Noname manuscript No.
(will be inserted by the editor)

An Unified Framework for Evaluating Test Criteria in Model-Checking-Assisted Test Case Generation

Bolong Zeng · Li Tan

Received: date / Accepted: date

Abstract Testing is often cited as one of the most costly operations in testing dependable systems [11]. A particular challenging task in testing is test-case generation. To improve the efficiency of test-case generation and reduce its cost, recently automated formal verification techniques such as model checking are extended to automate test-case generation processes. In model-checking-assisted test-case generation, a test criterion is formulated as temporal logical formulae, which are used by a model checker to generate test cases satisfying the test criterion. Traditional test criteria such as branch coverage criterion and newer temporal-logic-inspired criteria such as property coverage criteria [32] are used with model-checking-assisted test generation. Two key questions in model-checking-assisted test generation are how *efficiently* a model checker may generate test suites for these criteria and how *effective* these test suites are. To answer these questions, we developed a unified framework for evaluating (1) the effectiveness of the test criteria used with model-checking-assisted test-case generation and (2) the efficiency of test-case generation for these criteria. The benefits of this work are three-fold: first, the computational study carried out in this work provides some measurements of the effectiveness and efficiency of various test criteria used with model-checking-assisted test case generation. These performance measurements are important factors to consider when a practitioner selects appropriate test criteria for an application of model-checking-assisted test generation. Second, we propose a unified test generation framework based on *generalized Büchi automata*. The framework uses the same model checker, in this case, SPIN model checker [13], to generate test cases for different criteria and compare them on

Bolong Zeng
School of Electrical Engineering and Computer Science, Washington State University, Richland, WA 99352
E-mail: bzeng@wsu.edu

Li Tan
School of Electrical Engineering and Computer Science, Washington State University, Richland, WA 99352
E-mail: litan@tricity.wsu.edu

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 would be of interest to researchers and practitioners who want to use and extend this unified framework and its accompanying tools.

1 Introduction

Nowadays model checking [5] become an important player in the field of verification and validation (V&V) technologies. It has been widely used in a variety of applications such as verifying control modules in airplanes [27] and in deep-space vehicles [21]. Model checking provides a rigid and mathematically sound proof for the correctness of a variety of systems, many of which are used in safety-critical applications. On the other hand, testing remains an important force in the V&V field and its value is acknowledged by contemporary industrial processes and standards. For example, DO-178B [28], 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-case generation is devised as a strategy for harnessing the synergy of two major verification and validation (V&V) technologies: testing and model checking [1]. To improve the performance of model-checking techniques, over past three decades the model checking community has developed efficient algorithms to explore the state space of a system and reason its temporal behavior. The basic idea behind model-checking-assisted test-case generation is to utilize efficient model-checking algorithms to search for execution paths under a test criterion. These execution paths are then used to synthesize a test suite satisfying the test criterion. A benefit of model-checking-assisted test-case generation is that a model checker may automate the process of test-case generation with better efficiency and at a reduced cost. Additionally an efficient model-checking algorithm can help uncover a test case that otherwise may not be easily constructed manually.

In practice model-checking-assisted test generation makes use of counterexample generation capability provided by many contemporary model checkers [3]. The objectives of a test criterion are encoded as temporal logic properties, often called “trap properties” [9]. Various strategies are proposed to translate existing test criteria to trap properties in various temporal logics[10, 16, 26]. In [32], we proposed a syntax-based approach that extracted trap properties from a specification in Linear Temporal Logic. Our approach was based on the notion of “vacuity” [19]. In [29], we proposed a semantic approach to synthesize trap properties from temporal specification in Büchi automaton.

A question for model-checking-assisted test generation is its practical performance. This question can be further refined to two research questions: (i) how *effective* are test cases generated from different test criteria, and (ii) how *efficiently* a model checker can generate a test suite for a test criterion. Besides theoretical interests, studies on these questions are of practical importance: they would provide heuristic to practitioners on how to select test criteria to achieve desired test effectiveness, given time and resource constraints.

The main motivation behind our work is to develop a unified framework for evaluating and comparing the performance of various test criteria in context of model-checking-assisted test case generation. Our proposed framework addresses the aforementioned two questions. Specifically, the framework enables us to measure the efficiency of a model checker generating test cases for a given test criterion, and to assess the effectiveness of a given test criterion by measuring cross-coverage percentage with respect to other test criteria. The unified test framework also allows us to compare on a consistent basis test criteria used with model-checking-assisted test generation, yielding a less-biased comparison result.

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-case 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

Because of its potential benefits, model-checking-assisted test-case generation receives increasing interests in past several years in academia and industry practices. Commercial tools such as Simulink Design Verifier [22] are available to generate test cases for industrial designs using bounded model checking technique. In general, model-checking-assisted test-case generation utilizes the witness/counterexample generation capability of model checkers to generate test cases. Some of the early works on testing with model checking include [8], in which the *test objectives* are manually specified in the form of *never claims* for the underlying model checker. The approach has since evolved to become more systematic and provide a push-button solution for generating test cases using model checkers. The process may be briefly summarized as such: first, one needs to identify test objectives that define the characteristics of expected test cases. For example, a test objective could be a requirement on the final output of an execution, or a sequence of steps during the execution. In practice test objectives may be derived from a test criterion, which describes the overall goal of a test suite. A test objective is then translated to a temporal logic formula, often referred to as a “trap property”. A model checker is then deployed to examine a system model and search for an execution path satisfying such a trap property. The execution path is then used as the basis for constructing a test case that causes an implementation of the system to run the execution path. In practice, a trap property is often negated to form a *never-claim*, and a model checker such as a SPIN [14] is used to generate a counterexample for the *never-claim*, which is essentially a witness for the trap property [10]. It shall be noted that in some other works, such as [9], the authors refer to the never-claims used to extract counterexamples as trap properties. To avoid ambiguity, in this paper we refer to a trap property in its positive term, that is, a trap property describes the property that shall hold for an expected test case.

The problem then shifts to how to synthesize these trap properties. Coverage criteria are standards used in testing process to evaluate how thorough a system is tested under a certain set of test cases (a test suite). A coverage criterion is defined with respect to certain aspects of a system or a specification, such as statements and branches in a software program. If an objective described in the criterion is executed by a test case, we say it is *covered* by the test case. If every objectives of a test criterion is covered by some test cases in a test suite, the test suite achieves the full coverage for this criterion. In our study, we compare an array of existing coverage criteria and property-based coverage criterion in context of model-checking-assisted test-case generation. These coverage criteria serve as the basis for synthesizing test objectives and trap properties used by model checkers.

In our unified framework we will take into consideration different strategies that are used to synthesize test objectives from a coverage criterion. A variety of strategies [10,16,26] has been proposed to translate an existing coverage criterion to a set of trap properties. In [1], the authors presented the idea of constructing their implementation around mutation analysis to achieve coverage towards the Software Cost Reduction (SCR) specfications. With respect to the same standards, [10] developed the mechanism of extracting trap properties automatically from the specifications, by turning the operational specifications into an SPIN or SMV model, and then forming the never-claims from the SCR requirement properties.

Traditional coverage criteria may be used with model-checking-assisted test generation. Fraser *et al.* [9] studied various test criteria in the context of model-checking-assisted test-case generation. Structural coverage criteria are among the most frequently used coverage criteria in testing. [12] provided the general framework for generating test cases with respect to these criteria. Control and data flow coverage criteria emphasize on the coverage over the data-flow graph of a system model. Hong *et al.* investigated this particular topic in [15] with four data-flow coverage criteria being taken into consideration. Tan *et al.* proposed in [31,32] a method to extract trap properties from requirement properties in Linear Temporal Logic (LTL). The main motivation behind this work is to examine the property with respect to the notion of vacuity. In addition to some of the test criteria studied in [9], we also include a semantics-oriented coverage criterion [29], which is an extension of the work done in [31,32]. [29] presented the coverage criteria based on the Büchi automaton that is equivalent to the temporal property. It requires that each state in the automaton must be *strongly* and/or *weakly* covered by the test cases. This approach goes further into tackling the semantic of the properties, and is also capable of providing useful information over the quality of the property.

We briefly introduce a few research works that have been done over the years that are most relevant to our study presented in this paper. Most of these aforementioned studies discuss the methodology of conducting model-checking-assisted test case generation, while our focus is to compare these different methodologies in a unified framework, and present a way to evaluate them with respect to their effectiveness and efficiency. Another distinctive feature of our work lays on the formalism that we choose as our underlying representation of a temporal property. In model-checking-assisted test-case generation a trap property is formulated as a temporal property. Several formalisms representing temporal properties have been investigated for the purpose of assisting model-checking-assisted test case generation. In [9], the authors used Computational Tree Logic (CTL) as the underlying temporal logic to specify trap properties. It shall be noted that a counterexample for CTL may not necessarily be linear [4]. In this study we use Büchi automaton as the underlying formalism. For most of the test criteria, we translate them to trap properties in Linear Temporal Logic (LTL), and these properties are then translated to Büchi automata. Büchi-automaton-based coverage criteria [29] are also included in this study. These criteria are already expressed in the form of Büchi automata, thus no further translation is necessary. Using Büchi automaton as the underlying temporal formalism has two benefits: first, test cases used in most of applications are linear. Since a counterexample for a Büchi automaton is linear, a test case obtains its linearity from the counterexample that it is built up. Second, it enables us to compare existing test criteria as well as newer Büchi-automaton-based coverage criteria [29] on the same platform and using the same model checker, eliminating the performance bias that may be caused by using different temporal formalisms.

2 Preliminaries

2.1 Kripke Structure, Traces, and Tests

We use *Kripke Structure* 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 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 each state 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 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 [32].

Definition 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 [29], along with the usual states, transitions and acceptance condition.

Definition 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 that consists of sets of states, forms the acceptance condition. In addition, P is a set of literals, and $\Lambda : S \to 2^P$ is the labeling function that maps each state to a set of literals.

Similarly, we write $s \to s'$ in lieu of $(s, s') \in \Delta$. A run of a generalized Büchi automaton B is an infinite sequence $R = s_0 s_1 ...$ such that $s_0 \in S_0$ and $s_i \to s_{i+1}$ 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 $w \vDash B$ 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) [7]. A property in LTL is a *path formula* defined recursively as follows,

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

The basic semantics of a path formula are defined with respect to a Kripke structure $K = \langle \mathcal{S}, s_0, \to, \mathcal{T} \rangle$. If R is a trace of K and $a \in \mathcal{A}$ is an atomic propositon, we denote $R \vDash_K a$ if and only if $R[0] \in \mathcal{T}(a)$. Atomic proposition `true` may be satisfied 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 [6]. It is further shown that there always exists such witness and counterexample that are lasso-shaped.

Definition 5 Given a trace γ on a Kripke structure K. If $\gamma \vDash_K \phi$ holds on K, γ is a *linear witness* for the $\exists$LTL model-checking problem $\langle \mathbf{E}\phi, K\rangle$ and a *linear counterexample* for LTL model-checking problem $\langle \mathbf{A}\neg\phi, K\rangle$.

3 Coverage Criteria

This section gives a brief introduction of the test criteria we covered in this work. For each criteria, we explain how trap properties, or in the case of Büchi Automata state coverage, "trap automata" are generated. For clarity, in this paper we also 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 Computation Tree Logic (CTL) in [9]. Both LTL and CTL are sublogics of the CTL*. While they share a common subset, there are properties that can be described only by one of them. For the purpose of clarity, we explicitly write the top-level path quantifiers for LTL and its dual ELTL. We denote $\mathbf{A}$, $\mathbf{E}$ to distinguish them syntactically from the path quantifiers naturally presented in a CTL* formula.

3.1 Logic Expression Coverage Criteria

We investigate several logic expression coverage criteria in this study. Branch Coverage (BC) criterion belongs to the logic expression criteria [17], 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,

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

A witness produced for the trap property would indicate the i-th branch is covered. $\mathbf{X}\mathtt{true}$ is added to ensure 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.

Several other logic expression criteria extend the branch coverage criterion to check not only the truth value of a guard, but also the value of each clause of it and/or a particular combination of these values. Our experiments will cover some of these logic expression criteria as well.

Clause Coverage (CC) requires a test suite to cover both true and false outcomes of each clause of every transition guard. We use a boolean flag c_{ij} to represent the j-th boolean clause in the i-th branch, and similarly, the trap property is written as below,

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

Note that in Promela, the modeling language used by our underlying model checker SPIN, the branch transition guard can be a non-Boolean statement, for example, an action of sending message. In this case, c_{ij} is simply b_i, since Promela does not allow more than one clause in a non-boolean guard, and adding the clause after branch would change the semantics of the model. Simply put, such branches are each treated as one simple clause, and using the branch coverage trap property is sufficient. This applies to the clause coverage criteria described below as well.

Complete Clause Coverage (CoCC) is a stronger criterion over clauses than Clause Coverage. It requires a test suite to cover all possible combinations of truth values of each clause. For this criterion, the trap property is written as

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

in which C_i is the conjuction of every clause in the i-th branch being checked with different truth values: $C_i = \bigwedge c_{ij} == v_i$, v_i could be either `true` or `false`.

General Active Clause Coverage (GACC) evaluates the importance of a clause while covering its truth values. It requires that for each clause in the branch guard, there exists a configuration that the clause determines the truth value of the guard. Assume B_i is the i-th branch guard and c_{ij} is the j-th clause, $B_i(c_{ij}, x)$ means using x to substitute c_{ij} in B_i, then the following xor-expression needs to be true: $B_i(c_{ij}, \texttt{true}) \oplus B_i(c_{ij}, \texttt{false})$. Therefore, the trap property is

$$\mathbf{EF}(c_{ij} \wedge (B_i(c_{ij}, \texttt{true}) \oplus B_i(c_{ij}, \texttt{false})) \wedge \mathbf{X}\texttt{true})$$

It is worthwhile noting that the trap properties for the clause coverage criteria could only, and is only necessary to be applied to the boolean guards. In addition, since the branches in Promela is multiple-branch conditional construct, the coverage percentage of the logic expression criteria is calculated with respect to the cases when the branch can be covered, i.e. when the guard's truth value is true. For example, for a guard in the form of $c_1||c_2$ and the CoCC criterion, we consider the cases of $c_1 \wedge \neg c_2$, $\neg c_1 \wedge c_2$ and $c_1 \wedge c_2$, but not $\neg c_1 \wedge \neg c_2$, since the last configuration renders the branch unable to be covered, thus we cannot be sure how the clauses are configured. For the same reason, a branch with only one clause is considered fully covered if it can be evaluated to `true`, instead of being 50% by also considering the case of it being `false`. The "default" branches are treated as branches with one clause for the purpose of consistency.

3.2 Data-flow Coverage Criteria

It has been shown that Data-flow Coverage (DC) criteria may be reduced to model checking problems [15]. One of them emphasizes on covering definition-use pairs in a system [25]. 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:

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

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 [32]. 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 6 (Property-Coverage Metric [32]) 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 7 (Property-Coverage Criterion [32]) (PC) 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 $\square$ defines the polarity of a sub-formula. $\square(\phi) = \texttt{true}$ for a subformula ϕ in f if it is nested in odd number of negations; otherwise $\square(\phi) = \texttt{false}$ [32]. The trap property for a test covering the sub-formula ϕ may be defined as,

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

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. This criterion addresses the notion of vacuity [2, 19]. Aside from the trivial pass introduced by induction as presented in the previous example, there are also other types of vacuous passes [2]. Using the mutation technique on the temporal property prevents the vacuous passes in a logic formula to give trivial result, so that any justification made over the formula has more credibility. For the details of the property-coverage criterion, interested users may refer to [32].

3.4 Büchi Automata State Coverage Criteria

Recently Tan [29] proposed a semantics-oriented testing strategy for requirement specification in Büchi automata. The work is an extension of [31]. Unlike the property-based coverage criterion whose definition is based on the syntactical structure of an LTL formula, the coverage criteria in [29] are based on Büchi automata, which capture the semantics of a linear-time requirement specification.

Definition 8 (Covered States [29]) 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. Two test criteria are derived with respect to strongly or weakly cover the states in the Büchi automaton used to model the trap properties. 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'.

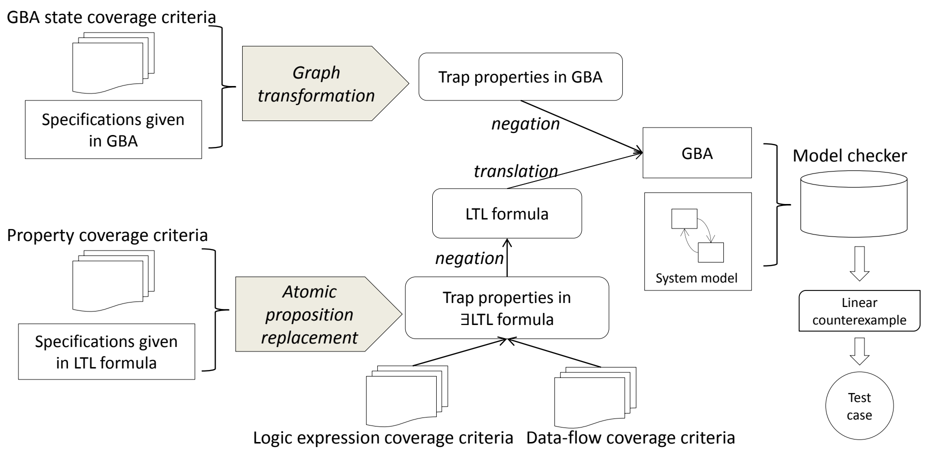
GBA state coverage criteria
Specifications given in GBA
Graph transformation
Trap properties in GBA
negation
GBA
Model checker
translation
LTL formula
System model
Property coverage criteria
negation
Atomic proposition replacement
Trap properties in ∃LTL formula
Specifications given in LTL formula
Linear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

Fig. 1 Test generation procedure

The test criteria developed for Büchi automata incorporate these two variants of state coverage. The quality of a test suite is measured against the percentage of states that it is able to cover. Since a Büchi automaton is essentially a high-level model of a temporal requirement, each state of the automaton contains a certain amount of semantic value of the requirements that are not straightforward. Enforcing and judging whether a test suite covers all the states extends testing to identify problems that are not obvious to even experienced engineers. For the details of SE-GBA, SM-GBA, and their constructions, interested readers may refer to [29]. We refer to the criteria as Büchi Automata State Coverage Criteria (SC).

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 [13] 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 the logic expression 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.

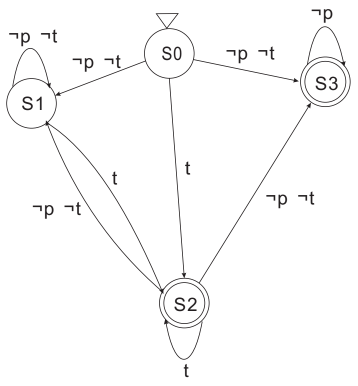

Fig. 2 GBA B for L_1

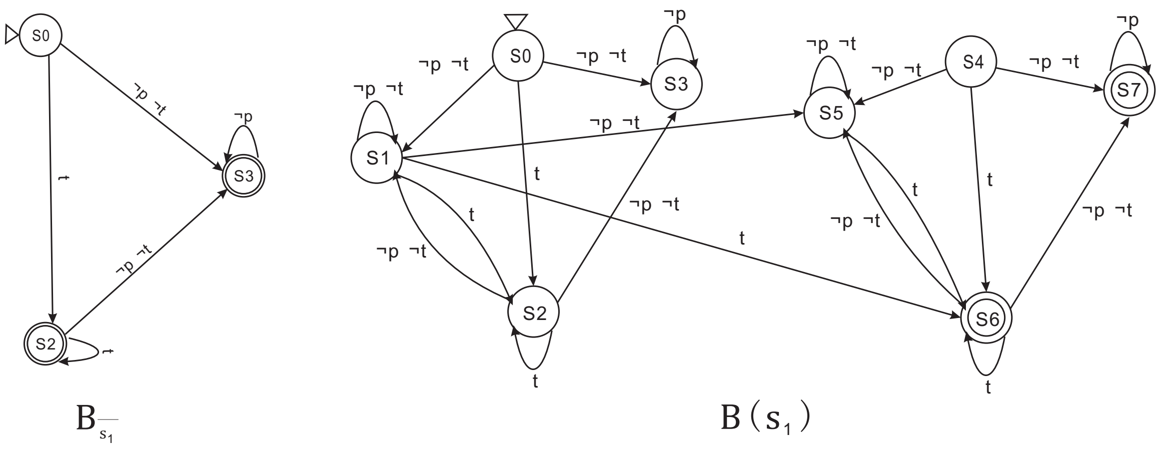

Fig. 3 SE-GBA $B_{\overline{s_1}}$ and SM-GBA $B(s_1)$

For Büchi Automata state coverage, the specifications presented in GBA needs to go through a graph transformation process using Goal [33]. We explain the process with an example taken from our experiments. Figure 2 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 [18]. 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 has been made. To produce test cases that strongly covers one state, for instance 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 3. 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 3). To construct $B(s_1)$, B is

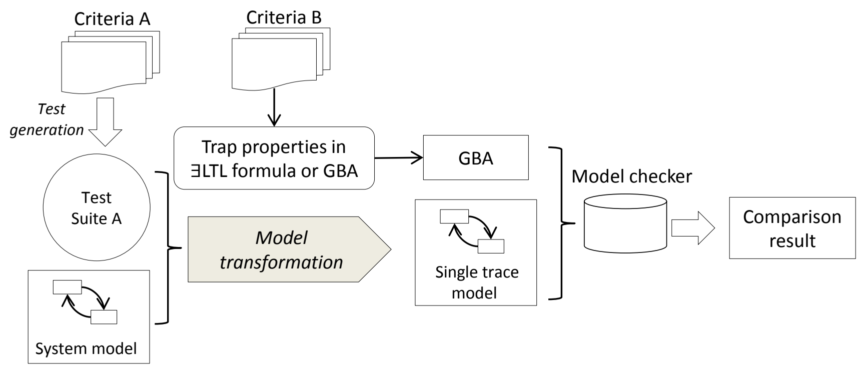

Fig. 4 Cross comparison procedure

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. As a result, any counterexample that is found by the model checker is equivalent to a successful run in the original automaton that goes through s_1, which 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.

We compare the practical performances of two criteria by measuring cross coverage. Figure 4 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 has 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.

Table 1 Cross-coverage comparison results

GIOP

	BC	CC	CoCC	GACC	SC-str	SC-wk	PC	DC
BC	(100%)	100%	100%	100%	67%	67%	75%	100%
SC-str	73%	73%	73%	73%	(100%)	100%	75%	82%
SC-wk	77%	77%	77%	77%	100%	(100%)	75%	82%
PC	73%	73%	73%	73%	100%	100%	(75%)	78%
DC	71%	71%	71%	71%	100%	100%	75%	(100%)

Sliding Window

	BC	CC	CoCC	GACC	SC-str	SC-wk	PC	DC
BC	(100%)	93%	94%	81%	50%	75%	75%	61%
CC	100%	(100%)	94%	81%	50%	75%	50%	51%
CoCC	100%	100%	(100%)	88%	50%	75%	75%	61%
GACC	100%	100%	98%	(92%)	50%	75%	50%	61%
SC-str	67%	81%	55%	63%	(75%)	75%	75%	70%
SC-wk	67%	81%	45%	55%	75%	(75%)	75%	70%
PC	72%	87%	79%	83%	75%	75%	(75%)	61%
DC	100%	93%	86%	60%	75%	75%	75%	(100%)

Lamport’s Bakery

	BC	CC	CoCC	GACC	SC-str	SC-wk	PC	DC
BC	(100%)	100%	95%	89%	33%	100%	60%	70%
CC	100%	(100%)	95%	89%	33%	100%	60%	70%
CoCC	100%	100%	(100%)	95%	33%	100%	60%	70%
GACC	100%	100%	100%	(95%)	33%	100%	60%	70%
SC-str	100%	100%	95%	89%	(100%)	100%	100%	100%
SC-wk	100%	100%	95%	89%	100%	(100%)	100%	100%
PC	75%	70%	65%	55%	100%	100%	(100%)	81%
DC	100%	100%	95%	89%	33%	100%	100%	(100%)

Peterson

	BC	CC	CoCC	GACC	SC-str	SC-wk	PC	DC
BC	(100%)	85%	72%	80%	67%	67%	60%	100%
CC	100%	(100%)	81%	80%	67%	67%	60%	100%
CoCC	100%	100%	(100%)	95%	67%	67%	60%	100%
GACC	100%	100%	81%	(95%)	67%	67%	60%	100%
SC-str	100%	85%	63%	70%	(100%)	100%	100%	100%
SC-wk	100%	85%	63%	70%	100%	(100%)	100%	100%
PC	100%	100%	81%	75%	100%	100%	(100%)	100%
DC	100%	85%	72%	80%	67%	67%	80%	(100%)

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 [18]. We also have a Promela model of a sliding window protocol, which depicts the behavior of the classic network protocol [24]. The other two examples we used are models of Lamport’s Bakery algorithm [20] and Peterson’s algorithm [23] for mutual exclusion problem.

All of these models have well defined LTL properties as their correctness requirments specifications.

In Table 1, the abbreviations in the beginning of each column and each row stands for the coverage criteria introduced in Section 3. In particular, 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. For each criterion, the maximal number of test cases can be generated is equal to the number of the trap properties. The figure in the parenthesis represent the percentage of feasible test cases, i.e. the number of trap properties that the model check is able to find a counterexample against, produced with respect to the criteria itself. For the GIOP model, since every branch in the model is only a single clause guard, which makes the clause coverage criteria equivalent to the branch coverage criterion, we omit the three corresponding rows.

The overall result indicates that the specification-based coverage criteria show more competent performance, and in most cases have better cross coverage results than the other 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 following 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 heavily overlapped. However, they fell short on evaluating the critical properties of the algorithms, which is the essence of the models. Büchi automata state coverage and property coverage criteria, however, are able to generate test cases that are more semantically oriented. Since the models are compactedly defined, these test cases can easily cover the structure elements in the source codes.

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. Such facts are more obviously demonstrated while comparing to the clause coverage criteria. Since even one branch is covered, there are several possible scenarios that each clause in the branch guard might be of different values and possible to be left out.

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. Lower coverage implies that the properties are likely to be relevant with parts of the models. Two directions are possible for the engineers to make use of the results. Either the property could be extended to describe the model's behavior more thoroughly, or it is more suitable to break the model down and focus on the

Table 2 Cross-coverage comparison results for BC+PC

Model	BC	CC	CoCC	GACC	SC-str	SC-wk	PC	DC
GIOP	100%	100%	100%	100%	100%	100%	75%*	100%
Sliding Window	100%	93%	94%	85%	75%	75%	75%*	61%
Lamport's Bakery	100%	100%	95%	89%	100%	100%	100%	85%
Peterson	100%	100%	81%	80%	100%	100%	100%	100%

parts that are correspondent to the 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 interpret 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.

One extension of the experiment is to generate a test suite with respect to two or more test criteria, and then see if such test suite would yield better result in cross-coverage percentage. We run the experiment again on the four models to explore this issue, choosing branch coverage criterion and property coverage criterion as the basis. They represent two different perspectives on testing, one structural, the other semantic-oriented. Therefore, the generated test cases are more likely to explore different aspects of the models. The cross-coverage results of the combined test suite is list in Table 2.

We shall first note that, as shown in Table 2, PC is not fully covered in GIOP and Sliding Window models (marked with an asterisk) by the new test suite. The reason is that for these two models, some of the trap properties of PC are unfeasible to generate test cases. The results show that the new test suite demonstrates an improved cross-coverage percentage over the test suites derived from the stand-alone test criteria. Such outcome should be expected since at the very least, the result would combine the higher percentage from each original test criterion. Furthermore, in some cases, the combined-criteria derived test suite would demonstrate a better coverage than both of the original ones. For instance in the Lamport's bakery algorithm model, the new test suite covers 85% of the data-flow coverage criterion, while the original percentages are 70% and 81% respectively. Such combination can make use of the strength of multiple test criteria, especially when they examine different aspects of the models. In a word, our approach can be used to conduct thorough testing with respect to both the structural elements of the model, and the semantic requirements established via temporal properties.

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. A systematic approach towards these two directions shall be designed to take full advantage of the result. From our observation, the result is able to unveil the effectiveness and strength of the test coverage criteria being used in model-checking-assisted test generation.

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 criterion), 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. We analyze the outcome with respect to the different characteristics of the various test coverage criteria, and explain the rationales behind the outcome.

Current model-checking-assisted test case generation technique mainly uses the existing counter-example generation capability of a model checker as its underlying mechanism. While searching in the state space of a model, a model checker collects much more information beyond a counter-example [30]. The additional information could be utilized to derive a more sophisticated testing strategy. This new testing strategy may guide a user interactively when debugging a system design, or suggesting possible ways of refining temporal properties. This new approach, which we refer to as *evidence-based* test case generation, is a future direction to extend our current work towards a more general model-checking-assisted test case generation framework.

References

1. Ammann, P.E., Black, P.E., Majurski, W.: Using model checking to generate tests from specifications. In: In Proceedings of the Second IEEE International Conference on Formal Engineering Methods (ICFEM'98), pp. 46–54. IEEE Computer Society (1998)
2. Beer, I., Ben-David, S., Eisner, C., Rodeh, Y.: Efficient detection of vacuity in actl formulas. In: Proceedings of the 9th International Conference on Computer Aided Verification, CAV '97. Springer-Verlag, London, UK, UK (1997)
3. Beyer, D., Chlipala, A.J., Majumdar, R., Henzinger, T.A., Jhala, R.: Generating tests from counterexamples. In: ICSE'04: Proceedings of the 26th International Conference on Software Engineering, pp. 326–335. IEEE Computer Society, Washington, DC, USA (2004)
4. Clarke, E., Jha, S., Lu, Y.: Tree-like counterexamples in model checking. In: Logic in Computer Science, (2002)
5. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching-time temporal logic. In: Logic of Programs, Workshop. Springer-Verlag, London, UK (1982)

6. Clarke, E.M., Grumberg, O., McMillan, K.L., Zhao, X.: Efficient generation of counterexamples and witnesses in symbolic model checking. In: Proceedings of the 32nd annual ACM/IEEE Design Automation Conference, DAC '95. ACM, New York, NY, USA (1995)
7. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press (1999)
8. Engels, A., Feijs, L.M.G., Mauw, S.: Test generation for intelligent networks using model checking. In: Proceedings of the Third International Workshop on Tools and Algorithms for Construction and Analysis of Systems, TACAS '97. Springer-Verlag, London, UK, UK (1997)
9. Fraser, G., Gargantini, A.: 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. IEEE Computer Society, Washington, DC, USA (2009)
10. Gargantini, A., Heitmeyer, C.: 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. Springer-Verlag, London, UK (1999)
11. Heimdahl, M., Rayadurgam, S., Visser, W.: Specification centered testing. In: Proceedings of the Second International Workshop on Automated Program Analysis, Testing and Verification (2001)
12. Heimdahl, M.P., Rayadurgam, S., Visser, W.: Specification centered testing. In: Second International Workshop on Analysis, Testing and Verification. Toronto, Canada (2001)
13. Holzmann, G.J.: The model checker SPIN. IEEE Trans. Softw. Eng. **23** (1997)
14. Holzmann, G.J.: The Model Checker {SPIN}. IEEE Transactions on Software Engineering **23**(5), 279–295 (1997)
15. Hong, H.S., Cha, S.D., Lee, I., Sokolsky, O., Ural, H.: Data flow testing as model checking. In: ICSE '03: Proceedings of the 25th International Conference on Software Engineering. IEEE Computer Society, Washington, DC, USA (2003)
16. Hong, H.S., Lee, I., Sokolsky, O., Ural, H.: 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. Springer-Verlag, London, UK (2002)
17. Jorgensen, P.C.: Software Testing: A Craftsman's Approach, 1st edn. CRC Press, Inc., Boca Raton, FL, USA (1995)
18. Kamel, M., Leue, S.: Validation of the general inter-orb protocol (giop) using the spin model-checker. In: In Software Tools for Technology Transfer. Springer-Verlag (1998)
19. Kupferman, O., Vardi, M.Y.: Vacuity Detection in Temporal Model Checking. Lecture Notes In Computer Science p. 82 (1999)
20. Lamport, L.: A new solution of dijkstra's concurrent programming problem. Commun. ACM **17** (1974)
21. Lerda, F., Visser, W.: Addressing dynamic issues of program model checking. In: Lecture Notes in Computer Science. Springer-Verlag (2001)
22. MathWorks: Simulink Design Verifier (2007). URL http://www.mathworks.com/products/sldesignverifier/
23. Peterson, G.L.: Myths about the mutual exclusion problem. Inf. Process. Lett. **12**(3) (1981)
24. Peterson, L.L., Davie, B.S.: Computer Networks: A Systems Approach, 3rd Edition. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2003)
25. Rapps, S., Weyuker, E.J.: Selecting software test data using data flow information. IEEE Trans. Softw. Eng. **11** (1985)
26. Rayadurgam, S., Heimdahl, M.P.: 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 (2001)
27. Rushby, J.: Using model checking to help discover mode confusions and other automation surprises. Reliability Engineering and System Safety **75**(2) (2002)
28. SC-167 Committee: Software Considerations in Airborne Systems and Equipment Certification. Tech. rep., Radio Technical Commission for Aeronautics (1992)
29. Tan, L.: State coverage metrics for specification-based testing with büchi automata. In: Proceedings of the 5th international conference on Tests and proofs, TAP'11. Springer-Verlag, Berlin, Heidelberg (2011)
30. Tan, L., Cleaveland, R.: Evidence-based model checking. In: In Computer-Aided Verification. Springer-Verlag (2002)
31. Tan, L., Sokolsky, O., Lee, I.: Property-coverage testing. Tech. rep., Department of Computer and Information Science, University of Pennsylvania (2003)

32. Tan, L., Sokolsky, O., Lee, I.: Specification-based Testing with Linear Temporal Logic. In: the proceedings of IEEE Internation Conference on Information Reuse and Integration (IRI'04). IEEE society (2004)
33. kuen Tsay, Y., fang Chen, Y., hsien Tsai, M., nien Wu, K., chin Chan, W.: Goal: A graphical tool for manipulating bchi automata and temporal formulae. In: In Proceedings of TACAS (2007), LNCS 4424. Springer (2007)