EXPLORING THE INTEGRATION OF MODEL-BASED FORMAL METHODS INTO SOFTWARE DESIGN EDUCATION

Except where reference is made to the work of others, the work described in this thesis is my own or was done in collaboration with my advisory committee. This thesis does not include proprietary or classified information.

Shuo Wang

Certificate of Approval:

John Hamilton Associate Professor Computer Science and Software Engineering	Levent Yilmaz, Chair Assistant Professor Computer Science and Software Engineering
Dean Hendrix Associate Professor Computer Science and Software Engineering	Stephen L. McFarland Acting Dean Graduate School

EXPLORING THE INTEGRATION OF MODEL-BASED FORMAL METHODS INTO SOFTWARE DESIGN EDUCATION

Shuo Wang

A Thesis

Submitted to

the Graduate Faculty of

Auburn University

in Partial Fulfillment of the

Requirements of the

Degree of

Master of Science

Auburn, Alabama
December 16, 2005

EXPLORING THE INTEGRATION OF MODEL-BASED FORMAL METHODS INTO SOFTWARE DESIGN EDUCATION

Shuo Wang

Signature of Author

Date of Graduation

VITA

Shuo Wang, son of Linfu Wang and Suran Guo, was born in October 1979 in Tianjin, the People’s Republic of China. He entered Georgia Institute of Technology in 1998 and received a Bachelor of Science degree in College of Computing in December 2002. Mr. Wang entered Graduate School at Auburn University in August, 2003.

THESIS ABSTRACT

EXPLORING THE INTEGRATION OF MODEL-BASED FORMAL METHODS INTO SOFTWARE DESIGN EDUCATION

Shuo Wang

Master of Science, December 16, 2005
(B.S., Georgia Institute of Technology, December 2002)

105 Typed Pages

Directed by Dr. Levent Yilmaz

Proper design analysis is indispensable to assure quality and reduce emergent cost due to faulty software. Teaching proper design verification skills early during the pedagogical development of a software engineer is crucial, as much analysis is the only tractable way of resolving software problems early when they are easy to fix. Besides, fundamental component of any engineering discipline, including software engineering, is the use of formal and sound techniques that facilitate analysis of artifacts produced by students. Yet, the impact of formal methods in software engineering practice, as well as education, is minuscule. The fundamental reasons why formal methods are not effectively utilized are attributed to (1) the impedance mismatch between the underlying mathematical underpinning of formal methods and students' semi-formal, if not informal, view of the design problem and (2) the lack of tool support for seamless and transparent integration of formal methods into software design education. This thesis suggests a strategy and tool support to improvement attainment of software design verification skills. The strategy illustrates how selective and pragmatic application of model-based verification methods can be used in software design education via tools that aim to bridge the gap between students' semi-formal design world-view and the formalism underlying formal methods.

ACKNOWLEDGEMENT

I would like express sincerely appreciation to Dr. Levent Yilmaz for his guidance, insight, and encouragement throughout the research to help me succeed.

I would also like to thank the rest of my thesis committee members, Dr. John Hamilton and Dr. Dean Hendrix for their valuable suggestions and comments.

Last but not least, I would like to thank my family and friends for their understanding, motivation and support.

Style manual or journal used: <u>Journal of SAMPE</u>

Computer software used: <u>Microsoft Word</u>

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

As modern systems are increasingly becoming reliant upon computing technologies, software that powers these platforms is emerging as a vital component for today's technology infrastructure. A study performed by the National Institute of Standards and Technology (NIST) reveals that erroneous and inefficient software products cost U.S. economy $59.5 billion annually in failed missions and lost productivity [42]. Clearly, the need for reliable software systems is critical as such systems are becoming pervasive in our lives. With the continuing growth of using software-intensive technology products, it will be even more important to attain higher levels of reliability and assurance. Various software verification methods have been introduced and applied [8] through software development stages to detect and eliminate errors as early as possible. Concomitantly, in academia, there has been considerable interest in developing more effective software design and verification techniques [1] and teaching them to new generation of software engineers [28].

1.1 The Need for Integration of Formal Methods into Software Design Education

The principle methods for complex system verification include simulation [8], testing [42], deductive verification [8], and model checking [6]. Simulation and testing both involve conducting experiments before deploying the system in the field. While simulation is performed using an abstraction (i.e. model) of the system, testing is performed on the actual product. In both cases, they usually involve providing certain inputs to the system or the system model and observing the corresponding outputs. These methods are common and cost-effective ways to find many errors. However, checking all possible interactions among all building blocks of the system using simulation and testing techniques is computationally intractable.

Deductive verification [8] refers to the use of axioms and proofs to prove the correctness of the software systems and usually applied in the verification of mission critical systems. Proofs were first constructed by hand and eventually software tools were built to facilitate the effort. Although deductive reasoning is widely recognized and accepted by computer scientists, the process is rather time consuming and can only be performed by experts who have proper education and experience in logical reasoning.

Model checking is a technique for formally verifying finite state concurrent systems based on formal methods [7]. Formal methods, mathematically based techniques that provide a framework to specify, define, and verify systems, can effectively reveal ambiguity, incompleteness, and inconsistency within complex systems. Use of formal methods does not automatically guarantee correctness. However, when used appropriately, these techniques have proven themselves to result in software products with higher levels of quality [9]. In model checking, specified systems are modeled as finite-state machines and its expected properties and behaviors are specified in temporal logic, and the process of verification can be performed automatically. The procedure normally uses an exhaustive search of the state space of the system to determine if some specification is true or not. While model checking has become increasingly popular in the industry, other formal methods have received relatively little attention. The introduction of formal methods, as a set of engineering disciplines and practices, into software design education has at least a two-fold benefit to the students [28]. First, it bridges the gap between theories in computer science and emerging industry practices. Second, it provides students valuable and pragmatic skills in the formal modeling and analysis of complex software systems that are beyond the scope of conventional informal verification methodologies.

1.2 The Challenges in Integrating Formal Methods into Software Design Education

The emerging trend of model-driven development [14] and model-driven architecture [30] suggests significant benefits when integrating model-based verification techniques into software design education. By learning the engineering discipline of applying these techniques to facilitate the development, analysis, and verification of software, students will grasp the significance of formal methods and gain valuable skills and experience in software design and modeling [28]. Strategies of integrating formal methods in the form of model-based verification into software design education have been difficult. Many academic institutions either completely avoid teaching formal methods or teach them in an isolated manner, with emphasis on notations rather than its underlying principles [15]. Carnegie Mellon University [15] has successfully integrated formal methods throughout its software engineering graduate program, but their approach cannot easily be adapted into undergraduate software design curriculum because undergraduate students do not possess the requisite skills to understand and apply formal methods. For these students, it is beneficial to participate in courses designed specifically toward them, such as courses that provide exposure to model checking without having students to burden themselves with high requirements in advanced background in mathematics. Concomitantly, new software tools should be developed to facilitate learning the pragmatic use of formal methods and to fill the gap between learning and practice to help the students accomplish following goals [47]:

- to use formal methods without getting into the quagmire of theoretical details
- to avoid steep learning curves about the syntax of a specific formal method by using alternative generic high-level constraint patterns to analyze designs
- to collaboratively analyze inconsistencies and design conflicts at least semi-formal reasoning within the realm of the actual industrial software development process

Using today's technology, students must rely on existing model checking tools to assist their learning experience. There are indeed capable model checking tools available, such as SMV [29] and SPIN [18]; yet, they share a common problem for a typical undergraduate student: These tools are difficult to learn and difficult to use, despite many accolades they have won from the industry, they are not designed for education purposes, and the learning curve is enormous, considering the range of functionalities they provide. Both model checkers are command-line applications requiring the user to memorize the meaning of all of their run-time parameters and options. In order to perform model checking, the software model which serves as an input to the model checkers need to be encoded using a separate but complex notation, almost as if the user is learning a new programming language. But perhaps the most serious problem of all, the model checkers do not facilitate the understanding or the learning of model checking by hiding the model checking processes from their users. No information is conveyed back to the users until the end, when model checking has been completed and lines of cryptic text-based results are dumped on the screen. To a typical undergraduate software engineering student, these text-based results probably make little sense.

The origin of the steep learning curve of the existing model checking tools can be traced back to almost twenty years ago when these tools were still in their infant stage of development. Computing at the time was a lot different than today. Computers were not only bulky and slow, but also extremely expensive. Computing resources were scarce and precious. As a result, most computer applications are written using structured programming languages in the most cost-effective manner to boast their performance. In order to do so, other aspects of the software, such as usability and maintainability, had been sacrificed. When building a software application to solve a particular problem, as long as the problem is solved within a reasonable amount of time and using a reasonable amount of computing resources, one could care less of how does the application solve the problem. As time has progressed and technology has been improved, modern computer platforms with the computation power one could only dream of just a few years ago have become accessible at much lower costs. The availability of high performance computing platforms has triggered a paradigm shift in the computing society regarding how to write software. Much more emphasis has been given to the learnability, flexibility, and robustness of the software instead of its crude performance, and the structured programming approach has been gradually shifted out and replaced by the object-oriented programming approach for the same purpose. It is quite ironic during the same period of time formal methods have gained acceptability in the industry and even some popularity when dealing with the verification of mission critical hardware systems [4]. This success has quickly led the application of the same approach on the verification of software systems. Model checking tools have become more robust, powerful, and feature-rich, and have a lot of potential to offer in the campaign against hidden software design flaws. However, it is built on top of an aging foundation. It is rather difficult to integrate these legacy tools directly with today's software design methodology being taught in undergraduate level classrooms.

1.3 Research Objective

Given the challenging obstacle of merging formal methods into software design education, the goal of this research is to enable the methodology of teaching formal methods in undergraduate level software design curricula, without having the students to be burdened by the vast amount of theoretical details and mathematical logic required to understand formal methods. This objective can be realized by developing a tool that integrates model checking into current software design methodology being taught in undergraduate software design courses. The design of this tool addresses the difficulties of teaching formal methods to undergraduate students in general, as well as the shortcomings of existing model checking tools for educational purposes. This tool abstracts unnecessary theoretical details away from its users while emphasizing flexible interaction with during the model checking process. As a result, the students may not only gain a novel verification technique to validate their software design in a quick and effective manner, but also attain fresh insights on how model checking works.

The thesis is organized as follows. Chapter 2 reviews the related work in formal methods, model checking and the integration of formal methods into software design education. Chapter 3 lays out the general strategy on how to realize the research objectives. Chapter 4 provides the conceptual design of the tool, and chapter 5 addresses finer design details related to components of the tool. Chapter 6 describes the implementation details of the tool and provides a case study to demonstrate its utility and effectiveness. Finally, in chapter 7 we conclude by discussing the benefits and limitations of the tool as well as future work to extend our research.

2. LITERATURE REVIEW

In meeting the challenge of software products' growing complexity, a major goal of software engineering is to enable the construction of reliable software systems [16]. The use of formal improves reliability by revealing inconsistencies, ambiguities, and incompleteness hidden in the system design [43]. As a result, a set of software engineering techniques and practices for software verification and testing based on formal methods, known as model-based verification, has been codified and adopted [16].

2.1 Formal Methods

Modeling and verification techniques employed by model-based verification involve the application of a formal methodology. A formal method in software development is defined as "a method that provides a formal language for describing a software artifact (e.g. specifications, designs, code) such that formal proofs are possible, in principle, about properties of the artifact [43]." In this formal methodology, essential models of a software system are created using a formalism, which is a collection of principles and practices that are built upon well-defined language of expression and inference and meaning assigned to the symbols of the language [9], and then analyzed and compared against its expected behaviors. Formal specification is the use of notations derived from formal logic to describe [26]

- the assumptions about the world in which a system will operate,

- the requirements that the system is to achieve, and
- the design to accomplish those requirements.

Essentially, a real system is represented, as a rule, in the form of labeled transition system (LTS) [26]. LTS is an oriented graph whose nodes are associated with the states of the system, and edges of this graph that connects the nodes, labeled by symbols of performed actions, are used for representation of the transition-action relation in the system. When the system starts, some state called initial is selected in the set of states of the LTS, and a sequence of transition-actions in the LTS is called its run or trace. The totality of all possible traces in the LTS is called the language of the system. An LTS is called finite if the sets of its states and transitions are finite, and infinite if otherwise. In formal specification, the basic types of properties that are usually specified include behavior properties over time, working characteristics, and internal structure. The behavioral properties are most important. Examples of such behavioral properties include safety and liveness properties, and they can be expressed in logic languages, such as temporal logic.

On the other hand, formal verification is the use of proof methods from formal logic to [26]

- analyze specifications for certain forms of consistency and completeness,
- prove that the design will satisfy the requirements, given the assumptions, and
- prove that a more detailed design implements a more abstract one

Two well established approaches to formal verification are model checking [8] and theorem proving [9]. Theorem proving, proposed by Burstall [5], Kröger [25] and Pneuli

[34], is a technique by which both the system and its desired properties are expressed in the form of mathematical logic. This logic is given by a formal system which defines a set of axioms and a set of inference rules. Theorem proving is the process of finding a proof of a property from the axioms of the system, and it is increasingly being used today in the mechanical verification of safety-critical properties of hardware and software designs [8]. The theorem proving tools consists of powerful collections of inference steps that can be used to reduce a proof goal to simpler sub-goals that can discharged automatically by the primitive proof steps of the prover. Given a property and a model, the prover is either able to verify the property by completing the proof or given back scenarios in which the property is violated. The advantage of theorem proving is that it can deal directly with infinite state spaces by relying on techniques such as structural induction to prove over infinite domains. Therefore, it is not limited by size of the state space. Large systems cannot be verified by a model checker for the same reason, but they can still by verified by the theorem prover. Unfortunately, theorem proving requires considerable amount of technical expertise. As a result, the process is often slow and error prone.

2.2 Model Checking

Model checking relies on building a finite model of a system and checking that a desired property is holding in the model [9]. It involves an exhaustive state space search which is guaranteed to terminate. During the search process, the model and the property are fed to a model checker and the model checker determines whether the system model satisfies the property. The result is either a claim that the property is true or a sequence of states from some initial state that violates the property, also known as a counterexample. Model checking can be applied to analyze specifications of software systems. Because checking whether a single model satisfies a formula is much easier than proving the validity of a formula for all models, model checking can be implemented fairly efficiently [8].

2.2.1 The Advantages and Disadvantages of Model Checking

Applying model checking to a design consists of several tasks: modeling [8], specification [8], and verification [39]. Modeling refers the conversion of a design into a formalism accepted by a model checking tool. In some cases, this is a straightforward compilation task. In other cases, owing to limitations on time and memory, modeling a particular design may require the use of abstraction to eliminate irrelevant or unimportant details. Before verification, it is necessary to state the properties that the design must satisfy. This specification is usually given in some logical formalism, such as temporal logic, which is able to assert the behavior of the system as it evolves over time. Although model checking provides means for checking a model of a design satisfies a given specification, it is impossible to determine whether the given specification covers all the properties that the system should satisfy. With modeling and specification in proper order, verification can take place. In theory, model checkers can perform verification automatically, given a model and a specification. However in practice, it often involves human assistance [8]. An example is the analysis of the verification results. In case of a negative result, the user is often provided with an error trace serving as a counterexample for the supplied property, which can be used to track down the exact location of the design fault. False negatives result from incorrect modeling of the system or incorrect specification Erroneous results or premature termination of verification can also emerge due to the size of model. In this case, it is necessary to decompose the model into fine-grain sub-models or change some parameters of the model checking tool.

Compared to theorem proving, model checking is relatively easy, systematic, and fast [8]. Model checking can be used to check partial specifications and provide valuable feedback about a system's correctness even if the system has not been completely specified. Model checker can produce counterexamples that reflect the errors in design, which can be invaluable for debugging. It is preferable to theorem proving, or deductive reasoning, whenever it can be applied. However, there will always be critical applications in which theorem proving is necessary for complete verification. There have been new research directions that attempt to integrate deductive verification and model checking to maximize benefits offered by both [36]. The main disadvantage of model checking is state explosion problem, as mentioned earlier. Many efforts have been invested to resolve the problem, such as McMillan's symbolic model checking [29]. Other approaches such as partial order reduction [33], localization reduction [22], and semantic minimization [13], are all designed to remove redundant states from a system model.

2.2.2 Symbolic Model Checking and Partial Order Reduction

In the original implementation of model checking algorithm, transition relations were represented explicitly by adjacency lists [8]. For a software system with small number of states and processes, the approach was quite practical. As the system model becomes more complex, the model checker simply could not handle the growing number of states. Since a model checker replies on an internal global state transition graph to keep track of the states and transitions during model checking, McMillan [29], in 1987, realized that by using a symbolic representation for the state transition graph, much larger systems could be verified. The new symbolic representation was based on Bryant's ordered binary decision diagrams (OBDDs) [3]. OBDDs provide a canonical form for Boolean formulas that is usually much more compact than conjunctive or disjunctive normal form, and very efficient algorithms have been developed in order to manipulate them. In this implicit representation, each state is encoded by an assignment of Boolean values to the set of state variables associated with the model. The transition relation can be expressed as a Boolean formula in terms of two sets of variables, one set encoding the old state and the other encoding the new. This formula is then represented by a binary decision diagram. The model checking algorithm is based on computing fix points of predicated transformers that are obtained from the transition relation. The fix points are sets of states that represent various temporal properties of the system. In the new implementations, both the predicate transformers and the fix points are represented with OBDDs. Thus, it is possible to avoid explicitly constructing the state graph of the system.

Besides symbolic model checking, partial order reduction is another popular technique designed to combat the state explosion phenomenon [33]. This technique exploits the independence of concurrently executed events. Two events are independent of each other when executing them in either order results in the same global outcome. A common model for representing concurrent software is the interleaving model, in which all of the events in a single execution are arranged in a linear order called an interleaving sequence. Concurrently executed events appear arbitrarily ordered with respect to one another. As a result, all possible interleaving of such events are normally considered and causing an extremely large state space. The partial order reduction technique makes it possible to decrease the number of interleaving sequences that must be considered. Thus, the number of states that are needed for model checking is reduced. Under the partial order reduction technique, when a specification cannot distinguish between two interleaving sequences that differ only by the order in which concurrently executed events are taken, it is sufficient to analyze only one of them.

2.2.3 Using Temporal Logic for Model Checking

Temporal logic is a formalism for describing sequences of transitions between states in a reactive system and has been proven to be useful for specifying concurrent systems, as they can describe the ordering of events in time without introducing time explicitly [8]. In temporal logic model checking, finite state machine models software or hardware system and a property specified as a formula in a certain temporal logic are given. The goal is to determine whether the system satisfies the formula. Since time is not considered explicitly, instead, a formula might specify that *eventually* some designated state is reached, or that an error state is *never* entered. Properties like *eventually* or *never* are specified using special temporal operators and these operators can be combined with Boolean connectives or nested arbitrarily.

CTL* is a powerful logic used for model checking as well as foundation for other logics [8]. Conceptually, CTL* formulas describe properties of computation trees. The tree is formed by designating a state in a Kripke structure [8] as the initial state and then unwinding the structure into an infinite tree with the designated state as the root. The computation tree shows all of the possible executions starting from the initial state. The logic formulas are composed of path quantifiers and temporal operators. The path quantifiers are used to describe the branching structure in the computation tree, and there are two of such quantifiers: A for all computation paths, and E for some computation path. These quantifiers are employed in a particular state to specify either paths starting from this state or some of the paths starting from this state contain certain properties. The temporal operators are used to describe properties of a path through the tree. There are five basic operators [8]:

- X – *next time*, which requires a property to hold in the second state of the path.
- F – *eventually* or *in the future*, which requires a property to hold at some state on the path.
- G – *always* or *globally*, which requires a property to hold at every state on the path.
- U – *until*, which requires the second property to hold at some state on the path, and the first property to hold at every proceeding state in the path.
- R – *release*, which requires the second property to hold along the path up and including the first state where the first property holds. However, the first property is not required to hold eventually.

There are two useful sub-logics based on CTL* [8]. One is branching-time logic called Computation Tree Logic (CTL) [7]. The other is linear-time logic called Linear Temporal Logic (LTL) [7]. CTL is a restricted subset of CTL* in which each of the temporal operators X, F, G, U, R must be immediately preceded by a path quantifier, resulting ten basic CTL operators:

- AX and EX
- AF and EF
- AG and EG
- AU and EU
- AR and ER

Examples of some typical CTL formulas include the following:

- AG safe: All reachable states are safe.
- AG AF stable: The system is stable infinitely often.
- AG (request $\rightarrow$ AF response): A request is always a response sometime in the future.
- AG EF restart: It is possible to restart the system in any reachable state

Formally, a finite state machine <Q, R, I> consists of a set of states Q, a state transition relation $R \subseteq Q \times Q$, and a set of initial state $I \subseteq Q$. A path is an infinite sequence of states such that each consecutive pair of states is in R. The set of states Q is often encoded by a set of state variables, such that each state corresponds to some value for the variables and no distinct states correspond to the same value. Basic on this foundation, a proposition is defined as any Boolean combination of predicates on the state variables. A formula is a proposition, a Boolean combination of formulas, or the combination of a temporal operator and a formula [8]. Each formula is evaluated at some state *q*. A proposition holds at *q* if *q* satisfies the proposition. The operator A means “for all paths starting at *q*”, E means “for some path starting at *q*”, G means “for every state along the path”, and F means “for some state along the path”. Therefore, AG safe holds at *q* is every state (G) along every path (A) starting at *q* satisfies the proposition safe. The system satisfies a formula if the formula holds at all initial states. If not, a model checker typically attempts to find a counterexample. For instance, if the formula AG safe is false, a counterexample is a finite path starting at some initial state and ending at a state that is not safe.

In explicit model-checking techniques, the truth value of a CTL formula is determined in a graph-theoretic manner by traversing the state diagram, with time complexity linear in the size of the state space and in the length of the formula [7]. Using symbolic model checking techniques, instead of visiting individual states as in conventional state space search, symbolic model checkers visit a set of states at a time [4, 28]. A state set can be represented by a predicate on the state variables such that a state is in the set if and only if the predicate is true at the state. When the state space is finite, we can assume that the state variables are Boolean and there are only finitely many of them. A predicate on these variables is simply a Boolean function, which can be represented by reduced ordered binary decision diagrams (OBDDs) [3]. An OBDD resembles a binary decision tree, except that isomorphic sub-trees must be combined resulting a directed acyclic graph. In addition, each path can contain a variable at most once, and must comply with a fixed linear order of the variables.

Linear Temporal Logic (LTL) is an extension of propositional logic to include discrete time information [8]. Formulas are interpreted as referring to events along an infinite path of time points. LTL formulas are built inductively from its set of atomic propositions. These atomic propositions and their operators are given below, and *p* and *q* are some states or events occurring in the path of time points:

• *And*	$p \wedge q$
• *Or*	$p \vee q$
• *Not*	$\neg p$
• *Next*	$\mathrm{X}\ p$
• *Always*	$\mathrm{G}\ p$
• *Eventually* / *Future*	$\mathrm{F}\ p$
• *Strong Until*	$p\ \mathrm{U}\ q$
• *Releases*	$p\ \mathrm{R}\ q$

One can model LTL by assigning to each natural number a set of true atomic propositions. The operators then define requirements on those propositions. The formula for the proposition “And” means that states *p* and *q* must both be true. “Or” means that either state *p* is true, or state *q* is true. “Not” means that state *p* is false. Atomic propositions “Next”, “Always”, “Eventually”, “Until” and “Release” have same meanings from CTL*, where they have been originated.

2.2.4 Existing Model Checking Tools

There are tools available that facilitate the checking of expected model based system behavior and properties of concurrent programs under different fairness assumptions. SMV [29], the Symbolic Model Verifier, is a popular model-checking system first developed by McMillan in 1993. It uses the OBDD-based symbolic model checking algorithm to perform verification and takes a finite state machine as the model of the system, expressed in its own input language, and properties of the system, expressed in CTL formula. The system model is often decomposed into a series of modules and each can be instantiated multiple times. A SMV module can be composed either synchronously, which means all modules perform an action concurrently at a time period, or using interleaving, which means exactly one module performs an action at a time period. The state transitions in the model can be either deterministic or nondeterministic. The state transitions in the model can be specified explicitly in terms of Boolean relations or implicitly as a set of parallel assignment statements. When performing model checking, a breadth-first searching procedure with fixed-point algorithms is used to check the satisfaction of the finite state machines against the expected properties.

An alternative model checking system is called SPIN [18], which uses explicit state enumeration and partial order reduction during model checking. It was developed at Bell Laboratories by Gerard Holzmann and Doron Peled, and primarily used for verifying asynchronous software systems such as communication protocols. It can check a system model for deadlocks or unreachable code or determines if it satisfies a particular property composed by LTL specification. The input language to describe the system model, called PROMELA [18], an acronym for Process Meta-Language, was developed by Gerard Holzmann. It uses syntactic constructs similar to several other programming languages, such as C. The basic building blocks of SPIN models are asynchronous processes, buffered and un-buffered message channels, synchronizing statements, and structured data. Unlike SMV, SPIN uses partial order reduction to limit the state space explosion problem to optimize the process of model checking.

2.3 The Role of Formal Methods in Software Design Education

Formal methods involve the use of discrete mathematics and mathematical logic in the study and practice of computer science and software engineering [1]. From its beginning, computing was regarded as an abstract, mathematical science. Pioneers like Turing, Church, and von Neumann used mathematics to establish the essence and boundaries of the computing discipline. Although computing technology is crucial in software engineering education and practice, the underpinnings are mathematical in nature and computing does deal with purely logical processes [30]. Students often resist the use of mathematics in the study of computing, usually for the following reasons:

- Students may lack the proper preparation or motivation.
- Many have neither an understanding of nor appreciation for the role of mathematics, or more explicitly, formal methods, in computing [31].
- Some feel intimidated or even fearful of the level of mathematical knowledge and capability required.

As the term software engineer becomes a popular title for software developers, there is little evidence to show that the practice of software design and engineering compares with the rigor and discipline that is required for practice in other engineering fields [32]. So the question seems to be whether software engineering programs should follow the traditional engineering approach to professional education. Quality problems arise from incomplete and imprecise requirements, specification, shoddy designs with poor documentation, and almost sole reliance on testing for software quality assurance, and there is increasing interest in the use of formal methods for specification and design [1].

With the explosive growth of software, the Internet, and electronic commerce, formal methods become a practical approach for achieving higher confidence in today and tomorrow's infrastructure system [1].

2.3.1 Formal Methods as Part of an Engineering Curriculum

Formal methods improve software reliability by providing mathematical frameworks to define, specify, and verify complex software systems. However, the majority of software engineering curricula have a low level of emphasis on formal methods [1]. This is partly due to a lack of interest on the part of the software industry, but much of the responsibility must be attributed to the state of the curriculum and course design. The computing education community has adopted a curriculum strategy of dividing curricula elements into areas of theory and practice. This causes both faculty and students to view the theory of computing as separate and distinct from the practice of computing. As a result, there are theorists who are viewed as the mathematical elite and practitioners with little respect for the applicability of formal methods to their work. This mindset inhibits the use and integration of formal methods into software development process, and ultimately, into software design and engineering education. Because of this, there is little guidance and support available to faculty, who would like to introduce formal methods into their software engineering courses.

The scope and scale of software projects today are increasing dramatically, along with shorter release cycles, and traditional software quality assurance methodologies are facing more challenges in attempting to meet quality standards [16]. Equipped with formal methods to address the complexity, model checking provides software engineers fresh insights on how to debug, verify, and validate designs. With the success of model checking and other formal approaches for software verification are attracting attentions, trying to integrate them into software design education has lead to following observations [28]:

- The theoretic foundations of model checking involve mathematical logic.
- The engineering principles and processes used for implementing model checking provide excellent training for students to solve complicated design and analysis problems.
- The skills and knowledge that students acquire from the course provide them with alternative approaches to solve problems in many important software engineering areas.

Strategies of integrating formal methods in the form of model-based verification into software design education have been difficult. Generally there are three strategies [15]. The first approach avoids teaching formal methods altogether and considers formal methods are impractical and mature enough to be beneficial in software engineering practices. The appropriateness of this argument for the general software engineering education is debatable. The second approach is to devote a specific course which emphasizes formal verification of source code using a number of formal methods, such as VDM [21] or Z [40]. The students are expected to learn about the methods, and then they are expected to apply the formal skills to software development activities. This approach involves broad coverage of a variety of formal methods that provide students with a larger scope of exposure, but may not enable them to be proficient in any specific approach. Furthermore, the methods tend to be taught in an isolated manner with emphasis in notations rather than its underlying principles. This isolated exposure generally prohibits students to apply such approaches in software engineering practices. Finally, the third approach is invented to redesign the curriculum so that formal methods are integrated throughout the entire curriculum [15]. Carnegie Mellon University [15], as an example, redesigned its software engineering graduate program to promote better understanding to formal models of software systems. This approach offers many benefits to the students as they incorporate finite state modeling and temporal logic for model checking interactive aspects of system. More specifically, the curriculum integrated with formal methods enables the analysis of software development products such as delivered code, specifications, designs, documentation, prototypes, and test suites. It also treats both static and dynamic analyses, such as type checking, verification, testing, performance analysis, hazard analysis, reverse engineering, and program slicing.

Although a novel strategy, the approach adopted at Carnegie Mellon is difficult to apply at the undergraduate level, as it assumes that students in the curriculum have already had exposure to advanced logic, discrete and combinatorial mathematics that facilitate the attainment of the requisite skills to understand and apply formal methods. Although many graduate students who have strong background in mathematics indeed possess such skills; many undergraduate students with comparatively limited background in advanced mathematics, a course delivery strategy that revolves around formal methods can be overwhelming for two reasons:

2. The impedance mismatch between the underlying mathematical underpinning of formal methods and students' semi-formal, if not informal, view of the design problem and
3. The lack of tool support in the seamless integration of formal methods into software design education

Considering the difficulties of teaching formal methods to undergraduate students, software tools should encourage learning by abstracting required material into relatively simple paradigms that novice users can easily learn and manage [9]. However, most tools, used to support the learning and the teaching of formal methods, are developed for practitioners, rather than for educators or learners. Some desired properties of tools that are attractive to this group of users include [9]:

- Ease of Use: Tools should be easy to use and their output should be easily interpreted by novice users.
- Ease of Learning: Tools should provide a starting point for writing formal specifications for users who would not otherwise write them. The knowledge requirement of formal methods on the users should be kept minimal.
- Focused Analysis: Tools should be good at analyzing at least one aspect of the system well.
- Early Payback: Tools should provide significant benefits almost as soon as the users start to use them.

- Incremental Gain for Incremental Effort: Tools should provide users increased benefits as the users are getting more adept or are putting more effort into writing specifications.
- Efficiency: Tools should make efficient use of users' time, and the amount of time used by the tool should be proportional to the extensiveness of the analysis.
- Integrated Use: Tools should work in conjunction with other common programming languages and techniques and should be integrated with traditional software development tools. Users should not have to look into another new methodology in order to receive benefits.
- Evolutionary Development: Tools should allow partial specification and analysis of selected aspects of a system.
- Orientation toward Error Detection: Tools should be optimized to find errors rather than confirming correctness.

Going one step further, rather than building a single tool, "meta-tools" can be built to automatically produce tools that are customized toward a particular problem domain [38], formal notation [10], or logic [17, 24]. It is also important for a tool to make the user aware its strengths, limitations, modeling assumptions, ease of integration with other tools, and start-up costs.

2.3.2 Common Formal Methods used in Software Engineering Education

The education of formal methods in a classroom environment often revolves around a particular technique. As a result, specific formal techniques have become foundations for certain curricula aimed to provide exposure of formal methods to students, due to their popularity in the industry. The most popular formal techniques are VDM [21] and the Z notation [40].

VDM is a model-oriented formal method based on a denotational semantic setting, intended to support stepwise refinement of abstract models into concrete implementations [21]. The method includes a formal specification language, VDM-SL [21], which supports various forms of abstraction. Representational abstraction is supported by data modeling facilities. These facilities are based on six mathematical data-structuring mechanisms: sets, sequences, maps, composite objects, Cartesian products and unions. At a lower level, the language provides various numeric types, Booleans, tokens and enumeration types. By using the data-structuring mechanism and the basic data types, compound data types can be formed with a specific mathematical structure, and these compound data types are denoted as domains. Sub-typing is supported by attaching domain invariants to domain definitions. Operational abstraction is supported by both functional abstraction and relational abstraction: the former by means of function specification and the latter by operation specification. Both functions and operations may be specified implicitly using pre and post conditions, or explicitly using applicative constructs to specify functions and imperative constructs to specify operations. Operations have direct access to a collection of global objects: the state of the specification. The state is constructed as a composite object and built from labeled components. A VDM specification typically consists of a state description augmented with invariant and initialization predicates, a collection of domain definitions augmented with invariants, a collection of constant definitions, a collection of operations and a collection of functions. An initial specification is usually kept as abstract as possible. Then the initial specification can be further developed and refined using two techniques: data reification, which addresses the refinements of state elements, and operation modeling, which addresses the refinements of the functions and operations. Data reification involves the transition from abstract to concrete data types, and a justification of this transition. Choosing a more concrete data model implies a redefinition of all operations and functions on the original model in terms of the new model, a process called operation modeling. Central to data reification is the notion of adequacy, expressed through two functions on the abstract and concrete domains, the abstraction-function and the retrieve-function. The abstraction-function maps abstract values onto concrete values; the retrieve-function does the opposite, mapping concrete values onto abstract values. The final step within the development is the transition of a low-level specification into the chosen programming language.

The Z notation is based upon set theory and mathematical logic [44]. The set theory used includes standard set operators, set comprehensions, Cartesian products, and power sets. The mathematical logic is a first-order predicate calculus. Together they make up the mathematical objects in Z. These objects and their properties can be collected in schemas, which are patterns of declaration and constraint. The schema language can be used to describe the state of a systems, and conditions in which that state may change. It can also be used to describe system properties, and to reason about possible refinements of a design. A characteristic feature of Z is the use of types. Every object in the mathematical language has a unique type, represented as a maximal set in the current specification. This notion of types suggests that an algorithm can be written to check the type of every object in the specification. Another important feature of Z is the use of natural language. In Z, mathematics is used to state the problem, to discover solutions, and to prove the chosen design meets the specification. Natural language is used to relate the mathematics to objects in the real world. This task is often partly achieved by the judicious naming of variables and additional comments in the specification. Z also supports the concept of refinement. A system can be developed by constructing a model of a design, using simple mathematical data types to identify the desired behavior. This description can be further refined later by constructing another model which respects the design decisions made, and yet is closer to implementation. This process of refinement can be continued until executable code is produced.

In summary, both VDM the Z notation are mathematical languages with powerful structuring mechanisms capable of producing formal specifications, and both require immense requisite skills in mathematical logic to be understood and used effectively. As a result, their user group is largely limited to seasoned professionals who have grasped the underlying principle over years of experience [43].

3. A PRACTICAL STRATEGY FOR INTEGRATING MODEL CHECKING INTO SOFTWARE DESIGN EDUCATION

The goal of the research is to integrate formal methods into current software design methodology being taught in undergraduate software design courses. This objective is realized by building a tool called the Behavioral Model Analyzer (referred as the BMA in the rest of the thesis). The BMA aims to address the difficulties of teaching formal methods to undergraduate students, as well as the shortcomings of existing model checking tools for educational purposes. The operation of the BMA can broadly be described in the following manner. It accepts as input a software design model (i.e., statechart) and a property specifying how the model is required to behave. The BMA then performs model checking using the model and the property to display the results back to the modeler. Although this description hardly differs from the operation of any other model checking tool, the BMA places emphasis on the following features that are not present in other model checkers:

- The input software design models are UML models, which are commonly used in software design, rather than tool-specific model description languages which modelers have little exposure.
- The properties defining the required behaviors of the models can either be supplied or derived in the form of abstract and user-friendly specification

- templates rather than temporal logic, thus eliminating the requisite mathematical skills involved in formal methods.
- The results of model checking are shown using graph visualization rather than cryptic text, so that students can have better grasps on where problems are detected and how to fix them.

3.1 Using UML Models as Input Design Models

Unified Modeling Language (UML) [2, 19, 38], is a general-purpose visual modeling language that is used to specify, visualize, construct, and document the artifacts of a software system and is capable of capturing information about the static structure and dynamic behavior of the system. The static structure defines the collection of discrete objects that make up the system and the dynamic behavior defines the history of objects over time and the communication among objects to accomplish goals. Software tools can provide code generators from UML into a variety of programming languages, as well as reverse engineered models from existing programs. UML is not a highly formal language designed for the theorem proving, but rather a modeling language for discrete systems. These reasons, coupled with the increasing popularity of object-oriented methodology, made UML ubiquitous in both the industry and the academia [2, 38].

Although UML offers a variety of diagrams to model different perspectives of a software system, we are interested of using the state machine view, modeled by a statechart diagram, to represent an input design model, since the state machine view describes the dynamic behavior of objects over time by modeling the lifecycles of objects of each class [2, 19, 38]. A state machine is a graph of states and transitions and it is attached to a class and describes the response of an instance of the class to events that it receives. Events represent the kinds of changes that an object can detect – the receipt of calls or explicit signals from one object to another, a change in certain values, or the passage of time. Anything that can affect an object can be characterized as an event. A state is a set of object values for a given class that have the same qualitative response to events that occur; therefore it describes a period of time during the life of an object of a class. In the state machine, a set of states is connected by transitions. A transition leaving a state and entering into another state defines the response of an object to the occurrence of an event.

To facilitate model checking of UML statecharts, the BMA translates input models into an intermediate format defined in terms of PROMELA language. This is similar in principle to the translation of UML models into code skeletons within the Model-Driven Architecture initiative [30]. PROMELA is an acronym for Process Meta-Language [18], which is a model description language for the model checker SPIN [18]. The BMA uses SPIN to perform model checking by translating abstract and user-friendly inputs from the modeler into tool-specific inputs for SPIN, and visualizing text-based model checking output in the form of graphs. As a result, UML statecharts need to be translated into PROMELA before the model checking process.

3.2 Substituting Temporal Logic with Specification Templates

In model checking, a property (i.e., required behavior) of the model is specified using temporal logic. The steep learning curve for the mathematical skills required to use temporal logic is one of the core reasons why formal methods have been almost absent in undergraduate design education. It is imperative for the BMA to circumvent this obstacle. As a result, the BMA uses the notion of specification templates [12] to describe the required property of the model. A specification template is a generalized description of a commonly occurring requirement on the permissible state sequences in a finite-state model of a system, and it describes the essential structure of some aspect of the system's intended behavior. The specification templates are generalized in a hierarchical structure in terms of their scopes for formal specification and verification. The scope of a template is the extent of program execution over which the template must hold. It is determined by specifying a starting and an ending state for the template. Therefore the scope consists of all states beginning with the starting state and up to but not including the ending state. There are five different scopes (Figure 1) [12]:

- *Global* – the entire program execution
- *Before* – the execution up to a given state
- *After* – the execution after a given state
- *Between* – any part of the execution from one given state to another
- *After-Until* – just like *Between* but the designated part of the execution continues even if the second state does not occur

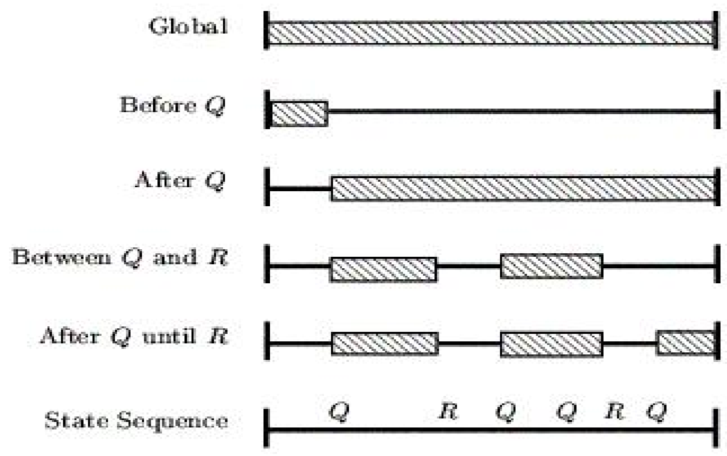


Figure 1 Scopes of specification templates

Each template can be translated into its corresponding temporal logic formula by the BMA. When working with a specification template, only states required by the particular template need to be supplied by the modeler in terms of simple mathematical logic. Some common specification templates are listed as follows (Figure 2) [12]:

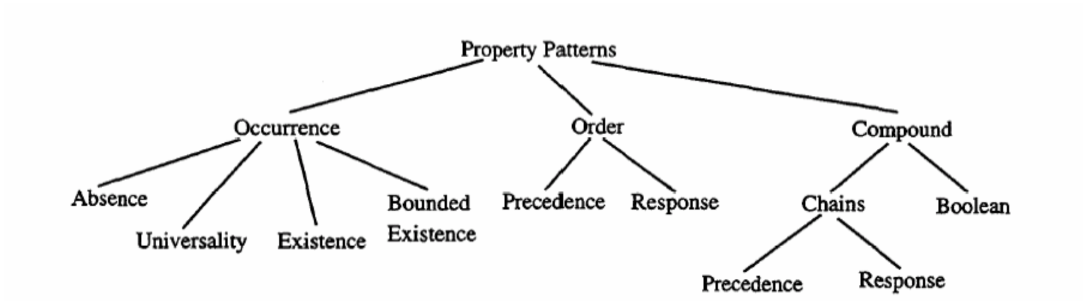


Figure 2 Specification templates in hierarchical order

Occurrence Templates include

- *Absence* – A given state or event does not occur within a scope. This template is also known as *Never*.
- *Existence* – A given state or event must occur within a scope. This template is also known as *Future* or *Eventuality*.
- *Bounded Existence* – A given state or event must occur k times within a scope.

- *Universality* – A given state or event occurs throughout a scope. This template is also known as *Globally, Always* and *Henceforth*.

Ordering Templates include

- *Precedence* – A state or event *P* must always be preceded by a state or event *Q* within a scope.
- *Response* – A state or event *P* must always be followed by a state or event *Q* within a scope. This template is also known as *Follows* and *Leads-to*.

Compound Templates include

- *Chain Precedence* – A sequence of states or events $P_1, \ldots, P_n$ must always be preceded by a sequence of states or events $Q_1, \ldots, Q_m$.
- *Chain Response* – A sequence of states or events $P_1, \ldots, P_n$ must always be followed by a sequence of states or events $Q_1, \ldots, Q_m$.
- *Boolean Combinations* – Sometimes we want to generalize the templates to allow for sets of states to describe scopes and properties. Some times this is straightforward and sometimes disjunctions and conjunctions of state or event descriptions can yield incorrect specifications when substituted into templates. These templates outline how Boolean combinations can be applied in different case.

Each of the hierarchical specification templates has its corresponding temporal logic specified, and the BMA translates these templates into temporal logic. The model checker SPIN can perform model checking using the translated temporal logic as input. Table 1 provides the temporal logic specifications for the *Existence* Templates.

Specification Scope	Temporal Logic Specification
Globally	`<>(P)`
Before R	`!R W (P & !R)`
After Q	`[](!Q) | <>(Q & <>P))`
Between Q and R	`[](Q & !R -> (!R W (P & !R)))`
After Q until R	`[](Q & !R -> (!R U (P & !R)))`

Table 1 Temporal logic specifications for the *existence* templates

In the table, *P*, *Q*, and *R* are events and the column *specification scope* indicates the scope where event *P* is true. The temporal operators in the temporal logic specifications have the following semantics:

• *Eventually*	<>
• *Always*	[ ]
• *Negation*	!
• *Or*	|
• *And*	&
• *Implies*	→
• *Until*	U
• *Strong Until*	W

Using the temporal logic for the specification templates as a foundation, the BMA can automatically derive the specification templates from UML sequence diagrams [2, 38], if they are included in the input design model, thus avoiding the necessity of having the modeler to input the required behavioral property of the design model. A sequence diagram specifies a set of messages arranged in time sequence to depict a scenario or the behavioral sequence of a use case. Each message on a sequence diagram corresponds to an operation on a class or an event trigger on a transition in a state machine. Since each specification template involves with the occurrence of one or more states or events present based on different scopes, it is possible to derive these specification templates by detecting the order of occurrence of the events in the sequence diagram.

3.3 Three Incremental Steps to Realize the BMA

The features of the BMA are implemented using three incremental steps. The first step includes the translation of UML statechart models for the purpose of model checking, the construction of the specification template input interface so that a modeler can supply the required behavioral property for the design model without the intricate details involving formal methods, and a graph visualization shows the counterexample, as a sequence of states, which violates the required property. The second step incorporates the capability to automatically recognize and derive the behavioral properties, in the form of specification templates, from the UML sequence diagrams. The third and final step adds the capability to visualize the specification finite state machine generated from the required behavioral properties before model checking. This visualization offers two benefits to the modelers. First, it provides detailed awareness and representation of the specification templates to the modeler. When deriving these templates from the UML sequence diagrams, it is unlikely the modeler knows the derived templates ahead of time. Second, since the BMA emphasizes learning formal methods through interaction, the detailed representation of specification templates in the form of graphs provides the modeler better intuitive grasp of the required behavior.

4. ARCHITECTURAL DESIGN OF THE BEHAVIORAL MODEL ANALYZER

The development strategy described in chapter 3 has outlined the required functions of the BMA. The architectural design of the BMA follows this strategy to form the functional components of the application. This goal of this chapter is to provide the design details at the component level in three phases to reflect distinct functions performed in each of one of the phases. The architectural design for the BMA can be divided into three subsystems as shown in Figure 3: the *Semi-Automated BMA*, the *Automated BMA*, and the *Advanced Visualizer*. The *DesktopBMA* is the front-end UI component of the entire application.

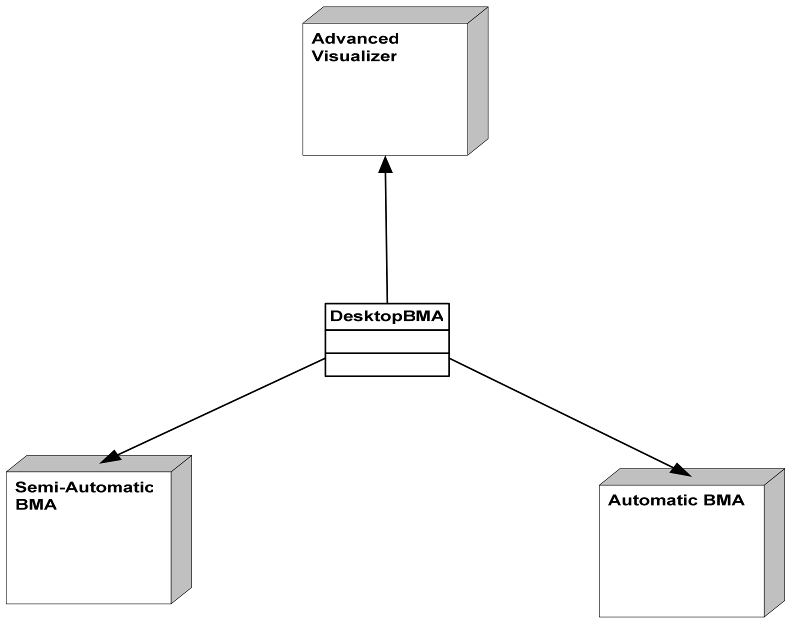


Figure 3 Three subsystems of the BMA

The *Semi-Automated BMA* fulfills Phase I requirements of the BMA. In this subsystem, model information from the UML statecharts, exported in XMI, are extracted and translated into PROMELA design models described by the PROMELA model description language. Specification templates are obtained from the modeler and translated into temporal logic. Model checking is then performed using the PROMELA model and behavioral constraints defined in terms of temporal logic. The results of model checking are visualized in an interactive and informative manner.

The *Automated BMA* fulfills Phase II requirements of the BMA. In this subsystem, messages in the UML sequence diagrams are examined and specification templates are derived based on the order of occurrence of these messages without intervention from the modeler.

The Advanced Visualizer fulfills Phase III requirements of the BMA. In this subsystem, the reachability graph of the states in the design model is generated. If errors are detected during model checking, an error trace is also shown to indicate the execution path of the error. The specification finite state machine is also generated from the specification templates either supplied by the modeler or derived from the UML sequence diagrams.

4.1 Architectural Design of the Semi-Automated BMA

The architectural design of the *Semi-Automated BMA* consists of eight functional components as shown in Figure 4: *DesktopBMA*, *XMIParser*, *TemplateInput*, *LTLEncoder*, *PromelaParser*, *SpinEvoker*, *OutputParser*, and *ModelVisualizer*. These components provide the following functions to perform model checking and display the results:

1. Convert UML statecharts to PROMELA model description language
2. Obtain specification templates and translate them into temporal logic
3. Perform model checking
4. Graphically display the errors detected during model checking

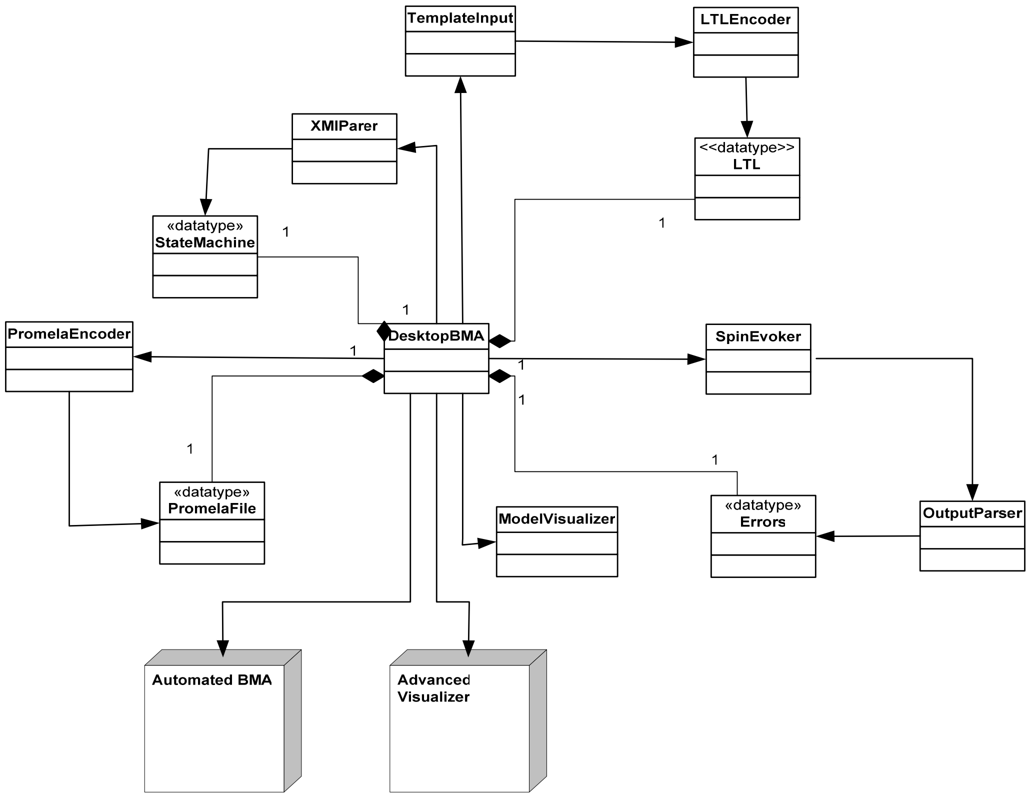


Figure 4 Static structure of the Semi-Automated BMA

The BMA is enacted when the *DestopBMA* initializes. This component contains functions to call other components to execute the work request by using interactive user interface widgets. Using these widgets, a UML design model is converted into PROMELA language; specification templates regarding the design model are supplied; and the model checking process is launched to find potential errors. When a UML input model has been chosen, the detailed structure information of the input model is extracted by the *XMIParser* and stored in a data structure called *StateMachine*. Afterwards, the graph visualization of the input model is generated by the *ModelVisualizer*. The model information stored in the *StateMachine* data structure is converted into PROMELA language and stored in a text file by the *PromelaEncoder*. Meanwhile, specialized user interface elements provided by the *TemplateInput* enable constraint templates to be supplied, and these templates are converted into temporal logic by the *LTLEncoder* and saved into a temporal text file. Once the design model and the specification templates are converted into PROMELA and temporal logic, respectively, the model checker is evoked by the *SpinEvoker* to perform model checking. Error information is extracted from the text-based model checking results by the *OutputParser* and saved into a data structure called *Errors*. Using this data structure, a colored trace linking the problematic states in the existing visualization is generated by the *ModelVisualizer*. The interactions among the components are specified in terms of the UML sequence diagram shown in Figure 5.

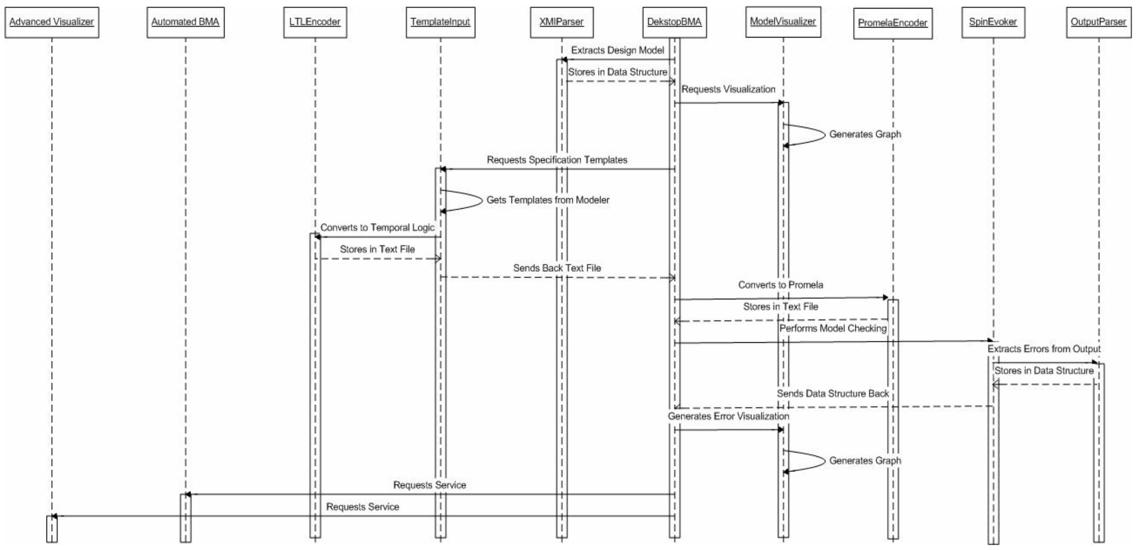


Figure 5 Sequence diagram of the Semi-Automated BMA

The deployment diagram shown in Figure 6 provides the physical view of the components grouped by their functions. The components in this system can be grouped into five subsystems. The subsystem *UI* contains the *BMADesktop*. Besides being the user interface, it facilitates communication among other components. The *Property Specification* handles the process of converting supplied specification templates to temporal logic and contains the *TemplateInput*, *LTLEncoder*, and the text file that includes the converted temporal logic from the specification templates. The *Model Checking* component performs model checking and retrieves the raw results from the model checker. It contains the *SpinEvoker*, *OutputParser*, and the data structure containing the errors found during model checking. Finally, the *Visualization* provides the results of model checking using graphs.

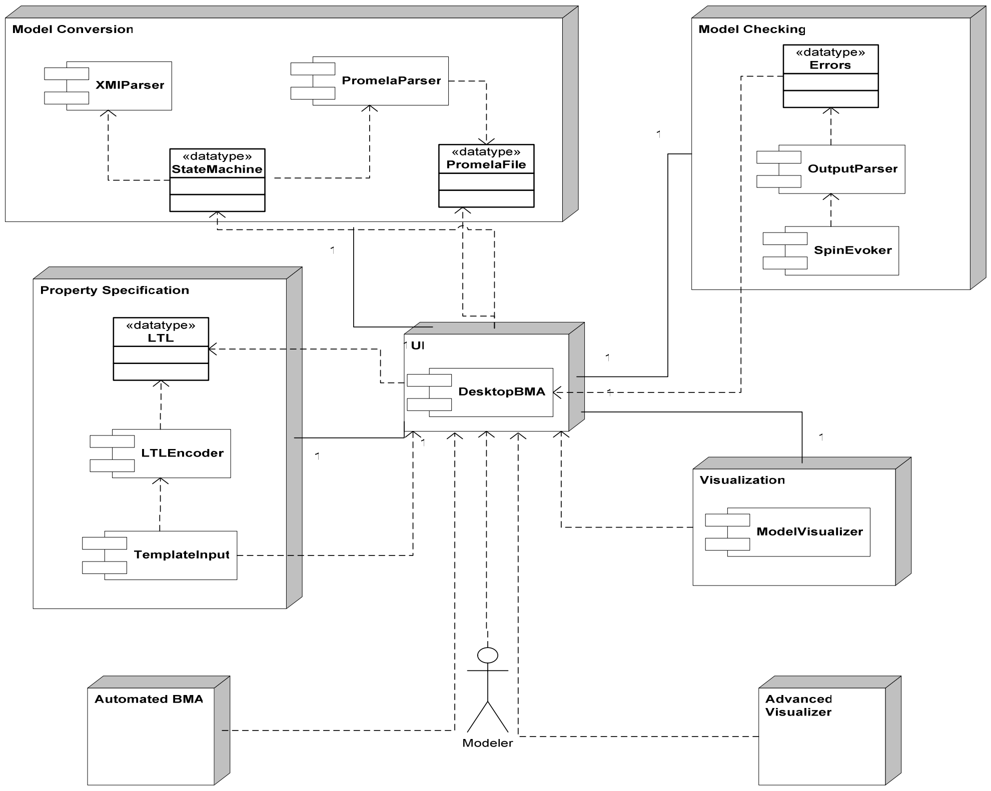


Figure 6 UML Deployment diagram of the Semi-Automated BMA

4.2 Architectural Design of the Automated BMA

The architectural design of the *Automated BMA* consists of four major components as shown in Figure 7: *DesktopBMA*, *TemplateHandler*, *XMISeqParser*, and *TempSelector*. These components facilitate the derivation of specification templates from the UML sequence diagram. When using the BMA, the specification templates can either be directly supplied or automatically derived from the UML sequence diagram, but not both at the same time.

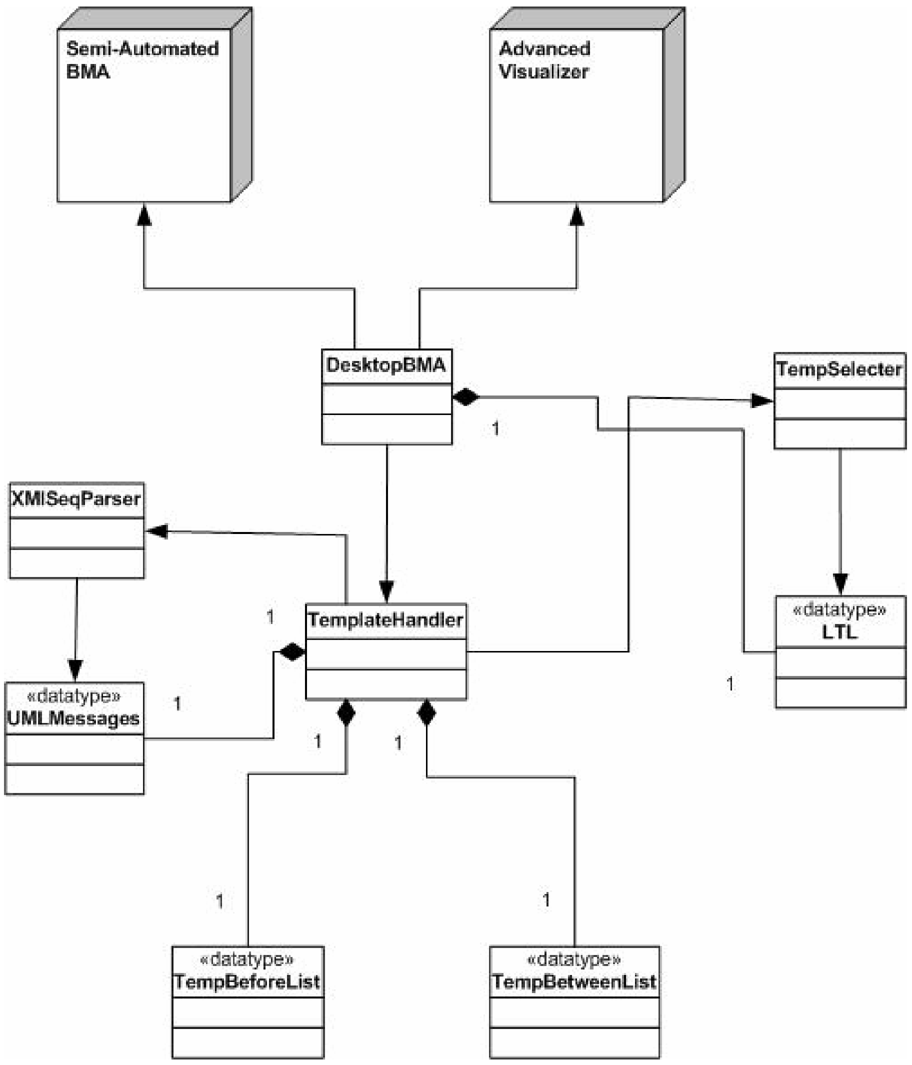


Figure 7 Static structure of the Automated BMA

The process of deriving specification templates starts when the messages in the UML sequence diagrams are extracted by the *XMISeqParser* and stored in a data structure named *UMLMessages*. Then the list of messages in the data structure is examined and relevant specification templates are derived by the *TemplateHandler*. Since the BMA is a prototype application, only the recognition of the *before* and the *between* occurrence templates defined by Dwyer et al [12] is implemented to demonstrate the effectiveness of our strategy. The recognized templates are saved into the data structures *TempBeforeList* and *TempBetweenList*, respectively. The list of recognized templates are shown to the modeler by the *TempSelector*, and the modeler needs to check off the particular ones to use as properties for the design model to perform model checking. Afterwards, the selected templates are converted into temporal logic and saved in a text file by the *TempSelector*. The interactions among the components are described in terms of the UML sequence diagram shown in Figure 8.

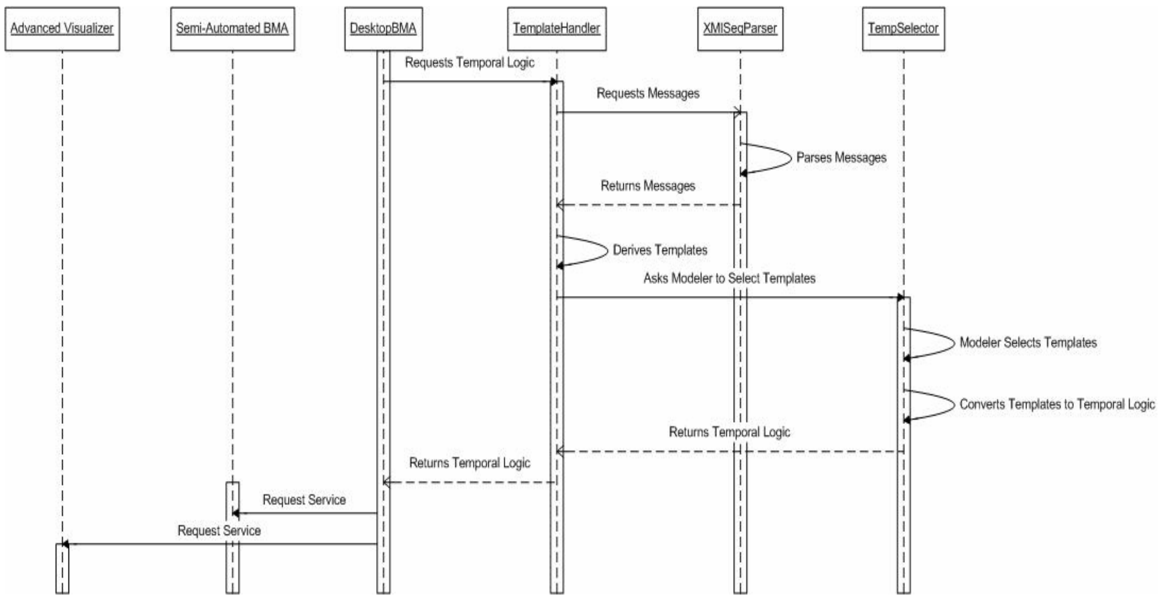


Figure 8 UML Sequence diagram of the Automated BMA

The deployment diagram shown in Figure 9 provides the physical view of the components in this system based on their functions. It contains two subsystems. The *UI* subsystem contains the *DesktopBMA*, which is the user interface. The rest of the components are all grouped under the *Template Generation*. This subsystem performs the extraction of the messages from the UML sequence diagrams, the derivation of specification templates based on their order of occurrence, and the conversion of the derived templates into temporal logic.

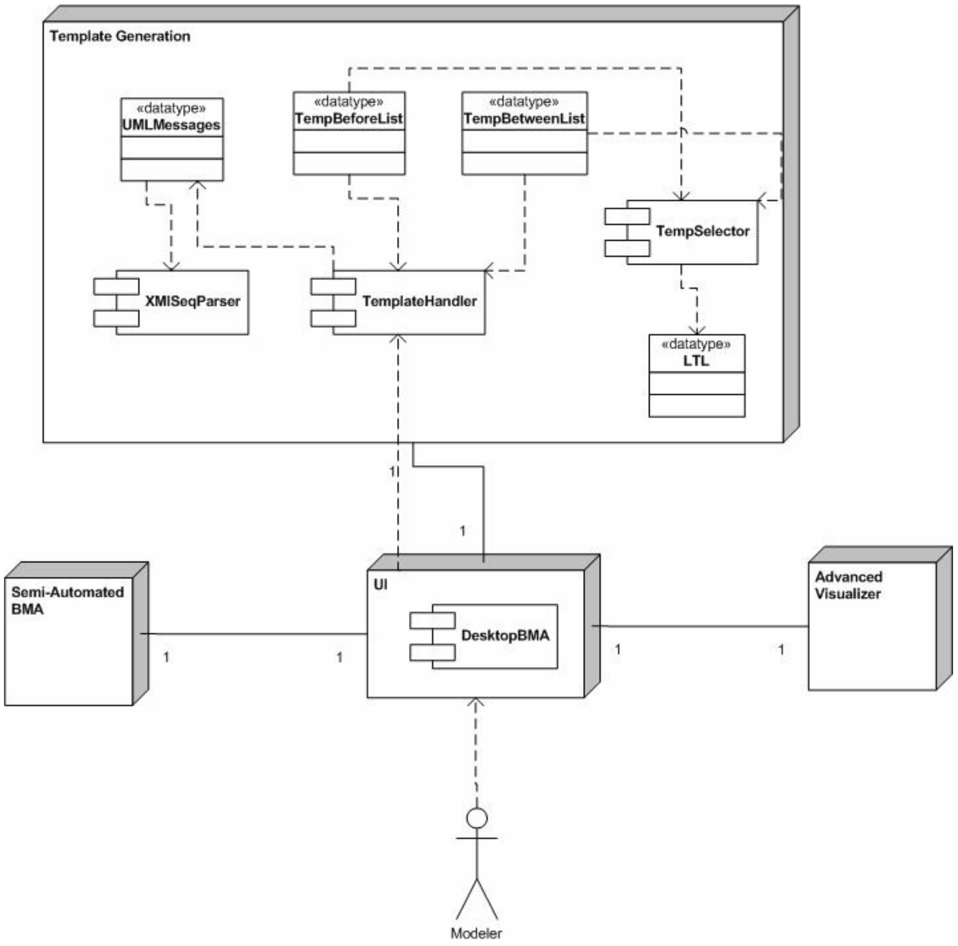


Figure 9 UML Deployment diagram of the Automated BMA

4.3 The Architectural Design of the Advanced Visualizer

The architectural design of *Advanced Visualizer* includes four components as shown in Figure 10: *DesktopBMA*, *ReachVisualizer*, *SpecParser*, and *SpecVisualizer*. These components enable two additional functions:

1. Generate a reachability graph and highlight the trace of errors detected during model checking.
2. Derive the specification finite state machine from the specification templates.

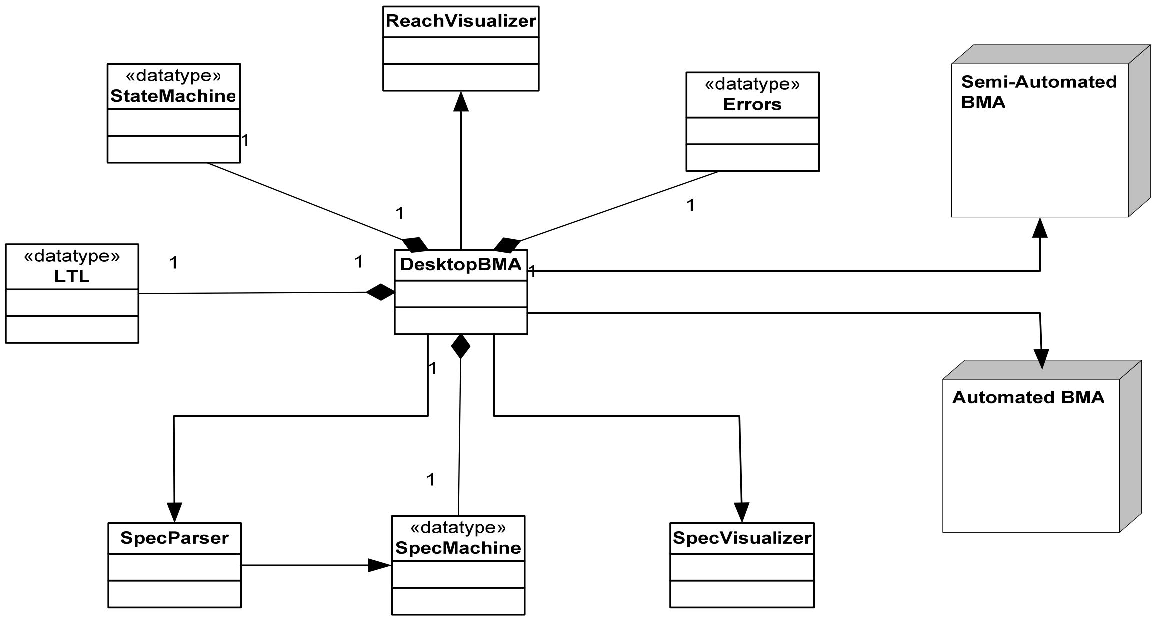


Figure 10 Static structure of the Advanced Visualizer

In this system, the main purpose of the *DesktopBMA* is to coordinate the communications among other components. The *ReachVisualizer* component is responsible for displaying the reachability graph derived from the finite state design model. To produce this graph, this component needs to access to two data structures: *LTL* and *StateMachine*, both are managed by *DesktopBMA*. *LTL* is the text file containing the property of the design model expressed in temporal logic, which is converted from the specification templates.

StateMachine is the data structure containing the finite state design model, which is extracted from the UML statecharts. To generate the specification finite state machine, structure information from *LTL* is extracted by the *SpecParser* and stored in another data structure called *SpecMachine*. This data structure is used by the *SpecVisualizer* to construct the specification finite state machine. The interactions among the components in this system are described by the UML sequence diagram shown in Figure 11. These two functions represent two separate processes that are completely independent of each other. When using the BMA, the specification finite state machine is generated earlier than the reachability graph because the temporal logic required to derive the specification finite state machine is available before model checking, and the errors required to produce the reachability graph becomes available after model checking.

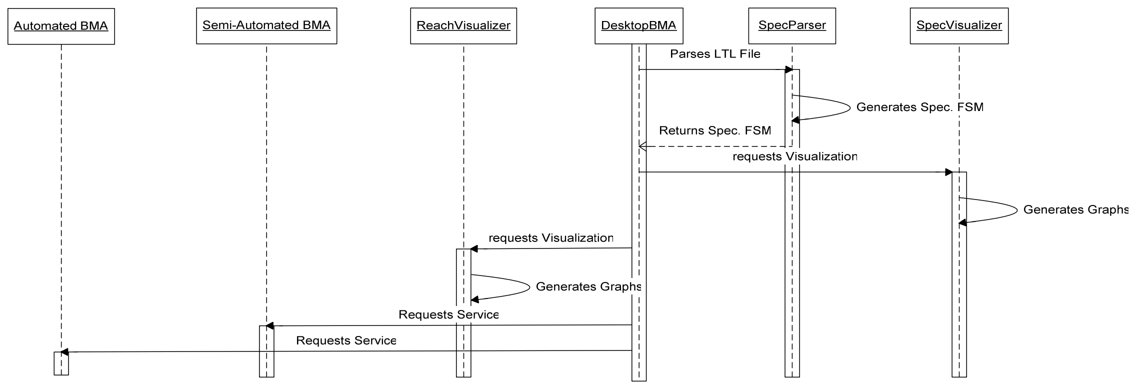


Figure 11 UML Sequence diagram for the Advanced Visualizer

The deployment diagram shown in Figure 12 provides the physical view of the components in the *Advanced Visualizer* grouped by their functions. There are three subsystems in this phase. The *UI* subsystem contains the component *DesktopBMA*, which is the user interface and performs the coordination of communications among other components. The *Specification FSM Generation* derives the specification finite state machine and contains the *SpecVisualizer,* as well as the *SpecParser*. It also contains the data structure enclosing the specification finite state machine and the text file containing the temporal logic needed to generate the specification finite state machine. The *Reachability Graph Generation* generates the reachability graph and contains the *ReachVisualizer* and the data structures that are required to generate the reachability graph.

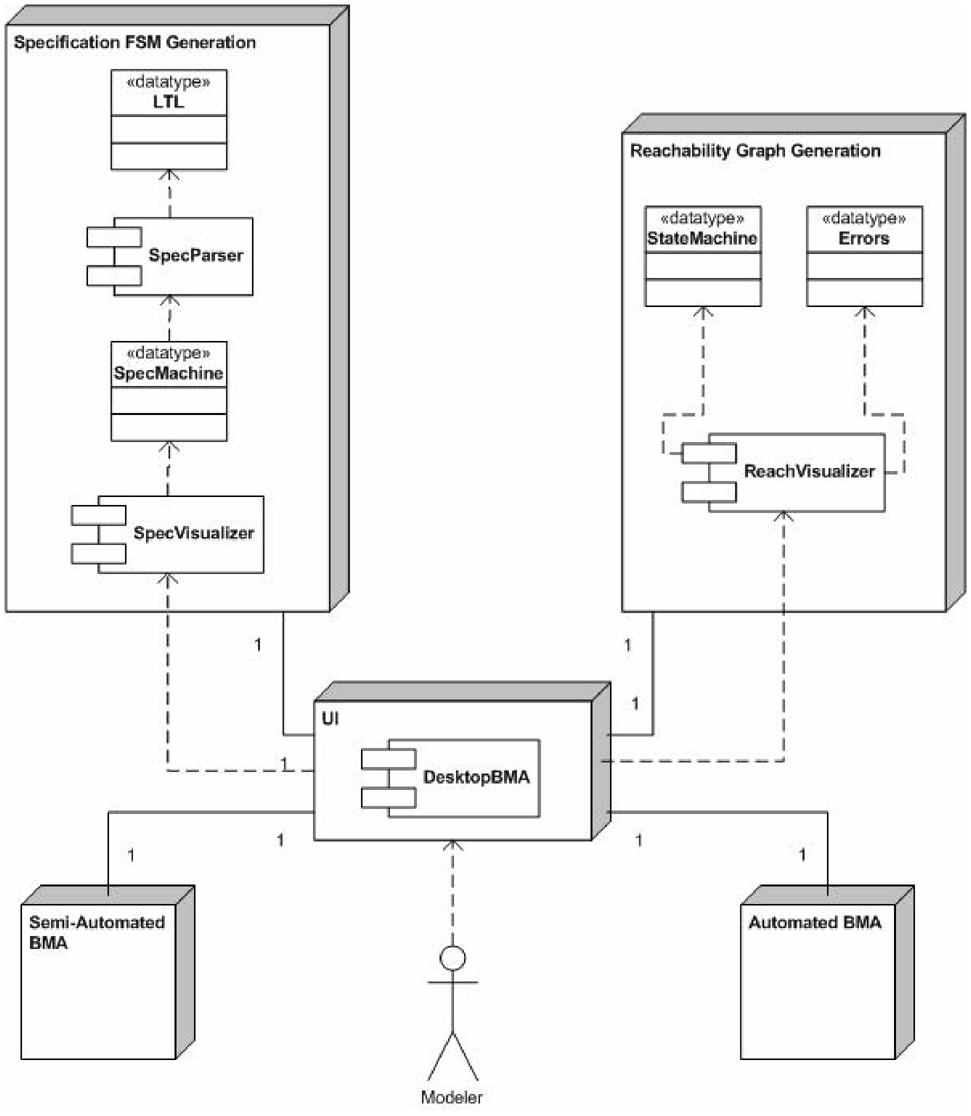


Figure 12 UML Deployment diagram for the Advanced Visualizer

5. THE VERIFICATION PROCESS USING THE BEHAVIORAL MODEL ANALYZER

The BMA provides various functions to improve the use of model checking as a formal method in software design education. These functions can be divided into the following processes:

- Converting UML statecharts to PROMELA model description language
- Augmenting state variables to the design model
- Obtaining supplied specification templates
- Deriving specification templates from UML sequence diagrams
- Visualizing model checking results
- Visualizing the reachability graph
- Visualizing the specification finite state machine

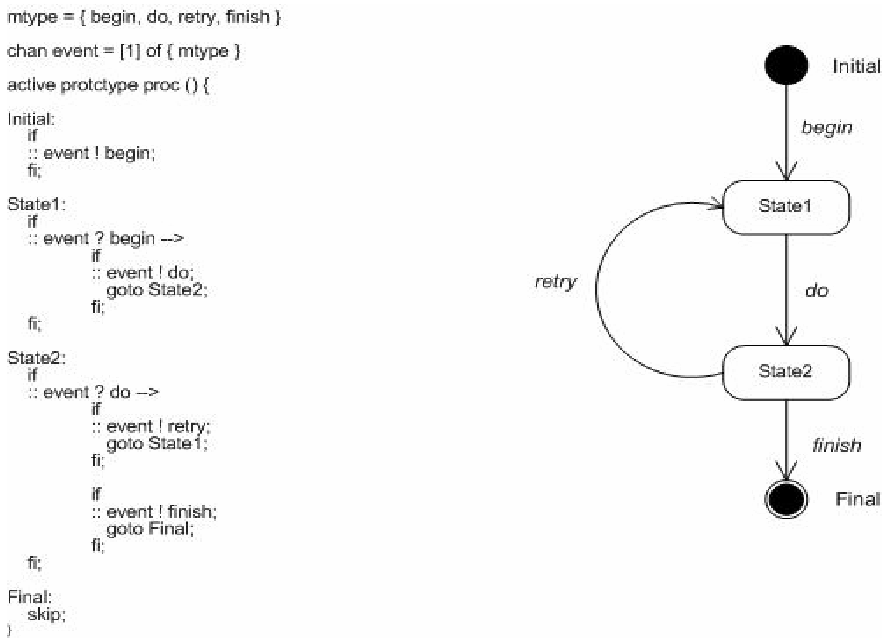
This chapter provides the detailed design motivation and procedure on the realization of these processes by the BMA.

5.1 Converting UML Statecharts to PROMELA Model Description Language

The BMA uses an existing model checker called SPIN [18] to perform model checking. SPIN is a formal verification tool for verifying distributed software systems and uses its own input verification language called PROMELA [18] to perform verification. As a result, the design model needs to be converted to its corresponding PROMELA input.

Before the conversion to the PROMELA model, a finite state machine representing thehierarchical structure of the software design model is obtained by extracting states and transitions from the UML statecharts which are exported to XMI. This finite state machine is stored in an internal data structure.

Based on the finite state machine, the PROMELA file can be divided into blocks, with each block representing the execution at a state in the state machine. The system always starts at the first block. Transition events are translated into signals in PROMELA. When a transition event is signaled, a GOTO statement causes the current block, representing the source state of the transition, to pass the execution control to another block, representing the target state of the transition. When the block representing the final state in the PROMELA file is reached, the system gracefully exists. Figure 13 provides a simple finite state model and its PROMELA description after the conversion.

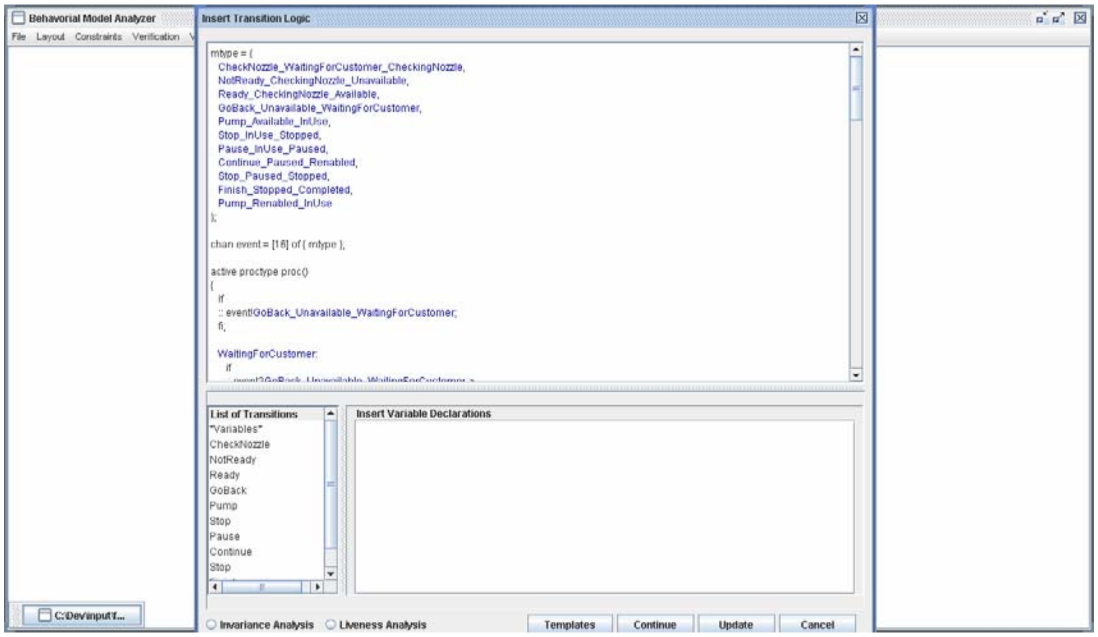


```
mtype = { begin, do, retry, finish }
chan event = [1] of { mtype }
active protctype proc () {
Initial:
   if
   :: event ! begin;
   fi;
State1:
   if
   :: event ? begin -->
            if
            :: event ! do;
               goto State2;
            fi;
   fi;
State2:
   if
   :: event ? do -->
            if
            :: event ! retry;
               goto State1;
            fi;

            if
            :: event ! finish;
               goto Final;
            fi;
   fi;
Final:
   skip;
}
```

Figure 13 A simple finite state machine and its PROMELA model

5.2 Augmenting State Variables to the Design Model

Although UML statecharts can model transitions in the finite state machine, it does not offer ways to describe finer details regarding each transition. Whenever an event is causing the system to transition from one state to another, this change should be captured using state variables. The absence of the use of state variables in the UML statecharts makes it impossible to express the transitional logic as an essential element of the model behavior. To fill this gap, the BMA allows the modeler to define state variables for each transition in the finite state machine. Figure 14 shows the specialized GUI that is used to define state variables.



```
mtype = {
  CheckNozzle_WaitingForCustomer_CheckingNozzle,
  NotReady_CheckingNozzle_Unavailable,
  Ready_CheckingNozzle_Available,
  GoBack_Unavailable_WaitingForCustomer,
  Pump_Available_InUse,
  Stop_InUse_Stopped,
  Pause_InUse_Paused,
  Continue_Paused_Renabled,
  Stop_Paused_Stopped,
  Finish_Stopped_Completed,
  Pump_Renabled_InUse
};

chan event = [16] of { mtype };

active proctype proc()
{
  if
  :: event!GoBack_Unavailable_WaitingForCustomer;
  fi;

  WaitingForCustomer:
    if
```

```
*Variables*
CheckNozzle
NotReady
Ready
GoBack
Pump
Stop
Pause
Continue
Stop
```

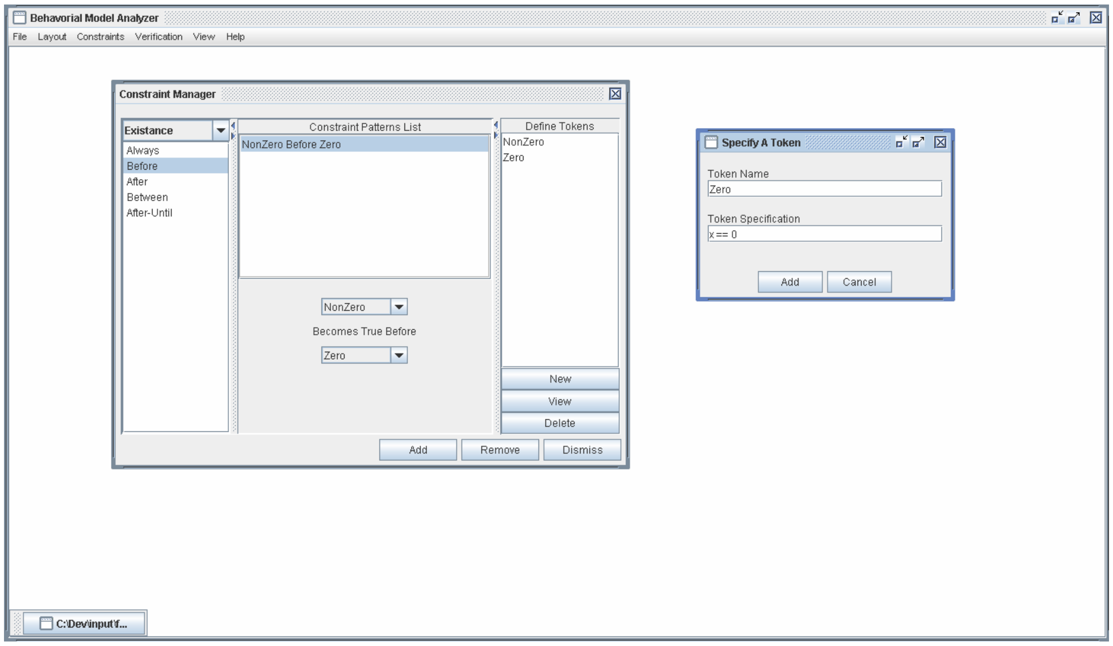
Figure 14 Defining state variables in the BMA

The previous section describes the process of converting the system model, represented in the UML statechart diagram, into the PROMELA model. This process produces the PROMELA skeleton code used for model checking. State variables are incorporated into this skeleton system to accurately describe the model's dynamic properties. The BMA offers two distinct approaches that allow the augmentation of the state variables. The first approach assumes that the modeler has no knowledge of PROMELA and does not understand how the BMA converts the system model into PROMELA. In this case, s/he can associate state variables with each transition in the system's finite state machine. A special GUI lists all transition events in the finite state machine and the modeler can assign different logic for different events. After the modeler has finished supplying all the logic, the BMA inserts the logic into the PROMELA file. The second approach assumes that the modeler may be familiar with PROMELA and prefers to insert the state variables directly into the PROMELA file. In this case, the BMA allows the modeler view and modify the contents of the PROMELA file and saves any modifications made by the modeler.

5.3 Specification Templates

In formal methods, the property of the design model is described by using temporal logic, which requires significant level of expertise in mathematical background. On the other hand, the hierarchical specification templates introduced by Dwyer et, al [12] offer an alternative mechanism to describe the model checking property in an intuitive manner. The list of hierarchical specification templates are described in the previous chapter. In the BMA, each specification template is encoded using one or more atomic propositions that are grouped together using operators such as *always*, *before,* and *until,* etc. Since each atomic proposition represents a specific predicate that is meaningful in the context of the particular design model, it is up to the modeler to define them. To simplify the process, variables can be defined to represent different logical propositions and these variables can be used to form multiple specification templates. Figure 15 displays how to define these variables when supplying specification templates.

It is typical for the modeler to supply one specification template before performing model checking. If more than one specification templates are supplied, they are concatenated into one complex template using the AND operator. If the design model violates one of the specification templates, the BMA will provides the visualization of a counter example for that particular specification template only.



Existance

Before

Constraint Patterns List

NonZero Before Zero

NonZero

Becomes True Before

Zero

Define Tokens

NonZero

Zero

Token Name

Zero

Token Specification

x == 0

Figure 15 Defining specification templates in the BMA

5.4 Deriving Specification Templates from UML Sequence Diagrams

Conventional model checking tools require the properties of the design model to be expressed in temporal logic. Our approach of substituting temporal logic with specification templates alleviates the problem to a degree, but having the modeler to supply the templates is not always desirable due to the following two reasons:

- The modeler may not be familiar with the concept of the hierarchical specification templates and cannot apply them in model checking.
- The modeler may not able to correctly originate the properties of the design model due to inexperience or complexity of the model.

Under these situations, the BMA's ability of deriving the templates directly from the UML sequence diagram is a valuable feature. The BMA extracts messages in the UML sequence diagrams, which are exported into XMI, and store them into an internal data structure. We use a top-down approach to recognize the specification templates creating a rule for each unique template, based on the availability and the sequence of the messages. Given the set of all messages in the UML sequence diagram, all possible combinations of messages and their sequences are examined to detect which rule has been met and what are their elements. Using this available information, the template is constructed using the elements in a detected rule. The prototype only implements the *global*, *before*, *after*, and *between* scopes of the *existence* templates and the *global* scope of the *universality* templates in the hierarchical template system.

Consider the detection of the *between-existence* template as an example, which states that a message or event *Q* should *eventually* occur between messages *P* and *R*. First, the BMA extracts all the unique messages from the UML sequence diagrams and forms all possible cases where one message can occur between two other unique messages. For instance, the UML sequence diagram below in Figure 16 leads to the following template combinations, where the template <A, B, C> is interpreted as Message *B* occurs between Messages *A* and *C*.

- <A, B, C>, <A, B, D>, <A, C, B>, <A, C, D>
- <A, D, B>, <A, D, C>, <B, A, C>, <B, A, D>
- <B, C, D>, <B, D, C>, <C, A, D>, <C, B, D>

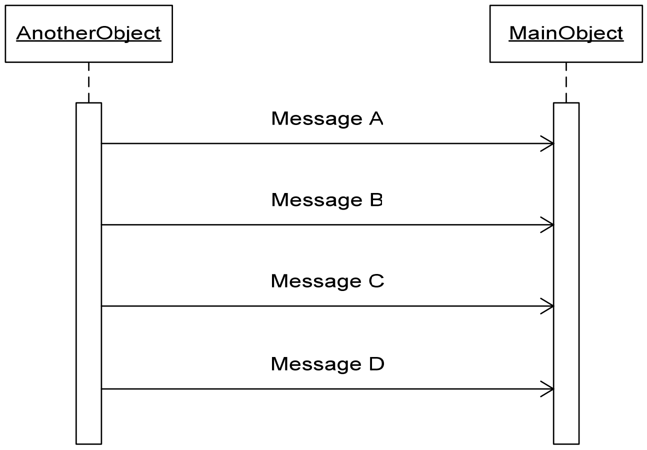


Figure 16 A simple UML sequence diagram

The BMA examines all of these combinations one by one. Redundant combinations that have the same meaning with an already derived template, such as <C, B, A> is the same as <A, B, C>, are eliminated. For the combination <A, B, C>, the BMA finds *one* occurrence of Message *B* in the sequence diagram, and finds whether there is a Message *A* occurring before *B* and *C* occurring after *B*. If so, the template is proved to be true. Otherwise, the BMA examines the next occurrence of Message B and applies the same procedure, until all occurrences of B are examined. For this simple sequence diagram, <A, B, C> is tested to be true, as well as the templates <A, B, D>, <A, C, D>, and <B, C, D>. Using a different example shown in Figure 17, the BMA derives the templates <A, B, C> and <A, C, B> using the same algorithm. For any *between-existence* template <*P*, *Q*, *R*>, the BMA does not support *P*, *Q*, or *R* to be the same message in the UML sequence diagram.

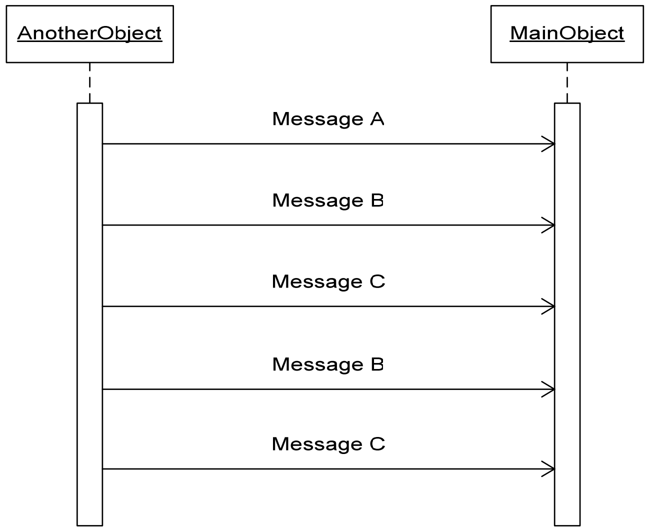
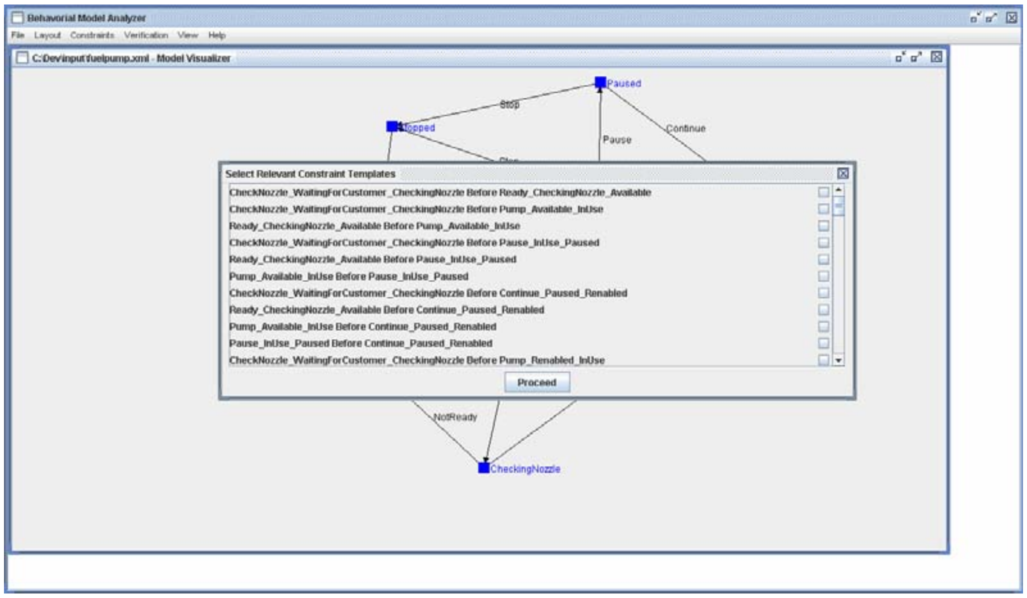


Figure 17 Another UML sequence diagram as an example

Even with limited templates available in implementation, a sequence diagram with a few messages can produce many templates that match the criteria. Some are relevant to model checking while others are not. Performing model checking with many templates at one time can be extremely slow, and the BMA facilitates this requirement by injecting a GUI containing all of the recognized templates, as shown in Figure 18. The modeler can select the templates that are relevant and only the selected templates are used for model checking. Another reason for this screening process is that occasionally the BMA will incorrectly derive a specification template. This problem is caused by the limitation that the BMA is unable to recognize the events that are unreached during execution. Looking at the UML statecharts of a design model, one observes that certain paths in the finite state machine remain unreached, and they are not known until the runtime. Figure 19 shows such an example.


Behavorial Model Analyzer
File Layout Constraints Verification View Help
C:Dev\input\fuelpump.xml - Model Visualizer
Paused
Stop
Pause
Continue
NotReady
CheckingNozzle
Select Relevant Constraint Templates
Proceed

CheckNozzle_WaitingForCustomer_CheckingNozzle Before Ready_CheckingNozzle_Available
CheckNozzle_WaitingForCustomer_CheckingNozzle Before Pump_Available_InUse
Ready_CheckingNozzle_Available Before Pump_Available_InUse
CheckNozzle_WaitingForCustomer_CheckingNozzle Before Pause_InUse_Paused
Ready_CheckingNozzle_Available Before Pause_InUse_Paused
Pump_Available_InUse Before Pause_InUse_Paused
CheckNozzle_WaitingForCustomer_CheckingNozzle Before Continue_Paused_Renabled
Ready_CheckingNozzle_Available Before Continue_Paused_Renabled
Pump_Available_InUse Before Continue_Paused_Renabled
Pause_InUse_Paused Before Continue_Paused_Renabled
CheckNozzle_WaitingForCustomer_CheckingNozzle Before Pump_Renabled_InUse

Figure 18 Confirming the derived templates

In this example, at *State 1*, the finite state machine either transitions to *State 2*, or to *State 3*, but not both at the same time. This means either Transitions *B* and *E*, or *C* and *F*, are reachable. Since we do not know which transitions can be reached for certain ahead of the runtime, it is impossible to produce accurate specification templates based on the occurrence of these transition events.

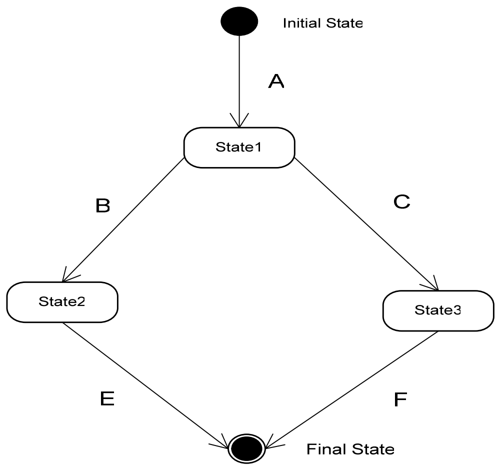


Figure 19 An unreachable path

5.5 Visualizing Model Checking Results

Once the design model has been converted to the PROMELA model and the specification templates converted to temporal logic, the BMA relies on the model checker SPIN to perform model checking. The model checker produces text-based model checking results which indicate whether the property is violated. If it is, then there is a list of states that have been traversed in order for the model checker to make such a conclusion. In other words, this list of traversed states is the counter example produced during model checking. The only problem is that these states do not map to the design model, but rather an optimized finite state machine based on the design model that is generated by the model checker at the beginning of model checking. However, the model checker does provide line numbers in the PROMELA text file that can be mapped to the states in its optimized model checking state machine. Given how the PROMELA text file is structured in Section 5.1, the BMA maps these line numbers to the states in the design model.

When the finite state design model is extracted from the UML statecharts, a graph containing the states and transitions among the states is generated and displayed. Once the list of traversed states from the model checker’s optimized finite state machine is mapped to the list of traversed states in the design model and stored into a data structure, the BMA locates the list of states in the graph and uses a different color to redraw them. Given the order of those states that have been reached during model checking, the transitions executed to reach these states is derived and colored as well. The result is a finite state graph containing the trace of errors found during model checking showing in Figure 20.

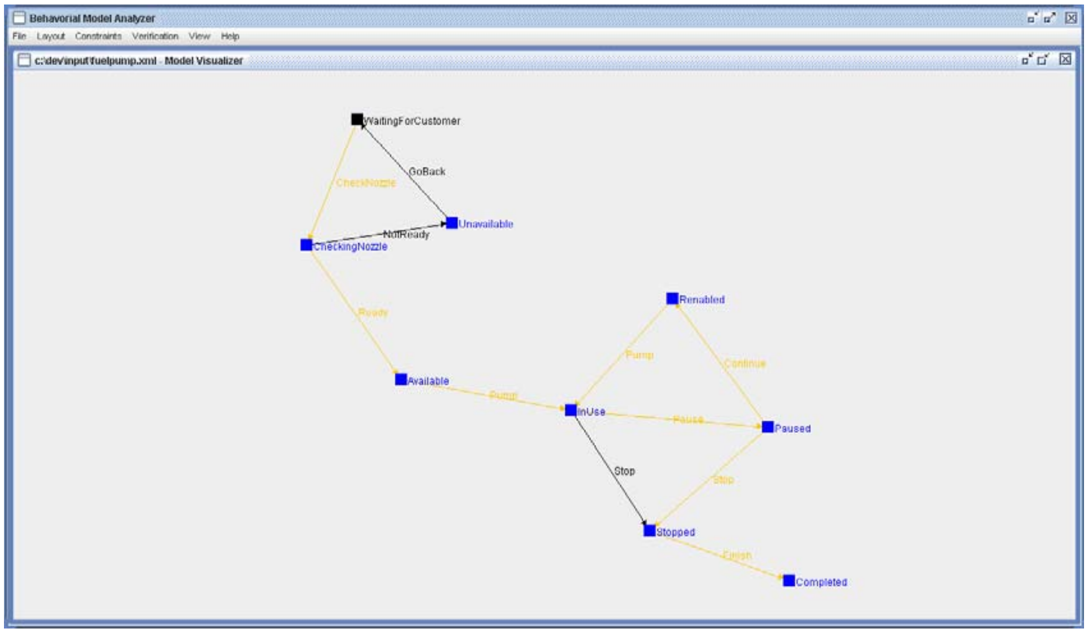


Figure 20 Visualizing model checking results in the BMA

5.6 Visualizing the Reachability Graph

Reachability analysis is a common practice in software verification and primarily used to verify the properties of synchronization structure, such as freedom from deadlock, starvation, and dangerous parallelism [46]. It describes the construction of a state-transition model of a system from models of individual processes. The composite state-transition model is called a reachability graph [46]. In this graph, each node represents a possible state in the system, whereas states represent the value of all variables in the system. Each edge represents progress in a single task. Figure 21 shows the reachability graph constructed from two simple interleaved tasks, T1 and T2, and they are synchronized prior to termination.

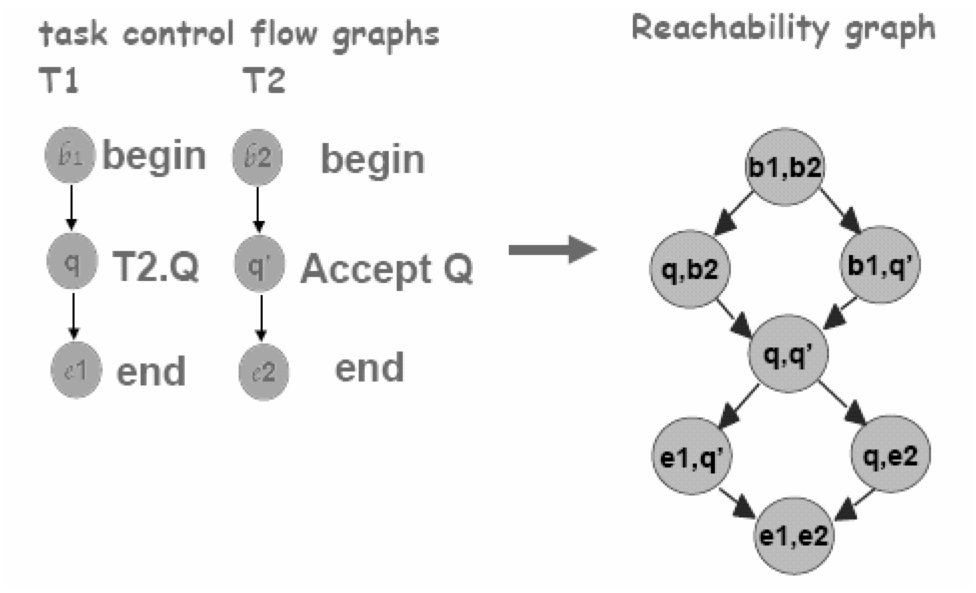


Figure 21 A simple reachability graph

The reachability graph provides an enhanced view of the counter example detected during model checking. The state-transition model is portrayed in a structure which resembles a tree. The root of the tree is the initial state in the model, and a leaf state of the tree is an ending state in the model. The error trace, which generated from the counter example produced by the model checker, is displayed. This error trace starts at the root of the reachability tree, and ends at one of the leaves of the tree. It resembles one path of the tree.

The error trace cannot be drawn in the same way in the visualization described in the previous section since the structure of a tree is different from a finite state machine. The BMA generates the reachability tree first, and then colors the states and transitions in the branch representing the counter example. Ideally, the reachability graph without the counter example should be available as soon as the finite state model is extracted from the UML statecharts and stored in a data structure. However, due to potential nonterminating paths in the tree, the error trace generated from the results of model checking is used as a factor to limit the size of the tree. Thus, the reachability graph feature of the BMA is only available after the model checking has been completed. The BMA uses a modified depth-first algorithm to visualize the reachability tree. The classic depth-first algorithm in this scenario involves drawing nodes in the graph based on transitions evoked at each state. The algorithm starts at the initial state, then it chooses one of transitions evoked from the initial state to draw the next state where the chosen transition leads to. This process will continue till a final state has been reached, suggesting a complete path of the tree is visualized. At the same time, the algorithm keeps a list, which contains all states in the finite state machine. Every time a state has been visualized, it is removed from the list, so the algorithm does not draw the same state again. When all the states have been drawn, the visualization of the complete reachability graph is finished.

This algorithm cannot be directly applied here because the reachability graph in the BMA involves the occurrence of the same state multiple times due to the potential infinite nature of some states in the design model. Therefore the marking strategy in the depth-first algorithm is modified. Now the same node in the tree is allowed to be drawn a number of times before the drawing stops, and this number should be defined based on the complexity of the design model. The strategy of picking a sub-branch of the tree to draw first is also modified and the priority is given to the states and transitions present in the error trace. This way, the counter example will not be partially left out of the reachability graph. Once the reachability graph is completely visualized, the BMA picks the branch that represents the counter example and colors it. Figure 22 shows a reachability graph produced by the BMA.

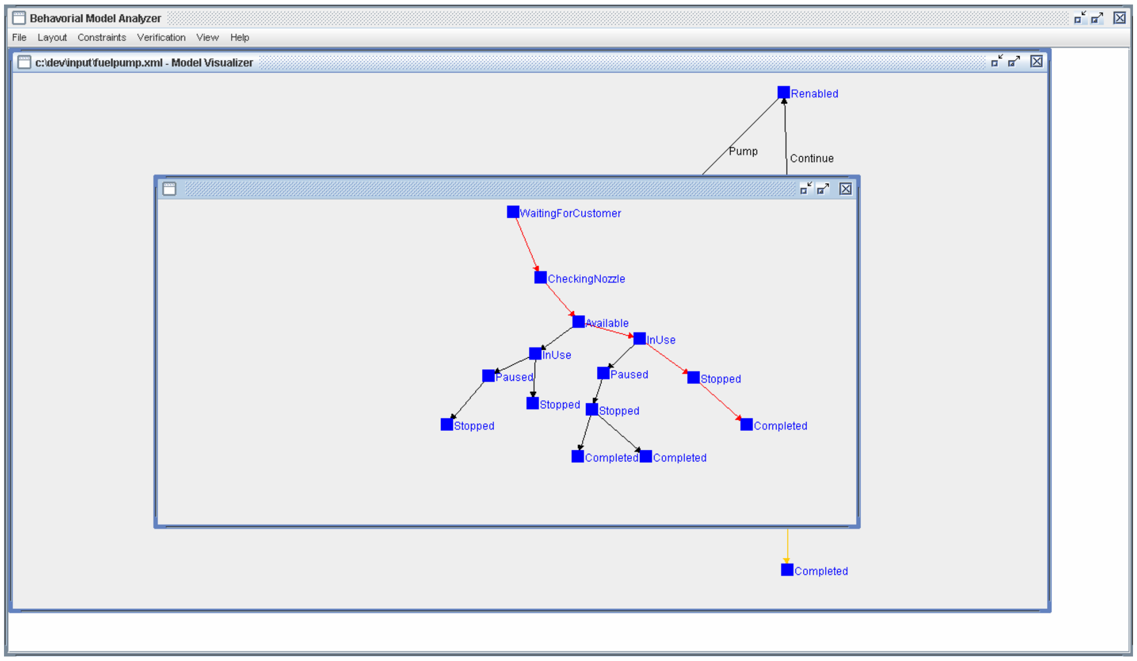
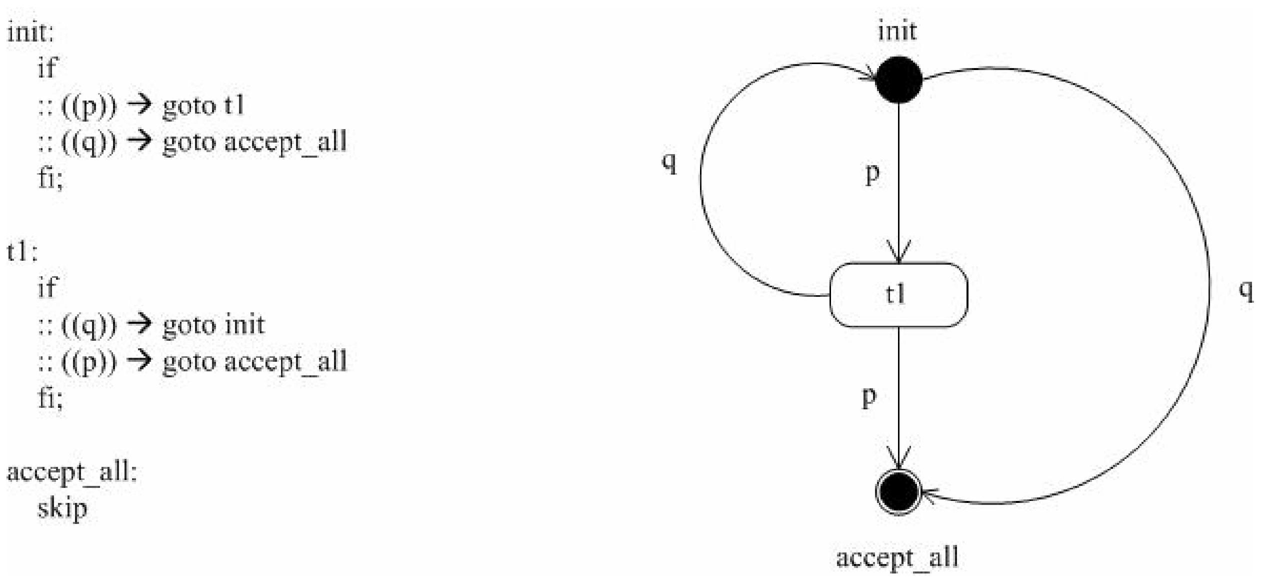


Figure 22 The reachability graph produced by the BMA

5.7 Visualization of the Specification Finite State Machine

The property defining the required behavior of the design model is converted to temporal logic by the BMA and the model checker converts the temporal logic into a specification text file so it can easily construct this finite state machine during model checking. The BMA visualizes this finite state machine to offer enhanced description of the specification templates. Since the conversion from the specification templates to the temporal logic is performed prior to model checking, this feature is available before the model checking process.

The format of the specification text file is identical to the PROMELA file generated by the BMA. Figure 23 shows the content of the text file representing a simple specification finite state machine as a toy example. The text file is divided into sections. Each section represents a state in the finite state machine and transitional behavior is specified under each section. The header of a section corresponds to the name of that particular state. At the state *init*, *p* and *q* are some events that cause the finite state machine to transition into other states. The state *accept_all* is the final state.



```
init:
   if
   :: ((p)) → goto t1
   :: ((q)) → goto accept_all
   fi;

t1:
   if
   :: ((q)) → goto init
   :: ((p)) → goto accept_all
   fi;

accept_all:
   skip
```

Figure 23 A simple specification text file and its Visualization

Since the state machine terminates at the final state, the section representing this state does not contain transitions to other states. Given this highly structured format of the specification text file, the BMA extracts the states and transitions in the text file and visualizes the specification finite state machine it represents. Figure 24 displays the visualization of a much more complex specification finite state machine produced by the BMA based on a specification template derived from the UML sequence diagram. Details of this design model can be found in the case study section of the next chapter.

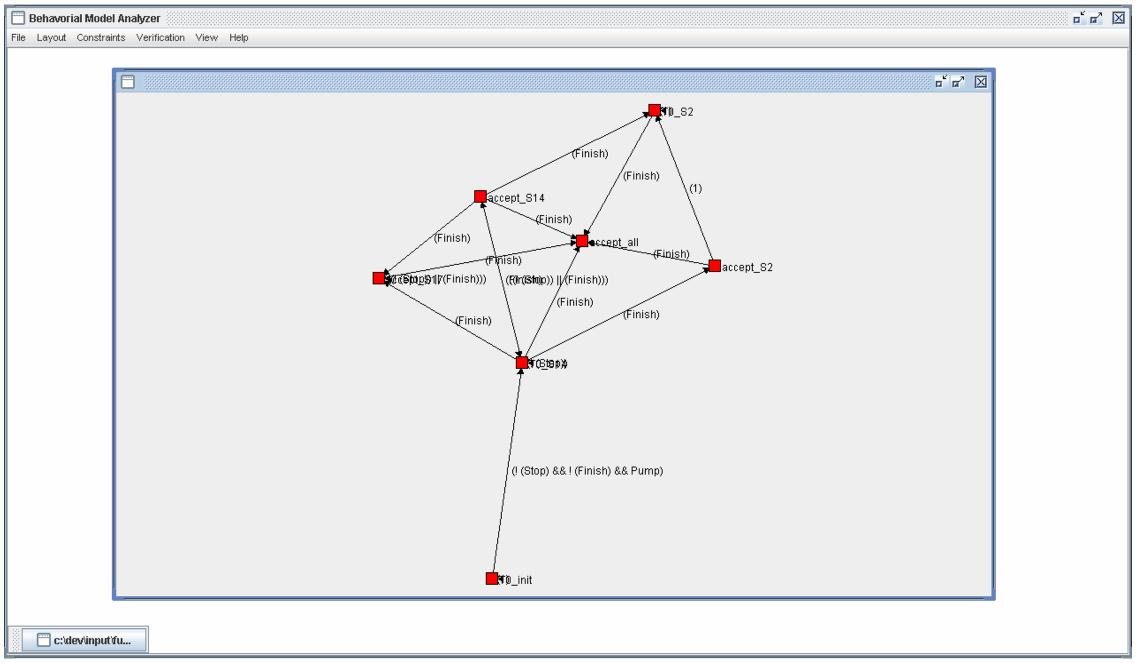


Figure 24 The specification finite state machine visualized by the BMA

6. IMPLEMENTATION AND CASE STUDY

6.1 Implementation

The BMA is developed in Java using the open source Eclipse IDE and it is compiled using Sun’s Java SDK 5.0. All GUI components in the application are built using the Java Swing toolkit. The XMI files containing the UML diagrams are parsed using the open source Xerces XML SAX parser for Java [45]. All graph-based visualizations in the application are generated by the Java Universal Network/Graph Toolkit (JUNG) [17]. The model checker SPIN [18] is used to perform model checking by the BMA. The BMA is developed on Windows, and it can be run on Linux with slight modifications.

6.2 Case Study

To demonstrate that the BMA is able to correctly detect problems giving a model and its property, this section provides a case study revolving around the following BMA features:

- Performing model checking using supplied specification templates
- Performing model checking using derived specification templates
- Visualizing the specification finite state machine
- Visualizing the reachability graph

One strategy is to require students to submit verification queries in terms of BMA templates along with their UML statechart designs. As part of their homework assignment, students can be provided with a set of constraints their designs need to satisfy. They are them asked to supply verification queries and demonstrate the correctness of their designs with respect to these queries. In evaluating submitted assignments the instructor can not only check the correctness of the design, but also students' verification performance. Verification performance can be tested by running the verification queries against the instructor provided reference models that are mutated to measure the fault revealing quality of the student-supplied queries. In the following scenario, we assume a student is submitting a design model, while the instructor uses specifications to verify the consistency of the student's design.

6.2.1 Scenario Description

For the case study, a simple gas pump design model is created and submitted by a student and its UML statechart diagram is shown in Figure 25. In this model, the gas pump starts in an idle start state waiting for a customer to interact with it. When a customer wants to use the pump by lifting a nozzle, the pump checks whether the nozzle is available to be used because sometimes a nozzle can be out of service. This causes the pump go to the state "*CheckingNozzle*". If the nozzle is out of service, the pump goes to the state "*Unavailable*" and eventually goes back to the idle start state and the customer is forced to choose another nozzle. If the nozzle is ready to be used, the gas pump goes to the state "*Available*" and once the customer starts pumping gas into his/her car, the gas pump goes to the state "*InUse*". At this state, the customer can pause the process and the pump goes to the state "*Paused*". The customer can also stop the process to finish fueling the car. In this case, the pump goes to the state "*Stopped*". While the pump is at the state "Paused", the customer can either continue the fueling process or stop the fueling process.

If continue, the pump goes to the state "*ReEnabled*" and the gas is again being pumped into the car and the pump goes back to the state "*InUse*". If stop, the pump goes to the state "*Stopped*". At the state "*Stopped*", the pump goes to the final state "*Completed*" and the fueling process is completed. This UML statechart model is created using Rational Rose and exported into XMI. The XMI file is opened by the BMA and the visualization of the statechart is generated by the BMA, as shown in Figure 26.

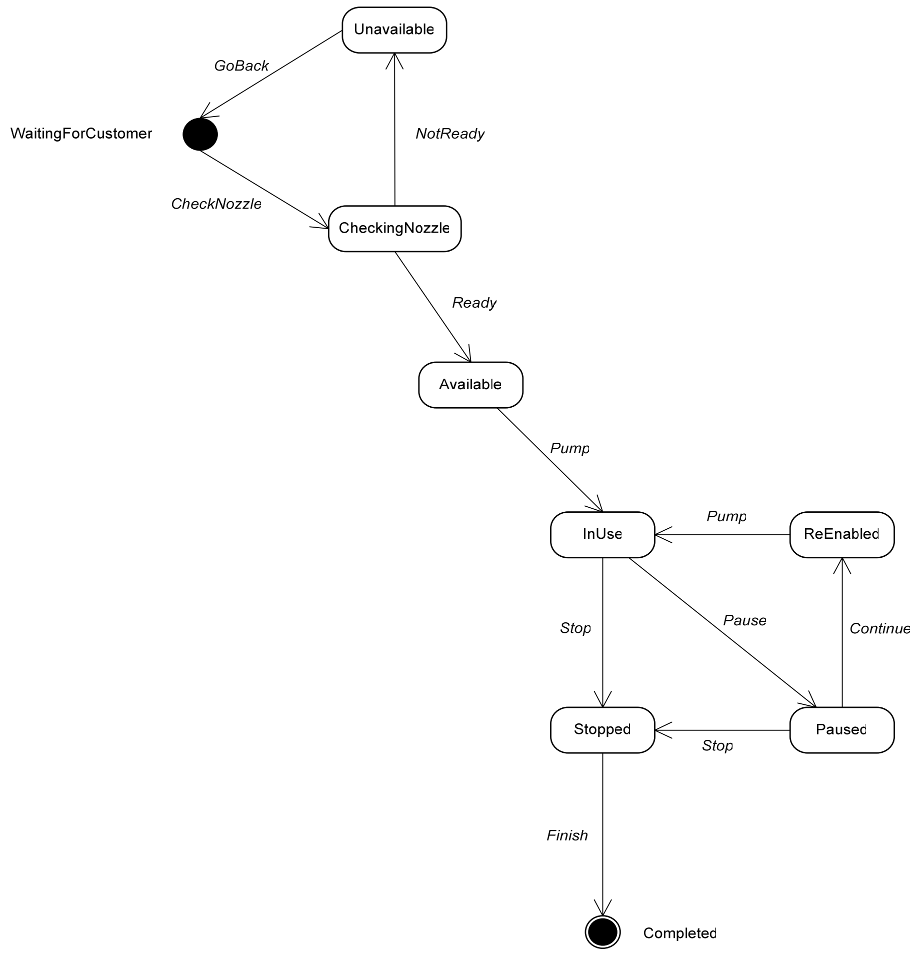


Figure 25 Statechart diagram of the gas pump model

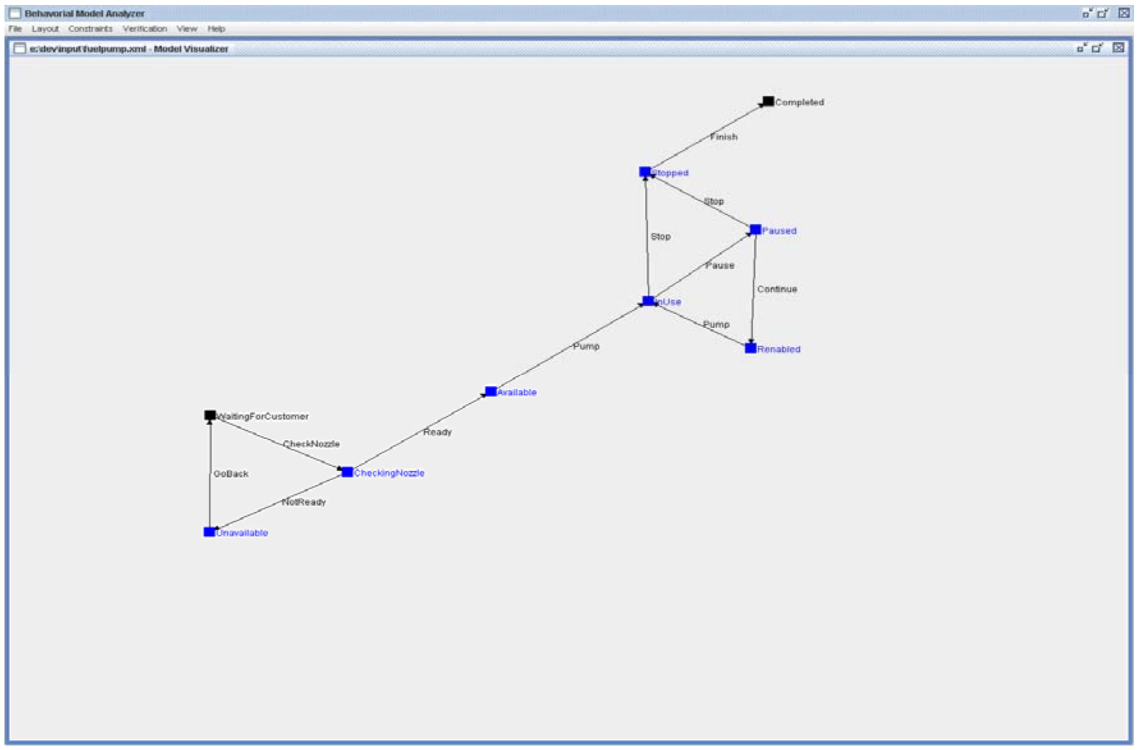
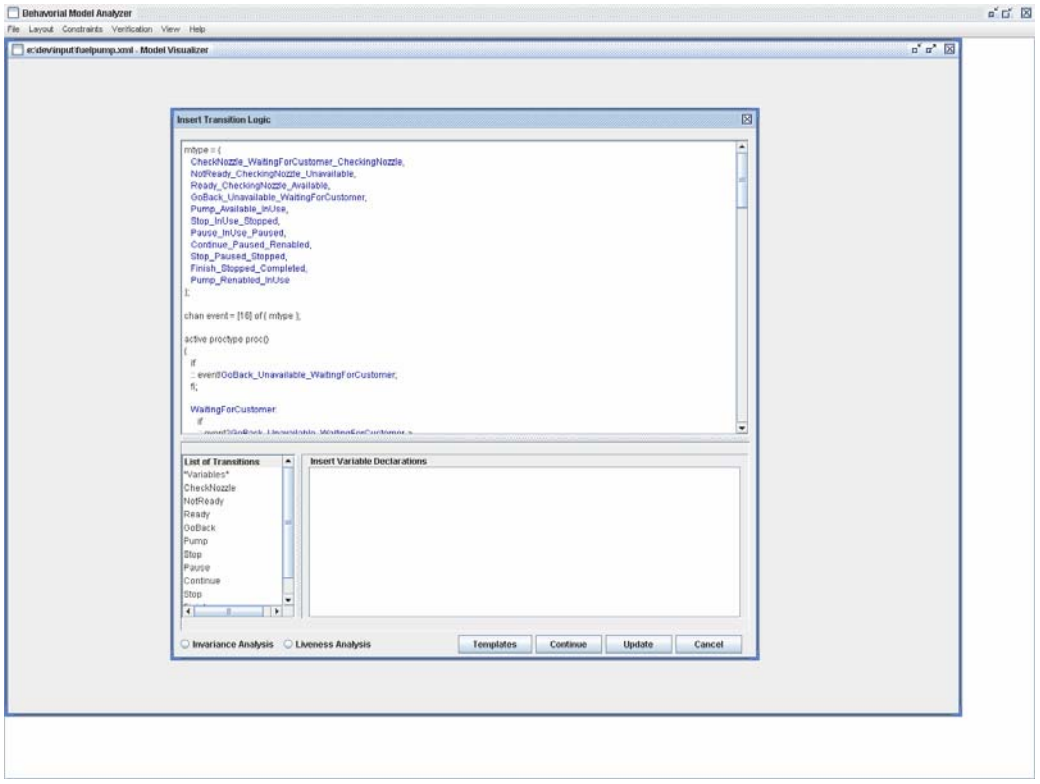


Figure 26 Visualization of the UML statechart model

6.2.2 Model Checking Using Supplied Specification Templates

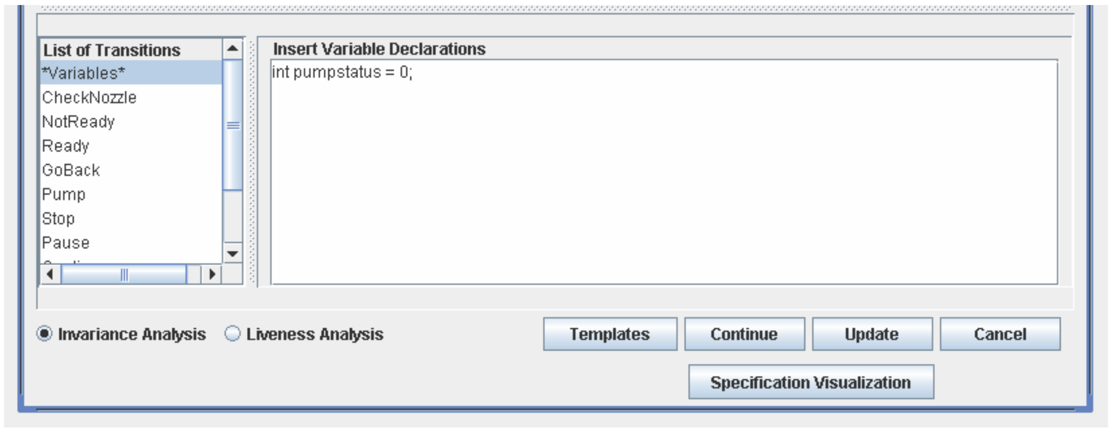
Specifications are needed before model checking takes place. The BMA can either obtain specification templates directly from the instructor or derive them from the UML sequence diagram. If the templates are supplied from the instructor, state variables need to be supplied beforehand. Figure 27 shows the interface for defining state variables. The top section of this interface displays the PROMELA model description of the design model. The PROMELA description can be modified by the instructor if s/he is comfortable in editing the PROMELA file. The bottom section of this interface enables the definition of state variables for each transition in this finite state model.


Behavorial Model Analyzer
File Layout Constraints Verification View Help
e:\devinput\fuelpump.xml - Model Visualizer
Insert Transition Logic

```
mtype = {
  CheckNozzle_WaitingForCustomer_CheckingNozzle,
  NotReady_CheckingNozzle_Unavailable,
  Ready_CheckingNozzle_Available,
  GoBack_Unavailable_WaitingForCustomer,
  Pump_Available_InUse,
  Stop_InUse_Stopped,
  Pause_InUse_Paused,
  Continue_Paused_Renabled,
  Stop_Paused_Stopped,
  Finish_Stopped_Completed,
  Pump_Renabled_InUse
};

chan event = [16] of { mtype };

active proctype proc()
{
  if
  :: event!GoBack_Unavailable_WaitingForCustomer;
  fi;

  WaitingForCustomer:
    if
```

Figure 27 Interface for state variables in the BMA

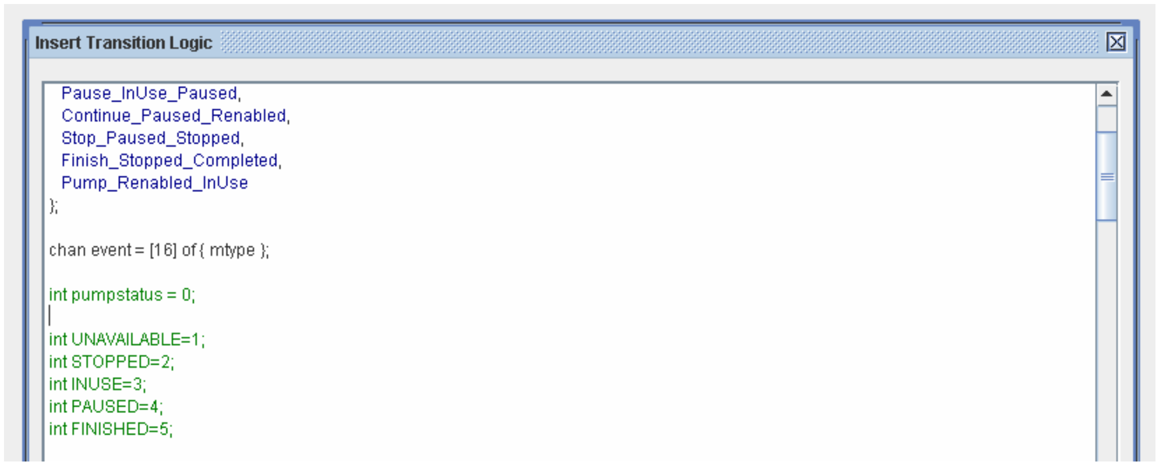
Here we assume the role of being the instructor and providing state variables and properties to the original model designed by the student. Before inserting state variables that are relevant to the case study, let us consider the following logic based on the scenario: The status of the gas pump can be described by a variable whose value represents five states including *unavailable*, *inuse*, *paused*, *stopped*, and *finished*. The state *unavailable* suggests the nozzle is out of service. The state *in-use* suggests that the nozzle is currently being used. The state *paused* suggests the fueling process is paused, and the state *stopped* suggests that the nozzle is currently idle. A state variable named *pumpstatus* is created to hold the value of these states. To create this variable in the BMA, we need to select the *variable* selection item under the *List of Transitions* and then type the declaration "*int pumpstatus = 0*" in the textbox on the right, as shown in Figure 28.



```
int pumpstatus = 0;
```

Figure 28 Initializing a state variable

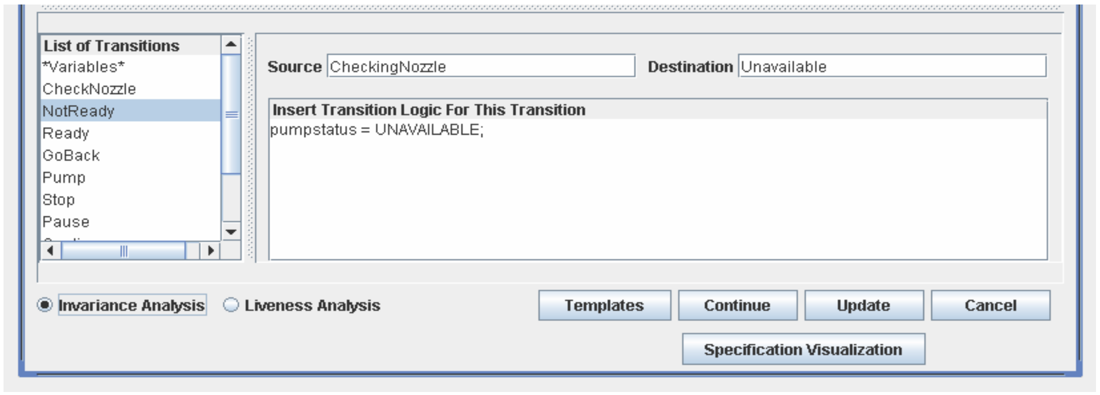
Besides initializing the state variable, additional values are needed to represent the five states. The exact values are irrelevant as long as they are different from each other, and we assign them from 1 to 5 respectively. This process is done by directly inserting expressions into the PROMELA language, as shown in Figure 29.



```
  Pause_InUse_Paused,
  Continue_Paused_Renabled,
  Stop_Paused_Stopped,
  Finish_Stopped_Completed,
  Pump_Renabled_InUse
};

chan event = [16] of { mtype };

int pumpstatus = 0;

int UNAVAILABLE=1;
int STOPPED=2;
int INUSE=3;
int PAUSED=4;
int FINISHED=5;
```

Figure 29 Assigning values for the states

While the finite state machines executes, transitions cause the state variable *pumpstatus* to change states. More specially, the state *unavailable* is assigned to the variable whenever the transitional event *NotReady* occurs. The state *stopped* is assigned whenever the transitional events *Ready* or *Stop* executes. The state *in-use* is assigned whenever the transitional event *Pump* takes place. The state *paused* is assigned whenever the transitional event *Pause* takes place, and finally, the state *finished* is assigned whenever the transitional event *Finish* takes place. Specifying transitional logic with respective to different transitional events in the BMA is demonstrated in Figures 30.

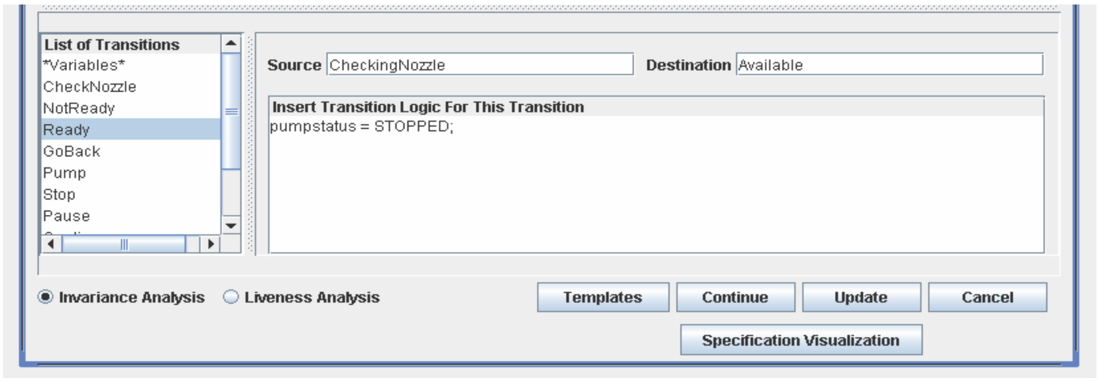


Source: CheckingNozzle Destination: Unavailable

Insert Transition Logic For This Transition

```
pumpstatus = UNAVAILABLE;
```

[✓ Invariance Analysis] Liveness Analysis

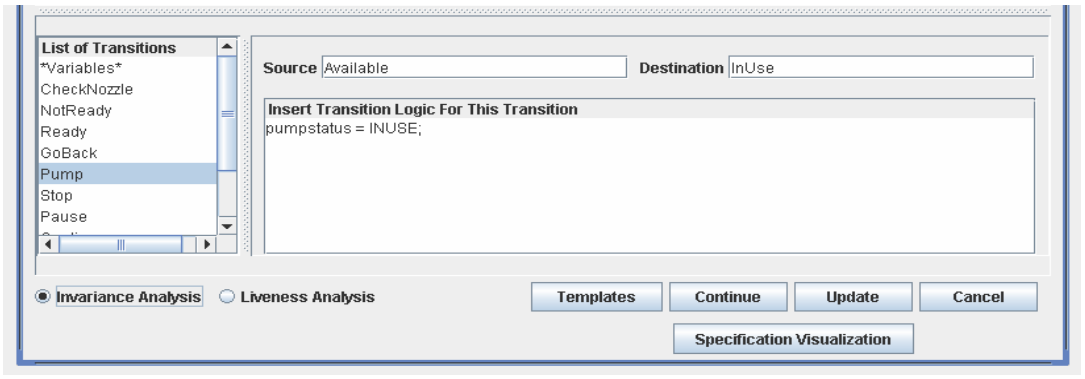


Source: CheckingNozzle Destination: Available

Insert Transition Logic For This Transition

```
pumpstatus = STOPPED;
```

[✓ Invariance Analysis] Liveness Analysis

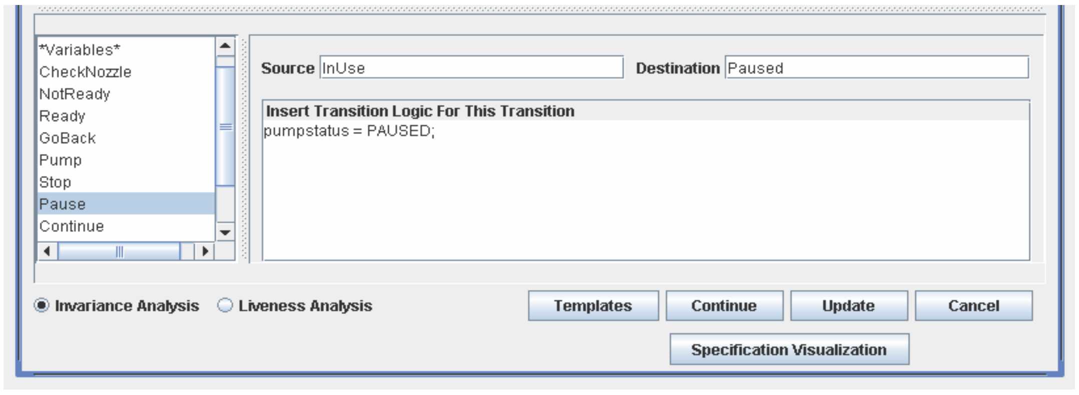


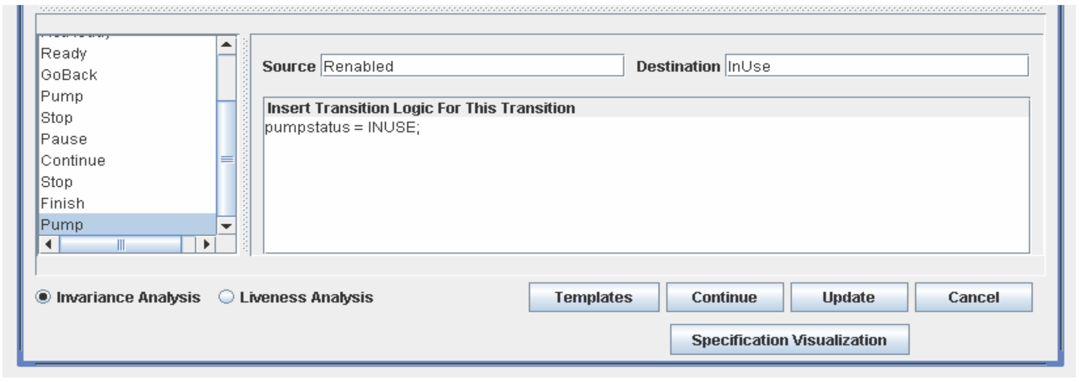
Source: Available Destination: InUse

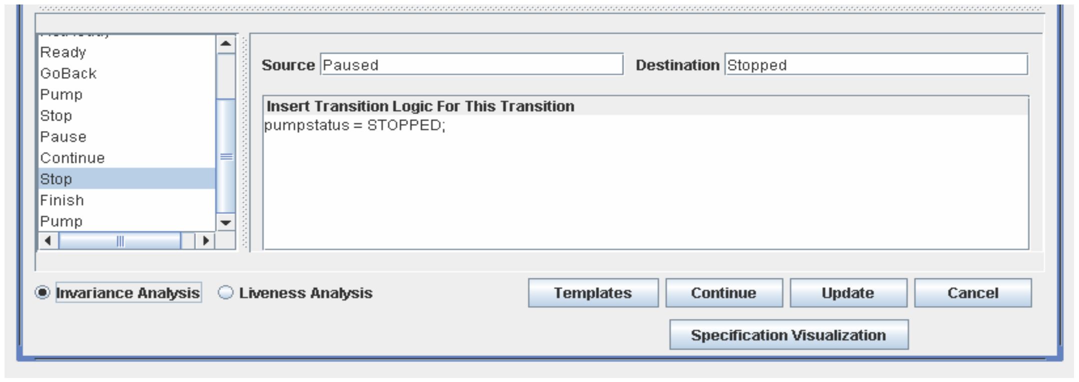
Insert Transition Logic For This Transition

```
pumpstatus = INUSE;
```

[✓ Invariance Analysis] Liveness Analysis


Source: InUse
Destination: Paused
Insert Transition Logic For This Transition
pumpstatus = PAUSED;


Source: Renabled
Destination: InUse
Insert Transition Logic For This Transition
pumpstatus = INUSE;


Source: Paused
Destination: Stopped
Insert Transition Logic For This Transition
pumpstatus = STOPPED;

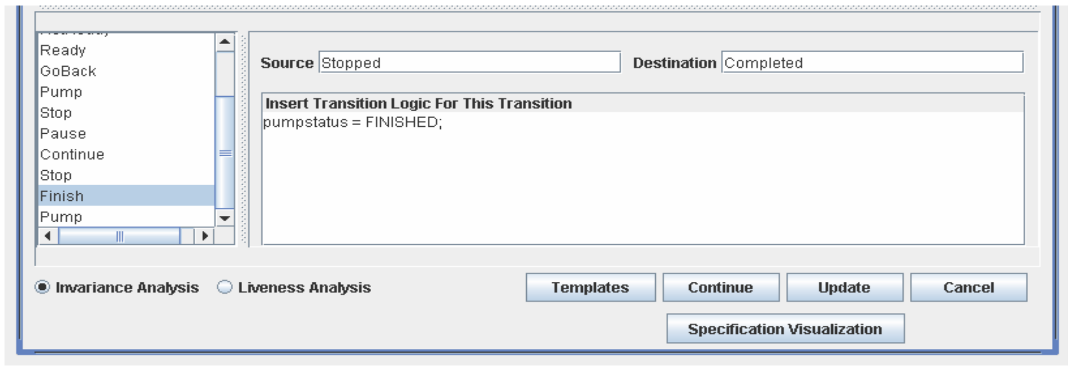
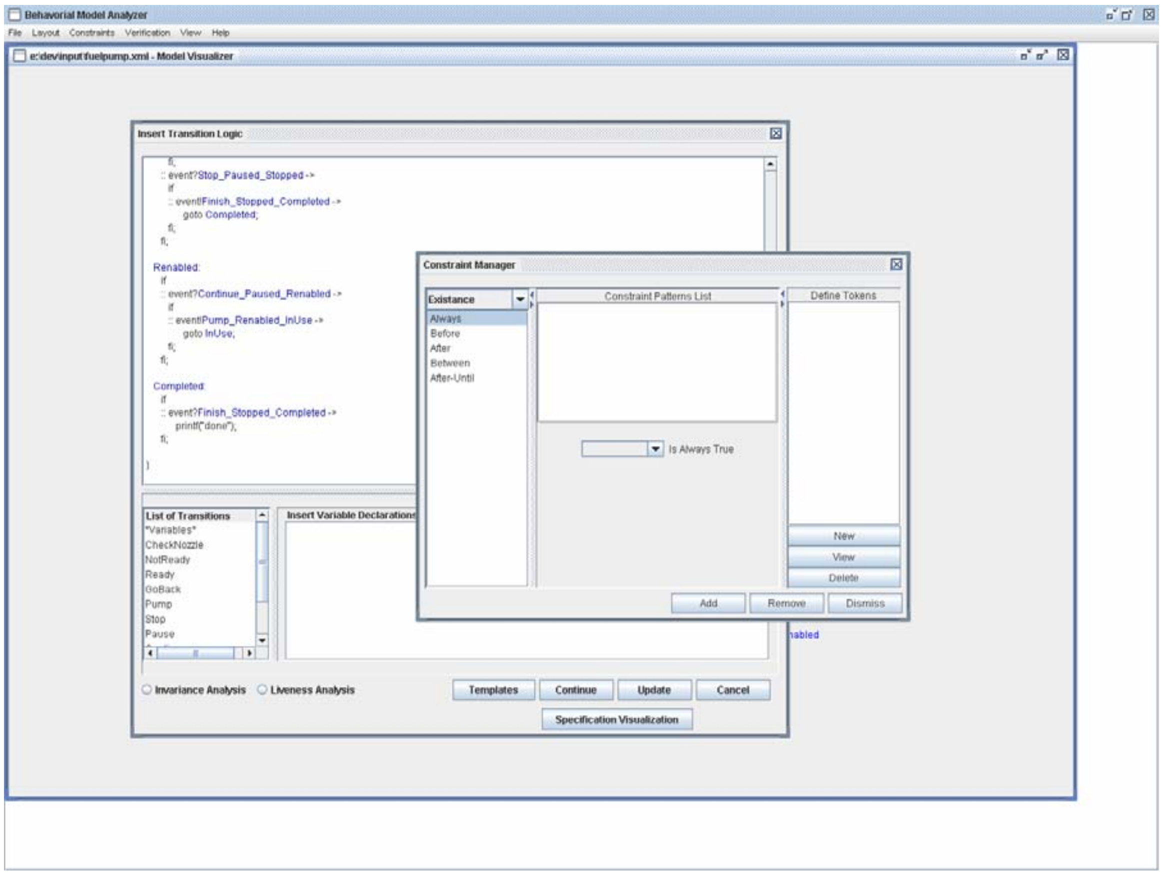

Source: Stopped
Destination: Completed
Insert Transition Logic For This Transition
pumpstatus = FINISHED;

Figure 30 Assigning transitional logic to different transitional events

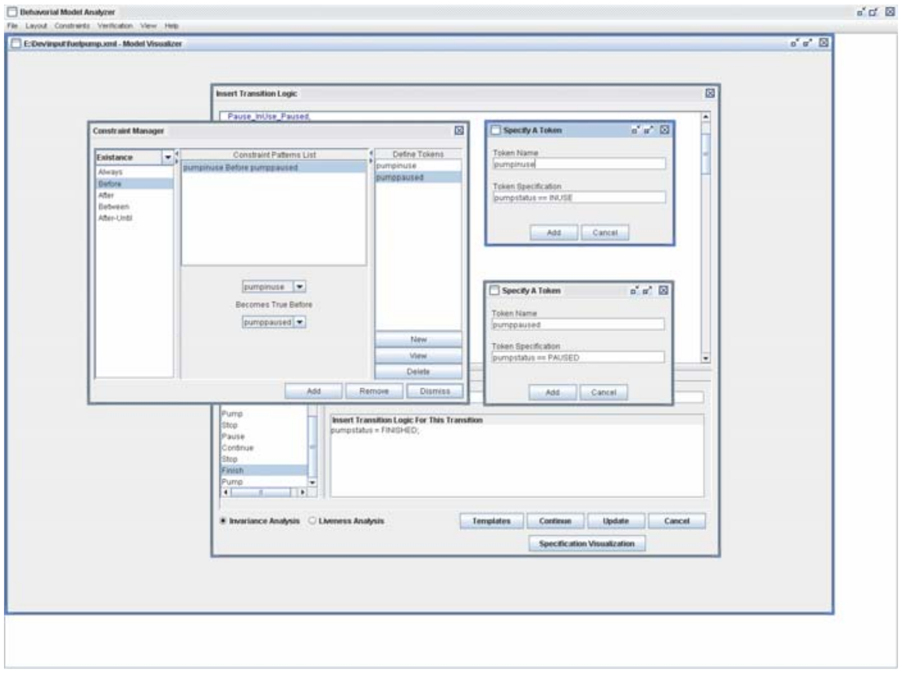
Once the transitional behaviors have been defined, the specification templates for the design model are supplied. In this scenario, we demonstrate by creating three templates for model checking. The first template can be described as “the *pumpstatus* will reach the *inuse* state before the *paused* state”. This refers to the fact that a customer must always start the fueling process before s/he can pause the pump. This template can be divided into two logical propositions: one representing the status of the pump is currently *inuse*, the other is currently *paused*. The two propositions are joined using the scope operator *before*. Figure 31 displays the interface which enables the specification of the templates.



```
      fi;
    :: event?Stop_Paused_Stopped ->
      if
      :: event!Finish_Stopped_Completed ->
          goto Completed;
      fi;
    fi;

  Renabled:
    if
    :: event?Continue_Paused_Renabled ->
      if
      :: event!Pump_Renabled_InUse ->
          goto InUse;
      fi;
    fi;

  Completed:
    if
    :: event?Finish_Stopped_Completed ->
        printf("done");
    fi;

}
```

Figure 31 Specifying specification templates in the BMA

In the scenario, we define two propositions: The first is called *pumpinuse* with the logical expression “*pumpstatus* == *INUSE*” assigned to it. The second is called *pumppaused* with the logical expression "*pumpstatus == PAUSED*" assigned to it. We also specify that the scope operator of this template is *before*, and *pumpinuse* is occurring before *pumppaused*. Figure 32 displays the declaration of the propositions and the template which made up by the two propositions and the *before* operator.



Pause_InUse_Paused

pumpinuse Before pumppaused

pumpinuse
pumppaused

pumpinuse
Becomes True Before
pumppaused

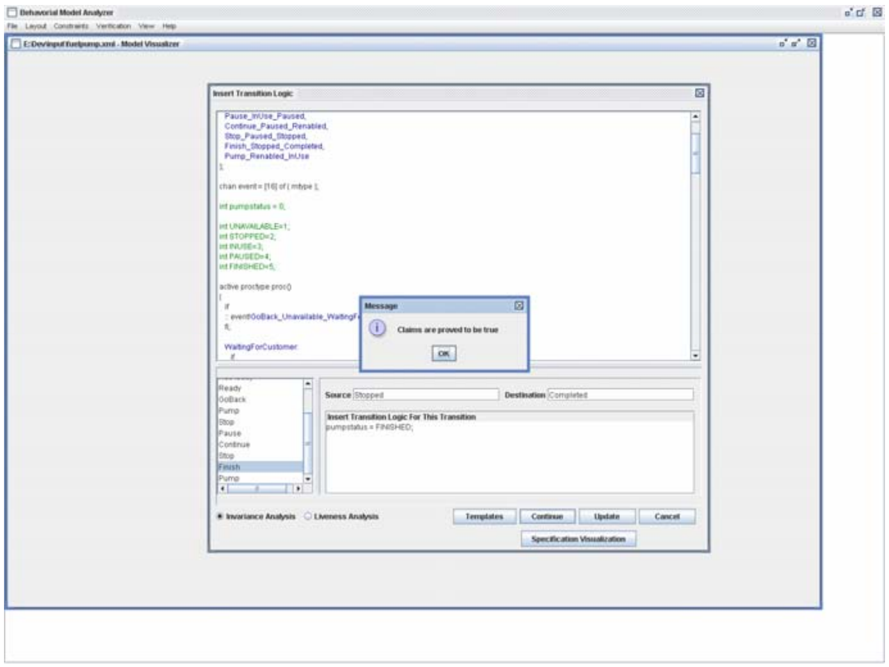
Token Name: pumpinuse
Token Specification: pumpstatus == INUSE

Token Name: pumppaused
Token Specification: pumpstatus == PAUSED

Insert Transition Logic For This Transition
pumpstatus = FINISHED;

Figure 32 Declaration of propositions during template specification

The model checking using the supplied template does not yield any errors, shown in Figure 33. The design model is consistent with the specification template.


Behavorial Model Analyzer
File
Layout
Constraints
Verification
View
Help
E:\Dev\input\fuelpump.xml - Model Visualizer
Insert Transition Logic

```
  Pause_InUse_Paused,
  Continue_Paused_Renabled,
  Stop_Paused_Stopped,
  Finish_Stopped_Completed,
  Pump_Renabled_InUse
};

chan event = [16] of { mtype };

int pumpstatus = 0;

int UNAVAILABLE=1;
int STOPPED=2;
int INUSE=3;
int PAUSED=4;
int FINISHED=5;

active proctype proc()
{
  if
  :: event!GoBack_Unavailable_WaitingF
  fi;

  WaitingForCustomer:
    if
```

Message

Claims are proved to be true

Source: Stopped   Destination: Completed

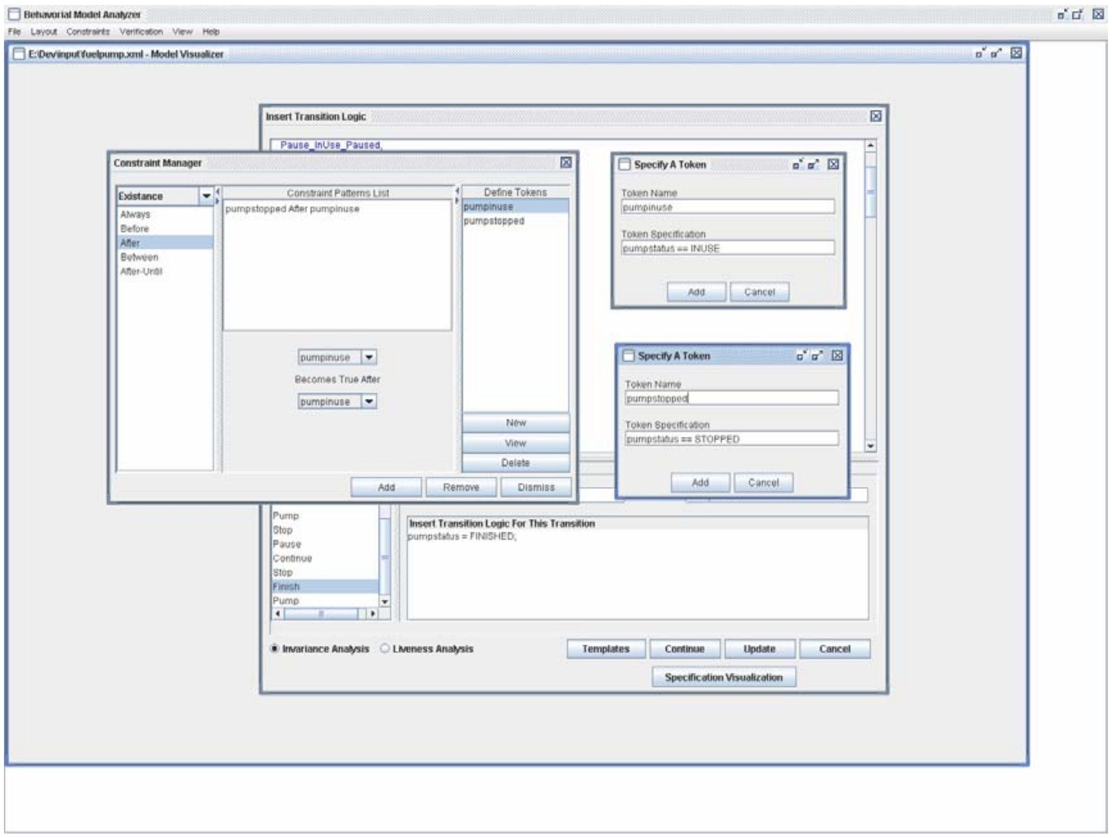
Insert Transition Logic For This Transition

```
pumpstatus = FINISHED;
```

[✓ Invariance Analysis] Liveness Analysis

Figure 33 No error is found during model checking

Let us look at another template: the *pumpstatus* will reach the state *stopped* after reaching the state *in-use*. Transitional logic is already defined in the demonstration of the previous template. Like the previous template, two propositions are declared: one representing the status of the pump is *inuse*, and the other representing the state is *stopped*. The two propositions are joined by the scope operator *after*, as shown in Figure 34. The result of model checking yields no errors, proving the design model is consistent with the template.



Pause_InUse_Paused,

pumpstopped After pumpinuse

pumpinuse
pumpstopped

pumpinuse Becomes True After pumpinuse

Token Name: pumpinuse
Token Specification: pumpstatus == INUSE

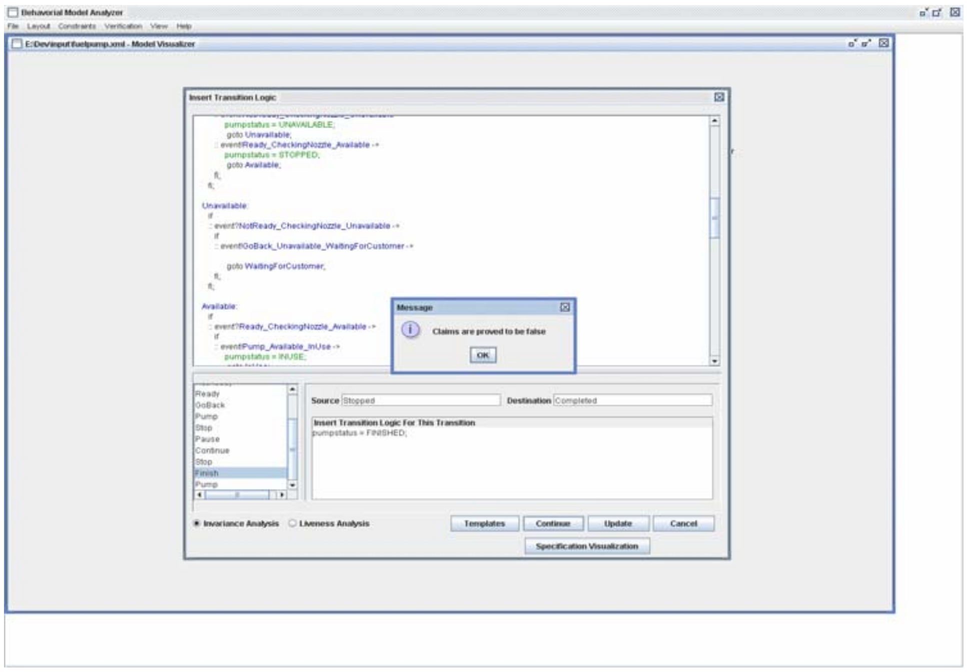
Token Name: pumpstopped
Token Specification: pumpstatus == STOPPED

Insert Transition Logic For This Transition
pumpstatus = FINISHED;

[● Invariance Analysis] ○ Liveness Analysis

Figure 34 Defining the second specification template

While the design model matches with the properties presented by the first two templates, the third template provides a mismatch between the design model and the specification. This template is described as “the *pumpstatus* will reach the state *paused* between the reaching the state *inuse* and the state *stopped*”. Although this template makes sense within the context of the scenario, upon careful inspection of the UML statechart diagram presented in Figure 25, the instructor finds that the transition *Pause* and the state “*Paused*” are not necessarily being reached during execution. In the scenario, the customer may complete the fueling process without pausing in the middle, thus the *pumpstatus* state variable may not be assigned to the state *paused*. Creating the template in the same way as described above and performing model checking proves that the design model violates the specification, shown in Figure 35, and the colored error trace is shown in Figure 36.


Behavorial Model Analyzer
File Layout Constraints Verification View Help
E:\Dev\input\fuelpump.xml - Model Visualizer
Insert Transition Logic
Message
OK
Invariance Analysis
Liveness Analysis
Templates
Continue
Update
Cancel
Specification Visualization

```
      pumpstatus = UNAVAILABLE;
       goto Unavailable;
    :: event!Ready_CheckingNozzle_Available ->
      pumpstatus = STOPPED;
       goto Available;
    fi;
  fi;

Unavailable:
  if
  :: event?NotReady_CheckingNozzle_Unavailable ->
    if
    :: event!GoBack_Unavailable_WaitingForCustomer ->

       goto WaitingForCustomer;
    fi;
  fi;

Available:
  if
  :: event?Ready_CheckingNozzle_Available ->
    if
    :: event!Pump_Available_InUse ->
      pumpstatus = INUSE;
```

Claims are proved to be false

Ready
GoBack
Pump
Stop
Pause
Continue
Stop
Finish
Pump

Source: Stopped   Destination: Completed

Insert Transition Logic For This Transition

```
pumpstatus = FINISHED;
```

Figure 35 Error detected in model checking

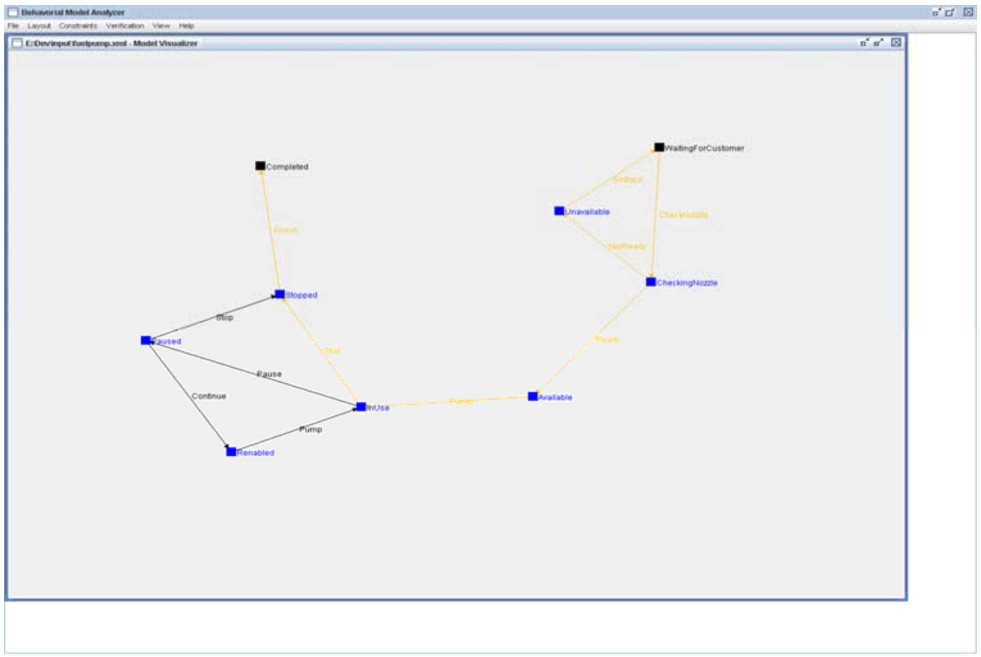


Figure 36 Error trace displayed by the BMA

In order to conform to the specification, the design model can be modified based on the results from model checking. The reason for which the model violates the specification template is that the transition event *Pause* can be unreachable. One alternative model shown in Figure 37 is consistent with the specification. In this model, the transition event *Pause* is guaranteed to be reached.

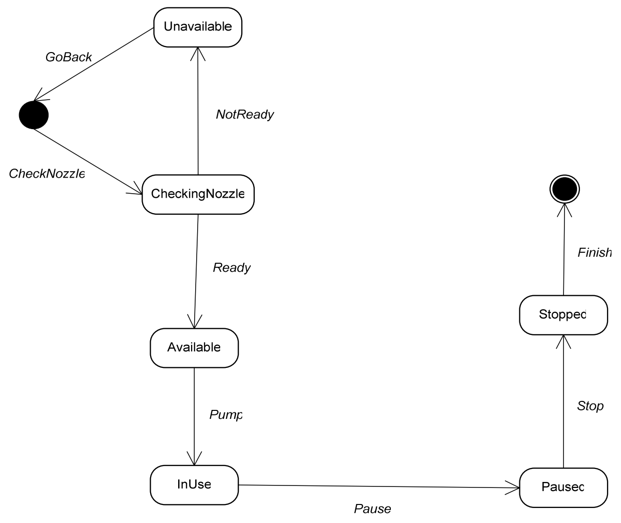


Figure 37 A model conforms to the specification

6.2.3 Model Checking Using Derived Specification Templates

If UML sequence diagram is present, the BMA is capable of deriving specification templates from the sequence diagram. Therefore the instructor can take advantage of this feature rather than design specifications from scratch to test the student's design. Figure 38 shows a sequence diagram used for this case study. It represents the following scenario: A customer wants to use the pump by lifting the nozzle, the gas pump first checks whether the nozzle is available to be used or not (*CheckNozzle*). Then it finds out that the nozzle is ready to be used and informs the customer (*Ready*). The customer connects the nozzle to the gas tank and starts pumping gas into the car (*Pump*). While fueling the car, the customer pauses the process (*Pause*). Then the customer continues fueling (*Continue*), and the nozzle once again starts pumping gas into the car (*Pump*). The gas tank is full and the customer stops fueling (*Stop*), and the customer disconnect the nozzle from the gas tank and the process is finished (*Finish*).

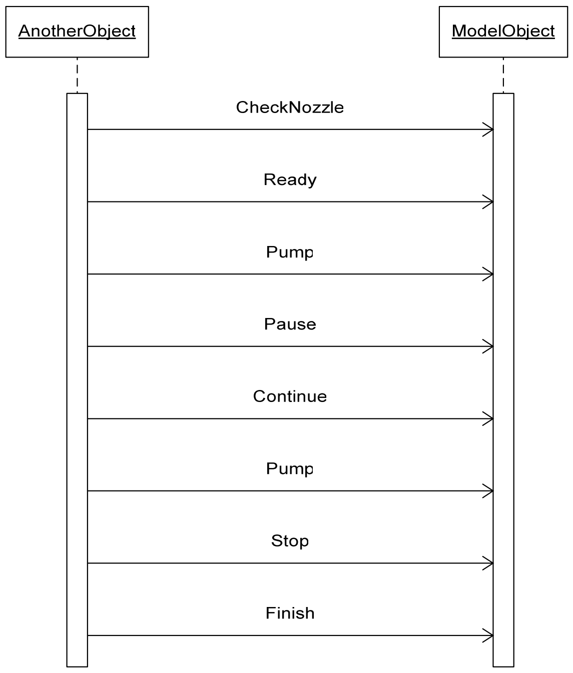
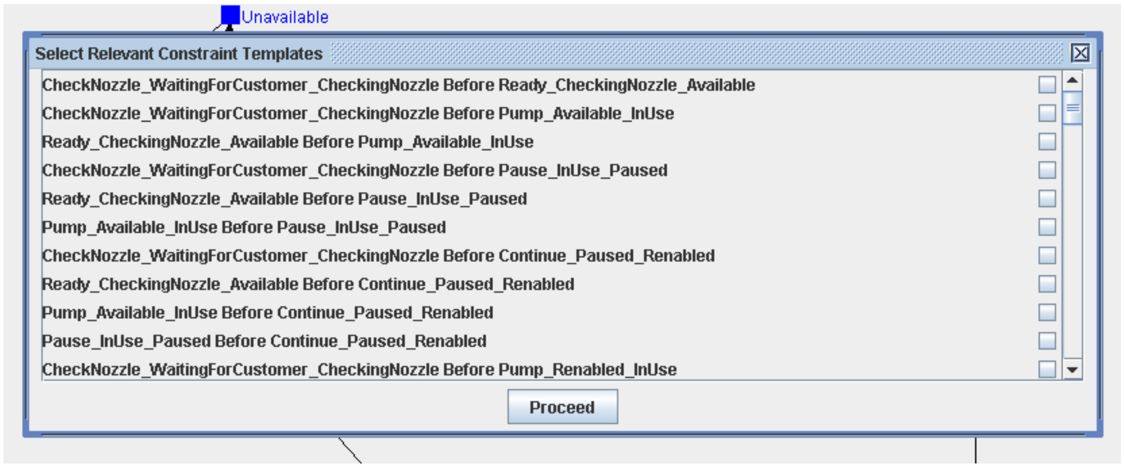


Figure 38 The UML sequence diagram for the case study

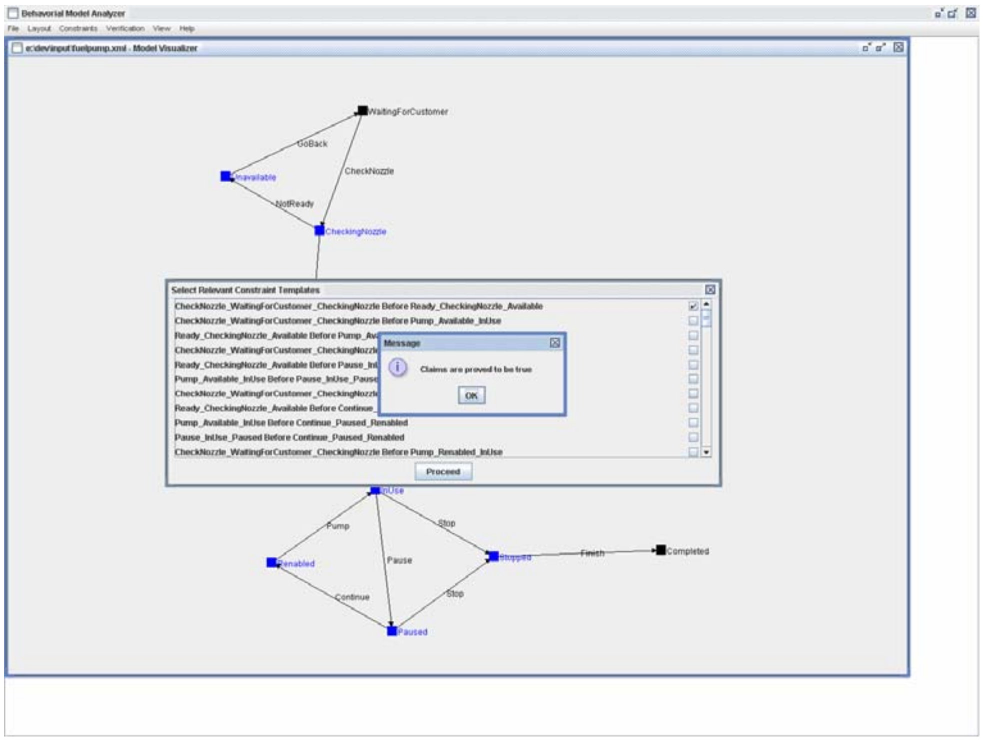
This UML sequence diagram is also created by using Rational Rose and exported into XMI. Once the XMI file has been loaded into the BMA, the BMA automatically applies the algorithm explained in chapter 5 to derive the specification templates. The list of the templates that have been detected is presented to the instructor, as shown in Figure 39.



CheckNozzle_WaitingForCustomer_CheckingNozzle Before Ready_CheckingNozzle_Available
CheckNozzle_WaitingForCustomer_CheckingNozzle Before Pump_Available_InUse
Ready_CheckingNozzle_Available Before Pump_Available_InUse
CheckNozzle_WaitingForCustomer_CheckingNozzle Before Pause_InUse_Paused
Ready_CheckingNozzle_Available Before Pause_InUse_Paused
Pump_Available_InUse Before Pause_InUse_Paused
CheckNozzle_WaitingForCustomer_CheckingNozzle Before Continue_Paused_Renabled
Ready_CheckingNozzle_Available Before Continue_Paused_Renabled
Pump_Available_InUse Before Continue_Paused_Renabled
Pause_InUse_Paused Before Continue_Paused_Renabled
CheckNozzle_WaitingForCustomer_CheckingNozzle Before Pump_Renabled_InUse

Figure 39 The list of specification templates derived by the BMA

The presentation of the templates remains primitive since there is no standard and effective way for displaying them using text-based interface. Each item in this list represents a derived template. Each template contains a scope operator, such as *before*, *after*, or *between*. Rather than using propositions, the templates are composed of the occurrence of messages in the UML sequence diagram. Each message is UML sequence diagram is related to a transitional event, therefore in the list, the name of each transitional event identifies each message in the sequence diagram. The name of the transition is followed by the source and target states of the transition. Upon examining the templates, one or more relevant templates are chosen as properties for model checking. In this case, we pick the first template in the list: *CheckNozzle* is occurring before *Ready*. The result of model checking confirms that the design model does not violate this property, as shown in Figure 40.



Select Relevant Constraint Templates

CheckNozzle_WaitingForCustomer_CheckingNozzle Before Ready_CheckingNozzle_Available	✓
CheckNozzle_WaitingForCustomer_CheckingNozzle Before Pump_Available_InUse	☐
Ready_CheckingNozzle_Available Before Pump_Av	☐
CheckNozzle_WaitingForCustomer_CheckingNozzle	☐
Ready_CheckingNozzle_Available Before Pause_In	☐
Pump_Available_InUse Before Pause_InUse_Pause	☐
CheckNozzle_WaitingForCustomer_CheckingNozzle	☐
Ready_CheckingNozzle_Available Before Continue_	☐
Pump_Available_InUse Before Continue_Paused_Renabled	☐
Pause_InUse_Paused Before Continue_Paused_Renabled	☐
CheckNozzle_WaitingForCustomer_CheckingNozzle Before Pump_Renabled_InUse	☐

Message

Claims are proved to be true

Figure 40 The design model does not violate the specification

The limitation of deriving templates fall into the problem which that the BMA is unable to recognize which transition can be skipped during execution. Since the templates are formed based on pure calculation of message sequences, it is unable to check which transition event can potentially be unreached. The sequence diagram shown in Figure 38 represents a valid scenario. One template generated by the BMA based on this scenario is that the event *Pause* occurs between the occurrences of events *Pump* and *Stop*. Based on the experiments from the previous section, the event *Pause* may not necessarily be reached during execution, thus the design model violates this template, if being used as a specification. This shortcoming places great emphasis upon human examination when the instructor checks off which template is relevant and to be us during model checking.

6.2.4 Visualizing the Specification Finite State Machine and the Reachability Graph

The visualization of the specification finite state machine becomes available immediately after the BMA has obtained the specification templates. If the BMA is relying on users to define templates, the visualization is generated after the templates are defined using propositions and scope operators. If the BMA is deriving templates from UML sequence diagram, the visualization is generated after the user checks off which template is relevant to the model, so that it be included as part of the properties list for model checking. Figure 41 displays the visualization produced using a derived template in the scenario: the event *CheckNozzle* occurs before *Ready*. The overlapping elements produced in the visualization are caused by limitation of JUNG, the library used to generate visualizations in the BMA.

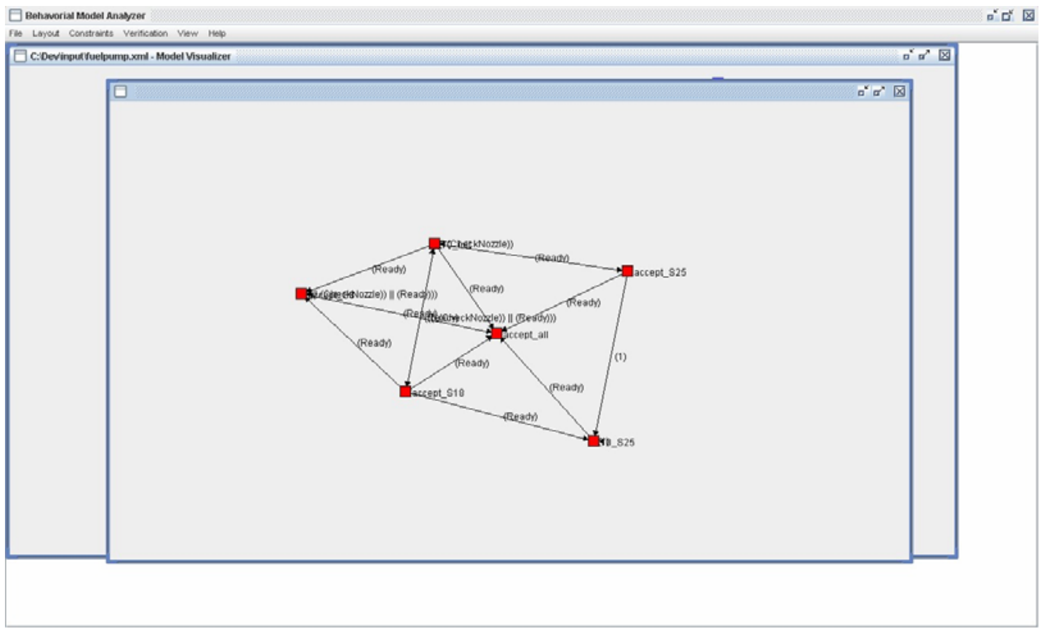


Figure 41 The specification finite state machine

The visualization of the reachability graph is available immediately after the model checking is completed. If an error is detected during model checking, it displays an error trace from the originating state to an ending state. The reachability graph shown in Figure 42 is generated after model checking using the scenario design model and the derived template, the event *Paused* occurs between the occurrences of events *Pump* and *Stop*, as the specification.

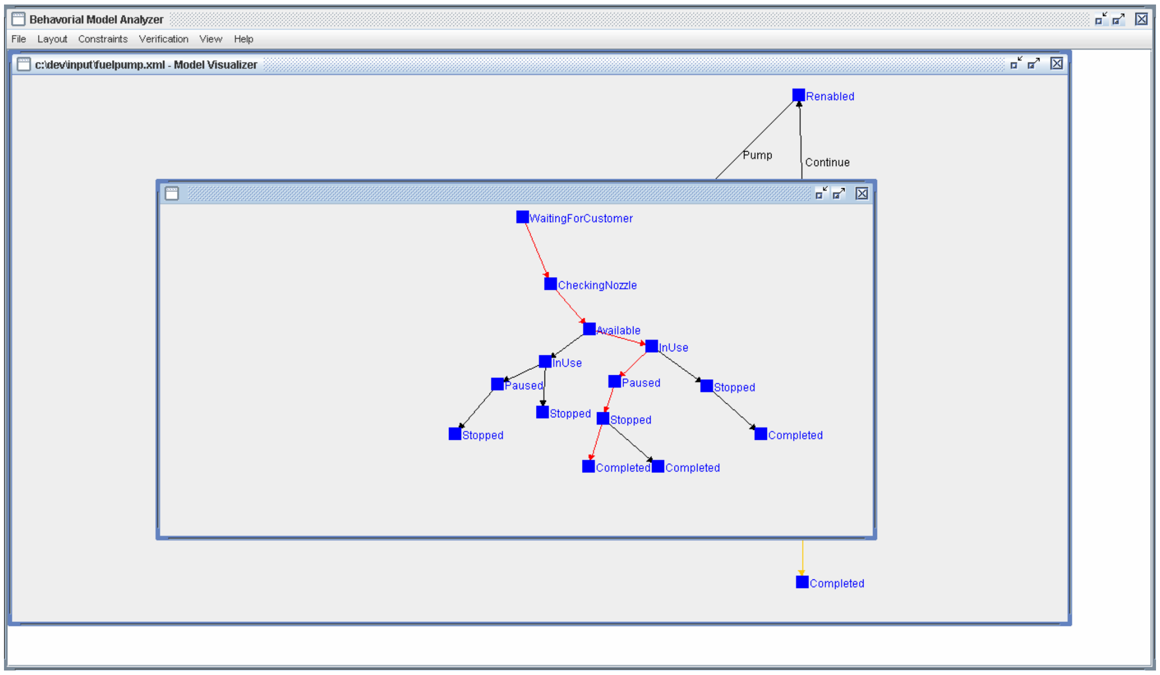


Figure 42 The reachability graph

7. CONCLUSIONS

Formal methods represent powerful techniques in meeting the challenges to reduce design flaws and increase overall quality in software. The need for formal methods increases tremendously in safety critical systems. The uses of formal methods enhance the insight into the understanding of software requirements. Therefore it is critical to bring formal methods into software design education. The BMA represents one step forward in this direction.

7.1 The Limitations of the BMA

Two critical limitations are encountered during the implementation of the BMA:

- Inadequate system model description provided by UML
- Information loss during system model translation to PROMELA

The BMA relies on the UML statecharts to obtain the behavioral model of the software system. Yet the UML statecharts cannot capture the transitional behaviors of the finite state machine. Recall that the UML statecharts are composed of distinct states with one or more state transitions between the states. Although UML statecharts provide mechanisms to specify guard conditions and triggers for every transition in the finite state machine, these mechanisms are inadequate to describe the runtime behavior of a vivid software system. State variables and their manipulation during a transition are not supported by UML. As a result, there is little or no restriction on the action specification during the transitions from one state to another. To get around this problem, an additional component is incorporated into the application that allows state variables to be defined, so that specification properties can be created based on these simple variables to test whether the model is behaving as expected during model checking. This component does not offer enough flexibility to enrich the model since the input is limited to simple variable declarations.

Another problem that has been encountered is potential information loss when converting the system model into the PROMELA input language that is required by the model checker. Information loss here refers as the original model description of system has changed once it has been converted into PROMELA, and suggesting that not all behaviors of the original model have been captured. The quality loss is largely due to the absence of a standard methodology that UML statecharts to be converted into PROMELA systematically since PROMELA is not designed to accommodate translations from UML statecharts when it was created. As one can expect, this problem occurs more frequently when the model becomes more complex. Finite state models with many transitions that are missing transitional logic often causes model checking to be trapped in an infinite loop. On the other hand, complex transitional logic coupled with guard conditions often cause model checking to end prematurely. To ensure the model checking process takes place without these side effects, problematic elements in PROMELA are rewritten at the sacrifice of slight variation to the original finite state model. Simple and moderate-sized finite state models with limited number of transitions are not being affected.

7.2 Future Work

Given the limitations from the previous section, the perfection of the BMA requires additional efforts from the computing community to recognize the potential offered by formal methods and to produce new methodology and better tools that integrates model checking with existing software design techniques. Model checkers such as SPIN need to be improved to accommodate verification of UML models, not just hardware systems. Methodology needs to be invented and standardized to convert UML models to the input model description language accepted by model checkers. The emergent trend toward model-driven development [15] and the adoption of the principles underlying the model-driven architecture [29] implicates the necessity of promoting critical analysis and verification skills within the context of software design education. The significance of teaching formal methods in the context of emerging trends in software development is also argued by Davies and Simpson [11] and Robinson [37].

The industry has already recognized the needs to produce new tools that complement this weakness in the UML specification. iUML [23], an application development environment offers support for executable UML modeling by incorporating Action Specification Language (ASL) [23] in UML models, represents one such effort. At this time, these new tools are neither sophisticated nor flexible enough to fill the gap.

The capability of the BMA can be augmented by implementing the recognition of additional specification templates. Currently the BMA only supports the *existence* templates for the purpose of demonstration. The visualization of the statechart design model and the specification finite state machine is problematic as the visualization library JUNG currently does not support arcs. Using an alternative visualization library or waiting a more mature version of JUNG will eliminate this problem. The flexibility of the BMA can be improved by more vigorous separation of modules in the design of the application by using specialized wrappers for SPIN and JUNG. This way the model checker and the visualization library can easily be swapped out and replaced with alternative tools.

7.3 Conclusion

The BMA offers potential benefits to its target users – undergraduate students who have little or no background in formal methods. With this tool, they can apply model checking, which is one of the most powerful verification techniques existing today. In this case, the students do not have to directly interact with a model checker, to manually design their model in the input language of the model checker, or to generate the sets of specifications for their designs. Through visualization, the application offers valuable output as compared to the text output of the original model checker. By closely working with UML models, the BMA provides a convenient bridge between the model checker and the current software modeling environment.

The idea of bringing formal methods into software design education has little to lose, but a lot to gain. Formal methods have proven themselves to be effective in formal verification of software systems [38]. It provides significant benefits to students if they utilize its potential for software design activities. The BMA represents an initiative to develop a methodology to realize such an idea.

References

[1] Almstrum, V. L., Dean, C. N., Goelman, D., Hilburn, T. B., and Smith, J. (2001). "ITiCSE 2000 Working Group Reports: Support for Teaching Formal Methods", *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education*.

[2] Borger, E., Cavarra, A., Riccobene, E, (2003). "Modeling the Meaning of Transitions from and to Concurrent States in UML State Machines", *Proceedings of the 2003 ACM Symposium on Applied Computing*, pp. 1086 - 1091

[3] Bryant, R. E. (1986). "Graph-Based Algorithms for Boolean Function Manipulation", *IEEE Trans. Comput*. C-35, 8.

[4] Burch, J. R., Clarke, E. M., Long D. E., McMillan, K. L., and Dill, D. L. (1994) "Symbolic Model Checking for Sequential Circuit Verification", IEEE Trans. *Computer-Aided Design of Integrated Circuits*, vol. 13, no. 4, pp. 401-424.

[5] Burstall, R. M. (1974) "Program Proving as Hand Simulation with a Little Induction", In IFIP Congress 74, pp. 308 – 312, North Holland.

[6] Clarke, E. M., Emerson, E. A. (1981). "Design and Synthesis of Synchronization Skeletons Using Branching Time Logics", in *Logic of Programs*: Workshop, Yorktown Heights, NY.

[7] Clarke, E. M., Emerson, E. A., and Sistla, A. P. (1986). “Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications”, *ACM Trans. Program Lang. Syst*. 8, 2, 244-263.

[8] Clarke, E. M., Grumberg, O., and Peled, D. A. (1999). Model Checking, Edmund M. Clarke, Jr., Orna Grumberg, and Lucent Technologies.

[9] Clarke, E. M. & Wing, Jeannette. (1996). “Formal Methods: State of the Art and Future

Directions”, *ACM Computing Surveys 28*, 4 (December 1996): 626-643.

[10] Cleaveland, R., Madelaine, E., and Sims, S. (1995). “*Generating Front Ends for Verification Tools*”, In *Tools and Algorithms for the Construction and Analysis of Systems* (TACAS ‘95), Vol. 1019 of Lecture Notes in Computer Science, E. Brinksma, R. Cleaveland, K. Larsen, and B. Steffen Eds., Springer-Verlag, 153-173.

[11] Davies J. and A. Simpson. (2004). “Teaching Formal Methods in Context,” In *Procedings of the CoLogNET/FME Symposium*, TFM 2004. LNCS 3294, pp. 185-202.

[12] Dwyer, M., Arvrunin, G. and Corbett, J. (1999). “Property Specification Patterns for Finite-State Verfication”, *Proceedings of the 21st international conference on Software engineering*, pp. 411 – 420.

[13] Elseaidy, W., Cleaveland, R., and Baugh, J. (1996). Modeling and Verifying Active Structural Control Systems”, *Sci. Comput. Program*.

[14] Emerson, E. A. (1981). “Branching Time Temporal Logic and the Design of Correct Concurrent Programs”, Ph.D. thesis, Harvard University.

[15] Garlan D. (1994). "Integrating Formal Methods into a Professional Master of Software Engineering Program", *Proceedings of The 8th Z Users Meeting.*.

[16] Gluch P. D. and Weinstock B. C. (1998). "Model-Based Verification: A Technology for Dependable System Upgrade", *CMU/SEI-98-TR-009*.

[17] Gordon, M. (1987). "HOL: A Proof Generating System for Higher-Order Logic", in *VLSIspecification, Verification and Synthesis*. Kluwer.

[18] Holzmann, G. (1997). "The Model Checker SPIN", *IEEE Trans. On Software Engineering*, Vol. 23, No. 5, May 1997, pp. 279-295.

[19] Jagger, D., Schleicher, A., Westfechtel, B (1999). "Using UML for Software Process Modeling", *Foundations of Software Engineering, Proceedings of 7th European Software Engineering Conference*, pp 91-108.

[20] Java Universal Network/Graph Framework. (2004). http://jung.sourceforge.net/.

[21] Jones, C. B. (1986). *Systematic Software Development Using VDM*. Prentice-Hall International, New York.

[22] Kelemen, C., Tucker, A., Henderson, P., Bruce, K., and Astrachan, O. (2002) "Has Our Curriculum Become Math-Phobic?" *Proceedings of the 5th Annual SIGCSE/SIGCUE Conference on Innovation and Technology in Computer Science Education*, pp. 132-135.

[23] Kennedy Carter iUML, http://www.kc.com.

[24] Kindred, D. and Wing, J. (1996). "Fast, Automatic Checking of Security Protocols", in *Proceedings of the USENIX Workshop on Electronic Commerce Protocols*.

[25] Kroger, F. LAR (1977): A logic for algorithmic reasoning. Acta Informatica 8:243–246.

[26] Kryvyi, S., Matveyeva, L. (2003) “Formal Methods of Analysis of System Properties”, *Cybernetics and System Analysis*, Vol. 39, No. 2.

[27] Kurshan, R. P. (1994). “The Complexity of Verification”, *In Proceedings 26th ACM Symposium on Theory of Computing (STOC)*, Montreal, 365 – 371.

[28] Liu, H., Gluch, D. P. (2002). “A Proposal for Introducing Model Checking Into an Undergraduate Software Engineering Curriculum”, *The Journal of Computing in Small Colleges*, Volume 18, Issue 2.

[29] McMillan, K. L. (1993). *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer.

[30] MDA. (2004). MDA: "The Architecture of Choice for a Changing World," http://www.omg.org/mda/executive_overview.htm.

[31] Mills, H., (1988) Software Productivity, Dorset.

[32] Parnas, D. L., (1999) “Software Engineering Programs are not Computer Science Programs”, *IEEE Software*.

[33] Peled, D. (1996). “Combining Partial Order Reduction with On-the-Fly Model-Checking,” *J. Formal Meth. Syst. Des*. 8 (1), 39-64.

[34] Pnueli, A. (1977). “The Temporal Logic of Concurrent Programs”, *Theoretical Computer Science 13*: 45 – 60.

[35] Queille, J. and Sifakis, J. (1982). “Specification and Verification of concurrent systems in CAESAR,” In *Proceedings of Fifth ISP*.

[36] Rajan, R., Shankar, N., Srivas, M. K. (1995). “An Integration of Model Checking with Automated Proof Checking”, *Proceedings of the 1995 Workshop on Computer-Aided Verification*, pp. 84 – 97.

[37] Robinson K. (2004). "Embedding Formal Development in Software Engineering," In *Procedings of the CoLogNET/FME Symposium*, TFM 2004. LNCS 3294, pp.32-46.

[38] Rumbaugh, J., Jacobson, I., and Booch, G. (1999). *The Unified Modeling Language Reference Manual*, Addison-Wesley Longman, Inc.

[39] Sistla, A. P., Gyuris, V. and Emerson, E.A. (2000). "SMC: A Symmetry-Based Model Checker for Verification of Safety and Liveness Properties," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, Volume 9, Issue 2.

[40] Spivey, J.M. (1988) Introducing Z: a Specification Language and its Formal Semantics. Cambridge University Press, Cambridge.

[41] Steffen, B., Margaria, T., Classen, A., and Braun, V. (1996) "The Meta '95 Environment", In *Proceedings of Computer-Aided Verification '96*, Lecture Notes Computer Science, Springer-Verlag.

[42] "The Economic Impacts of Inadequate Infrastructure for Software Testing". (2002) National Institute of Standards and Technology.

[43] Vienneau, Robert. *A Review of Formal Methods* (1993). Griffins AFB, N.Y.: Rome Laboratory.

[44] Woodcock, J., Davies, J. (1996) Using Z: Specification, Refinement, and Proof, rentice Hall Europe, 1996.

[45] Xerces XML Parser. (2004) http://xml.apache.org.

[46] Yeh, W. and Young, M. (1991) "Compositional Reachability Analysis Using Process Algebra", *Symposium on Testing, Analysis, and Verification*.

[47] Yilmaz, L. (2004) “Integrating Model-Based Verification into Software Design Education”, *ASEE Southeast Section Conference*.