AC 2009-2371: AN EXPERIMENT TO EVALUATE AN APPROACH TO TEACHING FORMAL SPECIFICATIONS USING MODEL CHECKING

Salamah Salamah, Embry-Riddle Aeronautical University, Daytona Beach

Steve Roach, University of Texas, El Paso

Omar Ochoa, University of Texas, El Paso

Veronica Medina, University of Texas, El Paso

Ann Gates, University of Texas, El Paso

Experiment to Evaluate Teaching Formal Specifications Using Model Checking

Salamah Salamah
Computer and Software Engineering Dept., Embry-Riddle Aeronautical University.
Steve Roach, Veronica Medina, Omar Ochoa, and Ann Gates
Computer Science Dept., University of Texas at El Paso.

Abstract

The difficulty of writing, reading, and understanding formal specifications remains one of the main obstacles in adopting formal verification techniques such as model checking, theorem and runtime verification. In order to train a future workforce that can develop and test high-assurance systems, it is essential to introduce undergraduate students in computer science and software engineering to the concepts in formal methods. This paper presents an experiment that we used to validate the effectiveness of a new approach that can be used in an undergraduate course to teach formal approaches and languages. The paper presents study that was conducted at two institutions to compare the new approach with the traditional one in teaching formal specifications. The new approach uses a model checker and a specification tool to teach Linear Temporal Logic (LTL), a specification language that is widely used in a variety of verification tools.

1 Introduction

In software engineering, formal techniques such as software runtime monitoring [5], and model checking [3, 8] require formal specifications that are based on mathematics. Formally specifying the behavior of a software system, however, is a difficult task because it requires mathematical sophistication to accurately specify, read, and understand properties written in a formal language. Natural language descriptions of software requirements are inherently ambiguous and often incomplete. Deriving a formal specification from a requirement written in natural-language of concurrent or sequential behavior is made more difficult because of the variety of aspects that must be considered when specifying software behavior. As such, a major impediment to the use of formal approaches in software development remains the difficulty associated with the development of $correct$ formal specifications (i.e, ones that match the specifier's original intent) [6, 7].

Currently, there exists multiple formal specification languages that can be used in a variety of verification techniques and tools. Linear Temporal Logic (LTL) [11], Computational Tree Logic (CTL) [10], and Meta Event Definition Language (MEDL) [9] are some of these languages. The aforementioned languages can be used in a variety of verification techniques and tools. For example, the model checkers SPIN [8] and NuSMV [2] use LTL to specify properties of software and hardware systems. On the other hand, the SMV [3] model checker verifies system behaviors against formal properties in in CTL. MEDL is used by JavaMac in runtime monitoring of java programs [9].

Many undergraduate curricula do not include the topic of formal methods. Certainly, a high level of mathematical sophistication is required for writing, reading, and understanding formal specifi-

cations. A number of tools have been developed to assist the generation of formal specifications, such as the Specification Pattern System (SPS) [4], the Property Elucidation tool (Propel) [17], and the Property Specification tool (Prospec) [12, 13]. While these tools support specification of a wide range of properties, they do not support specification of some common software properties. For example the recurrence property (i.e., if P happens to be false at any given point in a system execution, it is always guaranteed to become true) and stability property (i.e., there is always a point in a system execution where P will become invariantly true) [8], are not supported by the current tools. While pattern-based specifications can be adjusted to specify such properties, it requires someone who is knowledgeable in temporal logic.

Additionally, it is sometimes the case that formal specifications generated by the aforementioned tools do not match the natural language description of the specifications as provided by the tools [16]. As such, it is imperative that software engineers who use formal specifications can validate that the generated properties match their understandings.

In a previous work [14, 15] we introduced a novel technique for analyzing LTL specifications by using the SPIN model checker to illustrate the traces of computations that are accepted by the specifications. The educational component presented in the work can be used to introduce formal specifications and model checking or to supplement existing instructional material on temporal logic. In this paper we describe a case study to evaluate the effectiveness of the new approach in teaching and learning formal specifications (specifically LTL) compared to the traditional method of introducing formal specifications.

Section 2 of the paper presents an overview of essential concepts: model checking, Linear Temporal Logic (LTL), and SPS. Section 3 discusses the traditional approach for teaching formal specification the new approach. For the new approach, we include the teaching component lessons, and exercises in Section 4, and Section 5 provides description of the new approach including lessons and exercises. Section 6 provides the detailed description of the experiment including description of participating subjects, experiment objects, and variables. A summary and acknowledgments conclude the paper.

2 Background

2.1 Model Checking

Model checking is a formal technique for verifying finite or infinite-state concurrent systems by examining the consistency of the system against system specifications for all possible executions. The process of model checking consists of three tasks: modeling, specification, and verification.

Modeling. The modeling phase consists of converting the design into a formalism accepted by the model checker. In some cases, modeling is simply compiling the source code representing the design. In most cases, however, the limits of time and memory mean that additional abstraction is required to come up with a model that ignores irrelevant details. In SPIN, the model is written in the Promela language [8].

Specification. As part of model checking a system, it is necessary to specify the system properties to be checked. Properties are usually expressed in a temporal logic. The use of temporal logic allows for reasoning about time, which becomes important in the case of reactive systems. In

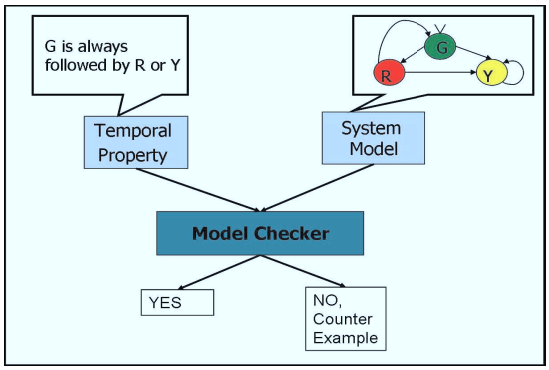

Figure 1. Model checking process.

model checking, specifications are used to verify that the system satisfies the behavior expressed by the property.

Verification. Once the system model and properties are specified, the model checker verifies the consistency of the model and specification. The model checker relies on building a finite model of the system and then traversing the system model to verify that the specified properties hold in every execution of the model [3]. If there is an inconsistency between the model and the property being verified, a counter example, in form of execution trace, is provided to assist in identifying the source of the error. Figure 1 shows the process of model checking.

2.2 Linear Temporal Logic (LTL)

Linear Temporal Logic (LTL) [11] is a prominent formal specification language that is highly expressive and widely used in formal verification tools such as the model checkers SPIN [8] and NuSMV [2] . LTL is also used in the runtime verification of Java programs [18].

Formulas in LTL are constructed from elementary propositions and the usual Boolean operators for not, and, or, imply ($\neg$, $\wedge$, $\vee$, $\rightarrow$, respectively). In addition, LTL provides the temporal operators next (X), eventually ($\diamond$), always ($\Box$), until, (U), weak until (W), and release (R). These formulas assume discrete time, i.e., states s = 0, 1, 2, . . . The meanings of the temporal operators are straightforward[1]

- The formula Xp holds at state s if p holds at the next state $s + 1$,
- $p\,U\,q$ is true at state s, if there is a state $s' \geq s$ at which q is true and, if s' is such a state, then p is true at all states s_i for which $s \leq s_i < s'$,
- the formula $\diamond p$ is true at state s if p is true at some state $s' \geq s$, and
- the formula $\Box p$ holds at state s if p is true at all moments of time $s' \geq s$.

Detailed description of LTL is provided by Manna et al.[11].

A problem with LTL is that the resulting LTL expressions can become difficult to write and understand. For example, consider the following LTL specification: $\Box(a \rightarrow \diamond(p \wedge \diamond((\neg p) \wedge \neg a)))$

[1] In this work, we only use the first four of these operators, as they are the ones supported by the model checkers SPIN and NuSMV. The other operators can be derived from these four.

represents the English requirement "If a train is approaching(a), then it will be passing(p), and later it will be done passing with no train approaching".

2.3 Specification Pattern System: Patterns and Scopes

Writing formal specifications, particularly those involving time, is difficult. The Specification Pattern System (SPS) [4] provides patterns and scopes to assist the practitioner in formally specifying software properties. These patterns and scopes were defined after analyzing a wide range of properties from multiple industrial domains (i.e., security protocols, application software, and hardware systems). *Patterns* capture the expertise of developers by describing software behavior for recurrent situations. Each pattern describes the structure of specific behavior and defines the pattern's relationship with other patterns. Patterns are associated with *scopes*, which define the portion of program execution over which the property holds.

The main patterns defined by SPS are: $universality$, $absence$, $existence$, $precedence$, and $response$. The descriptions given below are taken verbatim from the SPS website [19].

- $Absence$: To describe a portion of a system's execution that is free of certain events or states.
- $Universality$: To describe a portion of a system's execution which contains only states that have a desired property. Also known as Henceforth and Always.
- $Existence$: To describe a portion of a system's execution that contains an instance of certain events or states. Also known as Eventually.
- $Precedence$: To describe relationships between a pair of events/states where the occurrence of the first is a necessary pre-condition for an occurrence of the second. We say that an occurrence of the second is enabled by an occurrence of the first.
- $Response$: To describe cause-effect relationships between a pair of events/states. An occurrence of the first, the cause, must be followed by an occurrence of the second, the effect. Also known as Follows and Leads-to.

In SPS, each pattern is associated with a *scope* that defines the extent of program execution over which a property pattern is considered. There are five types of scopes defined in SPS as shown in Figure 2. A description of these scopes follows:

- $Global$: The scope consists of all the states of program execution.
- $Before\ R$: The scope consists of the states from the beginning of program execution until (excluding) the state where the proposition R first holds.
- $After\ Q$: The scope consists of the states from (including) the state where the proposition Q first holds and includes all the remaining states until program termination.
- $Between\ Q\ And\ R$: The scope consists of the states from (including) the state where the proposition Q holds and includes all the states until (excluding) the state where the proposition R holds.
- $After\ Q\ Until\ R$: This scope is similar to the previous one, with the exception that in the case where the proposition R does not hold then the scope contains all the states from (including) the state where Q holds and all those states until the end of program execution.

A detailed description of these patterns and scopes is provided by Dwyer et.al.,[4].

SPS is a website [19] that provides descriptions of the patterns, including intent, relationships, and known uses. The website provides a mapping of each pattern/scope combination into multiple formal specification languages including LTL. The user then simply replaces his/her propositions

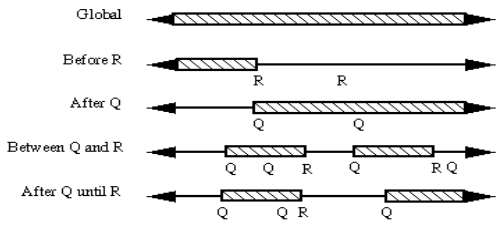

Figure 2. Scopes in SPS [19]

for L, R, P, and Q. For example, the property "Request (E) always triggers Acknowledgment (A), between Beginning of execution (B) and System shutdown (N)", can be described by the $S\ Responds\ to\ P$ pattern within the $Between\ Q$ and R scope. The LTL formula for the pattern and scope combination as provided by the SPS website is:
$\Box((Q \wedge (\neg R) \wedge \diamond R) \rightarrow (P \rightarrow ((\neg R)U(S \wedge \neg R)))UR)$.
Using the user's propositions E, N, A, and B, the resulting LTL specification is:
$\Box((B \wedge (\neg N) \wedge \diamond N) \rightarrow (E \rightarrow ((\neg N)U(A \wedge \neg N)))UN)$.

Tools such as the *Prop*erty *El*icitation (*Propel*) [17] and the *Pro*perty *Spec*ification (*Prospec*) [12, 13] build on SPS by completely automating the generation of formal specifications based on the notions of pattern and scope. These tools interact with the specifier through a series of predefined questions. Based on the specifier's answers, the corresponding pattern and scope combination is selected and the corresponding formal specification is displayed.

3 Traditional Approach to Teaching LTL: Lessons and Exercises

LTL is taught to undergraduates in a typical lecture environment. The objectives of the lesson are for students to be able to translate English statements into LTL, understand LTL statements, and be able to determine the whether a given execution trace satisfies an LTL statement.

The lecture begins by discussing the execution of a program as a sequence of states ordered in time. Then we introduce LTL syntax and show that an LTL valuation is a sequence of Boolean valuations, each of which represents a state. The satisfaction relation is described, as are equivalences and adequate sets of operators. Examples are used throughout. For example, we introduce the semantics of the until operator by stating that Q_1 U Q_2 means that Q_1 is true continuously until Q_2 becomes true, that Q_1 and Q_2 need not be true in the same state, and that Q_2 must be true eventually. Formally, this is written as: $\pi \models Q_1\, U\, Q_2 \iff \exists i \geq 1, \pi_i \models Q_2 \wedge \forall\ 1 \leq j < i, \pi_i \models Q_1$

Once the students are represented with the formal definitions of the LTL operators such as the Until operator above, they are then provided with exercises to translate English statements into LTL or vice versa. Some of these examples follow:

- It is impossible to get to a state where we are started but not ready.
  - $\Box\neg(started \wedge ready)$
- If a request is made, it will be serviced
  - $\Box(requested \rightarrow \diamond serviced)$
- Whatever happens, P will eventually become permanently deadlocked
  - $\diamond\Box deadlock$

- An elevator moving up at the second floor does not go down if it has passengers traveling to the 5th floor
  - $\Box(floor2 \wedge directionUP \wedge button5) \rightarrow (directionUp U\ floor5)$

4 Educational Component for Teaching Formal Specifications

4.1 Goals and Outcomes

The goal of the lessons designed in the new approach are to teach students how to: 1) write, read, and understand formal specifications using LTL, and 2) use tools (SPS and SPIN, respectively) in support of specifying software properties and analyzing the generated specifications using model checking. This section briefly describes the technique and activities that support the attainment of these goals. A detailed description of the technique, lessons, and exercises can be found in Salamah et al [14]. The prerequisite for the educational component is a course in discrete mathematics in which the students specify properties using propositional logic. The component described in the new approach is approximately six hours in length with tutorials on using the tools.

The educational outcomes, given below, are separated using Bloom's taxonomy [1], where level 1 outcomes represent knowledge and comprehension outcomes (those in which the student has been exposed to the terms and concepts at a basic level and can supply basic definitions.); level 2 outcomes represent application and analysis (those in which the student can apply the material in familiar situations); and level 3 represent synthesis and evaluation (those in which the student can apply the material in new situations). The educational outcomes are:

1-1. Students will be able to describe the behavior of simple LTL formulas.
2-1. Students will be able to use a model checker to determine if properties hold in a model.
2-2. Students will be able to identify the appropriate pattern and scope associated with a property and to apply them to generate a formal specification.
2-3. Students will be able to use a model checker to improve their understanding of LTL.
3-1. Students will be able to specify a property in LTL and will be able to define equivalence-class and boundary-value analysis test cases to test the property.

5 Overview of The New Technique

5.1 The Use of SPS: Exercise 1

The new technique uses SPS and model checking to enhance student's understanding of formal specifications. The use of SPS is straightforward. The students are first given an introduction to LTL and a tutorial on SPS. The students are then given hands-on exercises in which they use SPS to specify a series of properties. The lesson focuses on Outcome 2-2 and teaching students how to use SPS to specify patterns and scopes to generate an LTL formula. There are 25 possible SPS pattern and scope combinations. The concentration should be on the use of the response pattern, which is one of the most commonly used patterns in property specification [4].

In the first exercise the students are given a set of properties in natural language and are asked to: (1) define the property pattern, (2) define the property scope, (3) map the propositions used in the pattern and scope to the appropriate phrases in the property description, and (4) generate the LTL formula for the pattern and scope. A sample property is "*A request always triggers reply between start of execution and system shutdown.*"

5.2 Using Model Checking to Enhance Understanding

As opposed to using a model checker such as NuSMV to test the correctness of the model, the technique employed here uses a simple model written in the smv language to test whether an LTL specification holds for a given trace of computation. A *trace of computation* is a sequence of states that depicts the propositions that hold in each state. In this technique, the model produces a simple finite state automaton with exactly one possible execution and a small number of states.

The user models a trace of computation by assigning truth values to the propositions of the LTL formula for a particular state. For example, a user may examine one or more combinations of the following: a proposition holds in the first state, a proposition holds in the last state, a proposition holds in multiple states, a proposition holds in one state and not the next, an interval (scope) is built, an interval is not built, and nested intervals exist. This assignment of values is referred to as a test. The user runs SPIN using the Promela code, the test case, and the LTL specification. Each run assists the user in understanding a formula by checking expected results against actual results. The simplicity of the model makes inspection of the result feasible.

Specification of the LTL formula to be analyzed and the assignments of conditions to propositions are done using the LTLV tool which is graphical user interface to the NuSMV developed for the purpose of this study.

Inspection of LTL formulas helps with understanding the subtleties of the language simply by manipulating the propositions in the. For example, the LTL formula "$\diamond P$" asserts that P holds at some future state. However, a reasonable question would be "What if P holds in the current state where the formula is asserted. Would this be an acceptable behavior of the formula?" The answer to this question is "Yes", since the current state is part of the future in LTL. This piece of information can be easily missed by a naíve or beginner LTL specifier. Using our method, one can easily test such situation by simply asserting that P holds in the first state and test the formula $\diamond P$ and examine LTLV's output.

5.3 Exercises Using a Model Checker

The lessons described here follow a tutorial in which students are introduced to model checking. Students will have used the LTLV tool to check the correctness of simple models. In the exercises, students apply the new technique using the LTLV tool[2].

Exercise 2: The focus of Exercise 2 is to clarify the subtleties of the temporal operators of LTL (Outcome 2-3). For example, the LTL formula "$\diamond P$" asserts that P holds at some future state; however, in LTL, the current state is part of the future and, hence, the situation where P holds in the current state is accepted by this specification. Exercise 2 provides the students with a SPS pattern/scope combination, the corresponding LTL formula, and a list of traces of computations. The students are asked to run the traces of computations against the LTL formula using LTLV and then answer some questions. For example, the students are given the Response-Global pattern/scope and a set of traces. They are then asked to answer questions like *"in the response property, can the cause (P) and effect (S) hold at the same state? Explain your answer."*

Exercise 3: Exercise 3 focuses on teaching students the concepts of traces of computations, and how they can be used to visualize the appropriate behavior of an LTL formula. In addition, the

[2]The current LTLV tool can be requested from salamahs@erau.edu. Also a demo of the tool will be performed at the ASEE 09.

subtle properties of LTL are explained (Outcome 3.2). In the exercise the students are given an LTL formula and a set of traces of computations, and they are asked to predict the output of the LTLV tool when running the formula against each trace. The students are then asked to validate their expected results by actually running the formula and trace in LTLV.

Exercise 4 After completing the previous exercises, the students should be able to define more sophisticated LTL specifications. In addition, they should be able to define test cases in the form of traces of computations to validate their generated LTL formulas. Exercise 4 checks student's ability to satisfy Outcome 3.1. In Exercise 4, students' total understanding of LTL is put to the test. The students are given a list of natural language properties and they are asked to:

- specify a corresponding LTL formula for the property,
- define a set of traces of computations to validate the generated LTL formula,
- define the expected outcome for each trace of computation, and
- Use LTLV to test each trace of computation against generated LTL formula.

A sample property is *"When a connection is made to the SMTP server, all queued messages in the OutBox mail will be transferred to the server."*

6 Description of the Experiment

This section describes the experiment conducted at Embry-Riddle Aeronautical University (ERAU) in Daytona Beach, Florida, and the University of Texas at El Paso (UTEP). The results of the experiment are not available at the time of writing this paper. The results will be available for presentation at ASEE in June 2009. In addition, analysis of the results and supporting documentation of the experiment can be obtained from the authors.

6.1 Objective and Hypotheses

The formal experiment reported in this paper was a controlled investigation that examined the effectiveness of two alternative methods for teaching formal specifications, specifically LTL. The two methods are the traditional approach and the new one introduced in [14]. The objective of the experiment was to determine the effect that the two approaches have over the completeness and correctness of the generated software property specifications. The $null\ hypotheses$ were defined as:

- Students who are introduced to LTL using the new approach identify, on the average, the same number of correct mapping between an LTL formula and a set of traces of computations and the truth value of running each of these traces against the LTL formula as those who are introduced to LTL using the traditional approach.
- Students who are introduced to LTL using the new approach define, on the average, the same number of correct properties in LTL as those who are introduced to LTL using the traditional approach.

The $research\ hypotheses$ were defined as:

- Students who are introduced to LTL using the new approach identify, on the average, larger number of correct mapping between an LTL formula and a set of traces of computations and

the truth value of running each of these traces against the LTL formula as those who are introduced to LTL using the traditional approach.

- Students who are introduced to LTL using the new approach define, on the average, larger number of correct properties in LTL as those who are introduced to LTL using the traditional approach.

6.2 Subjects, Objects, and Variables

Subjects The $experimental\ subjects$ were undergraduate students, graduate students, and researchers from the Computer Science Department at UTEP, and the Computer and Software Engineering Department at ERAU. The subjects will report, through a pre-evaluation form, their degrees of knowledge and experience about requirements engineering, formal specifications, and LTL. Subjects will be given a 50 minute tutorial about: a) need for formal specifications and model checking, b) patterns and scopes; c) specification of properties using patterns and scopes; and d) navigation of LTLV tool.

This experiment will require two groups of participants. The first group will be obtained from UTEP advanced Computer Science classes, such as CS3331 or CS4310. This group will be known as the UTEP group. The second group will be obtained from the Department of Computer and Software Engineering at Embry-Riddle Aeronautical University. This group will be known as the ERAU group.

The UTEP group will have approximately sixty participants and the ERAU group will have approximately twenty five participants. The difference in numbers is due to the gap in the number of students registered in Computer Science at UTEP versus the number of students registered in Computer Science at ERAU. All participants are taking or will be taking advance courses of Computer Science. Neither the age, sex, or ethnic background of the participants is relevant for the purpose of this experiment; therefore, it will not be taken into account.

This experiment will take place on March 25-30. There will be two locations for this experiment. The first part of the experiment will take place at the University of Texas at El Paso in El Paso, Texas and the other part of the experiment will take place at Embry-Riddle University in Daytona Beach, Florida.

In order to recruit participants, an announcement will be made during the advanced CS classes, which will ask the students for their voluntary participation in this experiment. During the announcement, the students will be informed of the purpose of the experiment and what will be required as part of their participation.

Two faculty members from UTEP and ERAU will conduct the experiments. UTEP students will be volunteers from CS 3331 and CS4310 courses. ERAU students will be volunteers from SE300 and CS317 courses. Participants will be upper division students in Computer Science or Software Engineering who have exposure to propositional logic but not to temporal logic. Student proficiency in these areas will be assessed with a short pre-test.

Students from each institution will be randomly assigned to a study group and a control group. The control group will receive a 50 minute lecture on LTL using the traditional approach. The study group will receive a lecture using the new technique and the LTLV tool. The total instruction time for the study group will be identical to the instruction time given to the control group.

At the end of the instruction time, both sets of students will be given a set of approximately 10 exercises designed to assess the student's understanding of LTL. The control group will simply complete the 10 exercises. The study group will complete the exercises without the tool, then will

be allowed to use the tool and resubmit answers. Experiment facilitators will time the students as well as assess the correctness of their answers. Students will be given a 30 minute time span to complete the exercises. To encourage students to perform well on the assessments, students who answer at least 70% of the questions correctly will be entered into a pool for a drawing for token awards. Student have the option to not participate in the drawing.

The experiment is expected to last approximately two hours, including a ten minute break. To protect the privacy and the confidentiality of participants, they will be assigned a unique key to be used on the pretest and final assessments. The key table will be used to enter students into the drawing. At the end of the experiment, the key table will be destroyed. Students will not be given their individual results. However, the correct answers and the results of the drawing will be made public.

The analysis of data will consist of comparing the initial and final scores within each group and comparing the final scores of the control group, the final scores of the study group on the unsupported answers, and the final scores of the study group after using the supporting software tool. There are no foreseen risks or problems associated with the execution of this experiment.

Objects The $experimental\,objects$ in this experiment will be English descriptions of software properties. The domains of application for these properties are operating systems, avionics, and software applications. A complete list of properties used in this experiment, along with the corresponding LTL formulas and traces of computations can be obtained from the authors and will be presented at the the ASEE conference.

Variables The $independent\,variable$ or factor, i.e., the variable that can change the outcome, was the teaching method. There were two levels for this independent variable: traditional approach and the new approach. The $dependent\,variables$, i.e., the outcomes, were the correct LTL score ($corrLTL$) and correct traces mapping score ($corrTraceMap$) of the property specification. The $corrLTL$ score for each participant measures the number of correct LTL formulas defined for property descriptions. The $corrTraceMap$ score measures the number of correct evaluations of the truth value of each trace of computation against a specified LTL formula by the participant.

7 Summary

Software verification remains one of the most challenging aspects of software development. Testing continues to be the most widely used verification technique. However, testing can be costly and time consuming, and as software systems become more complex, testing is less able to provide assurance of correct software behavior. Formal verification approaches such as model checking, theorem proving, and runtime verification complement testing and assist in discovering subtle errors at earlier stages of software development. A major impediment to the use of these techniques remains the difficulty of writing and understanding the required formal specifications. A necessary step, therefore, is to educate and train future generation of software engineers to write and understand formal specifications.

This paper describes an experiment to test the efficiency of a new approach to teaching formal specifications using tools for writing and validating these formal specifications. The experiment will be conducted at the University of Texas at El Paso, and Embry-Riddle Aeronautical University. The participants will be upper level computer science and software engineering students.

8 Acknowledgment

The authors would like to extend their thanks to the students participating in this experiment as well as the lab assistants at both UTEP and ERAU for their help. This work is partially sponsored by the office of the provost at ERAU in Daytona Beach through the office of sponsored research.

References

[1] Bloom, B.S., "Taxonomy of Educational Objectives: The Classification of Educational Goals," Susan Fauer Company, Inc., 1956, pp. 201-207.

[2] Cimatti, A., Clarke, E., Giunchiglia, F., and Roveri, M., "NuSMV: a new Symbolic Model Verifer" International Conference on Computer Aided Verifcation CAV, July 1999.

[3] Clarke, E., Grumberg, O., and D. Peled. Model Checking. MIT Publishers, 1999.

[4] Dwyer, M. B., Avrunin, G. S., and Corbett, J. C., "Patterns in Property Specification for Finite-State Verification," Proceedings of the 21st Intl. Conference on Software Engineering, Los Angeles, CA, USA, 1999, 411-420.

[5] Gates, A., Roach, S., "DynaMICs: Comprehensive Support for Run-Time Monitoring," Runtime Verification Workshop, Paris, France, July 2001, pp. 61-77.

[6] Hall, A., "Seven Myths of Formal Methods," IEEE Software, September 1990, pp. 11-19.

[7] Holloway, M., and Butler, R., "Impediments to Industrial Use of Formal Methods," IEEE Computer, April 1996, pp. 25-26.

[8] Holzmann, G. J., "The model checker SPIN" IEEE Transactions on Software Engineering., 23(5):279-295, May 1997.

[9] Kim, M., Kannan, S., Lee, I., and Sokolsky, O., "Java-mac: a run-time assurance tool for java. In Proceedings of Runtime Verification (RV'01), volume 55 of Electronic Notes in Theoretical Computer Science. Elsevier Science, 2001.

[10] Laroussinie, F. and Ph. Schnoebelen, "Specification in CTL+Past for verification in CTL," Information and Computation, 2000, 236-263.

[11] Manna, Z. and Pnueli, A., "Completing the Temporal Picture," *Theoretical Computer Science,* 83(1), 1991, 97–130.

[12] Mondragon, O. and Gates, A., "Supporting Elicitation and Specification of Software Properties through Patterns and Composite Propositions," Intl. Journal Software Engineering and Knowledge Engineering, XS 14(1), Feb. 2004.

[13] Mondragon, O., Gates, A., and Roach, S., "Prospec: Support for Elicitation and Formal Specification of Software Properties," in Proceedings of Runtime Verification Workshop, ENTCS, 89(2), 2004.

[14] Salamah, S., and Gates, A., "A Technique for Using Model Checkers to Teach Formal Specifications" Proceedings of the 21st IEEE-CS International Conference on Software Engineering Education and Training (CSEE&T), Charleston, SC, April 2008, 181-188.

[15] Salamah, S., Gallegos, I., and Ochoa, O., "A Novel Approach for Software Property Validation" Proceedings of the International Conference for Software Engineering Theory and Practice (SETP), Orlando, Fl, July 2008

[16] Salamah, S., Gates, A., Roach , S., and Mondragon, O., "Verifying Pattern-Generated LTL Formulas: A Case Study. Proceedings of the 12th SPIN Workshop on Model Checking Software. San Francisco, California, August, 2005, 200-220

[17] Smith, R.L., Avrunin, G.S., Clarke, L.A., and Osterweil, L.,J. "PROPEL: an approach supporting property elucidation." In Proceedings of the 24rd International Conference on Software Engineering. 2002, pp. 11-21

[18] Stolz, V., and Bodden, E., "Temporal Assertions using AspectJ", *Fifth Workshop on Runtime Verification* Jul. 2005.",

[19] Spec Patterns, http://patterns.projects.cis.ksu.edu/, March 2009.