A Technique for Using Model Checkers to Teach Formal Specifications

Salamah Salamah
Computer and Software Engineering Dept., Embry-Riddle Aeronautical University.

Ann Q. Gates
Computer Science Dept., University of Texas at El Paso.

Abstract

The difficulty of writing, reading, and understanding formal specifications is one of the main obstacles in adopting formal verification techniques such as model checking and runtime verification. Introducing concepts in formal methods in an undergraduate program is essential for training a workforce that can develop and test high-assurance systems. This paper presents educational outcomes and outlines an instructive component that can be used in an undergraduate course to teach formal approaches and languages. The component 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. The paper also introduces a novel technique that analyzes LTL specifications by using the SPIN model checker to elucidate the behaviors accepted by the specifications.

1 Introduction

Formal verification techniques such as model checking [3] and runtime monitoring [9] are effective approaches for improving the dependability of programs. These techniques check the correctness of the system against specifications written in a formal specification language. Some advantages of using a formal specification language include: they are unambiguous; they can be used to capture pre- and post-conditions of methods and to generate test cases [4]; and, because they are mathematically based, it is possible to prove properties about the specifications themselves [6]. Because of the benefits of formal methods, it is important that students learn the basics of formal specifications and are able to apply them to define system properties.

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 specifications. A number of tools have been developed to assist the generation of formal specifications, such as the Specification Pattern System (SPS) [5], the Property Elucidation tool (Propel) [12], and the Property Specification tool (Prospec) [11]. 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) [7], 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.

This paper presents 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 this paper can be used to introduce formal specifications and model

978-0-7695-3144-1/08 $25.00 © 2008 IEEE
DOI 10.1109/CSEET.2008.25

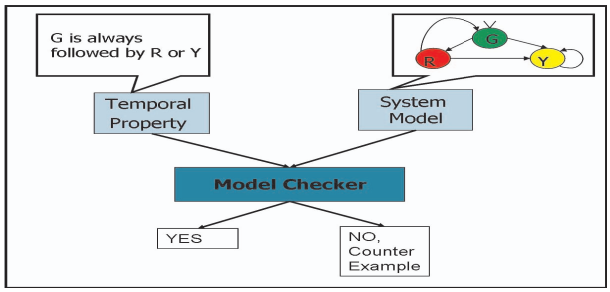

Figure 1. Model checking process.

checking or to supplement existing instructional material on temporal logic. Section 2 of the paper presents an overview of essential concepts: model checking, Linear Temporal Logic (LTL), and SPS. Section 3 discusses a teaching component and supports it with a brief description of lessons and exercises. The paper ends with a summary and discussion of future work.

2 Background

2.1 Formal Verification: 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 [7].

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 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, 7]. 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 Formal Specifications: Linear Temporal Logic

Linear Temporal Logic (LTL) [10] is a prominent formal specification language that is highly expressive and widely used in formal verification tools such as the model checkers SPIN [7] and NuSMV [2]. LTL is also used in the runtime verification of Java programs [8].

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 $s = 0, 1, 2, \ldots$ 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 $s' \geq s$.

One problem with LTL is that, when specifying software properties, the resulting LTL expressions can become difficult to write and understand. For example, consider the following LTL specification: $\Box(a \rightarrow \diamond(p \wedge \diamond((\neg a) \wedge \neg p)))$, where a denotes "Train approaches the station." and p denotes "Train passes the station." It is not immediately obvious that this specification states: "If a train approaches the station, then the train will pass the station and, after it passes, the train does not approach or pass the station."

2.3 Specification Pattern System: Patterns and Scopes

The Specification Pattern System (SPS) [5] provides patterns and scopes to assist the practitioner in formally specifying software properties. Dywer et al. [5] defined patterns and scopes 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 excerpts from the SPS website [14].

- $Absence$: 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.
- $Existence$: To describe a portion of a system's execution that contains an instance of certain events or states.
- $Precedence$: 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.
- $Response$: 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.

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 (see [5] for a detailed description of the scopes):

- $Global$: 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 the state immediately before the state in which proposition R first holds.

[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.

- $After\ Q$: The scope consists of the state in which proposition Q first holds and includes all the remaining states of program execution.
- $Between\ Q\ And\ R$: The scope consists of all intervals of states where the start of each interval is the state in which proposition Q holds and the end of the interval is the state immediately prior to one in which proposition R holds.
- $After\ Q\ Until\ R$: This scope is similar to the previous one except, if there is a state in which Q holds and proposition R does not hold, then the interval of the scope will include all states from and including the state where Q last holds until the end of program execution.

SPS is presented as a website [14] with links to descriptions of the patterns. The website provides a mapping of each pattern and scope combination into different formal specification languages. For example, the property "A request always triggers an acknowledgment, between the beginning of execution and system shutdown." can be described by the $S\ Responds\ to\ P$ pattern within the $Between\ Q\ and\ R$ scope, where S denotes "Acknowledgement is triggered.", P denotes "Request is made.", Q denotes "Execution begins.", and R denotes "System is shut down." The LTL formula for the pattern and scope combination as provided by the SPS website is:
$\Box((Q \wedge (\neg R) \wedge \diamond R) \rightarrow (P \rightarrow ((\neg R)U(S \wedge \neg R)))UR)$.

3 An Educational Component for Teaching Formal Specifications

3.1 Goals and Outcomes

The goal of the lessons described in this section 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 describes a technique and activities that support the attainment of these goals. The prerequisite for the educational component is a course in discrete mathematics in which the students specify properties using propositional logic. The component described here 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.

The lessons for Outcomes 1-1 and 2-1 are not given in this paper. The focus will be on the last two outcomes.

```
#define limit 20
byte i = 0;
active proctype seq( ) {
    do
      :: (i < limit) → i = i+1;
      :: (i == limit) → break;
    od; }
```

Figure 2. Promela Code of the Simple Model

3.2 Using SPS to Generate an LTL Formula

Following an introduction to the LTL language and a tutorial on SPS, the students will be given a hands-on exercise 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 [5].

Exercise 1: *For each of the properties below, use SPS 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) The OK button is enabled after the user enters correct data.
(b) A request always triggers reply between start of execution and system shutdown.
(c) No work will be scheduled before execution.

3.3 Using SPIN to Advance Understanding of Specifications

3.3.1 Overview of The New Technique

As opposed to using SPIN to test the correctness of the model, the technique described in this subsection uses a simple model written in Promela 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.

3.3.2 Steps in The New Technique

The Promela code consists of a do-loop that begins with the initial value of i set to zero, and terminates when i reaches a predefined value called $limit$. Setting $limit$ to the value 20 means that the model has a total of 20 states starting with the state i_0 and ending with the state i_{19}. The Promela code is given in Figure 2.

Test 1 : - - - - P - - R - - - - - - - - - - -	Test 2 : - - - - - R - - P - - R - - - - - - - -
Pattern: Existence of P	Pattern: Existence of P
Scope: Before R	Scope: Before R
Formula: $(\diamond R) \rightarrow ((\neg R)U(P \wedge \neg R))$	Formula: $(\diamond R) \rightarrow ((\neg R)U(P \wedge \neg R))$
P: (i == 5)	P: (i == 8)
R: (i == 8)	R: (i == 5 $\vee$ i == 11)
Expected Result: No violation	Expected Result: Violation

Figure 3. Sample Test Cases

The steps for applying the technique are as follows:
1) Insert the simple Promela model into the model checker.
2) Specify the LTL formula to be tested.
3) Use conditions to assign the states in which propositions from the LTL formula are set to true.

The Promela code remains the same in all test cases. In the third step above, the values of the propositions are set by assigning each proposition a truth value based on the variable i in the Promela model. For example, in Test 1 shown in Figure 3, P is true in the fifth state, and R is true in the eighth state. To assert P in the fifth state, we set P to be the truth value of the condition (i == 4) (note that the model starts with i set to zero). Similarly, we define R by the truth value of the condition (i == 7). Note that for this test case, P and R are only true in the fifth and eighth states respectively. However, it is possible to specify for a proposition to be true in more than one state using the "$\vee$" "or" operator as is the case for R in Test 2 in Figure 3.

A more detailed list of test suites for the patterns and scope is available in [13]. The test suite consists of a set of test cases for each scope, where a test case is a set of assignments for the propositions in the LTL formula under test, such that a particular trace of computation is created. In the following trace of computation: "- - - Q - - P - - - - R - - - - - - - -"
Q is true in the fourth state, P is true in the seventh state, and R is true in the twelfth state. Note that each character in the string represents a state, and a dash (-) implies that none of the propositions is $true$ at that state. A letter symbol, e.g., Q, P, and R in this example, denotes that the proposition is $true$ in the designated state. Displaying more than one letter between parentheses implies that the propositions represented by the letters are valid at that state. For instance, (PQ) denotes that P and Q both hold in the same state. It is worth mentioning that the test cases were selected using the equivalence classes and boundary value analysis testing strategies based on the patterns and scopes.

Specification of the LTL formula to be analyzed and the assignments of conditions to propositions are done in the XSPIN's LTL Property Manager. XSPIN is the graphical user interface to SPIN. A detailed tutorial with screen shots of using XSPIN with our approach is available at [13].

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 (i.e., define P (i == 0)) and test the formula $\diamond P$ and examine SPIN's output.

3.3.3 Exercises

The lessons described here follow a tutorial in which students are introduced to SPIN. Students will have used the tool to check the correctness of simple models. In the exercises, students apply

the new SPIN technique.

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. Instructions for Exercise 2 follow:

Consider the Response-Global pattern and scope combination and the following LTL formula: $\Box(P \rightarrow \diamond S)$. *Use SPIN to run the traces of computation given below against the specified LTL formula to answer the following questions: (1) Does the Response property guarantee the occurrence of effect* (S) *in all cases? Explain your answer. (2) In the response property, can the cause* (P) *and effect* (S) *hold at the same state? Explain your answer.*

(1) $- - - - - - - -$
(2) $- - - - - - - P$
(3) $- - - - - - - S$
(4) $- - -(PS) - -$

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 subtle properties of LTL are explained. This exercise also targets Outcome 3.2. Instructions for Exercise 3 follow:

Consider the following property: "When a connection is made to the SMTP server, all queued messages in the OutBox mail will be transferred to the server", and it representative LTL formula:

$$(\Box\neg R) \vee ((\neg R)U(P \wedge \neg R))$$

For each trace of computation given below, state the expected result of running SPIN. Check your results by running SPIN as shown in class to verify your predicted results. Finally, state whether the LTL specification matches the natural language property. Explain your answer.

(1) $- - - - - - - - - - - - -$
(2) $- - - - (PR) - - - - - - -$
(3) $- - - - PR - - - - --$
(4) $- - - - RP - - - - --$
(5) $- - - - - - - - - - - - R$
(6) $R - - - - - - - - - - - -$
(7) $- - R - -P - -R - -$
(8) $- - - - P - - - - - - -$

Exercise 4 After completing Exercises 1-3 and other assignments determined by the instructor, 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 their ability to satisfy Outcome 3.1. Instructions for Exercise 4 follow:

For each of the properties given below, specify the corresponding LTL formula. In addition, define a minimal set of traces of computations to validate your generated LTL formula. Use SPIN to test each trace of computation against your generated formula. The Properties are:

(1) *When a connection is made to the SMTP server, all queued messages in the OutBox mail will be transferred to the server.*

(2) *When the name of a mailbox is double-clicked, the mailbox will be opened.*
(3) *The OK button on the login window is enabled as soon as the login window is first displayed to the user.*

4 Summary and Future Work

One of the issues in employing formal verification techniques is the difficulty in writing, reading, and understanding formal specifications. Although there exists tool support for automated generation of formal specifications, there is still a need for software engineers to understand the subtleties of formal specifications and to gain experience in formal methods, especially for those working with high-assurance systems.

This paper describes a technique that assists in the understanding of LTL through the use of a model checker. The technique uses the SPIN model checker to elucidate the behavior of an LTL formula and promote understanding by analyzing the property (represented by the formula), test case (that represents a pre-defined trace of computation), and anticipated and actual result. The paper presents educational outcomes and exercises to support using the technique. Although the technique is being used with the SPIN model checker, it can easily be adopted to the NuSMV model checker and the Computational Tree Logic (CTL) [3].

Future work includes development of a website to support the use of the novel model checking technique presented in the paper. In addition, we are in the process of updating the Prospec tool to automatically generate LTL formulas and their corresponding traces of computations. Prospec extends SPS by specifying composite propositions, which are classifications for defining sequential and concurrent behavior to describe pattern and scope parameters [11].

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 Verifier", *11th Int'l Conference on Computer Aided Verification, CAV*, Trento, Italy, July 1999, pp. 495-499.

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

[4] Cheon Y., Leavens G. T., "A Simple and Practical Approach to Unit Testing: The JML and JUnit Way", *European Conference on Object-Oriented Programming, ECOOP*, Malaga, Spain, June 2002.

[5] Dwyer, M.B., Avrunin, G.S., and Corbett, J.C., "Patterns in property specifications for finite-state verification," *Int. Conf. on Software Engineering, ICSE*, Los Angeles, CA, May, 1999, pp. 411–420.

[6] Hall, A., "Seven Myths of Formal Methods," *IEEE Software*, September 1990, 11(8).

[7] Holzmann, G.J., *The SPIN Model Checker,* Addison-Wesley, 2004.

[8] Havelund, K., and Pressburger, T., "Model Checking Java Programs using Java PathFinder", *Int'l J. on Software Tools for Technology Transfer*, 2(4), 2000.

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

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

[11] Mondragon, O., and Gates, A.Q., "Supporting Elicitation and Specification of Software Properties through Patterns and Composite Propositions," *Int'l J. Software Engineering and Knowledge Engineering,* 14(1), February, 2004.

[12] Smith, R., Avrunin, G., Clarke, L., and Osterweil, L., "PROPEL: an approach supporting property elucidation" *22rd Int'l Conf. on Software Engineering, ICSE*, May 2002, Orlando, Florida, pp. 11-21.

[13] Salamah, S., "Supporting Documentation for the SPS-Prospec Case Study," UTEP-CS-05-14, the University of Texas at El Paso, April 2005.

[14] Spec Patterns, http://patterns.projects.cis.ksu.edu/, November 2007.