Test Criteria for Model-Checking-Assisted Test Case Generation: A Computational Study

Bolong Zeng and Li Tan*
School of Electrical Engineering and Computer Science
Washington State University
Richland, WA 99352
{bzeng, litan}@wsu.edu

Abstract

Test case generation is often cited as one of the most challenging tasks in testing dependable systems [9]. Besides benefits as a verification technique by its own right, model checking is emerging as an efficient method for automating test case generation. Existing testing criteria and a range of new criteria inspired by formal requirements have been used in model-checking-assisted test generation. This paper reviews some of these existing and new test criteria. We developed a unified framework for evaluating the effectiveness of these test criteria and the efficiency of model-checking-assisted test generation for these criteria. The benefits of this work are three-fold: first, the computational study carried out in this work assesses the practical effectiveness and efficiency of model-checking-assisted test case generation, which are important metrics to consider for selecting the right test criteria and test generation approach. Second, we proposed a unified test generation framework based on generalized Büchi automata*. The framework uses the same off-the-shelf model checker, in this case, SPIN model checker [10], to generate test cases for different criteria and compare them on a consistent basis. Last but not least, we describe in great details the methodology and automated test generation environment that we developed on the basis of our unified framework. Such details are of interest to researchers who needs to carry out their own experimental study on test criteria, and to practitioners who want to integrate model-checking-assisted test generation into their testing process.*

1 Introduction

Model-checking has become an important player in the field of verification technologies since its debut [4]. It is now widely used in an arrange of applications such as the autopilot modules in airplanes, and deep space vehicle [18]. Model checking provides a rigid and mathematically sound proof for the correctness of a variety of systems that are crucial to industries and society. On the other hand, testing remains an important force in the field and its value is acknowledged by industrial practices and standards. For example, DO-178B [23], the standard for avionics software, is known for its rigorous requirements for testing criteria, such as the Modified Condition and Decision Coverage (MC/DC).

Model-checking-assisted test generation is one of the strategies that harness the synergy of both verification technologies [1]. Over years model checking community developed efficient algorithms to search the state space of a system and reason its temporal behavior. The basic idea behind model-checking-assisted test generation is to use this powerful searching capability provided by a model checker to search for execution paths of a system that satisfy a given test criterion, and then use paths to synthesize test cases. An apparent benefit of model-checking-assisted test generation is that model checkers can be reused to improve the efficiency of producing test cases, and in many cases, a powerful model checker may be able to systematically search the state space of a system and find test cases that are not apparent to the eyes of developers and quality engineers.

In practices model-checking-assisted test generation makes use of counterexample generation capability provided by many contemporary model checkers [2]. The objectives of a test criterion are encoded as temporal logic properties, often called "trap properties" [7]. Various strategies are proposed to translate existing test criteria to trap properties [8, 12, 22]. In [25], we proposed a syntax-based approach that extracts trap properties from temporal specification in Linear Temporal Logic. The approach is based on the notion of "vacuity" [16]. In [24], we proposed a semantics-based approach to synthesize trap properties

*Corresponding author.

from temporal specification in Büchi automata.

A question for model-checking-assisted test generation is how it performs in practice. The question can be further refined to two research questions: (i) how effective are test cases generated from different test criteria, and (ii) how efficient is a model-checking-assisted approach for a given test criterion. Besides theoretical interests, studies on these questions are of practical importance: studies would provide heuristic to practitioners on how to select test criteria to achieve desired test effectiveness, given time and resource allocated to test case generation.

The main motivation behind our work is to present a unified framework for evaluating and comparing the performance of various test criteria in context of model-checking-assisted test case generation. Specifically, We measure the efficiency of a model checker generating test cases for a given test criterion, and we assess the effectiveness of a given test criterion by measuring cross-coverage percentage with respect to other test criteria. The unified test framework enables us to take measurement consistently on the same platform, reducing the performance variations introduced by other factors.

Last but not least, we describe in great details the tool and techniques we developed for automating test generation process. Such details are of interests to researchers who need to carry out their own experimental study on test criteria, and to practitioners who want to integrate model-checking-assisted test generation into their testing processes.

The rest of the paper is organized as follows: Section 2 reviews the notations and prior knowledge that will be used in the rest of the paper. Section 3 describes test criteria and the strategies used to translate the criteria to linear-temporal trap properties. Section 4 introduces the methodology and workflow that we developed for this computational study. Section 5 discusses the result of our computation study. Finally Section 6 concludes this paper.

1.1 Related Works

In our study we compare an array of existing test criteria and property-based test criteria. A variety of strategies [8, 12, 22] have been proposed to translate the existing test criteria to trap properties, and these strategies have been incorporated into our unified framework. Fraser *et al.* [7] studied various test criteria in context of model-checking-assisted test generation. In addition to some of test criteria studied in [7], we also include a semantics-based coverage criterion [24]. In [7] authors use CTL as the underlying temporal logic to specify trap properties. A counterexample for CTL may not necessarily be linear [3]. In this study we use Büchi Automata as the underlying formalism for temporal properties. For most of test criteria we translate them to LTL formulae, which are then translated to Büchi Automata. Using LTL and Büchi Automata as underlying temporal logic has two benefits: first, since counterexamples for Büchi Automata are linear, these counterexamples can be used to generate test cases, which are required to be linear for most of testing applications. Second, it enables us to compare existing test criteria with a new semantics-based test criterion in [24], which is based on Büchi Automata.

2 Preliminaries

2.1 Kripke Structure, Traces, and Tests

We use *Kripke Structrue* to model a system. A Kripke structure is a finite transition system in which each state is labeled with a set of atomic propositions. Each atomic proposition represents a primitive property held at a state.

Definition 2.1. Given the alphabet of atomic propositions $\mathcal{A}$, a *Kripke structure* is denoted as a tuple $\langle \mathcal{S}, s_0, \rightarrow, \mathcal{T} \rangle$. $\mathcal{S}$ represents the set of states while $s_0 \in \mathcal{S}$ is the starting state. $\rightarrow \subseteq \mathcal{S} \times \mathcal{S}$ denotes the transitions among them, and $\mathcal{T} : \mathcal{A} \rightarrow 2^{\mathcal{S}}$ labels a states with a set of atomic propositions.

For brevity and clarity, we use $s \rightarrow s'$ in lieu of $(s, s') \in \rightarrow$. We let atomic propositions range over $a, b, \ldots$ in the alphabet $\mathcal{A}$. The set of all the atomatic propositions and their negations forms the set of *literals* $\mathcal{L}$.

A *sequence* of Kripke structure is a series of states $\Sigma = s_{i_0} s_{i_1} \ldots$ in which for any integer $k \geq 0$, $s_{i_k} \rightarrow s_{i_{k+1}}$. A *trace* is a maximal sequence which starts with s_0, the start state.

Definition 2.2. A sequence Σ is *lasso-shaped* if it is in the form of $\sigma_1(\sigma_2)^\omega$, where σ_1 and σ_2 are both finite sequences.

A lasso-shaped sequence can be understood as a sequence that after traveling through a certain number of states, falls into a loop of sub-sequence which repeats infinitely. The sequence with such feature may be reduced to a bounded finite sequence for testing purpose [25].

Definition 2.3. A *test* is a sequence defined on $2^{\mathcal{L}}$, and a finite test is a *test case*. A finite set of such test cases is a *test suite*. Passing a test case t implies that the system has a trace R, in which the i-th element: $R[i] \in \mathcal{T}(t[i])$.

2.2 Generalized Büchi Automaton

A generalized Büchi automaton (GBA) is an ω-automaton, whose acceptance language is essentially an extended version of a regular language with infinite length. We define an extended version of GBA with an alphabet of literals [24], along with the usual states, transitions and acceptance condition.

Definition 2.4. A *generalized Büchi automaton* is denoted as a tuple $\langle P, S, S_0, \Delta, \Lambda, \mathcal{F}\rangle$, in which S is a set of states, $S_0 \subseteq S$ is the set of start states, $\Delta \subseteq S \times S$ represents the transitions, and the $\mathcal{F} \subseteq 2^S$, which is a set of sets of states, forms the acceptance condition. In addition, P is a set of literals, and $\Lambda : S \rightarrow 2^P$ is the labeling function that maps each state to a set of literals.

Similarly, we write $s \rightarrow s'$ in lieu of $(s, s') \in \Delta$. A run of a generalized Büchi automaton B is an infinite sequence $R = s_0s_1...$ such that $s_0 \in S_0$ and $s_i \rightarrow s_{i+1}$ for every $i \geq 0$. $inf(R)$ is used to represent a set of states that appear for infinite times on R. A successful run of B must satisfy the following condition that for every $F \in \mathcal{F}$, $inf(R) \cap F \neq \emptyset$. A GBA B with the complete set of literals $\mathcal{L}$ accepts infinite words over alphabet $2^{\mathcal{A}}$. Let w be a word from the alphabet, B has a run R induced by w, denoted as $w \vdash R$, if and only if for every $i < |w|$, $w[i] \in \Pi(\Lambda(R[i]))$, where Π is the function mapping the literals to the atomic propositions that are associated with the same states. B accepts w, denoted as $w \models B$ if and only if B has a successful run R such that $w \vdash R$.

2.3 LTL Model Checking

We present the system requirements in the form of Linear Temporal Logic (LTL) [6]. A *path formula*'s syntax is defined recursively as

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

The basic semantics of a path formula are defined with respect to a Kripke structure $K = \langle \mathcal{S}, s_0, \rightarrow, \mathcal{T}\rangle$. If R is a trace of K and $a \in \mathcal{A}$ is an atomic propositon, we denote $R \models_K a$ if and only if $R[0] \in \mathcal{T}(a)$. Atomic proposition `true` may be satisfies by any state, whereas no state may satisfy atomic proposition `false`.

The "next" operator $\mathbf{X}$ in $\mathbf{X}\phi$ means that ϕ has to hold starting from next state. The "until" operator $\mathbf{U}$ in $\varphi\mathbf{U}\psi$ requires that φ has to hold until eventually ψ becomes true. In addition, we define $\vee$ as a dual of $\wedge$, and the "release" operator $\mathbf{R}$ as a dual of $\mathbf{U}$. For convenience, $\mathbf{G}\phi$ and $\mathbf{F}\phi$ are often used to denote `false` $\mathbf{R}$ ϕ and `true` $\mathbf{U}$ ϕ respectively, bearing the meanings of ϕ always holds and ϕ will eventually hold.

$\mathbf{A}$ and $\mathbf{E}$ are path quantifiers that address formulae of LTL and its dual logic $\exists$LTL in the form of $\mathbf{A}\phi$ and $\mathbf{E}\phi$, meaning that ϕ holds on *all* paths or there *exists* a path on which ϕ holds. Obviously, an LTL formula could be negated into a $\exists$LTL formula, and vice versa. By definition, a single trace could be used to prove or disprove the holding of a $\exists$LTL or LTL formula, such trace is called a linear witness or a counterexample for a model-checking problem [5]. It is further shown that there always exists such witness and counterexample that are lasso-shaped.

Definition 2.5. Given a trace γ on a Kripke structure K. If $\gamma \models_K \phi$ holds on K, γ is a *linear witness* for the $\exists$LTL model-checking problem $\langle E\phi, K\rangle$ and a *linear counterexample* for LTL model-checking problem $\langle A\neg\phi, K\rangle$.

3 Coverage Criteria

This section gives a brief introduction of the test criteria we covered in this work. For each criteria, we explain how the trap properties, or in the case of Büchi Automata state coverage, the "trap automata" are generated for the experiments. In this paper, we refer to trap properties as desirable properties, i.e., test cases are generated with the purpose of satisfying them. We use LTL as the underlying logic to describe temporal properties as opposed to CTL in [7]. Both LTL and CTL belongs to the CTL* family. While they share a common subset, there are properties that can be described only by one of them. The motivation behind our choice is to build a unified framework that could incorporate other criteria such as the Büchi Automata based state coverage criteria that are not directly based on temporal logic formulae. Also, counterexamples for LTL are always linear, which is a favorable character for most testing applications.

3.1 Branch Coverage Criterion

Branch coverage criterion belongs to the logic expression criteria [13], and is one of the most commonly-used test coverage standards. The criterion focuses on the truth value of the guard of a transition in a transition system, for example, an "if-else" construct in a program. This criterion covers the dynamic behaviors of a system by testing both true and false outcomes of a logic expression. It needs to access the structure of the system and hence it is classified as a syntax-based white-box testing approach.

We orchestrate our experiments in the following manner. A Boolean flag b_i is attached to the i-th branch of a conditional construct of a system, thus checking the value of the flag could reveal that whether a branch is covered or not. We write the trap property as below,

$$\mathbf{EF}(b_i \wedge \mathbf{X}\texttt{true})$$

A witness produced for the trap property would indicate the i-th branch is covered. $\mathbf{X}$`true` is added to ensures that the transition induced by the i-th branch is completed, i.e., the transition reaches its destination state.

For a multiple-branch conditional construct such as "*switch*" statement in C/C++ or "if" block in Promela, the "*default*" branch is executed if all other branches fail. To make the criterion consistent, we assume that there is always a "default" branch, even it is undefined and/or has an empty code body.

3.2 Data-flow Coverage Criteria

It has been shown that data-flow coverage criteria may be reduced to model checking problems [11]. One of them emphasizes on covering definition-use pairs in a system [21]. We adopt the requirements of the all-Definitions coverage criterion in our work, which requires to cover all the definition-clear paths in the system.

Same as branch coverage criterion, data-flow coverage criteria are also white-box testing approaches. We apply a similar strategy as in Section 3.1 to the all-definition coverage criterion. We denote a definition and usage of a variable v as $d(v)$ and $u(v)$. We also write the disjunction of all the definitions of v as $D(v)$. The trap property is generated for every definition and usage of v as follows:

$$\mathbf{EF}(d(v) \wedge \mathbf{X}(\neg D(v)\mathbf{U}(u(v) \wedge \mathbf{X}\texttt{true}))$$

The property means that starting from the state that a definition is reached, no other definition can occur until the usage of x eventually happens. It also guarantees that the system is still executable after the usage. Any trace that the model checker is able to find is a witness of a definition clear path from $d(v)$ to $u(v)$.

3.3 Property Coverage Criterion

Inspired by the requirement of checking an implementation against a specific property instead of syntax-based standards, Tan *et al.* proposed the property-coverage metric and criterion towards model-checking-assisted test generation [25]. The property coverage metric measures how well an LTL property is tested by a test suite. A mutation of a formula f is written as $f[\phi \leftarrow \psi]$, in which ϕ is a subformula of f that is being replaced by ψ.

Definition 3.1 (Property-Coverage Metric [25])**.** Given a test t. Consider a mutation $f[\phi \leftarrow \psi]$, if every Kripke structure K that passes t is unable to satisfy the mutated formula, then t covers the subformula ϕ in f. The property-coverage metric is a preorder relationship $\gg_f$ for property f. For test suites TS_1 and TS_2, $TS_1 \gg_f TS_2$ if and only if for every subformula ϕ of f covered by a test $t \in TS_2$, there exists a test $t' \in TS_1$ that also covers ϕ.

The intuition behind the property-coverage metric and criterion is that a test suite shall test the relevancy of a system with respect to its requirement specification. One way to test the relevancy is to check whether every sub-formula plays an indispensable role in defining the requirement, that is, every sub-formula needs to be covered in test.

Definition 3.2 (Property-Coverage Criterion [25])**.** TS is a property-coverage test suite for a system K and an LTL property f if K passes TS and TS covers every subformula of f.

Function $\Box$ defines the polarity of a sub-formula. $\Box(\phi) = \texttt{true}$ for a subformula ϕ in f if it is nested in odd number of negations; otherwise $\Box(\phi) = \texttt{false}$ [25]. The trap property for a test covering the sub-formula ϕ may be defined as,

$$\mathbf{EF}(\neg f[\phi \leftarrow \Box(\phi)])$$

For example, consider a LTL property describing a vehicle at an intersection: $\phi_v = red \rightarrow \mathbf{X}(\neg red\, \mathbf{R}\, \neg acc)$, meaning that "after the light turns red, the driver will not accelerate the vehicle until the light switches". Using $\mathbf{R}$ instead of $\mathbf{U}$ means that the light may or may not switch. Using the property-coverage criterion on atomic propositions, we obtain three "trap" properties: $\neg(true \rightarrow \mathbf{X}(\neg red\, \mathbf{R}\, \neg acc)) = \mathbf{X}(red\, \mathbf{U}\, acc)$, $\neg(red \rightarrow \mathbf{X}(\neg true\, \mathbf{R}\, \neg acc)) = red \wedge \mathbf{X}(\mathbf{G}\, acc)$, and $\neg(red \rightarrow \mathbf{X}(\neg red\, \mathbf{R}\, \neg true)) = red \wedge \mathbf{X}(red\, \mathbf{U}\, true) = red$. For each of these properties, a test suite covering ϕ_v has a test case satisfying the property. For the details of the property-coverage criterion, interested users may refer to [25].

3.4 Büchi Automata State Coverage Criteria

Recently Tan [24] proposed a semantic-oriented testing strategy for requirement specification in Büchi automata. The work is an extension of [24]. Unlike the property-based coverage criterion whose definition is based on the syntactical structure of an LTL formula, the coverage criteria in [24] are based on Büchi automata, which capture the semantics of a linear-time requirement specification.

Definition 3.3 (Covered States [24])**.** Given a generalized Büchi automaton $B = \langle P, S, S_0, \Delta, \Lambda, \mathcal{F}\rangle$, if there exists a successful run R goes through s that can be induced by a test t, then t *weakly* covers s. If B accepts t, while every successful run R of B such that $t \vdash R$, it goes through s, then t *strongly* covers s.

Trap properties are given in the form of Büchi automata, transformed from the original Büchi automaton encoding linear-time requirements. To *strongly* cover a state s of the original Büchi automaton B, one may remove the state from B. The result is a *State Excluding Generalized Büchi Automaton (SE-GBA)* $B_{\bar{s}}$. The "trap automaton" for generating test strongly covering s is $\neg B_{\bar{s}}$, the negation of SE-GBA.

To *weakly* cover a state s of B, one may construct a *State Marking Generalized Büchi Automaton (SM-GBA)* $B(s)$ as follows: first we get a replica B' of the original Büchi automaton B, and then we add transitions from s of B to B''s counterparts of the destinations of these transitions. The start states of the resulting SM-GBA $B(s)$ are those of B, and the acceptance conditions of $B(s)$ is that of B'. For the details of SE-GBA, SM-GBA, and their constructions, interested readers may refer to [24].

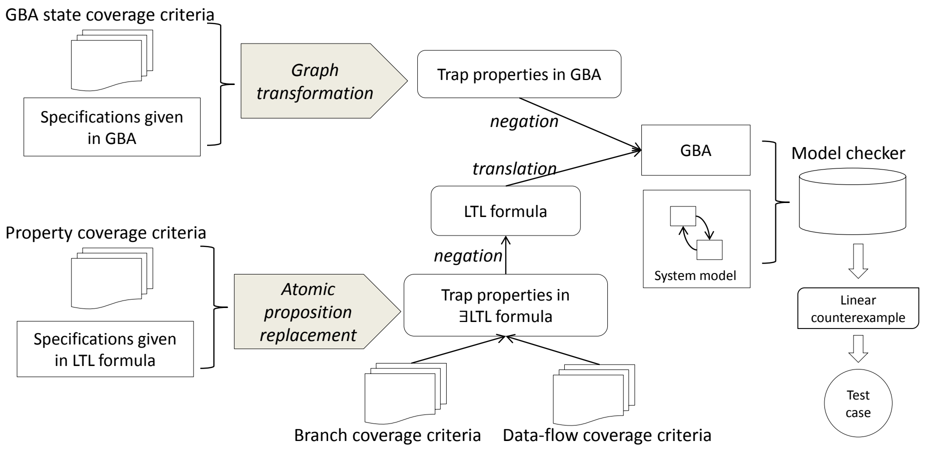

GBA state coverage criteria
Specifications given in GBA
Graph transformation
Trap properties in GBA
negation
GBA
translation
Model checker
LTL formula
System model
Property coverage criteria
negation
Atomic proposition replacement
Trap properties in ∃LTL formula
Specifications given in LTL formula
Linear counterexample
Test case
Branch coverage criteria
Data-flow coverage criteria

Figure 1. Test generation procedure

4 Experiment Methodology and Workflow

We propose a uniform framework for evaluating practical performance of testing criteria proposed for model-checking-assisted test generation. The framework contains two main components: a model-checking-assisted test generation platform capable of handling various test criteria; and a performance-comparison tool that computes cross-coverage between different test criteria.

Figure 1 shows the general workflow of our model-checking-assisted generation platform. We use SPIN as the underlying model checker [10] and model a system under test in Promela, the system modeling language used by the SPIN. For test criteria whose *trap properties* may be expressed in ∃LTL, such as branch coverage criterion (Section 3.1), data-flow coverage criteria (Section 3.2), and property-coverage criterion (Section 3.3), we first negate the trap properties to obtain LTL formulae. These LTL formulae are then fed to SPIN, along with the Promela model of the system under test. Internally SPIN translates a LTL formula to a Büchi automaton and performs Büchi-automaton-based model checking. If the system model does not satisify the LTL formula, SPIN produces a counterexample, which can be then translated to a test case.

For Büchi Automata state coverage, the specifications presented in GBA needs to go through a graph transformation process using Goal [15]. We explain the process with an example taken from our experiments. Figure 3 shows a generalized Büchi automaton B, which is semantically equivalent to LTL property $L_1 : \mathbf{G}(\neg t \rightarrow ((\neg p\ \mathbf{U}\ t) \vee \mathbf{G}\neg p))$. It specifies a temporal requirement for GIOP, the general Inter-Object Request Brokers (ORB) Protocol [14]. In L_1, t stands for a request being sent, and p stands for an agent receiving a reply. Semantically, L_1 holds only when an agent would never receive any reply until a request

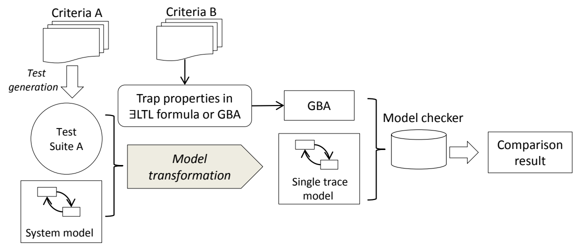
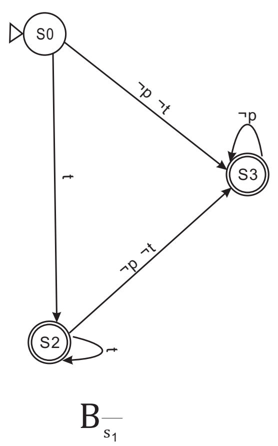


Figure 2. Cross comparison procedure

has been made. To produce test cases that strongly covers state s_1, simply removing s_1 from the automaton is sufficient to get the SE-GBA $B_{\overline{s_1}}$, as shown on the left side in Figure 4. Then we take the complement automaton $\neg B_{\overline{s_1}}$ as the "trap automaton". A counterexample produced by SPIN for the model-checking problem on "trap automaton" may be translated to a test case strongly covering s_1.

To generate a test case weakly covering s_1, the original GBA B is transformed to a SM-GBA $B(s_1)$ as the "trap automaton" (Figure 4). To construct $B(s_1)$, B is duplicated, then transitions from s_1 to s_5 and s_6 are also added, corresponding to the ones from s_1 to s_1 and s_2 in B. Another change is that the acceptance states, marked as double circled states, are all moved to the new automaton. Thus, any successful run of $B(s_1)$ must go through s_1, hence satisfies the requirements of weak state coverage.

It shall be noted that the graph transformation process changes the semantics of the original automaton. While the original GBA is equivalent to an LTL formula, after the transformation, we essentially treat the new automaton as an equivalent to a ∃LTL formula, since our objective is to find a linear counterexample, which can be used as the basis for a test case.



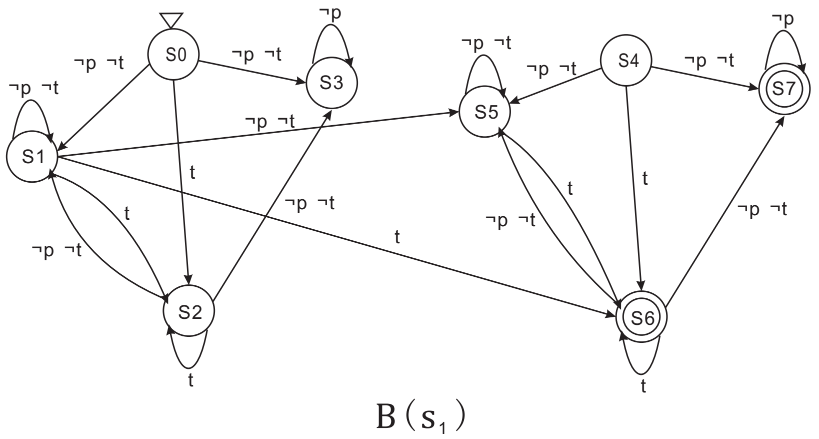


Figure 4. SE-GBA $B_{\bar{s_1}}$ and SM-GBA $B(s_1)$

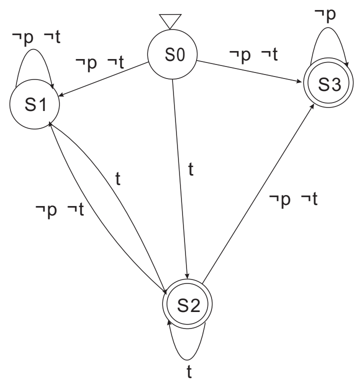


Figure 3. GBA B for L_1

We compare the practical performances of two criteria by measuring cross coverage. Figure 2 shows the workflow for measuring cross coverage between two criteria. The basic approach is to first generate a test suite for one test criterion, and then test the system using the generated test suite and measure the coverage against the other criterion. We conduct the experiments in the following ways. A script written in Java interprets the lasso-shaped counterexample produced and record each step it has taken along the way. Then the script transforms the system model accordingly into a new model that have only one possible execution path, which is identical to the counterexample. The execution path is one possible execution of the system under a test case extracted from the counterexample. By model checking a single-trace model against the trap properties of the other criterion, we may measure the coverage of a test case with respect to the other criterion. The cross coverage measures to what degree a test suite generated for one criterion may achieve the other test criterion. It serves as an indicator for a comparison of the practical effectiveness of two test criteria.

5 Experiment Results

Table 1 shows the result data for the cross-coverage experiments among the criteria. The first model we used in the experiment describes the general Inter-ORB Protocol (GIOP), a key component of the Object Management Group (OMG)'s Common Object Request Broker Architecture (CORBA) specification [14]. We also have a Promela model of a sliding window protocol, which depicts the behavior of the classic network protocol [20]. The other two examples we used are models of Lamport's Bakery algorithm [17] and Peterson's algorithm [19] for mutual exclusion problem. All of these models have well defined LTL properties as their correctness requirments specifications.

In Table 1, BC/SC/PC/DC stands for the following coverage criteria: branch coverage, Büchi automata state coverage, property coverage and data-flow coverage (all-definition-use path coverage), respectively. SC-str and SC-wk represents the strong and weak variants of Büchi automata state coverage. The left column lists test criteria from which we generate test suites, and the top row lists test criteria by which we measure the cross coverage of these test suites. The figure in the parenthesis represents the percentage of feasible test cases produced with respect to the criteria itself.

The overall result indicates that the two specification-based coverage criteria show more competent performance, and in most cases have better cross-coverage results than the other two traditional criteria. For example, in the experiments for the mutual exclusion algorithms, the Büchi Automata state coverage criterion triumphs with a complete full coverage over all the criteria, with property coverage criterion as a close second. The branch coverage and data-flow coverage are able to achieve a high percentage over each other, mostly because the models are so strictly defined that the branches and definition clear paths are heav-

Table 1. Cross-coverage comparison results

	GIOP					Sliding Window				
	BC	SC-str	SC-wk	PC	DC	BC	SC-str	SC-wk	PC	DC
BC	(100%)	67%	67%	75%	100%	(100%)	50%	75%	75%	61%
SC-str	73%	(100%)	100%	75%	82%	67%	(75%)	75%	75%	70%
SC-wk	77%	100%	(100%)	75%	82%	67%	75%	(75%)	75%	70%
PC	73%	100%	100%	(75%)	78%	72%	100%	100%	(75%)	61%
DC	71%	100%	100%	75%	(100%)	100%	75%	75%	75%	(100%)
	Lamport's Bakery					Peterson				
	BC	SC-str	SC-wk	PC	DC	BC	SC-str	SC-wk	PC	DC
BC	(100%)	33%	100%	60%	70%	(100%)	67%	67%	60%	100%
SC-str	100%	(100%)	100%	100%	100%	100%	(100%)	100%	100%	100%
SC-wk	100%	100%	(100%)	100%	100%	100%	100%	(100%)	100%	100%
PC	75%	100%	100%	(100%)	81%	100%	100%	100%	(100%)	100%
DC	100%	33%	100%	100%	(100%)	100%	67%	67%	80%	(100%)

ily overlapped. However, they fell short on evaluating the critical properties of the algorithms, which is the essence of the models.

One point worth noting, though, is that in the experiment for GIOP and sliding window model, the vacuity-based coverage criteria did not cover the logic branches and the data flow paths perfectly. The reason behind this is for both protocols, the properties being tested only focus on some of the behaviors described in the model. Take the GIOP model for instance, the property is only concerned about the recipient when it is waiting for or receives a message. It does not involve other functionalities of the model. Therefore the generated counterexamples bypassed some code segments and could not cover the branches and paths.

This observation leads to an important conclusion, that the quality of the temporal property plays a significant role in the property-based testing. A property that is more relevant with the model can result in better coverage. On the other hand, the result on cross coverage could prompt engineers to examine and refine a system design and its properties. Our future research includes the goal of developing a strategy of performing property refinement to enhance the testing approach with the help of the outcome of the aforementioned experiments.

Another difference among these criteria, is that while the Büchi automaton state coverage and property coverage criteria are more semantically oriented and better at finding errors, it is slightly more difficult to interprete the results since the test objectives are not directly source related. The syntax-based coverage criteria, however, benefit from their nature of being white-box testing methods, thus more suitable for debugging.

In general, the approach we present here can be viewed as a unified framework for evaluating multiple test criteria based on their cross-coverage performances. Many other test criteria can be fit into the framework for comparison purpose. It is also possible to incorporate other related techniques, such as property refinement or debugging strategy so that the framework could be further improved and thus serve a broader purpose.

6 Conclusions

We presented a uniform model-checking-assisted framework for generating test cases for various criteria and then comparing the performance of these criteria. We described in details the methodologies and techniques used in our framework. Our framework is able to incorporate a variety of test criteria, including traditional structure-based coverage criteria (e.g. branch coverage and data flow coverage criteria), syntax-oriented specification-based criteria (e.g. LTL property-based coverage criteria), and semantics-oriented specification-based criteria (e.g. Büchi automaton-based coverage criteria). The framework assesses the preformance of different test criteria by measuring cross coverage of generated test suites. We proposed an approach to streamline test case extraction and cross-coverage measurement. Our results validate the benefits of specification-based criteria used in model-checking-assisted test generation, but the results also indicate that such benefits largely depend on the quality of system specification. For furture works, we will extend the framework to additional test criteria, and also plan to develop a tool that fully automates model-checking-assisted test generation for different criteria, and the cross-coverage measurement.

References

[1] P. E. Ammann, P. E. Black, and W. Majurski. Using model checking to generate tests from specifications. In *In Proceedings of the Second IEEE International Conference on Formal Engineering Methods (ICFEM'98)*, pages 46–54. IEEE Computer Society, 1998.

[2] D. Beyer, A. J. Chlipala, R. Majumdar, T. A. Henzinger, and R. Jhala. Generating tests from counterexamples. In *ICSE'04: Proceedings of the 26th International Conference on Software Engineering*, pages 326–335, Washington, DC, USA, 2004. IEEE Computer Society.

[3] E. Clarke, S. Jha, and Y. Lu. Tree-like counterexamples in model checking. In *Logic in Computer Science,*, 2002.

[4] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, London, UK, 1982. Springer-Verlag.

[5] E. M. Clarke, O. Grumberg, K. L. McMillan, and X. Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. In *Proceedings of the 32nd annual ACM/IEEE Design Automation Conference*, DAC '95, New York, NY, USA, 1995. ACM.

[6] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.

[7] G. Fraser and A. Gargantini. An evaluation of model checkers for specification based test case generation. In *ICST '09: Proceedings of the 2009 International Conference on Software Testing Verification and Validation*, Washington, DC, USA, 2009. IEEE Computer Society.

[8] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. In *Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-7, London, UK, 1999. Springer-Verlag.

[9] M. Heimdahl, S. Rayadurgam, and W. Visser. Specification centered testing. In *Proceedings of the Second International Workshop on Automated Program Analysis, Testing and Verification*, 2001.

[10] G. J. Holzmann. The model checker SPIN. *IEEE Trans. Softw. Eng.*, 23, May 1997.

[11] H. S. Hong, S. D. Cha, I. Lee, O. Sokolsky, and H. Ural. Data flow testing as model checking. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, Washington, DC, USA, 2003. IEEE Computer Society.

[12] H. S. Hong, I. Lee, O. Sokolsky, and H. Ural. A temporal logic based theory of test coverage and generation. In *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS '02, London, UK, 2002. Springer-Verlag.

[13] P. C. Jorgensen. *Software Testing: A Craftsman's Approach*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1995.

[14] M. Kamel and S. Leue. Validation of the general inter-orb protocol (giop) using the spin model-checker. In *In Software Tools for Technology Transfer*. Springer-Verlag, 1998.

[15] Y. kuen Tsay, Y. fang Chen, M. hsien Tsai, K. nien Wu, and W. chin Chan. Goal: A graphical tool for manipulating bchi automata and temporal formulae. In *In Proceedings of TACAS (2007), LNCS 4424*. Springer, 2007.

[16] O. Kupferman and M. Y. Vardi. Vacuity Detection in Temporal Model Checking. *Lecture Notes In Computer Science*, 1999.

[17] L. Lamport. A new solution of dijkstra's concurrent programming problem. *Commun. ACM*, 17, August 1974.

[18] F. Lerda and W. Visser. Addressing dynamic issues of program model checking. In *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[19] G. L. Peterson. Myths about the mutual exclusion problem. *Inf. Process. Lett.*, 12(3), 1981.

[20] L. L. Peterson and B. S. Davie. *Computer Networks: A Systems Approach, 3rd Edition*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[21] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Trans. Softw. Eng.*, 11, April 1985.

[22] S. Rayadurgam and M. P. Heimdahl. Coverage based test-case generation using model checkers. In *Proceedings of the 8th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2001)*. IEEE Computer Society, April 2001.

[23] SC-167 Committee. Software Considerations in Airborne Systems and Equipment Certification. Technical report, Radio Technical Commission for Aeronautics, 1992.

[24] L. Tan. State coverage metrics for specification-based testing with büchi automata. In *Proceedings of the 5th international conference on Tests and proofs*, TAP'11, Berlin, Heidelberg, 2011. Springer-Verlag.

[25] L. Tan, O. Sokolsky, and I. Lee. Specification-based testing with linear temporal logic. IEEE International Conference on Information Reuse and Integration (IEEE IRI-2004), November 2004.