November 2004

Specification-Based Testing with Linear Temporal Logic

Li Tan
University of Pennsylvania, tanli@saul.cis.upenn.edu

Oleg Sokolsky
University of Pennsylvania, sokolsky@cis.upenn.edu

Insup Lee
University of Pennsylvania, lee@cis.upenn.edu

Specification-Based Testing with Linear Temporal Logic

Abstract

This paper considers the specification-based testing in which the requirement is given in the linear temporal logic (LTL). The required LTL property must hold on all the executions of the system, which are often infinite in size and/or in length. The central piece of our framework is a property-coverage metric. Based on requirement mutation, the metric measures how well a property has been tested by a test suite. We define a coverage criterion based on the metric that selects a finite set of tests from all the possible executions of the system. We also discuss the technique of generating a test suite for specification testing by using the counterexample mechanism of a model checker. By exploiting the special structure of a generated test, we are able to reduce a test with infinite length to an equivalent one of finite length. Our framework provides a model-checking-assisted approach that generates a test suite that is finite in size and in length for testing linear temporal properties on an implementation.

Comments

Specification-based Testing with Linear Temporal Logic *

Li Tan Oleg Sokolsky Insup Lee

Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA, USA
{tanli, sokolsky, lee}@ saul.cis.upenn.edu

ABSTRACT

This paper considers the specification-based testing in which the requirement is given in the linear temporal logic (LTL). The required LTL property must hold on all the executions of the system, which are often infinite in size and/or in length. The central piece of our framework is a property-coverage metric. Based on requirement mutation, the metric measures how well a property has been tested by a test suite. We define a coverage criterion based on the metric that selects a finite set of tests from all the possible executions of the system. We also discuss the technique of generating a test suite for specification testing by using the counterexample mechanism of a model checker. By exploiting the special structure of a generated test, we are able to reduce a test with infinite length to an equivalent one of finite length. Our framework provides a model-checking-assisted approach that generates a test suite that is finite in size and in length for testing linear temporal properties on an implementation.

1. INTRODUCTION

Recent years observe an increasing demand on reliable software and hardware systems. Software engineering community response to such demands by introducing an array of new techniques into software development cycles. One of such examples is the use of formal methods, which facilities the precise formulation of the requirement and the formal proof of an system. A formal specification provides the precise description of the requirement that facilities the automatic verification techniques, for example, model checking, in which the system is checked algorithmically against the requirement encoded in a temporal logic. A consequence of the use of formula method is that high quality formal specifications become increasingly available. These high quality specifications are also a valuable asset to other elements in software development processes. As Stocks and Carrington found in their case study [11], "A formal software specification is (also) one of the most useful documents to have when testing software". Despite the major limitation of testing that it can only show the presence of error and never their absence [5], testing plays an indispensable role in developing reliable software and hardware systems. It can work where automatic verification stops short. For instance, it doesn't suffer from the state explosion problem which renders state-of-the-art model checkers intractable for even moderate real-world software applications, and testing can be applied to an implementation directly. An important paradigm in testing is specification-based testing, in which test cases are generated from the behavioral and/or requirement specifications of a system. In this paper, we consider the specification-based testing in which the system requirement is formally specified in linear temporal logic (LTL).

Linear temporal logic is a widely-accepted and very expressive logic that can specify safety, fairness, and liveness properties. LTL is supported by popular model checkers like SMV [10] and SPIN [6]. An LTL formula specifies a property which must hold on all the paths, and such paths may be infinite both in number and in length. Restricted by the resources, a test suite must be finite. Our first and uttermost question is, how a finite test suite can be selected to test an LTL property on an implementation? We developed the following techniques to solve the discrepancy between the infinite paths on which the LTL property must hold and the reality that a tractable test suite must be finite in number and in length.

- *Property-coverage Metrics and Criteria.* To limit the number of test cases to finite, we start with a coverage metric that measures how well an LTL property is tested by a test suite. Based on mutations on the requirement, the *property-coverage metric* checks the subformulae of the LTL property covered by a set of tests. The precise definition of property-coverage metric is given in Section 4. We propose a coverage criterion based on property-coverage metric. In comparison to the traditional structural-based coverage, the property-coverage criterion we advocate selects a finite set of test cases with respect to the system requirement. We also discuss the issue of test generation under the property-coverage criterion: we show that a property-coverage test suite can be characterized by a set of ∃LTL formulae that are formally transformed from the target LTL formula, hence a model checker with counterexample mechanism like SMV and SPIN can be used to generate witnesses for ∃LTL formulae from which the test suite will form.

- Test-truncating Strategy. Although tests selected by the property-coverage criterion is finite in number, they may be still infinite in length. Our second step is to reduce the length of a test to finite. Indeed, the witnesses generated by model checkers for ∃LTL formulae have a special structure known as "lasso-shaped" structure [4]. By exploiting this special structure, we are able to replace an infinite test by a finite equivalent one. This

*This research was supported in part by NSF CCR-0086147, NSF CCR-0209024, and ARO DAAD19-01-1-0473

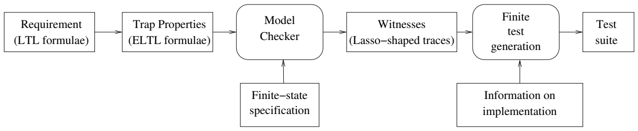

Figure 1: Property coverage test generation

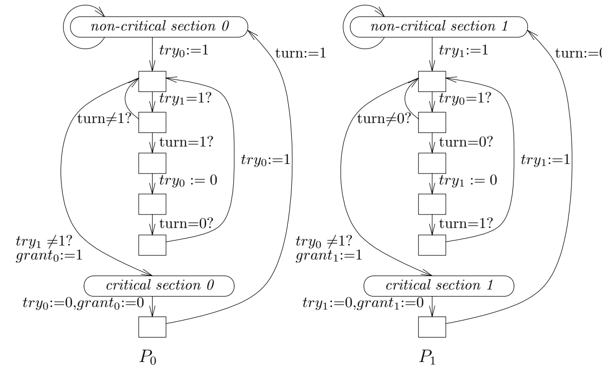

Figure 2: T_{dek}: the EFSM specification of Dekker's algorithm

approach is discussed in Section 5 in both a white-box test setting and a black-box test setting.

Our technique is inspired by the techniques from model checking in following sense: first, the requirement is encoded in linear temporal logic LTL; second, we use the notion of nonvacuity [2, 8] in model checking to explain the implications of *property-coverage metric and criterion*; finally, model checkers are used to automate the test generation.

The rest of paper is organized as follows: we outline the testing framework in Section 2 with the illustration of a motivating example. Section 3 prepares notations and definitions; Section 4 defines the property-coverage metric and criterion. Section 5 introduces *test-truncating strategy*, which reduces an infinite test to a finite equivalence one in either a black-box setting or a white-box setting. Section 6 shows our experiment on test generation using SMV; Finally, we summarize the results in Section 7. Proofs have been removed from the paper to save space. The full paper is available at [13].

2. TESTING FRAMEWORK

Figure 1 shows the workflow of our approach. The required property of a system is given as an LTL formula. We also assume that the specification (model) of the system is available. Test generation proceeds in three phases. In the first phase, each LTL property is transformed to a set of ∃LTL formulae called trapping formulae. The trapping formulae characterize test suites that satisfy property-coverage criterion. In the second phase, a *lasso-shaped* test is produced using model checkers for each ∃LTL property. A lasso-shaped test is a potentially infinite sequence, defined precisely in Section 3, represented as a finite sequence of steps leading to a loop. A lasso-shaped trace captures one possible way for the system to satisfy the property. In the last phase, a lasso-shaped test is truncated into a finite test case. The exact length of the resulting test case is determined by the targeted test setting. Our approach works on both white-box testing and black-box testing. As we will see shortly, less information revealed about the structure of the implementation means that longer tests for the same properties need to be generated and executed.

A motivational example. The example in Figure 2 illustrates our motivation. The specification used in this example is the Dekker's software solution to mutual exclusion problem. The specification is presented as the parallel composition of two extended finite state machines (EFSMs), as shown in Figure 2. Note that variables $grant0$ and $grant1$ are not required by the original algorithm. They are introduced to mark the granted accesses to critical sections.

The property of interest is encoded as an LTL formula $f_{mux} = \mathsf{A}\phi_{mux}$, where

$$\phi_{mux} = \mathbf{G}((try_1 = 1) \rightarrow \mathbf{F}(grant_1 = 1))$$

Note that $try_1 = 1$ only if P_1 makes its request to access the critical section 1, and hence the property ϕ_{mux} states that every request for the critical section 1 is eventually granted. The system being tested is an implementation of Dekker's algorithm. We assume that we may observe its behaviors via a predefined interface. In this example, the interface consists of the variables $turn$, try_0, try_1, $grant_0$, and $grant_1$.

There are two obstacles in testing f_{mux}. First, fully establishing f_{mux} on the implementation requires to check all its possible executions, which are potentially infinite in number. This renders testing infeasible. We instead aim at selecting nontrivial executions that f_{mux} is likely to fail. Clearly the property holds trivially if no requests to the critical section 1 has ever been made, hence we should check the executions in which such request is made at least once. The characteristic of such executions is captured by the following ∃LTL,

$$f_1' = \mathsf{E}(\mathbf{F}(try_1 = 1) \wedge \phi_{mux})$$

An sample test satisfying this property may be,

$$\rho_1 = \begin{array}{l} \emptyset\{try_0\}\{try_0, try_1\}\{try_0, try_1, grant_0\}\{try_1\} \\ \{try_1, turn\}\{try_1, grant_1, turn\}\{turn\} \cdot \emptyset^{\omega} \end{array}$$

We present a test as a sequence of sets of variables whose values are 1 in each step. In ρ_1 both processes make the request to the critical sections. Both processes have been granted the access sequentially and make further request afterwards.

Another nontrivial case is that the access is by "invitation only", that is, we want to make sure that if there is no access made after a time t, then no request is made after t. This is captured by the following ∃LTL formula:

$$f_2' = \mathsf{E}(\mathbf{FG}(grant_1 \neq 1) \wedge \phi_{mux})$$

An sample test satisfying this requirement may be

$$\rho_2 = \begin{array}{l} \emptyset\{try_0\}\{try_0, try_1\}\{try_0, try_1, grant_1\}\{try_0\} \\ \{try_0, grant_0\}\emptyset\{turn\} \\ \cdot\ \{try_0, turn\}\{try_0, grant_0, turn\}\{turn\}^{\ \omega} \end{array}$$

In ρ_2 each process makes a request and is granted an exclusive access to its critical section, and afterwards only P_1 makes requests to access its critical section.

Having selected ρ_1 and ρ_2, our next problem is that both of them are infinite in length; To be practical a test must be finite. Note that ρ_1 and ρ_2 have a so-called " lasso-shaped" structure; that is, they start with a finite prefix and end with a loop. Our strategy is to run ρ_2 for a finite number of times till the future behavior of a system can be projected. We consider two test settings: if the implementation is a white box, i.e., its structural is visible to the tester, we may end the test ρ_2 with a positive result if same states are encountered twice at the same position of the test, say, at $\{try_0, turn\}$ in ρ_2, because we are certain that ρ_2 can be extended from $\{try_0, turn\}$ to its full length by following the path already being tested; if the implementation is a black box but the number of its states is bounded by n, we only need to test the loop for at most n times since by then we are sure that the same states at the same position on the test has been encountered twice and the implementation passes ρ_2 in its full length.

The intuitions we just follow will be formalized in the rest of the paper: the notion of selecting non-trivial cases will be captured by "property-coverage criterion." In Section 4, we will extract the $\exists$LTL properties characterizing non-trivial test cases syntactically from the original LTL properties; the reason we are able to truncate ρ_1 and ρ_2 is their special lasso-shaped structures. In fact, such structures are possessed by the tests generated using model checkers to the $\exists$LTL formulae. Section 5 generalizes this test-truncating strategy in the context of white-box and the bounded black-box testings.

3. PRELIMINARIES

3.1 Kripke structures, traces, and test

In this paper systems are modeled as *Kripke structures.*

DEFINITION 3.1 (KRIPKE STRUCTURE). *Given a set of atomic proposition $\mathcal{A}$, a Kripke structure is a tuple $\langle \mathcal{S}, s_0, \rightarrow, \mathcal{V}\rangle$, where $\mathcal{S}$ is the set of states, $s_0 \in \mathcal{S}$ is the start state, $\rightarrow \subseteq \mathcal{S}\times\mathcal{S}$ is the transition relation and $\mathcal{V} : \mathcal{A} \rightarrow 2^{\mathcal{S}}$ is an evaluation for atomic propositions.*

We write $s \rightarrow s'$ in lieu of $\langle s, s'\rangle \in\rightarrow$. We use $a, b, \cdots$ to range over $\mathcal{A}$. We also denote $\mathcal{A}_\neg$ for the set of atomic propositions proceeding by the negation. Together, $\mathcal{L} = \mathcal{A} \cup \mathcal{A}_\neg$ defines the set of *literals*. We let $l, l_1, \cdots$ and $L, L_1, L_2, \cdots$ range over $\mathcal{L}$ and $2^{\mathcal{L}}$, respectively. We may abuse the use of $\mathcal{V}$ so that $\mathcal{V}(l) = \mathcal{S} - \mathcal{V}(a)$ if $l = \neg a$, and $\mathcal{V}(L) = \bigcap \{\mathcal{V}(l) \,|\, l \in L\}$.

We will also use the following notations: Let $\beta = p_0p_1\cdots$ be a sequence, we refer to $\beta[i] = p_i$ as i-th element of β, $\beta^{(i,j)}$ as the subsequence $p_i \cdots p_j$, and $\beta^{(i)} = p_i \cdots$ as the i-th suffix of β. A *trace* of a Kripke structure $\langle \mathcal{S}, s_0, \rightarrow, \mathcal{V}\rangle$ is defined as a maximal sequence of states starting with s_0 which respects the transition relation $\rightarrow$, i.e., $P[0] = s_0$ and $P[i-1] \rightarrow P[i]$ for every $i < |P|$.

DEFINITION 3.2 (LASSO-SHAPED SEQUENCE). *A sequence β is* lasso-shaped *if it has the form $\alpha_1(\alpha_2)^\omega$, where α_1 and α_2 are finite sequences. $|\alpha_2|$ is called the* repetition factor *of β. The length of β is a vector $\langle |\alpha_2|, |\alpha_1|\rangle$ with $|\alpha_2|$ as the most significant bit.*

DEFINITION 3.3 (TEST AND TEST SUITE). *A test is a sequence defined on $2^{\mathcal{L}}$, where $\mathcal{L}$ is the set of literals. A test case is a finite test. A test suite Ξ is a finite set of test cases. A system $T_i = \langle \mathcal{S}, s_0, \rightarrow, \mathcal{V}\rangle$ passes a test case ξ if T_i has a trace R such that $R[i] \in \mathcal{V}(\xi[i])$ for $i \le |\xi|$. A system T' conforms to T if every test passed by T must also be passed by T'.*

We define a function Π that extracts a test from a trace by projecting R on atomic propositions, that is, $(\Pi(R))[i] = \{l \mid R[i] \in \mathcal{V}(l)\}$.

3.2 LTL model checking

System requirements are given in Linear Temporal Logic (LTL) . The definition of LTL and its dual logic $\exists$LTL relies on the notion of *path formula*, which is defined recursively as below,

$$\phi ::= a \mid \neg\phi \mid \phi \wedge \phi \mid \mathsf{X}\,\phi \mid \phi\,\mathsf{U}\,\phi$$

LTL formulae and $\exists$LTL formulae have the form $\mathsf{A}\phi$ [1] and $\mathsf{E}\phi$, respectively. A and E are called *path quantifiers*, and X, U are *path modalities.* A formula is said *simple* if it is a path formula without path modality. f is a *state* formula if f is an $\exists$LTL formula, or an LTL formula, or a simple formula. In what follows, we use $f, g, \cdots$ to range over state formulae and $\phi, \psi, \cdots$ to range over path formulae. We allow the syntactic sugaring of LTL formula: We write $\mathbf{G}\phi$ and $\mathbf{F}\phi$ in lieu of $false\ \mathsf{R}\ \phi$ and $true\ \mathsf{U}\ \phi$, respectively, and we use R as the dual of U.

LTL and $\exists$LTL are interpreted with respect to a *Kripke structure* $T = \langle \mathcal{S}, s_0, \rightarrow, \mathcal{V}\rangle$. Formally, the semantics of a path formula ϕ is defined as follows, where R is a trace of T,

1. $R \models_T a$ iff $R[0] \in \mathcal{V}(a)$.
2. $R \models_T \neg\phi$ iff $R \not\models_T \phi$
3. $R \models_T \mathsf{X}\phi$ iff $R[1] \models \phi$.
4. $R \models_T \varphi\mathsf{U}\psi$ iff $\exists i \in \omega$ such that $R^{(i)} \models \psi$ and $R^{(j)} \models \varphi$ for all $j < i$.
5. $R \models_T \varphi \wedge \psi$ iff $R \models \varphi$ and $R \models \psi$.

The semantics for LTL or $\exists$LTL associate formulae with a set of states that satisfy the formula: $s \models_T \mathsf{A}\phi$ if $R \models_T \phi$ for every path R from s, and $s \models_T \mathsf{E}\phi$ if $R \models_T \phi$ for some path R from s. We will write $T \models f$ in lieu of $s_0 \models_T f$. It can be shown that $\wedge$ and $\vee$, U and R, A and E are dual to each other, and X is self-dual. Apparently, the negation of a LTL formula falls into $\exists$LTL, and vice versa.

By the definition, the holding of a $\exists$LTL formula or the refusal of a LTL formula may be evidenced by a single trace. This observation induces the notion of linear witness and counterexample (cf. [3]).

DEFINITION 3.4. *Let $\mathsf{E}\phi$ be a $\exists$LTL formula and $T = \langle \mathcal{S}, s_0, \rightarrow, \mathcal{V}\rangle$ be a Kripke structure, if P is a trace of T such that $P \models_T \phi$, then P is a linear witness for the $\exists$LTL model-checking problem $\langle \mathsf{E}\phi, T\rangle$ and a linear counterexample for the LTL model-checking problem $\langle \mathsf{A}\neg\phi, T\rangle$.*

THEOREM 3.5. *Given a finite Kripke structure T, for every LTL property f such that $T \not\models f$, there exists a lasso-shaped counterexample for model-checking problem $\langle f, T\rangle$; for every $\exists$LTL property g such that $T \models g$, there exists a lasso-shaped witness for $\langle g, T\rangle$.*

[1] We explicitly write the primary path quantifier A in a LTL formula to distinguish it from $\exists$LTL formulae

3.3 Vacuity

The notion of vacuity [2] in model checking is introduced to capture the problem that properties may be trivially satisfied. Since its introduction, the problem inspires much interests on how well a system is checked on a property. Later in Section 4 the results from the vacuity research help us develop the notion of "property coverage" metric and criterion. We use $f[\phi \leftarrow \psi]$ to denote the formula obtained by replacing a designated *occurrence* of the formula ϕ by ψ. [2]

DEFINITION 3.6 (AFFECT). *A sub-formula ϕ of f affects f in model T if there is a formula ψ such that the truth value of f and $f[\phi \leftarrow \psi]$ are different with respect to T.*

DEFINITION 3.7 (VACUITY). *T satisfies f vacuously with respect to a subformula ϕ if $T \models f$ and ϕ doesn't affect f in T. T satisfies f vacuously if there exists a subformula ϕ such that T satisfies f vacuously with respect to ϕ.*

By Definition 3.7, we have to check all the possible replacement of each subformula to decide the non-vacuity of the formula, which is practically impossible. Nevertheless, Theorem 3.9 shows that one only needs to check the occurrences of atomic propositions by replacing them by true or false depending their polarities.

DEFINITION 3.8 (POLARITY OF SUB-FORMULA). *The polarity of f's sub-formula is recursively defined on the structure of f as follows: let ψ be a sub-formula of ϕ, then ψ has the positive (negative polarity) if it is nested in even (odd) number of negation.*

THEOREM 3.9. *[8] A Kripke structure T satisfies the formula f vacuously if and only if $T \models \neg f[a \leftarrow \Box(a)]$ for some (occurrence of) atomic proposition a, where $\Box(a) = false$ if a has positive polarity in f and $\Box(a) = true$ otherwise.*

4. PROPERTY-COVERAGE CRITERIA

Now we consider the test generation for LTL properties. An LTL formula describes a property that holds on all the paths of the system and hence fully establishing an LTL property on an implementation requires checking all the possible executions, which is potentially infinite. We instead concentrate on those tests that provide the sufficient coverage on the property being tested. Intuitively, the property-coverage metric in Definition 4.1 describes how well the different mutations of an LTL property can be excluded by tests. We consider a general notion of *mutation* defined as replacing some subformula ϕ of an LTL property f with an arbitrarily different formula ψ, written as $f[\phi \leftarrow \psi]$.

DEFINITION 4.1 (PROPERTY-COVERAGE METRICS). *Given an LTL property f, a test t covers a subformula ϕ of f if there is a mutation $f[\phi \leftarrow \psi]$ such that every Kripke structure T that passes t will not satisfy the formula $f[\phi \leftarrow \psi]$. The property-coverage metrics for the LTL property f is a preorder $\gg_f$ such that for every test suites ST_0 and ST_1, $ST_0 \gg_f ST_1$ iff every subformula ϕ of f covered by a test $t \in ST_1$ is also covered by some test $t' \in ST_0$.*

[2] Vacuity may also be defined based on the replacement of all occurrences of a subformula. For a comparison of different notions of vacuity, readers may refer to [8]

Think T_s in Definition 4.2 as the model of the system, a test suite satisfying the property-coverage criteria shall be passed by the model, just as other test suites generated from the model for specification-based testing, and in addition, it also achieves the maximal coverage on the target LTL property.

DEFINITION 4.2 (PROPERTY-COVERAGE CRITERIA). *ST is a property-coverage test suite for a system T_s and an LTL property f if T_s passes ST and ST covers every subformula of f.*

As stated before, an LTL property can hardly be established on an implementation by tests alone because one has to check all the possible executions of the implementation, which is potentially infinite. Nevertheless we consider a set of tests *nontrival* if it can exclude some unwanted types of implementations, in case of property-coverage criterion, those implementations that satisfies some mutation of the LTL requirement. To better understand the implication of property-coverage criterion, we contrast it with the notion of non-vacuity in model checking. Both of them are introduced to measure how well a logic property (requirement) captures the system (implementation). The different is that property coverage does it by testing, while non-vacuity analysis uses model checking. Lemma 4.3 links property coverage with the notion of affect assuming that the property holds on the system model, and Theorem 4.4 links property-coverage criterion with the notion of non-vacuity under the same condition.

LEMMA 4.3. *A subformula ϕ of f affects f in the system T_i if $T_i \models f$ and T_i passes a test that covers the subformula ϕ of f.*

THEOREM 4.4. *A system T_i satisfies a property f non-vacuously if $T_i \models f$ and the system T_i passes a property-coverage test suite for some system T_s and the property f.*

Now we need to find a way to generate a property-coverage test suite from the specification and the property: we turn to the witness (counterexample) generation mechanism of model checkers for help.

LEMMA 4.5. *Given a system T and an LTL formula f, if a subformula ψ of f affects f on T, then,*

1. *there is a lasso-shaped witness for the model checking problem $\langle \neg f[\psi \leftarrow \Box(\psi)], T\rangle$.*
2. *For every witness R for $\langle \neg f[\psi \leftarrow \Box(\psi)], T\rangle$, the test $\Pi(R)$ is a test which covers the subformula ψ of f.*

Test generation using model checker has been studied before [1, 7]. The idea is to use model checkers to generate witnesses as tests for a set of properties characterizing coverage criteria. Lemma 4.5 lays out the path one may follow to generate a property-coverage test suite: to obtain a test that covers a subformula ψ of f on T, we model check T on an $\exists$LTL property $\neg f[\psi \leftarrow \Box(\psi)]$, the mutation of f in which ψ is replaced by true or false depending on its polarity; the test can be obtained by projecting the witness on atomic propositions. Furthermore, by Lemma 4.6 we only need to generate tests for every atomic proposition in the target property.

PropertyCoverage($f \equiv \mathsf{A}\phi$, T)

for every atomic proposition $a \prec f$

$\langle result, witness\rangle$:= ModelCheck($\mathsf{E}(\phi \wedge \neg\phi(a \leftarrow \Box(a)))$, T)

if $result$=**true then**

% Project the witness on atomic propositions

$test$:=$\Pi(witness)$

ST:=$ST \bigcup \{test\}$

return ST

Figure 3: Generating a property-coverage test suite for T and f

Lemma 4.6. *Let $\psi' \prec \psi \prec \phi$, a test that covers the subformula ψ' of ϕ also covers the subformula ψ of ϕ.*

To generate a property-coverage test suite for the specification T and an LTL formula f, we first define a set of *trapping properties* for f as follows,

$$G(f,T) = \{\mathsf{E}(\phi \wedge \neg\phi[\psi \leftarrow \Box(\psi)] \mid \psi \text{ is a subformula of } f\ \}$$

then we generate a witnesses set $W_{G(f,T)}$ for $G(f,T)$ such that for each $g \in G(f,T)$, there is a $R \in W_{G(f,T)}$ that is a witness for g. By Theorem 4.7, $\Pi(W_{G(f,T)})$, the set of tests projected from the the witness set forms a set of tests covering LTL formula f on the specification T. Figure 3 shows the algorithm for computing a set of tests covering f on T.

Theorem 4.7. *Given a Kripke structure T and an LTL formula f such that T satisfies f nonvacuously, the set of tests $\Pi(W_{G(f,T)})$ covers f on T.*

Finally, we consider the practical meaning of property-coverage testing by revisiting the motivational example in Section 2. Recall that in the motivational example the requirement is $f_{mux} \equiv A(\phi_{mux})$ and the specification is T_{dek} in Figure 2. As the first step, we extract a set of trapping properties $G(f_{mux}, T_{dek})$ from f_{mux} that characterizes the property-coverage criteria,

$$\begin{aligned} &G(f_{mux}, T_{dek}) \\ &= \{\mathsf{E}(\phi_{mux} \wedge \neg(\phi_{mux}[(grant_1 = 1) \leftarrow false])), \\ &\quad \mathsf{E}(\phi_{mux} \wedge \neg(\phi_{mux}[(try_1 = 1) \leftarrow true]))\} \\ &= \{\mathsf{E}(\mathbf{F}(try_1 = 1) \wedge \phi_{mux}), \mathsf{E}(\mathbf{FG}(grant_1 \neq 1) \wedge \phi_{mux})\} \end{aligned}$$

The first formula $\mathsf{E}(\mathbf{F}(try_1 = 1) \wedge \phi_{mux})$ in $G(f_{mux}, T_{dek})$ characterizes a test in which a request to access the critical section 2 is made. This is equivalent to the first criteria for non-trivial test we draw in Section 2; the second formula $\mathsf{E}(\mathbf{FG}(grant_1 \neq 1) \wedge \phi_{mux})$ characterizes a test on which eventually no access to the critical section 2 is made (and hence no request should be made afterwards). This is equivalent to the second criteria in Section 2. Furthermore, model-checking $G(f_{mux}, T_{dek})$ produces two lasso-shaped tests similar to the tests in Section 2. Instead of relying on our intuitions, now we obtain "nontrivial" test cases by an automated formal reasoning.

5. TEST-TRUNCATING STRATEGY

Property-coverage criterion limits the number of tests to finite. By Theorem 3.5 such witnesses are lasso-shaped, potentially infinite in length. Next we will showshow that a lasso-shaped test may be reduced to a finite equivalent one in either a black-box or a white-box test settings.

5.1 Black-box testing

In black-box testing the detail of the implementation being tested is unknown, but just like other black-box testing paradigms such as conformance testing (cf. [9]) we assume the knowledge of the uppper-bound n on the number of states. With the known upperbound n, it is possible to test only a finite prefix of the lasso-shaped test to project the future behavior of the implementation. The idea is that, if we repeat the loop part of the test for enough times, for example, n times, then the same states must be encountered twice at the same position on the loop, therefore, an infinite trace extended by repeating same configuration at end should also pass the test in its full length. More specifically, let's assume that a black-box implementation T_i passes the finite test $t = \alpha(\beta)^n$ and the finite trace of T_i in response to t is R, then the same state must be repeated at the beginning of some iterations on β, i.e., there are $i, j < n$ such that $R[|\alpha| + |\beta| \cdot i] = R[|\alpha| + |\beta| \cdot j]$. Therefore, T_i has a trace $R^{(0,|\alpha|+|\beta|\cdot j)} \cdot (R^{(|\alpha|+|\beta|\cdot i+1, |\alpha|+\beta|\cdot j)})^\omega$ and clearly such trace will pass the infinite test $\alpha(\beta)^\omega$. Thus, it is sufficient that we test only a truncated finite test $\alpha \cdot (\beta)^n$ instead of the infeasible job of testing $\alpha(\beta)^\omega$ in its full length. Theorem 5.1 shows that the cut for lasso-shaped tests is also tight.

Theorem 5.1. *For a black-box system T_i with at most n states and a lasso-shaped test $t = \alpha(\beta)^\omega$, n is the least number such that T_i passes t if and only if T_i passes $t^{(0,|\alpha|+n)}$.*

5.2 White-box Testing

In white-box testing, we assume that the detail of implementation is visible to a tester. For white-box testing a tester can track the states traversed and terminate whenever the same state has been visited twice at the same position on the loop. The following procedure outlines the strategy for applying a lasso-shaped test $\alpha \cdot (\beta)^\omega$ to an white-box implementation T_i,

1. Apply $\alpha[0], \alpha[1], \cdots$ to T_i.
2. Start with $i := 0$ and then repeat the following steps till T_i fails.
   (a) apply β^i to T_i. Let s_k be the current state of T_{lmp}
   (b) if $s_k \in S_i$ then test terminates with the report that T_i passes the test.
   (c) add s_k to S_i and $i := (i+1) mod |\beta|$

Clearly there are only two ways out under the above strategy: either T_i fails in test or the same state are encountered twice in the same position on the loop part. For the latter, we can project from this finite testing that T_i has an infinite trace which can pass $\alpha(\beta)^\omega$ in its full length. Such the infinite trace can be constructed from the finite path R in response to the truncated test: assume that s is the state that causes the termination of testing, i.e., there is a position i on the loop such that $R[(i)] = R[|R| - 1] = s$, $i \geq |\alpha|$, and $(|R| - 1 - i)\ mod\ (|\beta|) = 0$, then the infinite trace $R \cdot (R^{(i+1,|R|-1)})^\omega$ obtained by repeating the tail of R

	$P_1 : \mathbf{G}(\phi_0)$					$P_2 : \mathbf{G}(\phi_0 \rightarrow \mathbf{G}(\phi_1))$			
# of Atom. Prop.	5					4			
Interesting Prop.	$\mathbf{P_{11}}$	$\mathbf{P_{12}}$	$\mathbf{P_{13}}$	P_{14}	P_{15}	$\mathbf{P_{21}}$	$\mathbf{P_{22}}$	$\mathbf{P_{23}}$	$\mathbf{P_{24}}$
# of BDD nodes	781168	66630	76524	75922	75922	134749	103576	110160	204701
Time (sec.)	0.77	0.45	1.18	1.26	0.99	13.02	8.92	12.03	28.54
$\|\alpha\|$	5	6	12	6	6	34	19	8	22
$\|\beta\|$	7	11	15	11	11	15	26	8	8

Table 1: Test suite generated for a digital shuttle controller

	$P_1 : \mathbf{G}(\phi_0 \rightarrow \phi_1 \mathsf{U} \phi_2)$					$P_2 : \mathbf{G}(\phi_0 \rightarrow \phi_1)$	
# of Atom. Prop.	5					2	
Interesting Prop.	$\mathbf{P_{11}}$	$\mathbf{P_{12}}$	P_{13}	$\mathbf{P_{14}}$	P_{15}	$\mathbf{P_{21}}$	$\mathbf{P_{22}}$
# of BDD nodes	325026	345204	315076	416876	350898	124483	121910
Time (sec.)	101.83	145.35	103.98	198.20	194.53	8.89	7.73
$\|\alpha\|$	0	0	0	6	6	0	0
$\|\beta\|$	1	10	10	11	11	1	11

Table 2: Test suite generated for PCI bus protocol

from $i+1$ will also pass the test $\alpha \cdot (\beta)^\omega$ in its full length. The truncated test may be as short as $|\alpha| + |\beta|$, but in any case the truncated test is at most $|\alpha|+|\beta|\cdot n$ in length, where n is the number of states in the implementation.

6. EXPERIMENT

To assess the feasibility of our approach, we use the model checker SMV to generate tests under property-coverage criterion. The examples we chose are from the benchmark applications collected by Bwolen Yang [14]. In these examples, we choose a variety of properties, including safety and liveness properties. Each property is translated to a set of interesting properties characterizing property-coverage criterion, and then SMV is used to generate tests for these properties. All the experiments are done on 1.2 GHz Mobile Pentinum III machine with 512 MB memory. We use the Cadence SMV release 10-11-02p36 for the Windows.

The first example is a digital shuttle controller. In Table 1 we present two properties, where each ϕ_i represents a state formula. A set of trapping properties are extracted from these properties under property-coverage criteria. A property P_{ki} is obtained from P_k by replacing its i-th atomic proposition with true or false, depending on the proposition's polarity. The second example is a PCI bus protocol. We choose a safety property P_1 and a liveness property P_2. We report the length of tests in term of the finite prefix as well as the loop part.

7. CONCLUSIONS

Model-checking-assisted test generation recently receives much attention. In this work we consider specification-based testing in which the requirement is encoded in linear temporal logic, a popular temporal logic supported by many model checkers and the variants of which are widely adopted in industry today. We proposed an framework for testing linear temporal (LTL) properties. We are not trying to establish the correctness using testing. Instead, we want to provide a practical approach to enable the testing of linear temporal properties on the implementation. For such purpose, we propose the property-coverage criteria that limits the tests to those non-trivial ones. Under the property-coverage criterion, the property being tested are transformed to a set of $\exists$LTL properties characterizing non-trivial tests, which are in turn used by model checkers for generating tests via witness (counterexample) generation mechanism. We use the notion of nonvacuity in model checking to interpret the implication of property-coverage testing. Moreover, we argue that by exploiting their "lasso-shaped" structure the generated tests can be reduced to finite equivalent ones in either a white-box or a black-box testings.

The work presented in this paper can be extended in several ways. For instance, it is possible to further reduce the length of an test by minimizing proof structure for $\exists$LTL formulae. The techniques presented here for LTL may also be generalized to more expressive logics such as CTL* or μ-calculus. Finally, the approach can also utilize more generic proof structures such as support sets [12].

8. REFERENCES

[1] P. Ammann and P. Black. A specification-based coverage metric to evaluate test sets. In *International Journal of Reliability, Quality and Safety Engineering*, volume 8. World Scientific Publishing, 2001.

[2] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh. Efficient detection of vacuity in ACTL formulas. In *CAV'97*, 1997.

[3] E. Clarke, O. Grumberg, K. McMillian, and X. Zhao. Efficent generation of counterexamples and witnesses in symbolic model checking. In *Proc. 32nd Design Automaion Conference*, 1995.

[4] E. M. Clarke and I. A. Draghicescu. Expressibility results for linear-time and branching-time logics. In *REX Workshop*, volume 354 of *LNCS*, 1989.

[5] O.J. Dahl, F. W. Dijkstra, and C. A. R. Hoare. Structural programming. In *APIC Studies in Data Processing*, 1972.

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

[7] H. Hong, I. Lee, O. Sokolsky, and H. Ural. A temporal logic based theory of test coverage and generation. In *TACAS'02*, 2002.

[8] O. Kupferman and M. Vardi. Vacuity detection in temporal model checking. In *Proceedings of CHARME'99*, 1999.

[9] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - A survey. In *Proceedings of the IEEE*, volume 84, pages 1090–1126, 1996.

[10] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic, 1993.

[11] P. Stocks and D. Carrington. Test template framework: A specification-based testing case study. In *Proceedings of the 1993 International Symposium on Software Testing and Analysis*, 1993.

[12] L. Tan and R. Cleaveland. Evidence-based model checking. In *CAV'02*, 2002.

[13] L. Tan, O. Sokolsky, and I. Lee. Property-coverage testing. Technical Report MS-CIS-03-02, University of Pennsylvania, 2003.

[14] B. Yang. SMV models. http://www.cs.cmu.edu/b̃wolen/software, 1998.