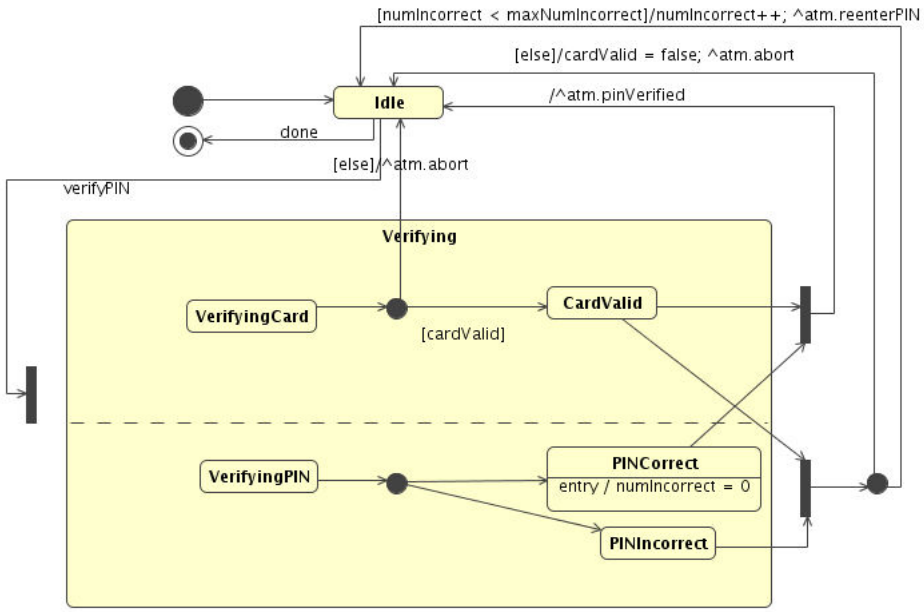


Figure 6.2: The Screenshot of MagicDraw

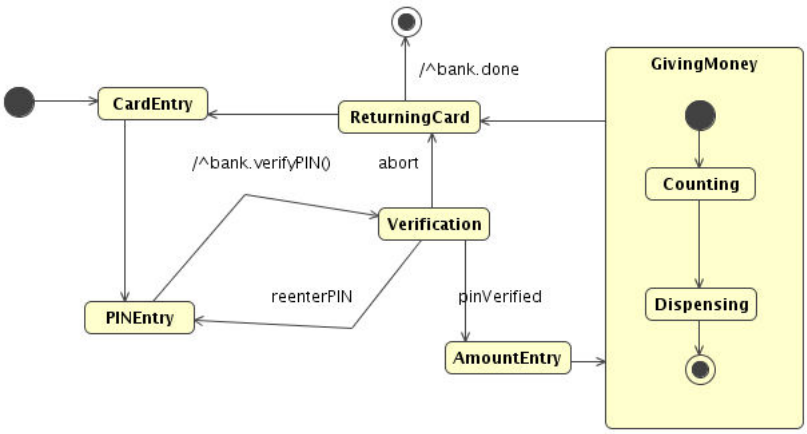
`ATM.xmi` is the input XMI file for DAMAS which represents the ATM state machine, whereas ATM.gxx is the output file represents the Host-graph model of the ATM state machine.

DAMAS creates the appropriate file with the extension `gxx`. The following shows part of the created file:

```
<?xml version="1.0" encoding="UTF-8"?>
<Document version="1.0">
 <GraphTransformationSystem ID="I1" name="GraGra">
  <TaggedValue Tag="AttrHandler" TagValue="Java Expr">
  <TaggedValue Tag="Package" TagValue="java.lang"/>
  <TaggedValue Tag="Package" TagValue="java.util"/>
  <TaggedValue Tag="Package" TagValue="com.objectspace.jgl"/>
```



(a) State Machine of Bank with MagicDraw



(b) State Machine of ATM with MagicDraw

Figure 6.3: Designing of ATM with MagicDraw

```
<TaggedValue Tag="Package" TagValue=
             "genged.alphabet.datatypes"/>
</TaggedValue>
<TaggedValue Tag="CSP" TagValue="true"/>
<TaggedValue Tag="injective" TagValue="true"/>
<TaggedValue Tag="dangling" TagValue="true"/>
```

```
  <TaggedValue Tag="NACs" TagValue="true"/>
  <TaggedValue Tag="showGraphAfterStep" TagValue="true"/>
  <TaggedValue Tag="TypeGraphLevel" TagValue="DISABLED"/>
  <Types>
  <NodeType ID="I2" name="State%:ROUNDRECT:
      java.awt.Color[r=0,g=0,b=0]::[NODE]:">
  <AttrType ID="I4" attrname="name"
          typename="String" visible="true"/>
```

6.2 DAMAS and Graph Transformation Engine

We use usually AGG transformation tool [AGG] to see the created file via DAMAS. We have already explained in chapter 3 how we define the state machines as graph models with nodes and edges. Some nodes represent the state in the UML state machine and other nodes represent the actions between the states. Figure 6.4 shows the created graph model of the ATM case study with the AGG tool.

We have already determined the graph model of our case study, so we can choose the pre-defined rules to transform the host-Graph to new appropriate graph model which must be suitable to software model checking design. Suppose we want to transform the host-Graph into new model represents the predicate diagram. In this case, we have to add the transformation rules for predicate diagrams. Figure 6.5 shows the transformation rules that must be added to our host-Graph model.

We have now the required grammar to transform the host-Graph model into predicate graph model. The rule is actually executed by clicking the "S" button in the toolbar of AGG. We can now save the created graph (the one which represents the predicate diagram) as `gxx` file, let's call it for example `predicate.gxx`. The following is a part of `predicate.gxx` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<Document version="1.0">
 <GraphTransformationSystem ID="I1" name="GraGra">
  <TaggedValue Tag="AttrHandler" TagValue="Java Expr">
  <TaggedValue Tag="Package" TagValue="java.lang"/>
  <TaggedValue Tag="Package" TagValue="java.util"/>
  <TaggedValue Tag="Package" TagValue="com.objectspace.jgl"/>
  <TaggedValue Tag="Package" TagValue=
              "genged.alphabet.datatypes"/>
```

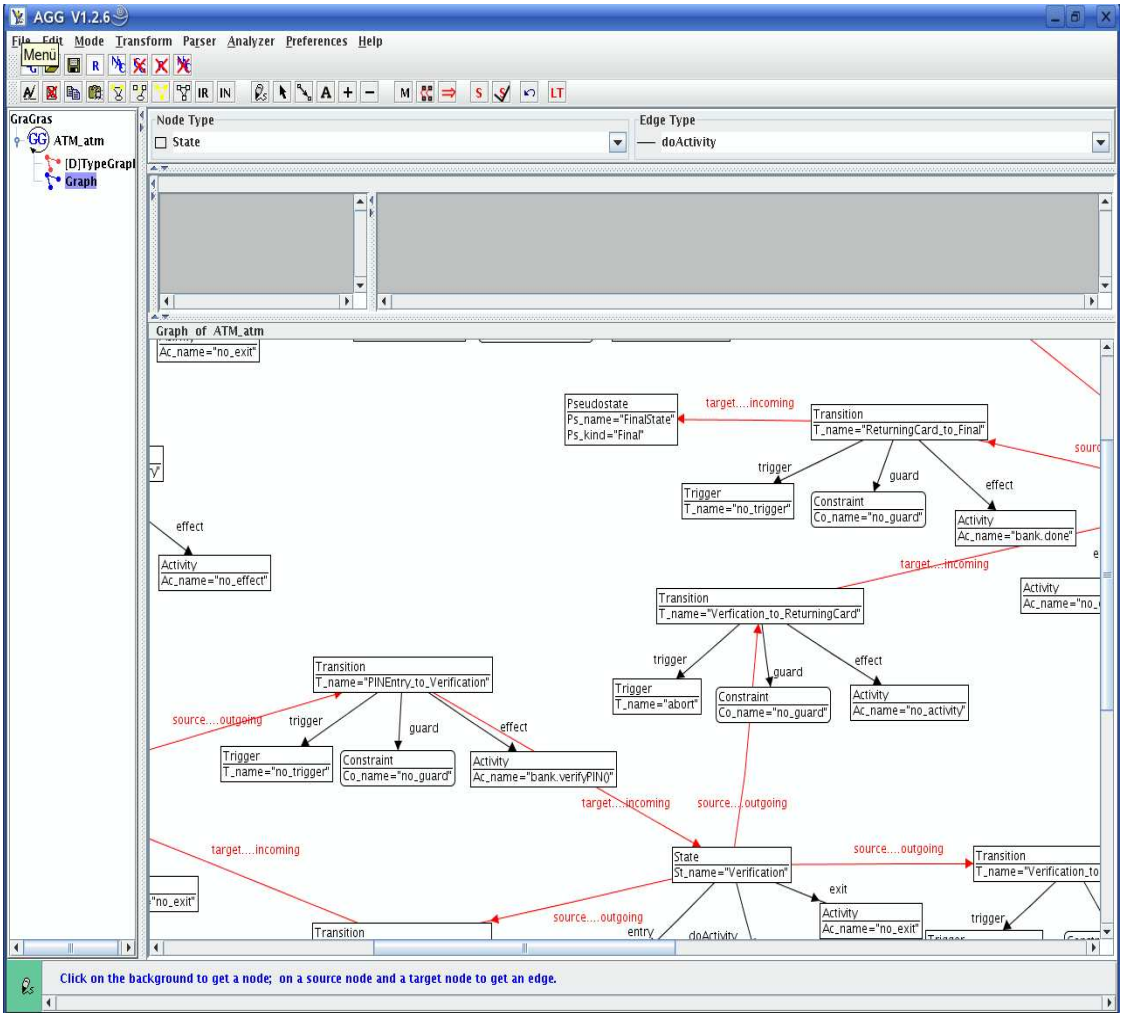


Figure 6.4: The Graph Model of ATM

```
<TaggedValue Tag="Package" TagValue=""/>
<TaggedValue Tag="Package" TagValue=
             "agg.attribute.handler.impl.javaExpr"/>
</TaggedValue>
<TaggedValue Tag="CSP" TagValue="true"/>
<TaggedValue Tag="injective" TagValue="true"/>
<TaggedValue Tag="dangling" TagValue="true"/>
<TaggedValue Tag="NACs" TagValue="true"/>
<TaggedValue Tag="showGraphAfterStep" TagValue="true"/>
<TaggedValue Tag="TypeGraphLevel" TagValue="DISABLED"/>
<Types>
<NodeType ID="I2" name="State%:ROUNDRECT:
```

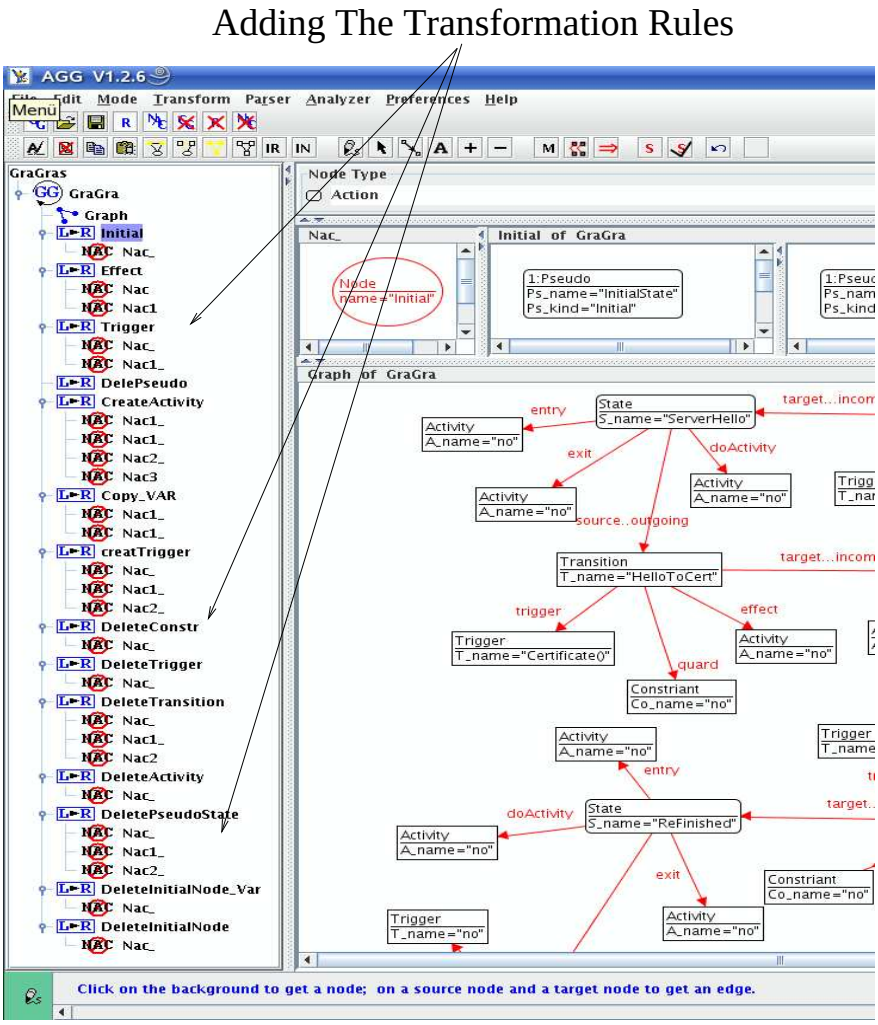


Figure 6.5: The Transformation Rules to Predicate Diagram

```
    java.awt.Color[r=0,g=0,b=0]::[NODE]:">
<AttrType ID="I4" attrname="S_name"
  typename="String" visible="true"/>
</NodeType>
<NodeType ID="I5" name="Pseudo%:ROUNDRECT:
   java.awt.Color[r=0,g=0,b=0]::[NODE]:">
```

6.3 DAMAS and Model Checking

We use usually the model checker DIXIT [FMM05] to verify the required properties. The DIXIT toolkit provides support for the verification of systems using Boolean abstractions in the

form of predicate diagrams. It is organized around a visual editor that allows a user to draw a predicate diagram and enter node and edge annotations. Properties expressed in temporal logic can be verified from the interface by calling the SPIN model checkers, The specification of predicate diagrams are written with TLA+ (temporal logic of action). If we want to compile our final graph model into predicate diagram, we have to write the specification of our case study ATM. The following is the specification of ATM in TLA+:

```
------------MODULE ATM ---------
EXTENDS  Naturals
VARIABLE Amount, Card, Pin
-------------------------------------
EnterCard == Card ="no" /\ Card' ="yes"
         /\ Pin' =Pin /\ Amount' =Amount
VerifyCard == Card = "yes" /\ Card' =Card
  /\ Pin="no" /\ Pin' ="yes"
          /\ Amount' =Amount
EnterAmount == Pin ="yes" /\ Card ="yes"
            /\ Amount =0 /\ Amount' =Amount +1
            /\ Card' =Card /\ Pin' =Pin
GiveMoney ==Amount # 0 /\ Amount =0 /\ Card="yes"
          /\ Card' = Card /\ Pin = "yes" /\
             Pin' = Pin
ReturnCard == Card ="yes" /\ Card' ="no" /\
         Pin ="yes" /\ Pin' ="no"
         /\ Amount # 0 /\ Amount'=0
-------------------------------------
Init == Card ="no" /\ Pin ="no" /\ Amount =0
Next == EnterCard \/ VerifyCard \/
      EnterAmount \/ GiveMoney \/ ReturnCard
v == <<Amount, Card, Pin >>
---------------------------------
Spec == Init /\ [][Next]_v /\ WF_v(Next)
```

DAMAS has a special rule to add the specification in the graph model, in this case, we have to execute the rule given in figure 6.6.

After executing the TLA rule, a small dialog called Al-Dehni is opened to write the required TLA specification. Let's save the TLA file under the same name `predicate.gxx`. Our case study ATM is ready now to translate it into new file represents the input file of DIXIT. The

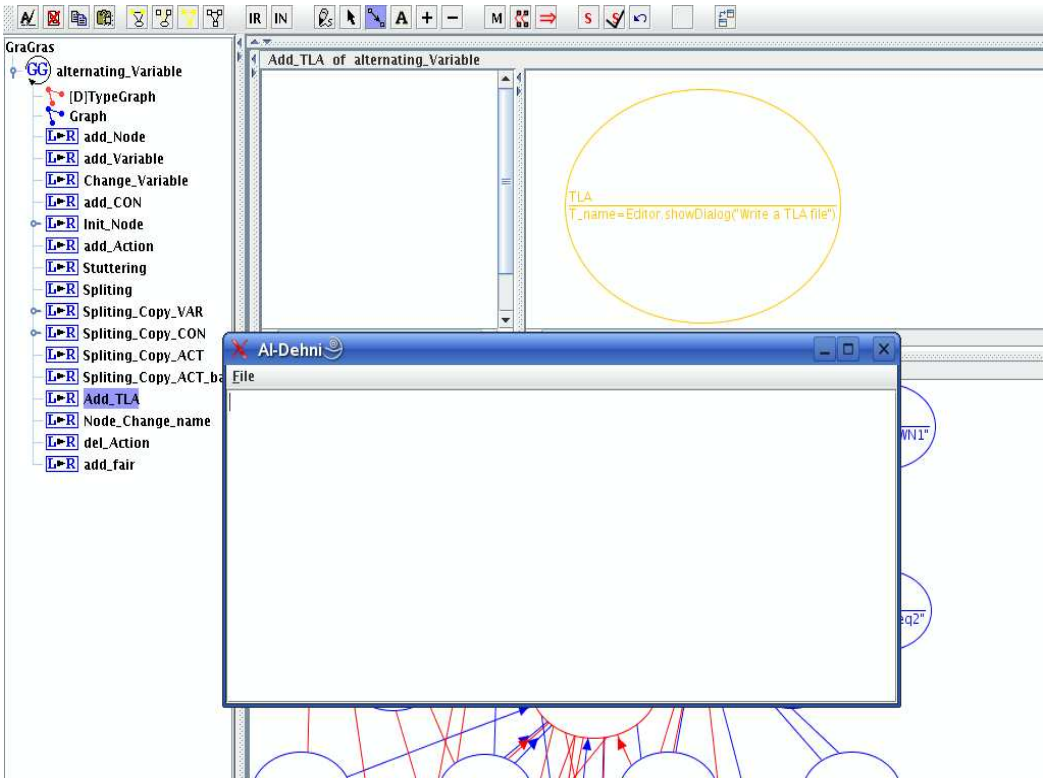


Figure 6.6: Rule for Adding TLA+ Specification

following is the command to compile the graph model to DIXIT:

```
DAMAS Predicate "predicate.gxx" "dixit.xml"
```

The new file `dixit.xml` is an input "XML" file for DIXIT model checker. Part of this file is shown as following:

```
<project base="file:/lfe/pst/home/users/
 aldehni/dixit/dixit/shadi/ATM/"
 xsi:schemaLocation="http://www.loria.fr
 /equipes/mosel/dixit/ dixit.xsd">
<name>Untitled Project</name>
 <diagram id="Diagram1">
<name>ATM</name>
 <node height="97" id="Node2"
    width="146" x="56" y="32">
<name>Node1</name>
 <initial>true</initial>
```

```
 <predicate>Card="no"</predicate>
 <predicate>Pin="no"</predicate>
 <predicate>Amount=0</predicate>
</node>
<node height="109" id="Node3"
 width="164" x="57" y="325">
 <name>Node2</name>
 <initial>false</initial>
 <predicate>Card="yes"</predicate>
 <predicate>Pin="no"</predicate>
 <predicate>Amount = 0</predicate>
 <predicate/>
</node>
<node height="93" id="Node4"
  width="146" x="448" y="336">
```

We have now the required file for verifying the predicate diagram using DIXIT. Figure 6.7 shows the predicate via the DIXIT model checker.

6.4 Verifying Properties using DAMAS

To verify some properties which must hold in ATM design, we write the properties as temporal logic in DIXIT. For example, to verify that the ATM model is indeed give the required money to the customer, if he/she give the right card with the right PIN-number. Such this property is written in temporal logic as following:

```
[] <> (Card = "yes" && Pin = "yes" => Amount # 0)
```

The result of DIXIT using SPIN model checker is: `verified.`

Another example:

```
[] (Card = "yes" && Pin = "no" => Amount = 0)
```

The result of DIXIT using SPIN model checker is: `not verified.`

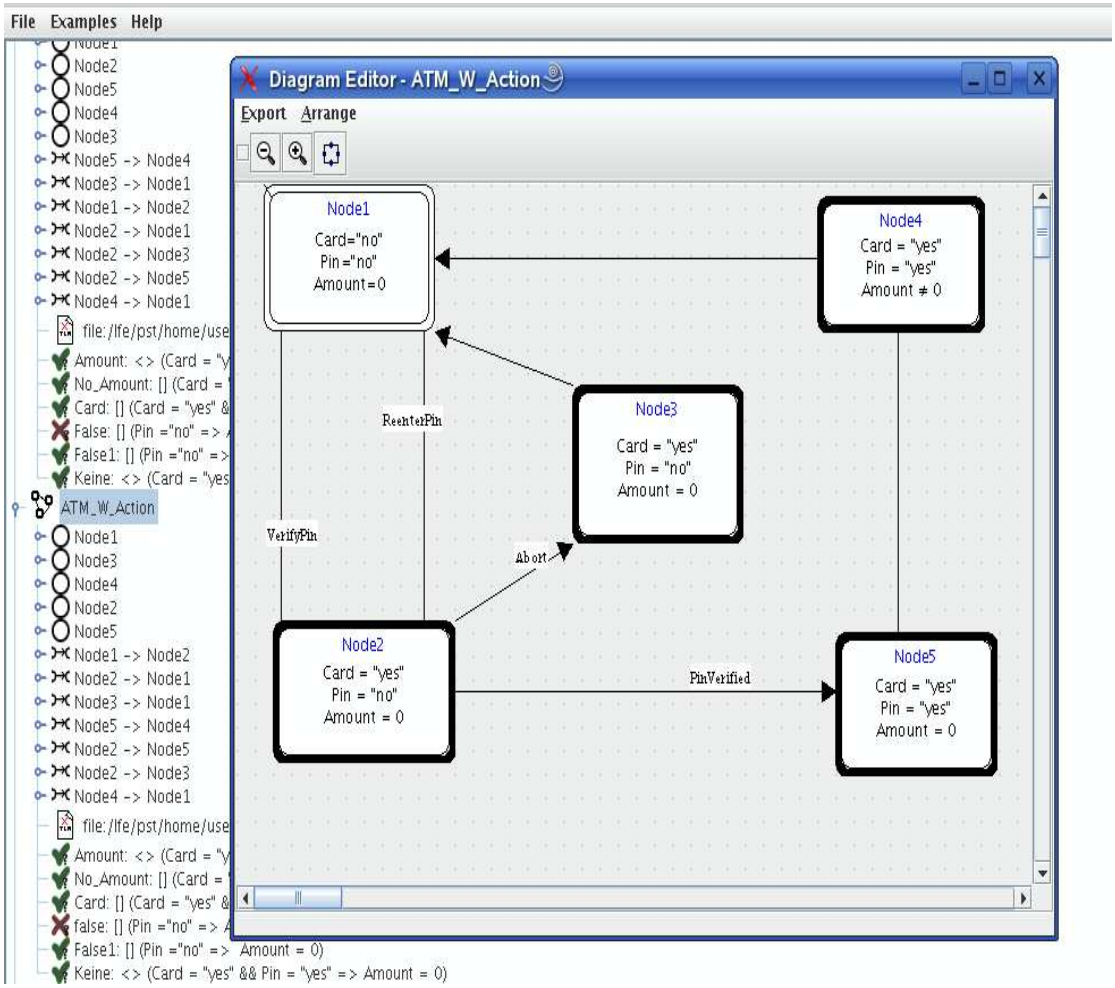


Figure 6.7: Predicate Diagram of ATM

6.5 Result and Discussion

In the current chapter, we proposed our prototype tool DAMAS. DAMAS is used to compile the UML state machines into graph models and to verify the desired features automatically. In particular, we use DAMAS to generate the executable UML state machines and the predicate diagrams of UML software designs.

We illustrated how we apply our prototype DAMAS to use our pre-defined rules for transforming the graph models of the software designs into new models that represent the approach of predicate diagrams. We used DAMAS to compile the graph models into the standard XMI file for DIXIT model checker. In this case, we can use the DIXIT model checker to verify the desired properties against the diagrams.

We have introduced how we reduce the state space of the software model checking by using

the approach of executable state machines. DAMAS is used here to transform automatically the graph models of UML state machines into executable state machines models and to create the UML state machines to be checked using HUGO model checker.

In Appendix B we give a short introduction to the use of DAMAS by presenting the SSL-Handshake protocol verification with DAMAS.

7 Conclusion

The work presented in my thesis focuses on model transformations of UML software designs and verifying properties of software designs by using the software model checking designs.

A fundamental discussion about temporal logic and model checking theories is presented in the second chapter. In this chapter we introduce two case studies: the state machine of ATM and the state machine of two-phase commit protocol. We use the HUGO model checker to verify whether the desired properties are indeed valid against the model.

The third chapter illustrates basic principles of graph grammar and graph transformation system. We discuss in this chapter how we create the graph models from the observed scenario. A general theory of graph formalism and semantic definitions of graph models (host-graph, final-graph, left hand side, ... etc) are also presented in this chapter. The transformation rules are used here to transform the host−graph into final one by adding or removing vertices as specified in the pre-defined rules. We introduced several useful tools and choose the tool AGG to run our case studies.

The fourth chapter contains an overview in the Unified Modeling Language (UML topics). So far, we have covered the basics principles of UML diagrams, we tried to research in the semantic definitions of UML state machines. We presented in this chapter the concepts of executable state machines to flat the UML state machines into simple states and actions with guards. We compiled then both of UML state machines and executable state machines to graph models and used the HUGO model checker to get a very useful result. We reduced the state space of model checking software using our transformation strategy about 10 percent. We used also in this chapter HUGO to see whether the functional properties, expressed in a formal logic like LTL (linear temporal logic), do hold in our case studies.

In chapter 5 we provided the reader with an overview of the security protocols especially SSL-record protocol, which is world wide used protocol to assure the connection between the server and client. As a case study we choose the SSL-Handshake protocol and verify this protocol in JESSIE software, we used JML (Java Modeling Language) assertions which must inserted inside the Java code of JESSIE. The Bandera tool are also used here to verify special security properties. Our transformation techniques are implemented on SSL−Handshake protocol to transform it into new designs that are suitable for software model checking. We compiled the

created graph models to the input language of the DIXIT model checker. The DIXIT model checker shows us if the desired security properties are verified.

The last chapter presented our prototype tool DAMAS. We designed DAMAS to compile and to transform the model (case study) into the desired graph automatically. DAMAS provides the user the ability to transform automatically the UML state machine diagram either into executable state machines or into new models suitable for software model checkers like HUGO and DIXIT.

7.1 Further Work

The results presented in my thesis indicate that the future for applying the approach of graph grammar and graph transformation systems on UML state machine and model checking systems is extremely promising. The integration of graph transformation techniques with software model checking designs allows to make use of the variety of verification concepts. Near future work in UML state machines is to add more transformation rules to manipulate the graph model of UML state machines into other model like Petri nets or others.

There is much further work that should be done in the graph transformation strategies. We point out a possible direction for further research in the abstraction techniques to create a new UML state machine models that guarantee to reduce the state space of the software model checking more than 30 percent.

For our practical implementation, the tool that we developed is a prototype tool and needs to be improved. We could extend our tool to manipulate the graph model using directly implementation of graph with model checker Spin, we could also design a user friendly interface for DAMAS to show directly the result of transformations and the result of the model checker.

Appendix A

Textual UML format(UTE) We introduce in appendix A the UTE text format of the case study state machines of ATM. UTE reflects the UML features in a text format, it is required as an input file to the HUGO model checker.

The UTE textual format for the state machines of ATM case study is shown as following:

```
model untitledModel {
  class Bank{
      signature{
      attr atm : ATM;
      attr cardValid : boolean = true;
      attr numIncorrect : int = 0;
      attr maxNumIncorrect : int = 2;
      operation verifyPIN();
      reception done();
    }
    behaviour {
      states {
        simple Idle;
        initial top_initial0;
        final top_final0;
        concurrent Verifying {
        composite Verifying_region0{
        simple VerifyingPIN;
        simple PINCorrect{
        entry numIncorrect = 0;
         }
```

```
 simple PINIncorrect;
 junction Verifying_region0_junction0;
 }
 composite Verifying_region1{
 simple VerifyingCard;
 simple CardValid;
 junction Verifying_region1_junction1;
}
}
join top_join0;
join top_join1;
junction top_junction2;
fork top_fork0;
}
transitions{
Verifying.Verifying_region0.VerifyingPIN ->
Verifying.Verifying_region0
Verifying_region0_junction0{
  guard true;
  effect ;
}
Idle -> top_final0{
  trigger done;
  guard true;
  effect;
}
Idle -> top_fork0{
  trigger verifyPIN;
  guard true;
  effect ;
}
top_join1 -> Idle{
  guard true;
  effect atm.pinVerified();
}
Verifying.Verifying_region1
 Verifying_region1_junction1 ->
```

```
        Idle {
  guard !(cardValid);
  effect atm.abort();
}
Verifying.Verifying_region1
Verifying_region1_junction1 ->
Verifying.Verifying_region1.CardValid{
  guard cardValid;
  effect;
}
Verifying.Verifying_region0
Verifying_region0_junction0 ->
Verifying.Verifying_region0.PINCorrect{
  guard true;
  effect;
}
top_junction2 -> Idle{
  guard numIncorrect < maxNumIncorrect;
  effect numIncorrect++; atm.reenterPIN();
}
top_junction2 -> Idle{
guard !(numIncorrect < maxNumIncorrect);
effect cardValid = false; atm.abort();
}
Verifying.Verifying_region1.CardValid ->
  top_join1{
  guard true;
  effect;
}
top_fork0 -> Verifying.Verifying_region0.
    VerifyingPIN{
  guard true;
  effect;
}
Verifying.Verifying_region1.CardValid ->
top_join0{
  guard true;
```

```
  effect;
}
Verifying.Verifying_region1.VerifyingCard ->
Verifying.Verifying_region1
Verifying_region1_junction1{
 guard true;
 effect;
}
top_join0 -> top_junction2{
  guard true;
  effect;
}
Verifying.Verifying_region0.PINCorrect ->
top_join1{
  guard true;
  effect;
}
top_fork0 -> Verifying.Verifying_region1.
VerifyingCard{
  guard true;
  effect;
}
Verifying.Verifying_region0.PINIncorrect ->
top_join0{
  guard true;
  effect;
}
Verifying.Verifying_region0
Verifying_region0_junction0 ->
Verifying.Verifying_region0.PINIncorrect{
  guard true;
  effect;
}
top_initial0 -> Idle{
  guard true;
  effect;
}
```

```
    }
  }
}
class ATM {
  signature {
    attr bank : Bank;
    reception abort();
    reception reenterPIN();
    reception pinVerified();
  }
  behaviour{
    states{
      initial top_initial0;
      simple Idle;
      simple Verifying;
      simple ReturningCard;
      composite GivingMoney{
        initial GivingMoney_initial1;
        simple Counting;
        simple Dispensing;
        final GivingMoney_final0;
      }
      final top_final1;
      simple AmountEntry;
      simple PINEntry;
    }
    transitions {
    GivingMoney -> ReturningCard {
       guard true;
       effect;
      }
      GivingMoney.Counting ->
      GivingMoney.Dispensing{
        guard true;
        effect;
      }
      GivingMoney.GivingMoney_initial1 ->
```

```
GivingMoney.Counting{
  guard true;
  effect;
}
Verifying -> AmountEntry{
trigger pinVerified;
guard true;
effect ;
}
GivingMoney.Dispensing ->
GivingMoney.GivingMoney_final0{
  guard true;
  effect ;
}
Verifying -> ReturningCard{
  trigger abort;
  guard true;
  effect;
}
ReturningCard -> top_final1{
  guard true;
  effect bank.done();
}
Idle -> PINEntry {
  guard true;
  effect ;
}
Verifying -> PINEntry {
  trigger reenterPIN;
  guard true;
  effect ;
}
PINEntry -> Verifying {
  guard true;
  effect bank.verifyPIN();
}
AmountEntry -> GivingMoney {
```

```
          guard true;
          effect ;
        }
        ReturningCard -> Idle {
          guard true;
          effect ;
        }
        top_initial0 -> Idle {
          guard true;
          effect ;
        }
      }
    }
  }
}
```

Appendix B

Model checking SSL-Handshake protocol via DAMAS In Appendix B we present the verification of the SSL-Handshake protocol via our tool DAMAS. As we explained in chapter 4 we first have to represent the protocol as UML state machine. Usually we use MagicDraw to design our case studies. Figure 7.1 shows the UML state machines of SSL-Handshake protocol. The left side of figure 7.1 shows the client's state machine, whereas the right side shows the server's state machine. Let's save these state machines as XMI file using MagicDraw and name

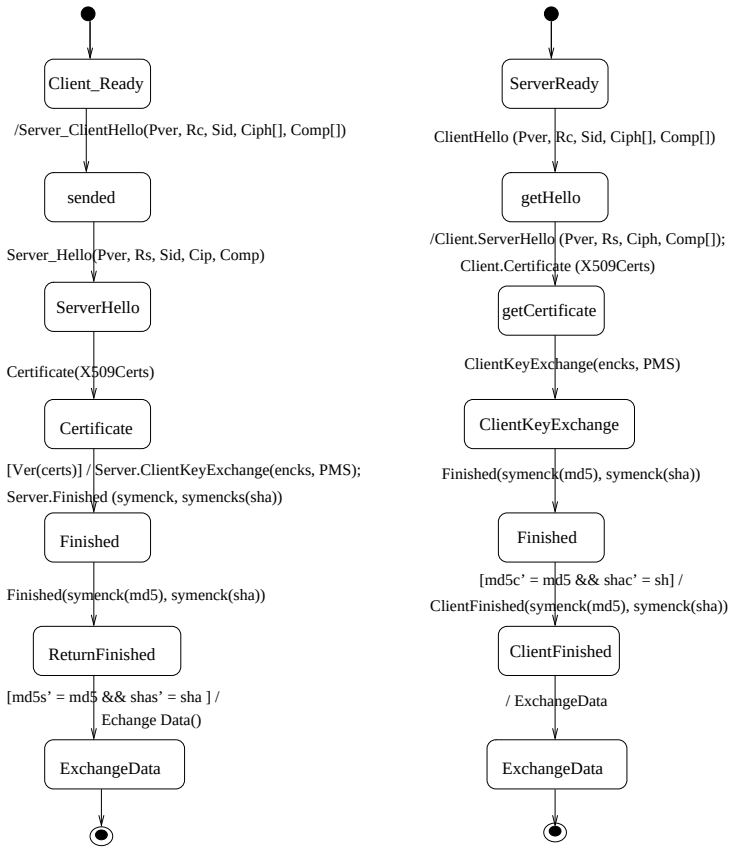


Figure 7.1: SSL Handshake protocol as UML State Machines

the file $SSL.xmi$. To compile $SSL.xmi$ into graph model using our prototype tool DAMAS, we have to write the following command:

```
DAMAS Graph "SSL.xmi" "SSL.gxx"
```

The new created file `"SSL.gxx"` represents the Host-Graph model of SSL-Handshake protocol. We can open the file using AGG Editor to show the Host-Graph model. Figure 7.2 shows the Host-Graph via AGG. We must now choose the appropriate rules to transform the Host-Graph

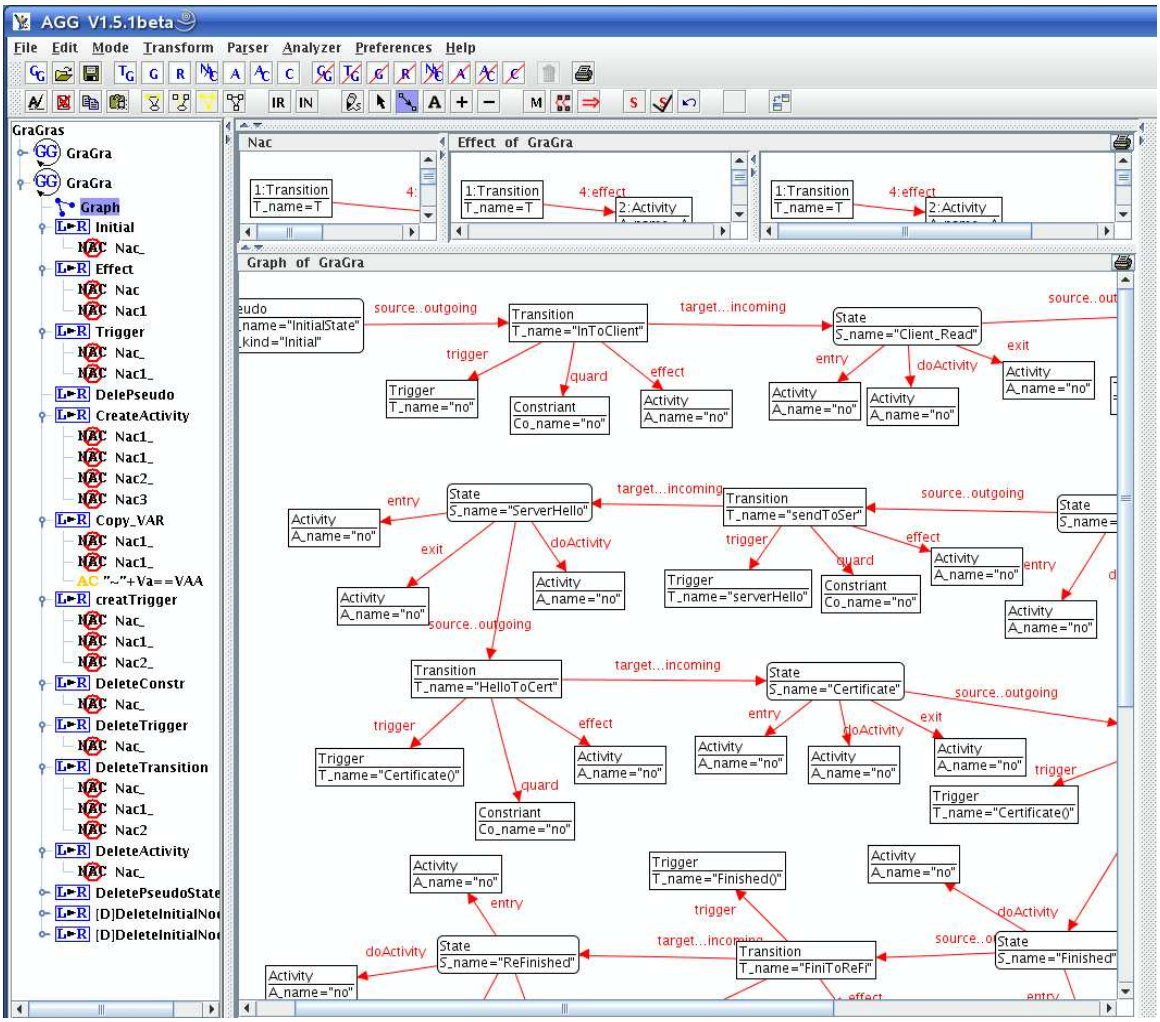


Figure 7.2: SSL Handshake protocol as Graph Model

into predicate graph model. The transformation rules are executed by clicking the "S" button in the Toolbar of AGG. Figure 7.3 shows the final graph which represents the predicate diagram in AGG. To compile the final graph to new xmi file that specifies the predicate diagram and open

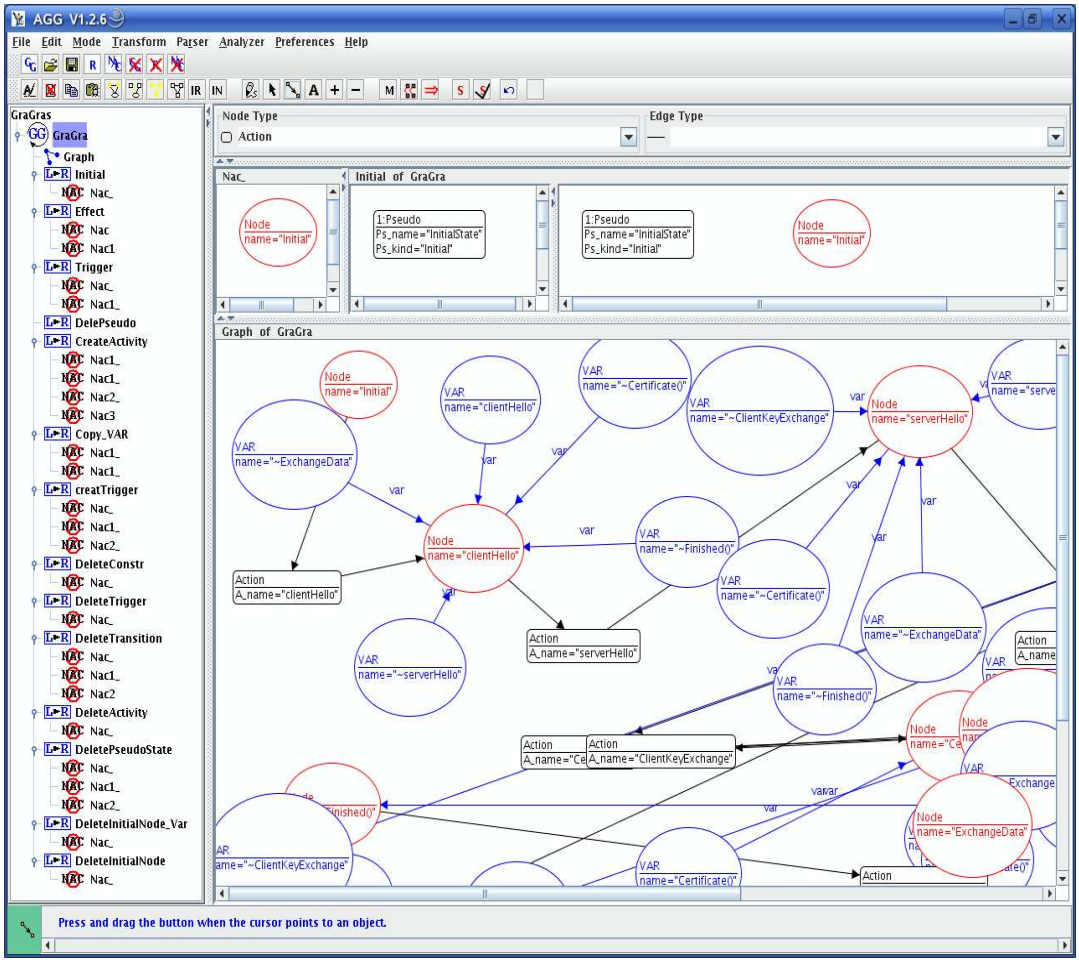


Figure 7.3: SSL Handshake protocol as Graph Model

it with $DIXIT$ model checker, we have first to add TLA specification file. We use usually our pre-defined rule to add the required TLA specification.

To compile the final graph into new input file for DIXIT, we have to write the following command:

```
DAMAS Predicate SSL.gxx SSLPredicate.xmi
```

Figure 7.4 shows the predicate diagram of SSL Handshake protocol after opening the file `SSLPredicate.xmi` using DIXIT Editor.

We can now use the tool DIXIT to verify the security properties. For example, to verify if the client sends the message `HelloServer`, and the next step is to receives the message `HelloClient`, this property is written in temporal logic as follows:

```
[] (ClientHello  => ServerHello)
```

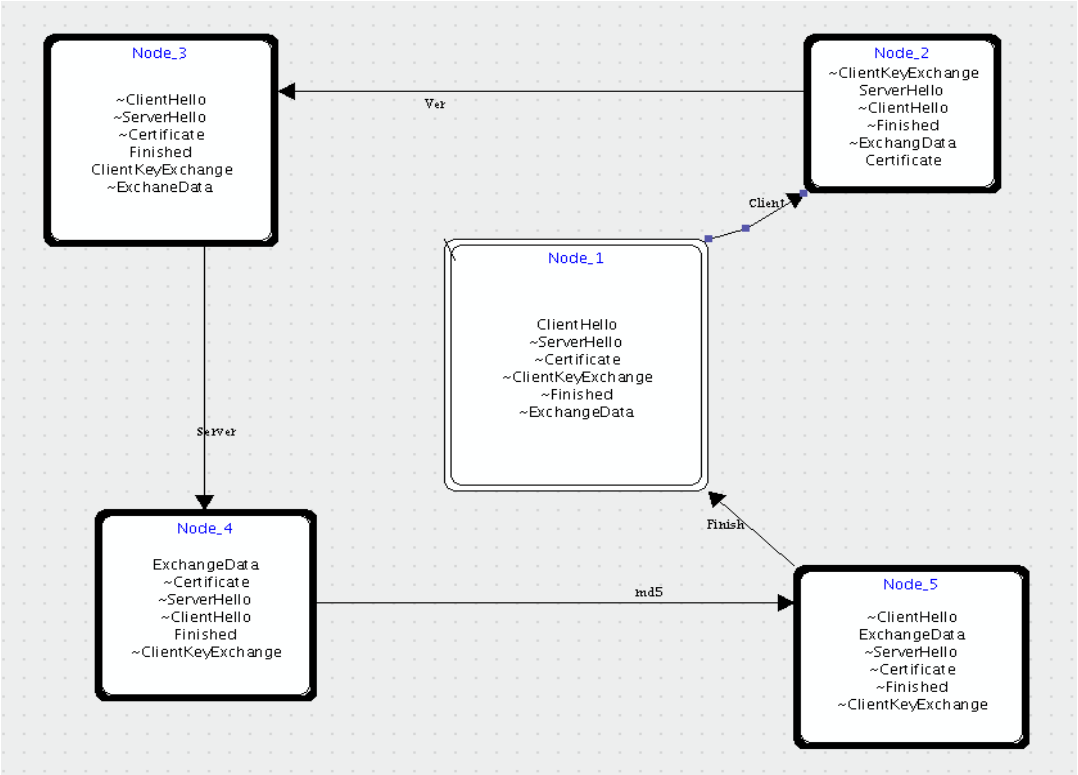


Figure 7.4: SSL Handshake protocol as Predicate Diagram

The result of DIXIT to verify the previous property using SPIN is :`verified.`

Another example is the property:

```
[] (ClientHello  && Finished => ExchangeData)
```

The DIXIT model checker shows the result: `verified.`

Appendix C

Verifying SSL-Handshake via Bandera Bandera [BA00,CDH+00] is a tool-set for translating Java programs to the input of existing model checkers, such as SMV and SPIN to test and verify the concurrent system. We use Bandera in our specification to check if the required properties are verified in SSL-Handshake protocol or not.

Bandera takes as input Java source code and a software requirement formalized in Bandera's temporal specification language, and it generates a program model and specification in the input language of one of several existing model-checking tools. Both program **slicing** and **user extensible abstract interpretation** components are applied to customize the program model to the property being checked. When a model-checker produces an error trail, Bandera renders the error trail at the source code level and allows the user to step through the code along the path of the trail while displaying values of variables and internal states of Java lock objects.

We introduce the abstracted Java code of SSL-Handshake protocol and insert some Bandera assertion to check if the Handshake protocol is indeed implemented in the software or not.

Bandera Interface Figure 7.5 displays the main window of the BUI (Bandera User Interface) with some example *code loaded*. The main window contains two panels: the left panel is the *project panel* and the right panel is the *code panel*. The Project panel contains a tree that organizes the packages, classes, fields and methods of the Java software being analyzed. Selecting a node in the project panel brings up a detailed view of the selected object in the code panel. For instance, in the example display in figure 7.5, selecting the **HandshakeClient** class from the left hand side displays the class and his code structure in the code panel (right hand side).

Bandera Specification Language Source code properties to be checked are written in the Bandera Specification Language (BSL). BSL is based on a collection of fieldtested temporal specification patterns [DAC99a] that allow users to write specifications in a stylized English format. These patterns essentially are parameterized macros that can be instantiated to one or more temporal logics such as LTL or CTL.

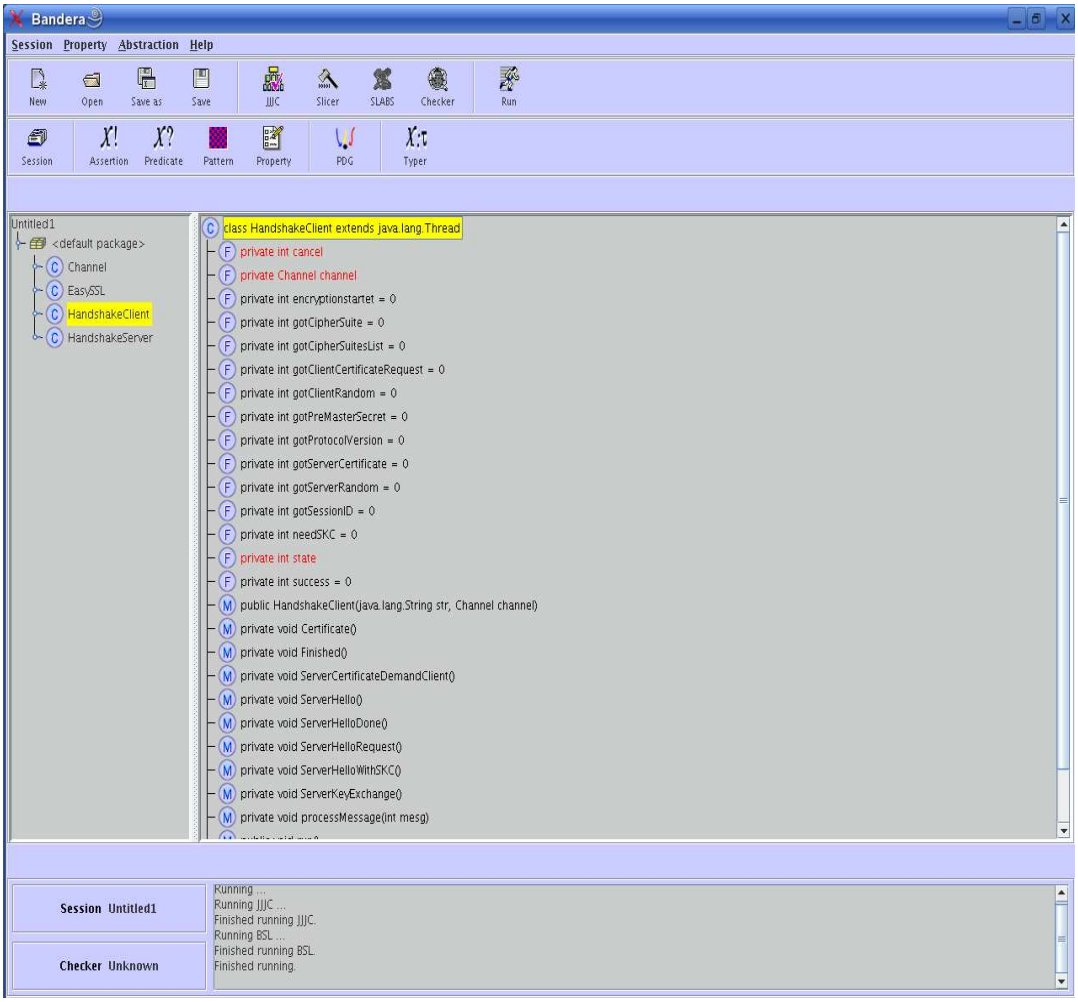


Figure 7.5: The Interface of Bandera Tool

Sessions Runs of Bandera are configured using *sessions*. A session is a record holding information about the file name(s) of the source code to be checked during the run, the property to be checked, the tool components that are to be enabled during the run, options and settings for the selected components, the particular back-end model checker to be used, and other miscellaneous information such as location of working directories into which temporary output should be dumped.

Multiple session records are held in a sessions file. When performing a new run of Bandera, the session record can be saved in a session file and loaded at a later time. This allows the user to avoid restating all option information, etc. Session records in a session file can also be processed in batch mode using a command line flag. This is useful for performing regression tests on software under development. For example, you might consider creating a session file holding all the checks that you usually run on a piece of software, then using the batch mode

facility to run all of the model-checker specified in the session file overnight.

Implementing SSL-Handshake with Bandera Figure 7.5 shows the implementation of the SSL-Handshake protocol in Java code. The implementation consists of four classes, **HandshakeClient**, **HandshakeServer**,**EasySSL** and **Channel** class. The HandshakeClient class implements sending and receiving the messages of the Client. The HandshakeServer Class implements sending and receiving the Server's messages. We use the class channel to save the message between the client and the server whereas the class EasySSL responsible for running the protocol.

We write the Bandera assertions BSL inside the Java Code of the SSL-Handshake protocol to verify sending and receiving messages between the server and the client.

The following shows the BSL assertions in the Java classes:

- To verify if the client indeed sends the message ServerHello, we define the boolean variable `SendClientHello` as the following.

  ```
  private boolean SendClientHello = false;
  ```

  if the variable has the value true, it means that the client sent the Message `HalloServer` to the server. We add the following pre and post BSL conditions before the $ServerHalloRequest$ procedure to verify the previous feature as follows:

  ```
  /**
  * @assert
  *  PRE  SendClientHello: (SendClientHello == false);
  *   POST SendClientHelloTrue: (SendClientHello==true);
  */
  private void ServerHelloRequest()
  {
  //create and send Client Hello
  //set ProtocolVersion
  gotProtocolVersion=1;
  //create random ...
  gotClientRandom=1;
  //create session ID
  gotSessionID=1;
  //available CipherSuites
  ```

```
gotCipherSuitesList=1;
writeToChannel(Channel.CLIENTHELLO);
SendClientHello = true;
}
```

- To verify if the server indeed receive the message ServerHello from the client, we define the boolean variable:

```
private boolean ReceiveServerHello = false;
```

if $ReceiveServerHello$ has the value true, it means that the server receive the message `HalloServer` from the client. We add the following pre and post Bandera condition before the $ServerHello()$ procedure to verify the previous feature as follows:

```
/**
* @assert
*  PRE  ReceiveServerHello: (ReceiveServerHello == false);
*   POST ReceiveServerHello: (ReceiveServerHello==true);
*/
private void ServerHello()
{
if (gotProtocolVersion==1)
{
gotProtocolVersion=2;
}
else
{
//Protocol failure
channel.cancel();
}
if (gotClientRandom==1)
{
gotServerRandom=1;
}
else
{
//Random failure
channel.cancel();
```

```
}
if (gotSessionID==1)
{
gotSessionID=2;
}
else
{
//Session failure
channel.cancel();
}
if (gotCipherSuitesList==1)
{
gotCipherSuite=1;
ReceiveServerHello=true;
}
else
{
//Cipher failure
channel.cancel();
}
}
```

- To verify that the client receives the server's certificate, we define the boolean variable:

```
private boolean ReceiveCertificate = false;
```

  if $ReceiveCertificate$ has the value true, it means that the client receive the the server's certificate. We add the following pre and post Bandera condition before the $Certificate()$ procedure to verify receiving the server's certificate as follows:

```
/**
* @assert
*  PRE  ReceiveCertificate: (ReceiveCertificate == false);
*  POST SendClientHelloTrue: (ReceiveCertificate == true);
*/
private void Certificate()
{
gotServerCertificate=1;
```

```
ReceiveCertificate = true;
}
```

- To verify that the server receives the client's KeyExchange message from the client, we define the boolean variable:

```
ReceiveServerKeyExchange = false;
```

if $ReceiveServerKeyExchange$ has the value true, it means that the server receives the client's KeyExchange message. We add the following pre and post Bandera condition before the $ServerKeyExchange()$ procedure to verify receiving the client's KeyExchange as follows:

```
/**
* @assert
*  PRE  ReceiveServerKeyExchange:
  (ReceiveServerKeyExchange == false);
*   POST ReceiveServerKeyExchange:
 (ReceiveServerKeyExchange == true);
*/
private void ServerKeyExchange()
{
if (needSKC==1)
{
needSKC=2;
ReceiveServerKeyExchange = true;
}
else
{
//The Message was not requested
channel.cancel();
}
}
```

- To verify that the server receives the client's Finished message, we define the boolean variable:

```
ReceiveFinished = false;
```

if $ReceiveFinished$ has the value true, it means that the server receives the client's Finished message. We add the following pre and post Bandera condition before the $Finished()$ procedure to verify receiving the Finished message as follows:

```
/**
* @assert
*  PRE  ReceiveFinished:  (ReceiveFinished == false);
*   POST ReceiveFinished: (ReceiveFinished == true);
*/
private void Finished()
{
encryptionstartet=2;
ReceiveFinished = true;
//ServerFinished verarbeiten
}
```

We define also inside the HandshakeServer Java class the following Bandera assertion to verify some properties of the server actions:

- To verify that the server receives the clientHello message and send the ServerHello message and his Certificate, we define the boolean variables inside the HandshakeServer class;

```
ReceiveClientHello = false; SendServerHello  = false;
                SendCertificate  = false;
```

if $ReceiveClientHello$ or $SendServerHello$ or $SendCertificate$ has the value true, it means that the server receives the clientHello message from the client or the server sends ServerHello message to the client or the server sends the Certificate message to the clients. We add the following pre and post Bandera condition before the $ClientHello()$ procedure to verify the previous properties:

```
/**
* @assert
*  PRE  ReceiveClientHello: (ReceiveClientHello == false)
* && (SendServerHello == false)
* && (SendCertificate == false);
*  POST ReceiveClientHello: (ReceiveClientHello == true)
* && (SendServerHello == true)
```

```
* && (SendCertificate == true);
*
*/
private void ClientHello()
{
//check ClientHello
gotClientRandom=1;
gotProtocolVersion=1;
gotSessionID=1;
gotCipherSuitesList=1;
//create and send Server Hello
gotServerRandom=1;
if (gotProtocolVersion==1)
{
gotProtocolVersion=2;
}
else
{
//SSL-Protocol-Version not supplied
channel.cancel();
}
if (gotSessionID==1)
{
gotSessionID=2;
}
else
{
//Session failure
channel.cancel();
}
if (gotCipherSuitesList==1)
{
gotCipherSuite=1;
}
else
{
//CipherSuite failure
```

```
channel.cancel();
}
if (gotCipherSuite==0)
{
//No CipherSuite
channel.cancel();
ReceiveClientHello = true;
}
if (ServerHellowithKeyExchange==0)
{
writeToChannel(Channel.SERVERHELLO);
SendServerHello = true;
}
else
{
writeToChannel(Channel.SERVERHELLOWITHSKC);
}
//create ServerCertificate message
writeToChannel(Channel.CERTIFICATE);
//create ServerKeyExchange message
if (ServerHellowithKeyExchange==1)
{
writeToChannel(Channel.SERVERKEYEXCHANGE);
ServerHellowithKeyExchange=2;
}
//create ServerCertificate message
if (ClientCertificateRequest==1)
{
writeToChannel(Channel.SERVERCERTIFICATEDEMANDCLIENT);
ClientCertificateRequest=2;
SendCertificate = true;
}
//create ServerHelloDone message
writeToChannel(Channel.SERVERHELLODONE);
}
```

- To verify that the server receives the client certificate's message, we define the boolean variable $ReceiveClientCertificate$ inside the HandshakeServer class;

```
ReceiveClientCertificate = false;
```

if $ReceiveClientCertificate$ has the value true, it means that the server receives the client's certificate. We add the following pre and post Bandera condition before the $Certificate()$ procedure for receiving the client's certificate:

```
/**
* @assert
*  PRE ReceiveClientCertificate:
       (ReceiveClientCertificate == false);
*  POST ReceiveClientCertificate:
  (ReceiveClientCertificate == true);
*/
private void Certificate()
{
if (ClientCertificateRequest==0)
{
//not requested
channel.cancel();
}
if (ClientCertificateRequest==1)
{
//no request sent yet
channel.cancel();
}
if (ClientCertificateRequest==2)
{
ClientCertificateRequest=3;
ReceiveClientCertificate = true;
}
}
```

- To verify that the server receives the client KeyExchange message from the client, we define the boolean variable $ReceiveClientKeyExchange$ inside the $HandshakeServer$ class;

```
ReceiveClientKeyExchange = false;
```

if $ReceiveClientKeyExchange$ has the value true, it means that the server receives the client's KeyExchange message. We add the following pre and post Bandera conditions before the $ClientKeyExchange()$ procedure to verify receiving the client's KeyExchange message:

```
/**
* @assert
*   PRE ReceiveClientKeyExchange:
(ReceiveClientKeyExchange == false);
*   POST ReceiveClientKeyExchange:
(ReceiveClientKeyExchange == true);
*/
private void ClientKeyExchange()
{
if ((gotClientRandom==1)&&(gotServerRandom==1))
{
gotPreMasterSecret=1;
ReceiveClientKeyExchange = true;
}
else
{
//missing random for PMS-Create
channel.cancel();
}
}
```

- To verify that the server sends the Finished message to the client, we define the boolean variable $SendFinished$ inside the $HandshakeServer$ class;

```
SendFinished = false;
```

if $SendFinished$ has the value true, it means that the server sends the Finished message to the client. We add the following pre and post Bandera conditions before the $Finished()$ procedure to verify sending the Finished message to the client.

```
/**
* @assert
*   PRE SendFinished: (SendFinished == false);
```

```
    *   POST ServerHashPost: (SendFinished == true);
    */
    private void Finished()
    {
    //ClientFinished verarbeiten
    if (gotPreMasterSecret==1)
    {
    encryptionstartet=1;
    writeToChannel(Channel.FINISHED);
    SendFinished = true;
    }
    else
    {
    //Cannot change to encryption because of missing PMS
    channel.cancel();
    }
    }
```

Verifying BSL via Spin Bandera generates a program model and specification in the input language of one of several existing model-checking tools including SPIN. We use the model checker SPIN to verify our Bandera specification assertions, the result of the model checker SPIN is shown as following:

```
pan.exe -n -m1000000 -w18 -e
(Spin Version 4.1.3 -- 24 April 2004)

Full statespace search for:
        never claim             - (not selected)
        assertion violations    +
        cycle checks            - (disabled by -DSAFETY)
        invalid end states      +

State-vector 888 byte, depth reached 21963, errors: 0
   75524 states, stored
  139406 states, matched
  214930 transitions (= stored+matched)
  548201 atomic steps
```

```
hash conflicts: 19397 (resolved)
(max size 2^18 states)

97.145  memory usage (Mbyte)

*** END ***
```

Bandera shows us also a small window that written inside *is verified*, it's means that our BSL assertions in the Java code of the protocol are valid. That is, the Java code specifies the properties of SSL-Handshake protocol.

List of Tables

List of Figures

Index

Bibliography

[ABKS] M. Ankerst, M.M. Breunig, H.-P. Kriegel, and J. Sander. The internet encyclopedia of philosophy. In *http://www.iep.utm.edu/*.

[AGG] Agg homepage. In *http://tfs.cs.tu-berlin.de/agg*.

[arg] In *http://argouml.tigris.org*.

[BA00] Bandera. In *Extracting safe finite-state models from source code. URL: www.cis.ksu.edu/santos/bandera/*, 2000.

[BAMP83] M. Ben-Ari, Z. Manna, and A. Pnueli. The temporal logic of branching time. In *Acta Informatica 20*, pages 207–226, 1983.

[BJ08] A. Bauer and J. Juerjens. Security protocols, properties, and their monitoring. In *The 4th International Workshop on Software Engineering for Secure Systems (SESS'08 @ ICSE)*, 2008.

[BLPY97] J. Bengtsson, K.G. Larsen, P. Pettersson, and Wang. Yi. Uppaal - a tool suite for automatic verification of real-time systems. In *R. Alur, T.A. Henzinger, and E.D. Sonntag, editors, Hybrid Systems III - Verification and control, volume 1066 of LNCS*, pages 232–243, Springer, 1997.

[BRJ99] Grady. Booch, James. Rumbaugh, and Ivar. Jacobson. The unified modeling language user guide. In *Addison Wesley Longman, Inc, Springer-Verlag*, 1999.

[Bry] R. E. Bryant. Graph-based algorithms for boolean function manipulation. In *IEEE Transactions on Computers -35(8)*, pages 677–691.

[CDH+00] J.C. Corbett, M.B. Dwyer, J. Hatcliff, S. Laubach, C. Pasareanu, S. Robby, and H. Zheng. Bandera: Extracting finite-state models from java source code. In *Proceedings of the 22nd International Conference on Software Engineering. IEEE Computer Society Press*, 2000.

[CE81] E. M. Clarle and A. Emerson. Design and synthesis of synchronisation skeletons using branching time temporal logic. In *Logic of Programs. Proceedings of Workshop, volume 131 of Lecture Notes in Computer Science*, pages 52–71, 1981.

[CER79] V. Claus, H. Ehrig, and G. Rozenberg. Proc. int. workshop on graph grammars and their application to computer science and biology. In *LNCS 73, Springer Verlag*, 1979.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications. In *ACM Transactions op Programming Languages and Systems*, pages 244–263, 1986.

[Cho56] Noam. Chomsky. Three models for the description of language. In *IRE Transactions on Information Theory (2)*, pages 113–124, 1956.

[CL] Yoonsik. Cheon and Gary.T. Leavens. Extended static checker for java version 2. In *http://secure.ucd.ie/products/opensource/ESCJava2/*.

[CL02] Yoonsik. Cheon and Gary.T. Leavens. A runtime assertion checker for the java modeling language (jml). In *Hamid R. Arabnia and Youngsong Mun, editors, Proceedings of the International Conference on Software Engineering Research and Practice (SERP '02), Las Vegas, Nevada, USA*, pages 322–328, June 24-27, 2002.

[CMMag] Dominique. Cansell, Dominique. Me'ry, and Stephan. Merz. Predicate diagrams for the verification of reactive systems. In *2nd Intl. Conf. on Integrated Formal Methods (IFM 2000), volume 1945 of Lecture Notes in Computer Science, Dagstuhl, Germany*, pages 40–51, November 2000. Springer-Verlag.

[CRY99] In *http://www.gnu.org/software/gnu-crypto/*, 1999.

[DAC99a] M.B. Dwyer, G.S. Avrunin, and J.C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st International Conference on Software Engineering*, 1999.

[DAC99b] M.B. Dwyer, G.S. Avrunin, and J.C. Corbett. Pattern in property specifications for finite-state verification. In *Proceedings of teh 21st International Conference on Software Engineering*, May, 1999.

[dLV02] J. de Lara and H Vangheluwe. Atom3: A tool for multi-formalism modelling and meta-modelling. In *LNCS 2306, Springer. See: http://atom3.cs.mcgill.ca*, pages 174–188, 2002.

[EE05] H. Ehrig and K. Ehrig. Overview of formal concepts for model transformations based on typed attributed graph transformation. In *Proceedings of GraMoT 2005, Electronic Notes in Theoretical Computer Science. Elsevier*, 2005.

[EEHP04] H. Ehrig, K. Ehrig, A. Habel, and K.H. Pennemann. Constraints and application conditions: From graphs to high-level structures. In *F. Parisi- Presicce, P. Bottoni, and G. Engels, editors, Proc. 2nd Int. Conference on Graph Transformation (ICGT'04), LNCS 3256, Rome, Italy*, pages 287–303, October 2004.

[EEKR99] H. Ehrig, G. Engels, H.-J. Kreowski, and G Rozenberg. Handbook of graph grammars and computing by graph transformation. In . *Vol 1. Foundations. World Scientific.*, 1999.

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. Fundamentals of algebraic graph transformation. In *(Monographs in Theoretical Computer Science. An EATCS Series), Springer-Verlag New York, Inc., Secaucus, NJ*, 2006.

[EKRR91] H. Ehrig, H.J. Kreowski, G. Rozenberg, and A. Rosenfeld. Proc. 4th int. workshop on graph grammars and their application to computer science. In *LNCS 532, Springer Verlag*, 1991.

[FKK96] A.O. Freier, P.P. Karlton, and C. Kocher. The ssl protocol version 3.0. In *Internet-draft, draft-freier-ssl-version03-02.txt*, Nov. 1996. http://wp.netscape.com/eng/ssl3/draft302.txt.

[FMM05] Loic. Fejoz, Dominique. M'ery, and Stephan. Merz. a graphical toolkit for predicate abstractions. In *Fourth International Workshop on Automated Verification of Infinite-State Systems-AVIS'05*, pages 39–48, 2005.

[GJM91a] C. Ghezzi, M. Jazayeri, and D. Mandrioli. Fundamentals of software engineering. In *Prentice Hall Int.*, 1991.

[GJM91b] M. Ghezzi, Jazayeri., and D. Mandrioli. Fundamentals of software engineering. In *Prentice Hall Int.*, 1991.

[Gro03a] Object Management Group. Omg meta object facility (mof), version 1.4. In *URL: http://www.omg.org/cgi-bin/apps/doc?formal/02-04-03.pdf access: 2004-06-28*, 12th June 2003.

[Gro03b] Object Management Group. Omg idl syntax and sematics. In *defined in the Common Object Request Broker: Architecture and Specification, version 2,*

The latest version of CORBA version 2.0 is available at https://www.omg.org/technology/documents/formal/corba_2.htm, March 2003.

[HDR02] J. Hatcliff, M. Dwyer, and Robby. Specification and verification of reactive systems (cis 842). In *Lecture notes," https://www.cis.ksu.edu/ hatcliff/842"*, 2002.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. In *Communications of the ACM 12(10)*, pages 576–583, October 1969.

[Hol] G. Holzmann. Basic spin manual. In *SPIN Online Documentation, http://spinroot.com/spin/Man/Manual.html.*

[Hol93] G.J. Holzmann. Design and validation of protocols: A tutorial. In *Computer Networks and ISDN Systems, vol. 25, no. 9*, pages 981–1017, 1993.

[Hol04a] G. J. Holzmann. The spin model checker: Primer and reference manual. In *Addison-Wesley*, 2004. ISBN 0-321-22862-6.

[Hol04b] Gerad J. Holzmann. The spin model checker. In *Lucent Technology Inc, Bell Laboratories*, pages http://netlib.bell–labs.com/netlib/spin/whatispin.html, 2004.

[Hom02] Fujaba Homepage. Universität paderborn. In *http://www.fujaba.de*, 2002.

[Itl] In *http://nusmv.irst.itc.it*.

[JES] In *http://www.nongnu.org/jessie/*.

[JML] In *http://en.wikipedia.org/wiki/Java_Modeling_Language*.

[Jue04] Jan. Juerjens. Secure systems development with uml. In *Springer-Verlag, Heidelberg,*, 2004.

[Jue06] Jan. Juerjens. Security analysis of crypto-based java programs using automated theorem provers. In *21st International Conference on Automated Software Engineering, IEEE/ACM*, pages 167–176, ASE 2006.

[Jue07] Jan. Juerjens. Automated security verification for crypto protocol implementations: Verifying the jessie project. In *Seventh International Workshop on Automated Verification of Critical Systems, Oxford*, Sep. 10-12, 2007.

[KAF] In *http://www.kaffe.org/*.

[Kir06] David Kirscheneder. Methode zum vergleich von java-programmen und uml-modellen. In *Diplomarbeit, TUM*, April, 2006.

[KIV86] KIV. Karlsruhe interactive verifier. In *http://i11www.ira.uka.de/ kiv/*, 1986.

[KKHK] H.J. Kreowski, R. Klempien-Hinrichs, and S. Kuske. In *Some Essentials of Graph Transformation. University of Bremen, Department of Computer Science. Bremen*.

[KM11] Alexander. Knapp and Stephan. Merz. Model checking and code generation for uml state machines and collaborations. In *Dominik Haneberg. Gerhard Schellhorn, and Wolfgang Reif, editor Proc. 5th Wsh. Tools for System Design and Verification, Institut für Informatik, Universität Augsburg*, pages 59–64, Technical Report 2002-11.

[Kwo00] G. Kwon. Rewrite rules and operational semantics for model checking uml statecharts. In *: A. Evans, S. Kent and B. Selic, editors, Proc.* 3^{nd} *Int. Conf. UML, Lect. Notes Comp. Sci. 1939*, pages 528–440, 2000.

[KZDS] Jun. Kong, Kang. Zhang, Jing. Dong, and Guanglei. Song. A graph grammar approach to software architecture verification and transformation. In *the university of Texas at Dallas. Richardson, Texas 75080-0688, USA*.

[Lam94] L. Lamport. The temporal logic of actions. In *ACM Transactions on Programming Languages and Systems*, pages 16(3): 872–923, 1994.

[LLMC04] T. Levendovszky, L. Lengyel, G. Mezei, and H. Charaf. A systematic approach to metamodeling environments and model transformation systems in vmts. In *2nd International Workshop on Graph Based Tools (GraBaTs); workshop at ICGT 2004, Rome, Italy*, 2004.

[LMM99] D. Latella, I. Majzik, and M. Massink. Automatic verification of a behavioural subset of uml statechart diagrams using the spin model-checker. In *Formal Aspects Comp. 11*, pages 637–664, 1999.

[LP99] J. Lilius and I. P. Paltor. Formalising uml state machines for model checking. In *: R. B. France and B. Rumpe, editors, Proc.* 2^{nd} *Int. Conf. UML, Lect. Notes Comp. Sci. 1723*, pages 430–445, 1999.

[(Lt] F.S.E. (Ltd). Fdr 2.0. failure divergence refinement. In *User Manuel 2003*.

[Mag04] Magicdraw. In *http://www.magicdraw.com*, 2004.

[Mey92] Betrand. Meyer. Applying design by contract. In *Computer, 25(10)*, pages 40–51, 1992.

[MLS97] E. Mikk, Y. Lakhnech, and M. Siegel. Hierarchical automata as model for statecharts. In *: R. K. Shyamasundar and K. Ueda, editors, Proc. 3^{nd} Asian Computing Science Conf., Lect. Notes Comp. Sci. 1345*, pages 181–196, 1997.

[MLSH99] E. Mikk, Y. Lakhnech, M. Siegel, and G.J. Holzmann. Implementing statecharts in promela/spin. In *Proc. Wsh. Industrial-Strength Formal Specification Techniques*, 1999.

[MP92] Z. Manna and A. Pnueli. The temporal logic of reactive and concurrent systems. In *Springer-Verlag*, 1992.

[MQR95] M.X. Moriconi, L. Qian, and R. A. Riemenschneider. Correct architecture refinement. In *IEEE Trans. Software Eng., 21(4)*, pages 356–372, 1995.

[NSN99] F. Norman, Schneidewind., and Allen.P. Nikora. Issues and methods for assessing cots reliability, maintainability, and availability. In *Proceedings of the First Workshop on Ensuring Successful COTS Development, 2 1 st International Conference on Software Engineering, Los Angeles, California*, May 22nd, 1999. 4 pages.

[Nus02] Nusmv. In *http://nusmv.irst.itc.it/*, 2002.

[oP] The Internet Encyclopedia of Philosophy. In *http://www.iep.utm.edu/*.

[ORG] In *http://gcc.gnu.org/*.

[PA69] J.L. Pfaltz and Rosenfeld A. Web grammars. In *Int. Joint Conference on Artificial Intelligence*, pages 609–619, 1969.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, pages 46–57, 1977.

[Poe04] Poseidonuml. In *http://www.gentelware.com/product/index.php3*, 2004.

[Pra71] T.W. Pratt. Pair grammars: graph languages and string-to-graph translations. In *Journal of Computer and System Sciences*, pages 5:560–595, 1971.

[Qua98] Terry. Quatrani. Visual modeling with rose and uml. In *Addison-Wesley. Reading, MA*, 1998.

[Rat] In *http://www.rational.com*.

[Roz97] Grzegorz. Rozenberg. Handbook of graph grammars and computing by graph transformation. In *World Scientific Publishing Co. Pte. Ltd, LNCS. Springer-Verlag*, pages 5:560–595, 1997.

[RU71] N. Rescher and A. Urquhart. Temporal logic- springer-verlag. 1971.

[RVR+99] Alejandro. Ramirez, Philippe. Vanpeperstraete, Andreas. Rueckert, Kunle. Odutola, Jeremy. Bennett, and Linus. Tolke. Argouml. In *A tutorial and reference description*, 1999. The last version is presently available at http://www.opencontent.org/openpub/.

[Sch] A Schürr. Specification of graph translators with triple graph grammars. In *LNCS 903*, pages 151–163.

[SG95] M. Shaw and D. Garlan. Software architectur: Persepectives on an emerging discipline. In *Prentice Hall*, 1995.

[SKM01] Tim Schaefer, Alexander Knapp, and Stephan. Merz. Model checking uml state machines and collaborations. In *Scott D. Stoller and Willem Visser, editors, Proc. Wsh. Software Model Checking, volume 55(3) of Electr. Notes Theo. Comp. Sci.*, 2001. 13 pages.

[SM05] T. Schattkowsky and W. Müller. Transformation of uml statemachines for direct execution. In *Proc. 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05), Dallas, TX, USA*, September 2005.

[SP] In *http://spinroot.com/spin/whatispin.html*.

[SS99] H. Saidi and N. Shankar. Abstraction and model checking while you prove. In *Computer-Aided Verification, Volume 1633 of LNCS*, pages 443–454, 1999.

[TER99] G. Taentzer, Ermel, and C. Rudolf. Agg-approach. In *Language and Tool Environment, Graph Grammar Handbook 2: Specifications and Programming, World Scientific, LNCS*, 1999.

[Tog04] Borland Together. Togethersoft. In *Integrated and Agile Design Solutions. Borland*, 2004.

[VMT] Vmts web site. In *http://avalon.aut.bme.hu/tihamer/research/vmts*.

[VP04] D. Varro and A. Pataricza. Generic and meta-transformations for model transformation engineering. In *Baar, T., Strohmeier, A., Moreira, A., Mellor, S., eds.: Proc. UML 2004: 7th International Conference on the Unified Modeling Language. Volume 3273 of LNCS., Lisbon, Portugal*, pages 290–304, Springer 2004.

[VVP02] D. Varro, G. Varro, and A. Pataricza. Designing the automatic transformation of visual languages. In *Volume 44(2) of LNCS*, pages 205–227, 2002.