An Approach to Enhance Students' Competency in Software Verification Techniques

Omar Ochoa, Salamah Salamah
Department of Computer Science
University of Texas at El Paso
El Paso, Texas, USA
{omar, isalamah}@utep.edu

***Abstract*— In this paper we present an approach used to enhance students' competency in software verification. Students were asked to apply software verification techniques to a complex formal specification system. The complexity of the system stems from its sophisticated requirements. Selecting such system for this study was intentional for the following two reasons 1) the system is difficult to understand and analyze because of the domain knowledge required to generate formal specifications in temporal logic and 2) the system is large and complex which lends itself to a wide range of applicable verification techniques, and thus highlights the differences in the capabilities of each of the software verification approaches. Students were assessed using multiple criteria including; examination in applying learned techniques, students' attitude toward the technique, perceived efficiency of the techniques in discovering software defects, and the ability of the technique to locate errors in the code beyond simply indicating their presence. The results of this work show that the students applied the learned techniques successfully and their attitudes towards software verification improved.**

Keywords—Software Engineering, Computer Science Education, Debugging, Software Testing

I. Introduction

As our dependency on computer-based critical systems continues to increase, software systems will continue to grow in complexity and size. It has been reported that a significant number of software systems do not meet users' requirements and are of poor quality [13]. In addition a study from the National Institute of Standards and Technology revealed that software defects are so prevalent and detrimental that they cost the U.S. economy an estimated $59.5 billion annually, or about 0.6 percent of the gross domestic product [14]. The estimated cost due to software errors in the aerospace industry alone was $6 billion in 1999 [15]. Thus it is increasingly more important that efforts in software development focus on decreasing the number of defects within developed software systems. However, studies have found that senior-level college students and recent graduates lack adequate preparation in software verification, and have difficulty in testing software thoroughly [16, 17, 18, 19]. Therefore, there is an increasing demand for software professionals who have a strong background in software verification.

This paper presents the results of an approach in enhancing students' competency in software verification. Students are asked to apply the techniques of interest to verify a system used to generate formal specification [7, 9]. The complexity of the system stems from its sophisticated requirements (specified using first order logic). The choice of such system was intentional for the following two reasons 1) the system is difficult to understand and analyze because of the domain knowledge required to generate formal specifications in temporal logic and 2) the system is large and complex which lends itself to a wide range of applicable verification techniques emphasizing the differences in the capabilities of each of the software verification approaches. Students were assessed using multiple criteria including; examination in applying learned techniques, students' attitude toward the technique, perceived efficiency of the techniques in discovering software defects, and the ability of the technique to locate errors in the code beyond simply indicating their presence.

The hands-on approach used in this work aims at exposing the students to the subtleties of each technique by allowing them to explore and learn by practice, and thus, gain expertise in software verification. This paper presents the results of the approach, focusing on the increased proficiency of the student's ability to leverage these software verification techniques in finding software defects and in the applicability of the chosen system in teaching these techniques..

II. Background

One of the key aspects in software development is the assurance that the software behaves correctly, i.e., as specified by requirements. The most common software verification technique is that of testing. Software testing is the process of creating and applying test cases to a software system in order to check if the system's behavior was correctly implemented. A test must be efficient in such way that it either exposes a defect or gives some level of confidence on the absence of defects. The problem of determining efficient inputs for test cases has been tackled by different approaches, ranging from treating the system as a black-box [20] to white-box approaches [21].

A. Black-box Testing

In a black-box testing approach, which focuses on selecting input for test cases from the specified requirements and input ranges, it is easier to determine inputs for a test case, but a failed test case will not tell us the potential location of the defect in the system. Our approach focuses on three black-box testing techniques:

Equivalence Class Partitioning. Equivalence class testing is a technique in which the input domain is partitioned into a finite number of valid and invalid classes. Input values within each equivalence class should behave the same. As such one test input selected from each equivalence class should be sufficient to determine the behavior of the whole equivalence class. By executing one test from each equivalence class, the amount of necessary testing is reduced.

Boundary Value Analysis. Boundary value analysis is a technique that selects tests from the boundary range of input values. These ranges can be defined from specifications or via equivalence classes. The test is selected from above, below and at the boundary of the range. Software defects are commonly encountered at these boundaries, and testing them is an efficient way to detect defects.

Pair-wise Testing. Software systems of any significance are, typically, too large to be tested for all possible input combinations. For these software systems, pair-wise testing [22] is useful because it focuses on testing all pairs of test case inputs instead of all combinations of inputs. The generation of test case input under pair-wise testing is mechanical in nature, only ensuring that each pair of input variables is accounted for from an orthogonal array of inputs. This leads to a automatic way of generating test case input data for the testing of a wide range of large systems.

B. *White-box Testing*

A white-box testing approach requires the analysis of the internal structure of a system in order to select sets of test input cases that provide adequate coverage of the source code. Determining test input data in white-box testing can be more difficult since it requires construction of control flow graphs (CFG) from source code to determine what input must be used to meet the desired coverage criteria. Because the executions paths are directly tied to the test case input, a failed test can provide the location of the defect.

Branch Coverage. A branch coverage criterion is one that requires that each branch of every control structure has been touched at least once by the testing. Test case input is selected by analyzing the CFG and determining what branches need to be tested. Test cases are created until every branch has been touched at least once. When every branch has been touched at least once, branch coverage reaches 100%.

Path Coverage. A path coverage criterion requires that every possible path is executed by testing at least once. In order to determine test case input, an analysis of the CFG must be done to generate test input data for every possible path that can be executed. This coverage criterion is deemed the most expensive type of coverage as the number of tests required grows exponentially with the number of conditions in the source code.

Def-use Coverage. A def-use coverage criteria focuses on generating tests that cover paths through the definition and use of variables in a program. The def-use requires that testing covers every path from every variable definition to every use of that variable. This is done by examining the paths stemming from where a variable is first defined, down to every single use of that definition of the variable.

III. PROPERTY SPECIFICATION (PROSPEC) TOOL

A. *Formal Specification Languages*

Software specifications refer to properties that the system must adhere to. Specifications can be defined and used at the different stages of software development and can range in formality from completely informal (i.e., natural language) to completely formal (i.e., mathematical description). Informal specifications provide a simple mean by which stakeholders, who are, usually, not immersed in logic, can easily understand the desired system properties. On the other hand, formal specifications are more succinct and unambiguous. More importantly, formal specifications allow for the use of formal verification techniques such as model checking [1] and runtime monitoring [2].

There are numerous formal languages that can be used to express system behavior, e.g., first order logic, the specification languages Z and VDM, which have been used in the specification of software and hardware systems.

In the case of dynamic software systems, temporal logic has been used for many decades. In order to be able to describe truth in different moments of time, researchers have designated extensions of traditional logic called temporal logics. Linear Temporal Logic (LTL) [3] and Computation Tree Logic (CTL) [4] have been commonly used by formal verification techniques, especially model checking.

Formulas in LTL are constructed from elementary propositions (i.e., statements with truth value describing system conditions or events), and the usual Boolean operators for not, and, or, imply (!, &, |, and →, respectively). In addition, LTL provides the temporal operators *next* (X), *eventually* (F), *always* (G), *until*, (U), *weak until* (W), and *release* (R). These formulas assume discrete time, i.e., state s may be denoted as 0, or 1, or 2, or … The meaning of the temporal operators is straightforward [3]:

- The formula Xp holds at state s if p holds at the next state $s + 1$,
- the formula $p\ U\ q$ holds 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 $F\ p$ holds at state s if p is true at some state $s' \geq s$, and
- the formula $G\ p$ holds at state s if p is true at all states $s' \geq s$.

B. *Automated Support for Generating Formal Specifications*

Specification Patterns System (SPS). A major reason for the lack of application of formal methods in software development is the difficulty in writing, reading, and validating formal specification, particularly those involving time. To assist users in the generation of formal specifications, Dwyer et al. [5, 6] developed the Specification Pattern System (SPS). The work defined a set of patterns to represent the most commonly used formal properties and to guide the practitioner through the specification process. *Patterns* capture the expertise of developers by describing solutions to recurrent problems. SPS also defined a set of scopes of system execution where the pattern of interest must hold. Each pattern and scope combination can be mapped to specifications in multiple formal languages including LTL and CTL. Using the notions of patterns and scopes a user can define formal system properties without being an expert in the language.

The main patterns defined by SPS are: *Universality*, *Absence*, *Existence*, *Precedence*, and *Response*. The descriptions given below are taken verbatim from the SPS website [S6]:

- *Absence*(*P*): To describe a portion of a system's execution that is free of event or state (P).
- *Universality*(*P*): To describe a portion of a system's execution which contains only states that have the desired property (P). Also known as Henceforth and Always.
- *Existence*(*P*): To describe a portion of a system's execution that contains an instance of certain events or states (P). Also known as Eventually.
- *Precedence*(*P*, *Q*): To describe relationships between a pair of events/states where the occurrence of the first (Q) is a necessary pre-condition for an occurrence of the second (P). It can be said that an occurrence of the second is enabled by an occurrence of the first.
- *Strict Precedence*(*P*, *Q*): To describe relationships between a pair of events/states where the occurrence of the first (Q) is a necessary pre-condition for an occurrence of the second (P). This pattern differs from the previous one in that in regular precedence the pattern is satisfied if both *P* and Q hold in the same state, while the *Strict Precedence* is only satisfied when *Q* holds in a state that *strictly precedes* the one in which *P* holds.
- *Response*(*P*, *Q*): To describe cause-effect relationships between a pair of events/states. An occurrence of the first (P), the cause, must be followed by an occurrence of the second (Q), 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:

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

SPS is presented as a website [6] 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 ***Q*** *Responds to* ***P*** pattern within the *Between* ***L*** *and* ***R*** scope, where *Q* denotes "Acknowledgement is triggered.", ***P*** denotes "Request is made.", ***L*** denotes "Execution begins.", and ***R*** denotes "System is shut down.". Based on the selected pattern and scope combination, the LTL formula provided by the SPS website [S6] is:

$$G((\mathbf{L} \ \& \ (!\ \mathbf{R}) \ \& \ F\ \mathbf{R}) \rightarrow (\mathbf{P} \rightarrow ((!\mathbf{R})\ U\ (\mathbf{Q}\ \&!\mathbf{R})))U\ \mathbf{R}).$$

Composite Propositions (CP). Composite propositions (CP) [7] expand the expressiveness of patterns and scopes to include the specification of sequential and concurrent behaviors. In practical applications, we often need to describe properties where one or more of the pattern or scope parameters are made of multiple (i.e., composite) propositions. An example of such property is the one for sending data by a client and storing the data by the server. The English description of the property is "every time data is sent at state s_i, data is read at state $s_j \geq s_i$, the data is processed at state $s_k \geq s_j$, and data is stored at state $s_l \geq s_k$." Although SPS provides a significant support for property specifications, specifying such a property using SPS would require a significant effort by the user. In order to arrive at the formal specification for this property, the user will have to manipulate the different propositions within the property in order to make it fit the available patterns and scopes. This extra effort by the user might lead to erroneous specifications or user frustration, or both.

Even with the introduction of SPS' patterns and scopes, defining formal specifications for concurrent and sequential properties similar to the above example remains a huge obstacle in adapting formal methods in software verification. To describe such properties, SPS was extended by introducing a classification for defining sequential and concurrent behavior to describe pattern and scope parameters. Specifically, the work [7] defined the following CP classes along with their English description:

- AtLeastOne$_C$: At least one of the propositions in the set of propositions holds.
- AtLeastOne$_E$: At least one of the propositions in the set of propositions becomes *true*.
- Parallel$_C$: All propositions in the set of propositions hold.

- Parallel$_E$: All propositions in the set of propositions become *true* simultaneously.
- Consecutive$_C$: Each proposition in the sequence of propositions is asserted to hold in a specified order, one at each successive state.
- Consecutive$_E$: Each proposition in the sequence propositions becomes *true* in a specified order, one at each successive state. Once they become *true*, their truth value in subsequent states does not matter.
- Eventually$_C$: Each proposition in the sequence of propositions is asserted to hold in a specified order and in distinct and possibly non-consecutive states.
- Eventually$_E$: Each proposition in the sequence of propositions becomes *true* in a specified order and in distinct and possibly non-consecutive states. Once they become *true*, their truth value in subsequent states does not matter.

The subscripts C and E describe whether the propositions within a CP class are asserted as conditions or events respectively. A proposition defined as a condition holds in one or more consecutive states. A proposition defined as event means that there is an instant at which the proposition changes truth value in two consecutive states. [7] Provides the LTL semantics for each of the CP classes.

Combining patterns and scopes with this notion of CP makes the formal specification of the sending/storing property above much easier. This property can be described using the *Existence of P* pattern within the *Between L and R* scope where *L* is defined with the proposition signifying "data is sent," *R* is defined by the proposition "date is stored," and *P* is a CP of type Consecutive$_C$ composed of the two propositions p_1 and p_2 (data is read and data is processed, respectively).

One of the advantages of utilizing CPs in generating formal specifications is that it forces the user to consider the different possible relations among proposition within a property. This leads to more complete and correct specifications of sequential and concurrent behaviors which are characteristics of reactive systems.

Combining patterns, scopes and CPs. Although SPS provides LTL formulas for basic patterns and scopes (ones that use single, "atomic", propositions to define L, R, P, and Q) and Mondragon et al. [7] provided LTL semantics for the CP classes described above., in most cases it is not adequate to simply substitute the LTL description of the CP class into the basic LTL formula for the pattern and scope combination. Consider the following property: "The delete button is enabled in the main window only if the user is logged in as administrator and the main window is invoked by selecting it from the Admin menu". This property can be described using the Existence (Eventual$_C$(p_1, p_2)) Before(r) where p_1 is "the user logged in as an admin", p_2 is "the main window is invoked", and r is "the delete button is enabled". As mentioned above, the LTL formula for the Existence(p) Before(r) is [S6] "(G !r) | (! r U (p & !r))", and the LTL formula for the CP class Eventual$_C$, as described in [7], is (p_1 & X(!p_2 U p_2)). By replacing P by (p_1 & X(!p_2 U p_2)) in the formula for the pattern and scope, we get the formula: "(G !r) | (!r U ((p_1 & X(!p_2 U p_2)) & !r))". This formula however, asserts that either *r* never holds or *r* holds after the formula (p_1 & $X(!p_2\ U\ p_2)$) becomes true. In other words, the formula asserts that it is an acceptable behavior if *r* ("the delete button is enabled") holds after p_1 ("the user logged in as an admin") holds and before p_2 ("the main window is invoked") holds, which should not be an acceptable behavior.

As seen by the above example, the temporal nature of LTL and its operators means that direct substitution could lead to the description of behaviors that do not match the actual intent of the user. For this reason, it is necessary to provide abstract LTL formulas that can be used as templates for the generation of LTL specifications for all combinations of patterns, scopes, and CP classes. To address this issue, Salamah et al [8] defined a set of templates that can be used to ease the generation of LTL specifications for all pattern, scope, and CP combinations. The work defined nine templates to generate formulas within the *global* scope, 14 for formulas within the *before R* scope, and another five templates to generate the formulas within the *between L and R* and the *after L until R* scopes. The work in [8] also showed how formal proofs and software inspections were used to validate the correctness of the defined templates.

Using the templates defined by [8] allows for the generation of over 31,000 types of properties as described in the breakdown shown in Table 1.

The numbers for each combination of pattern/scope are results of number of possible ways to combine the different representation of P, Q, L, and R. For example, in the *Absence* of *P* pattern within the *Before R* scope, there are eight possible ways of presenting each of *P* and *R*, and as result there are 8x8 (64) combinations. On the other hand, the number of combinations for *Q Responds to P* within *Between L and R* is calculated as 8x8x8x8 to yield 4096 combinations.

Table 1. Breakdown of Property Types

Pattern/ Scope	*Global*	*Before R*	*After L*	*Between L and R*	*After L Until R*	*Total*
Absence	8	64	64	512	512	1160
Existence	8	64	64	512	512	1160
Universalit y	8	64	64	512	512	1160
Precedence	64	512	512	4096	4096	9280
Strict Precedence	64	512	512	4096	4096	9280
Response	64	512	512	4096	4096	9280
Total	216	1728	1728	13824	13824	31320

Prospec Tool. The Property Specification (Prospec) [7, 9] is a prototype tool with a graphical user interface developed at the University of Texas at El Paso to help software developers create software specifications using SPS' patterns and scopes as well as the aforementioned notion of composite propositions (CP). Prospec generates formal specification in Future Interval Logic (FIL) [10] and LTL. The tool guides practitioners in the creation of formal specifications by assisting the user in the identification and elucidation of events or conditions that are used to specify behavior. Prospec makes use of decision trees to guide the user through a series of decisions to select an appropriate pattern, scope, and CP classes to match his/her property.

C. *Using Prospec in Teaching Software Testing*

Considering the complexity associated with generating a large number of LTL specifications using the Prospec tool, we envision this as a perfect platform to teach software testing. The applicability of the Prospec tool stems from multiple factors:

1. The complexity of the LTL specifications generated by the tool requires significant domain knowledge in order for students to adequately test the generated specifications. This is typical of a real-world environment in which software quality assurance personnel must gain domain expertise in the system under test (SUT) before attempting to test such system.
2. The large number of LTL specifications (over 31,000 possible specifications) supported by the tool forces students to employ adequate testing techniques to select sets of test cases that provides appropriate coverage of the system.
3. The characteristics of patterns, scopes, and CPs considerably lend themselves to testing techniques such as equivalence classes and boundary testing. For example, in order to generate black-box test cases, students have to break the input domain into different possibilities, such as, a) the pattern holds but the scope does not, b) the scope holds, but the pattern does not, among many other cases.
4. The fact that the domain associated with the Prospec tool is that of formal specifications is an attractive one as it provides a way to introduce students to formalisms and formal specifications as part of formal methods within software development.

IV. Approach for Using the Propsec Tool in Teaching Software Testing

The Prospec tool and the associated patterns, scopes, and CPs were introduced as a semester project for the CS4387/5387 (Software Integration and V&V) course at the University of Texas at El Paso (UTEP) in Spring 2015. The course is a core requirement for students in the Masters of Science in Software Engineering (MSSwE) program at UTEP. However, the course also attracts students from other programs offered by the Computer Science Department.

A. *Students' Profile and Team Structure.*

The current addition of the course has a total of 22 students. Of those students one is a PhD. candidate in Computer Science, another is pursuing a Master's in Computer Science, two others are undergraduate students in Computer Students, and the remaining 18 students are MSSwE students.

Students in the class were assigned into five teams of four to five students. The course teams work on a semester long project where they develop a complete test plan for a particular project. The test plan includes:

- Plans to test the system's requirements and design specifications as well as implementation using black-box testing techniques,
- Plans to perform white-box testing on system implementation, and
- Plans for system integration, and
-

B. *Team Assignments*

Using our approach, student teams were assigned three major assignments to verify correctness of LTL specifications generated by the Prospec tool. Below, we discuss each of these assignments, along with students' expectation, and actual students' performance in each of these assignments.

Using Pair-wise Testing. Each of the five teams in the class was assigned a pattern/scope combination where all four propositions are present. Namely, each team was assigned one of the following pattern/scope combinations:

- Q Responds to P, Between L and R
- Q Responds to P, After L Until R
- Q Precedes to P, Between L and R
- Q Precedes to P, After L Until R
- Q Strictly Precedes to P, Between L and R

According to Table 1, each one of these combinations yields a total of 4,096 possible LTL specifications. Because of the large number of formulas to consider when testing the system, students were introduced to the technique of pair-wise testing in which possible values of each pair of input are combined to generate a single test case. The students were asked to use pair-wise testing to reduce the number of candidate formulas to be tested under each combination.

While using pair-wise testing is part of the outcomes of this course, students were not required to manually generate the orthogonal arrays as part of pair-wise testing. Instead, students were asked to research tools that automatically generate pair-wise tests. All teams used the web tool Hexawise [11]. Using the Hexawise tool, each team generated a list of 78 formulas to consider for testing of their particular pattern/scope combination. The teams were also asked to verify that each pair of CPs representing the different proposition (*P, Q, L, and R*) is present in the 78 formulas suggested by Hexawise.

Black-box Testing. Once the teams generated a list of formulas to test using pair-wise testing, the generated formulas were divided among team members to test using black-box testing. As a result, each student was responsible for testing no more than 20 formulas.

Students were asked to use equivalence classes and boundary value testing techniques in testing each of their formulas. To start, the team as a whole was required to develop equivalence class partitioning for the general pattern/scope combination (without the introduction of CPs).

Considering the patterns and scope characteristics described previously, we can find a perfect symmetry with the conditions emphasized in equivalence class partitioning and testing boundary values. For example, scopes formally define the states of interest within system execution (i.e., define boundaries where a pattern is to hold). As a result, scopes characteristics provide sufficient examples to explain what constitutes a boundary value. For example, using the scope Before R, one can test whether the pattern of interest is upheld if that pattern holds immediately before the state where R holds, at the same state as R, and at the state immediately after R. In a similar fashion, patterns can be used to better understand equivalence class testing. For example, using the Q Responds to P pattern, we can think of multiple equivalence classes to choose tests from:

- P never holds, and Q holds,
- P and Q never hold,
- P holds and Q never holds,
- P holds and Q holds after P,
- P and Q hold in the same state,
- …

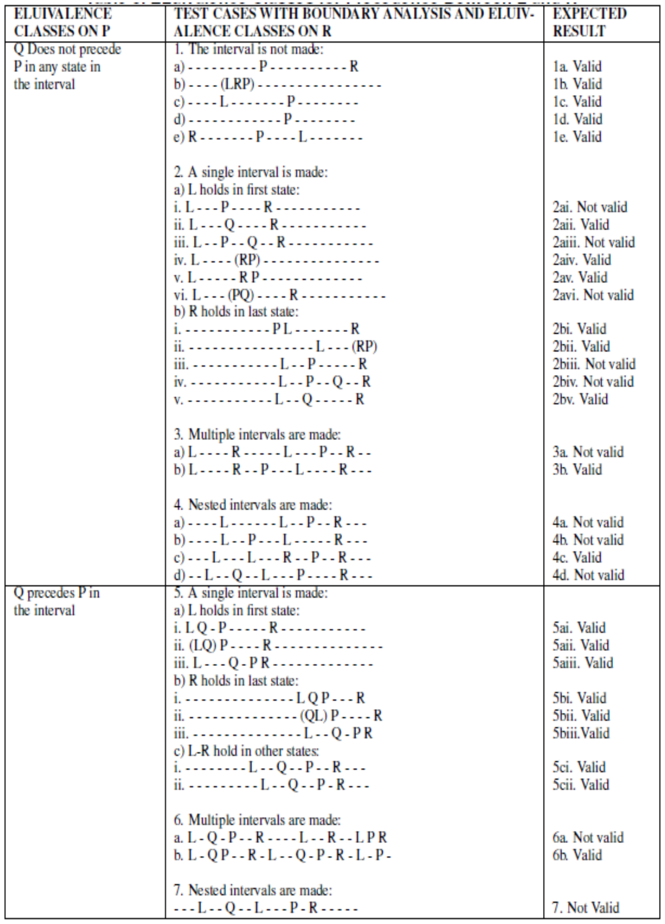
Figure 1 shows the equivalence class partitioning and boundary testing of the Q Precedes P pattern within the Between L and R scope as produced by the designated team of students for that pattern/scope combination. The teams were required to provide multiple drafts of their equivalence partitioning and boundary test cases. The provided figure is a sample of the final breakdown of test cases after consultation with the teaching team (both authors). In Figure 1 the students used the notion of Traces of Computation to describe test cases and the behaviors accepted or rejected by a pattern and scope combinations. A trace of computation is a string representing a sequence of states that depicts the propositions that hold in each state. Each character in the string represents a state and a dash (-) implies that no proposition is true at that state. A letter symbol, e.g., P, Q, L, and R, 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 example, in the trace of computation "- - L - - R - - (QP) - -", L is true in the third state, R is true in the sixth state, and both Q and P are true in the ninth states.

Once the candidate tests for each patterns/scope combination was agreed on, the students were then asked to provide equivalence partitioning of CP classes. For all CP classes the possible inputs fall into one of two cases; the CP holds or the CP does not hold. For example, in the case where P is of type EventualC (p1, p2, p3), possible equivalence classes are as follows (using traces of computations):

- P Holds:
  - p1p2p3
  - p1-p2p3
  - p1-p2-p3
  - p1p2--p3
  - …
- P Does not hold
  - p1-p2---
  - p2-p1-p3
  - p2-p3
  - ….

Once both the equivalence partitioning of the simple pattern/scope combination and those for each of the CP classes was finalized, students were required to translate their original test cases (similar to those in Figure 1) into ones that consider CP classes. For example, the first test cases in Table 1 can be translated into the following case when both P and R are of type EventualC: "------p1p2---p3------------r1r2r3".

All the pattern/scope combinations had known defects to the teaching team but not the students. As is the case with any software system of significance, there are defects that are not obvious even for the original developers of the system. In the case of the Prospec tool, it is assumed that such unknown defects are present in the implementation. The expectations for students' teams were that they will discover the majority of these defects. The teams discovered defects at the following rate:



ELUIVALENCE CLASSES ON P	TEST CASES WITH BOUNDARY ANALYSIS AND ELUIV-ALENCE CLASSES ON R	EXPECTED RESULT
Q Does not precede P in any state in the interval	1. The interval is not made:	
	a) - - - - - - - - - P - - - - - - - - - - R	1a. Valid
	b) - - - - (LRP) - - - - - - - - - - - - - - - -	1b. Valid
	c) - - - - L - - - - - - - P - - - - - - - -	1c. Valid
	d) - - - - - - - - - - - - P - - - - - - - -	1d. Valid
	e) R - - - - - - - P - - - - L - - - - - - -	1e. Valid
	2. A single interval is made:	
	a) L holds in first state:	
	i. L - - - P - - - - R - - - - - - - - - - -	2ai. Not valid
	ii. L - - - Q - - - - R - - - - - - - - - - -	2aii. Valid
	iii. L - - P - - Q - - R - - - - - - - - - - -	2aiii. Not valid
	iv. L - - - - (RP) - - - - - - - - - - - - - - -	2aiv. Valid
	v. L - - - - - R P - - - - - - - - - - - - -	2av. Valid
	vi. L - - - (PQ) - - - - R - - - - - - - - - - -	2avi. Not valid
	b) R holds in last state:	
	i. - - - - - - - - - - - P L - - - - - - - R	2bi. Valid
	ii. - - - - - - - - - - - - - - - - L - - - (RP)	2bii. Valid
	iii. - - - - - - - - - - - L - - P - - - - - R	2biii. Not valid
	iv. - - - - - - - - - - - L - - P - - Q - - R	2biv. Not valid
	v. - - - - - - - - - - - L - - Q - - - - - R	2bv. Valid
	3. Multiple intervals are made:	
	a) L - - - - R - - - - - L - - - P - - R - -	3a. Not valid
	b) L - - - - R - - P - - - L - - - - R - - -	3b. Valid
	4. Nested intervals are made:	
	a) - - - - L - - - - - - L - - P - - R - - -	4a. Not valid
	b) - - - - L - - P - - - L - - - - - R - - -	4b. Not valid
	c) - - - L - - - L - - - R - - P - - R - - -	4c. Valid
	d) - - L - - Q - - L - - - P - - - - R - - -	4d. Not valid
Q precedes P in the interval	5. A single interval is made:	
	a) L holds in first state:	
	i. L Q - P - - - - - R - - - - - - - - - - -	5ai. Valid
	ii. (LQ) P - - - - R - - - - - - - - - - - - - -	5aii. Valid
	iii. L - - - Q - P R - - - - - - - - - - - - -	5aiii. Valid
	b) R holds in last state:	
	i. - - - - - - - - - - - - - - L Q P - - - R	5bi. Valid
	ii. - - - - - - - - - - - - - - (QL) P - - - - R	5bii. Valid
	iii. - - - - - - - - - - - - - - L - - Q - P R	5biii.Valid
	c) L-R hold in other states:	
	i. - - - - - - - - L - - Q - - P - - R - - -	5ci. Valid
	ii. - - - - - - - - - L - - Q - - P - R - - -	5cii. Valid
	6. Multiple intervals are made:	
	a. L - Q - P - - R - - - - L - - R - - L P R	6a. Not valid
	b. L - Q P - - R - L - - Q - P - R - L - P -	6b. Valid
	7. Nested intervals are made:	
	- - - L - - Q - - L - - - P - R - - - - -	7. Not Valid

Figure 1. Equivalence Partitioning and Boundary Analysis for the Precedence Between L and R Combination

- Two out of two known defects in *Response Between L and R*
- *Two out of three* defects in *Response After L Until R*
- Two out of two defects in *Precedence Between L and R*

- One out of two defects in *Precedence After L Until R*
- Two out of three defects in *Strict Precedence Between L and R*

White-box Testing. Through course lectures, the students were introduced to the notion of control flow graphs (CFGs) and the different types of white-box testing techniques and coverages such as *statement, branch, path,* and *def-use* coverages. The students were also asked to research tools that automatically generate CFGs for a particular implementation.

Using black-box testing techniques, student teams are asked to *discover the presence* of defects. The next phase of our approach is to require students to employ white-box testing techniques to *locate* defects within the Prospec implementation. For this purpose we have packaged two implementations of the *Response Between L and R* combination. Both implementations have representations of the two known defects within this combination. Both implementations were in Java. The reason for choosing two implementations was to make sure students do not share results with each other. The teams were asked to conduct white-box testing at the unit level, where a method is considered a unit. Teams were also asked to conduct white-box testing to ensure branch (edge) coverage.

While teams used different pattern/scope combination to conduct black-box testing, the choice to have all teams work on the same combination for white-box testing was two-folds; 1) We believe that the implementation for the *Response Between L and R* combination is the most straightforward one of all the other combinations. It is also one that we have had the most experience working with [12], 2) we feel that giving the teams the same combination allowed us to have the teams compete against one another to discover code defects, and as such, injecting some excitement into the class assignments.

The results of students' teams testing were as follows:

- Two teams located both known code defects, and provided the correct suggestions to fix the code,
- One team located both known defects and provided the correct suggestions to fix the code. In addition, this team located an extra defect within the implementation. This defect, however, was not domain-specific as it was I/O-related.
- Each of the remaining two teams located one of the two defects in the code, and correctly suggested a way to remove the defect.

V. ASSESSMENT

In this section we shed some light on the impact of using this described approach on students' experiences in learning the different testing techniques described in the paper. In doing so we assess students' abilities in using the different verification techniques, and we compare the performances of the students involved in this semester's course with those of previous semesters. We also, report on a survey conducted at the end of the semester which was meant to gauge students' feelings toward the semester-long project and the associated assignments and material.

A. *Students' Performances Compared to Previous Semesters*

To assess students' learning and mastery of the testing approaches discussed in this paper, we compare students' performances against students in previous editions of the course. This comparison is in the shape of final exam questions related pair-wise, black-box, and white-box testing. Students' performance on the same set of exam questions from the previous two additions of the course will be compared to those the current students. Below, we describe the expectations for each type of questions used in the comparisons.

Questions Related to Pair-wise Testing. In this question, students are given a description of a system with a large number of discrete inputs and asked to 1) identify an appropriate testing strategy for testing the system and provide a justification for this choice, and 2) provide test cases for testing the system.

Successful answer for this particular problem consists of Identifying pair-wise testing as the appropriate strategy as well as providing a correct mapping of input variables to elements of an orthogonal array (the students are allowed to use tools such as Hexawise [11] for this purpose.). Table 2 below provides the results of students in the current semester (22 students in total) compared to those in spring semester of 2013 and spring semester of 2014 (total of 21 students).

Table 2. Students' Answers to Pair-wise Testing Questions

Question Element	**Percent of Students Correctly answering the question**	
	Springs 13 & 14	**Spring 15**
Correctly identifying pair-wise testing as the testing strategy	90	95
Correctly mapping input variables into an orthogonal array	86	95

Questions Related to Black-box Testing. In this question, students are given a description of a problem and asked to 1) break the input domain into appropriate set of equivalence classes and identify boundary conditions within each equivalence class, and 2) develop appropriate test cases for each identified equivalence class and boundary condition. Table 3 below provides the results of students in the current semester compared to those in spring semester of 2013 and spring semester of 2014.

Table 3. Students' Answers to Black-box Questions

Question Element	**Percent of Students Correctly answering the question**	
	Springs 13 & 14	**Spring 15**
Identify correct set of equivalence classes and boundary conditions	71	77
Identify correct test cases for each equivalence class and boundary condition. A test case consist of input and expected output	62	68

Questions Related to White-box Testing. In this question, students are given a piece of code and are asked to 1) manually develop a control flow graph (CFG) of the code, 2) identify test cases to ensure a) branch coverage, path coverage, and def-use coverage based on a particular variable. Table 4 below provides the results of students in the current semester compared to those in spring semester of 2013 and spring semester of 2014.

B. *Students' Assessment of the Described Approach*

In addition to assessing students' performance in the testing techniques discussed in this paper, we asked the students to complete an anonymous survey to gauge their feelings about the use of the described approach. The survey questions were meant to elicit students' attitude toward software testing both before and after completing the course.

Table 4. Students' Answers to White-box Questions

Question Element	Percent of Students Correctly answering the question	
	Springs 13 & 14	Spring 15
Correctly construct a CFG for the code	86	86
Identify correct set of test cases to ensure branch coverage	76	77
Identify correct set of test cases to ensure path coverage	71	69
Identify correct set of test cases to ensure def-use coverage	67	69

The students were also probed to provide their feelings towards the use of the Prospec tool material as a semester-long project in this course. Finally, the students were questioned regarding the applicability of the testing approaches in finding defects versus the ability to locate defects. In the following we provide the survey questions and Table 5 provides the summary of students' responses.

Question 1*:* My knowledge and awareness of the difficulties associated with software verification ***before taking this*** course were:

a)very high b)high c)average d)below average e)low

Question 2*:* My knowledge and awareness of the difficulties associated with software verification ***after taking this*** course are:

a)very high b)high c)average d)below average e)low

Question 3*:* The use of the semester project highlighted the subtle differences between the different verification techniques:

a)strongly agree b)agree c)neutral d)disagree e)strongly disagree

Question 4*:* Of the two major approaches, the one that is easier to use in generating test cases is:

a)white-box testing b) black-box testing

Question 5*:* Of the two major approaches, the one that is most effective at locating defects in software systems is:

a)white-box testing b) black-box testing

Table 5. Students' Answers to the Assessment of the Approach

	Number of Responses				
	a	**b**	**c**	**d**	**e**
Q1	0	6	8	7	1
Q2	8	9	4	1	0
Q3	5	9	5	0	2
	a			**B**	
Q4	13			9	
Q5	15			7	

Results show that overall the students performed slightly better than in previous semesters. However, this must be taken with a grain of salt due to small sample space and variances among the student's experiences and abilities. The general consensus among the student's responses is that using such hands-on approach yields a better appreciation for software verification.

VI. CONCLUSIONS AND FUTURE WORK

The use of this approach supports and enhances the teaching of software verification, such as black-box and white-box techniques, assisting in improving the education of software engineers. In particular, this approach requires significant domain knowledge in order for students to adequately test the system. Thus students are exposed to formal specifications domain knowledge which can serve them as they advance in their software engineering careers. This reflects a typical real-world environment in which software quality assurance personnel must gain domain expertise. One of the results of this work is that the Prospec tool is very appropriate for teaching classes in software engineering due to its complexity, large range of output formulas and need for high assurance.

A future goal of this work is to develop standalone teaching modules based on Prospec such that they can be used in courses that are not necessarily focused on testing, but can complement other software engineering topics. In addition, there is a need to conduct a more formal study to assess the applicability of this approach within other software quality assurance courses.

VII. ACKNOWLEDGMENTS

The authors would like to thank the students in the CS 4387/5387 (Software Integration and V&V) class. The students are listed alphabetically based on their last name:

1. Juan Angulo
2. Salah Atiyeh
3. Roberto Camacho
4. Arih Carrera
5. Julio Corral
6. Erick Ferreira
7. Roberto Fierro
8. Adrian Garcia
9. Alan Garcia
10. Luis Garnica
11. Rebekah Gruver
12. Yadira Jacquez
13. Hima Kondepati
14. Florencia Larsen

15. Eduardo Martinez
16. Troy McGarity
17. Jesus Medrano
18. David Reyes
19. Carols Romero
20. Oliver Stone
21. Jesus Tabares
22. John Vasquez

References

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

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

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

[4] Emerson, E, A., and E. M. Clarke, "Using Branching Time Temporal Logic to Synthesize Synchronization Skeletons," Science of Computer Programming, vol. 2, no. 3, pp. 241–266, 1982.

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

[6] Specification Patterns System, http://patterns.projects.cis.ksu.edu accessed April 2015

[7] Mondragon, O., A. Q. Gates, and S. Roach, "Prospec: Support for Elicitation and Formal Specification of Software Properties," Proceedings of 4th Runtime Verification Workshop, Barcelona, Spain, April 2004.

[8] Salamah, S., Gates, A., and Kreinovich, V., "Validated Templates for Specification of Complex LTL Specifications" in the Journal of Systems and Software, Volume 85 Issue 8, August, 2012

[9] Gallegos, I., Ochoa, O., Gates, A., Roach, S., Salamah, S., and C. Vela, "A Property Specification Tool for Generating Formal Specifications: Prospec 2.0," Proceedings of the Software Engineering and Knowledge Engineering Conference, San Francisco, CA, July 2008

[10] Ramakrishna, Y. S., Smith, P.M., Moser, L.E., Dillon, L.K., and G. Kutty, "Interval Logics and their Decision Procedures - Part I: An Interval Logic," Theoretical Computer Science, 166(1-2), Oct. 1996, pp. 1-47.

[11] HexaWise, https://www.hexawise.com accessed April 2015.

[12] Salamah, S., Ochoa, O., and Jacquez, Y., "Using Pairwise Testing to Verify Automatically-Generated Formal Specifications", Proceedings of the IEEE 16th International Conference on High Assurance Systems Engineering (HASE), Jan, 2015.

[13] Rubinstein, D., Standish Group Report: There's Less Development Chaos Today, Software Development Times, March 2007. http://www.sdtimes.com/article/story-20070301-01.html accessed April 2015.

[14] National Institute of Standards and Technology (NIST), June 2002, http://www.nist.gov/director/planning/upload/report02-3.pdf accessed April 2015.

[15] RTI, "The Economic Impacts of Inadequate Infrastructure for Software Testing," Health, Social, and Economic Research, Project No. 7007001, Research Triangle Park, NC.

[16] Carver, J.C. and N.A. Kraft, "Evaluating the Testing Ability of Senior-level Computer ScienceStudents," Proceedings of the 24th IEEE-CS Conference on Software Engineering Education and Training, Honolulu, Hawaii, May 2011.

[17] Astigarraga, T., Dow, E.M., Lara, C., Prewitt, R., and M.R. Ward, "The Emerging Role of Software Testing in Curricula," Transforming Engineering Education: Creating Interdisciplinary Skills for Complex Global Environments, pp. 1 – 26, April 2010.

[18] Garousi, V. and A. Mathur, "Current State of the Software Testing Education in North America Academia and Some Recommendations for the New Educators," Proceedings of the 22nd IEEECS Conference on Software Engineering Education and Training, Pittsburg, PA, March 2010.

[19] Bhattacherjee, V., Neogi, M.S., and R. Mahanti, "Are our Students Prepared for Testing Based Software Development?" Proceedings of the 22nd IEEE-CS Conference on Software Engineering Education and Training, Hyderabad, India, February 2009.

[20] B. Beizer, "Black-box testing: techniques for functional testing of software and systems", John Wiley & Sons, Inc., New York, NY, 1995

[21] B. Beizer, "Software testing techniques" (2nd ed.), Van Nostrand Reinhold Co., New York, NY, 1990

[22] J. Bach and P. Schroeder. "Pairwise testing: A best practice that isn't", in Proceedings of 22nd Pacific Northwest Software Quality Conference, pages 180–196, 2004.