State Coverage Metrics for Specification-Based Testing with Büchi Automata

Li Tan

School of Electrical Engineering and Computer Science
Washington State University, Richland, WA 99354
litan@wsu.edu

Abstract. Büchi automata have been widely used for specifying linear temporal properties of reactive systems and they are also instrumental for designing efficient model-checking algorithms. In this paper we extend specification-based testing to Büchi automata. A key question in specification-based testing is how to measure the quality (relevancy) of test cases with respect to system specification. We propose two state coverage metrics for measuring how well a test suite covers a Büchi-automaton-based requirement. We also develop test generation algorithms that use counter-example generation capability of an off-the-shelf model checker to generate test cases for the coverage criteria inferred by these metrics. In our experiment we demonstrate the feasibility and performance of the coverage criteria and test generation algorithms for these criteria. In [13] we proposed testing coverage metrics and criteria for properties in Linear Temporal Logic (LTL) and referred to the new approach as property-coverage testing. This research shares the same motivation as in [13] and it extends property-coverage testing to specifications in Büchi automata. Since automaton minimization techniques can be used to reduce the structural diversity of semantically equivalent Büchi automata, we argue that a coverage metric based on Büchi automata is less susceptible to syntactic changes of a property than a LTL-based coverage metric, and hence the proposed coverage metrics measure the relevancy of a test suite to the semantics of a linear temporal property. We also discuss an algorithm for refining a Büchi-automaton-based requirement based on its strong state coverage metric. Our experiment demonstrates the feasibility and performance of our coverage criteria and test generation algorithms.

1 Introduction

Testing and formal verification are considered as two important and yet complemental methods in verifying and validating reactive systems. Reactive systems refer to those dynamic systems that continuously operate and interact with their environment. Examples of reactive systems include engine control modules (ECMs) in automobiles, and autopilot modules in airplanes. Ensuring the correct functioning of reactive systems is of uttermost interest to automobile, aerospace, and many other industries where reactive systems are widely deployed in safe-

and/or mission-critical applications. Whereas testing is to check the behavior of a system under a controlled environment, formal verification is to algorithmically analyze a system. A critic of testing may be best summarized by Dijkstra's notable statement "testing shows the presence, not the absence of bugs". Nevertheless, despite this shortcoming, testing can work where formal verification stops short. Compared with formal verification, testing usually has a better scalability, and it can be applied to implementation directly, for instance, in a setting such as hardware-in-loop testing. In the foreseeable future testing will continue to play a predominant role on validating and verifying reactive systems.

A direction of our research is to study how to harness the synergy of testing and formal verification. For instance, formal verification techniques such as model checking (cf. [3]) have enjoyed a great deal of successes in past two decades. As the result, rigorous formalisms such as temporal logics and Büchi automata are increasingly popular for specifying requirements for high-dependable reactive systems. A research question is how testing can benefit from the proliferation of these high-quality formal specifications. One of important formal verification techniques is linear temporal model checking (cf. [15]), in which a system design is checked against a linear temporal property. Büchi automata have been used for specifying linear temporal properties. Since other formalisms such as Linear Temporal Logic (LTL) can be translated into Büchi automata, Büchi automata are also used as a unified theoretical platform for reasoning about linear temporal model checking algorithms. For instance, efficient linear temporal model checking algorithms such as those in [5, 6] have been proposed with the use of Büchi automata.

The purpose of this research is to develop a framework for specification-based testing with Büchi automata. We will address the following issues:

1. We need to define the relevancy of a test case to a specification in a Büchi automata. For this purpose, we propose two variants of state coverage metrics that measure how a Büchi automaton is covered during a test. A weak variant indicates that a particular state *may* be reached during a test, whereas a strong variant requires that the state *must* be reached during a test;
2. We need to develop a practical way to produce test cases for the testing criteria inferred by the proposed metrics. For this purpose, we propose the algorithms that can use the counterexample mechanism of an off-the-shelf model checker to generate test cases for state coverage criteria.

In addition, we consider property refinement based on the proposed metrics. Lack of state coverage may be caused by incorrect/incomplete implementation, and/or imprecise/loose specification. Whereas testing helps identify the first problem, a careful examination and refinement of the specification will address the latter issue. We will discuss how to enhance our test generation algorithms to systematically refine a property expressed as a Büchi automaton.

The rest of the paper is organized as follows: Section 2 prepares the notations that will be used in the rest of the paper; Section 3 proposes two variants of state coverage metrics for Büchi automata; Section 4 gives the algorithms for generating tests for state coverage using a model checker; Section 5 discusses the property refinement based on state coverage metrics; Section 6 discusses the experimental results; and finally Section 7 concludes the paper.

Related Works Test generation using model checkers attracted much research efforts in recent years as a way to harness the synergy of testing and model checking. The first and foremost question in model-checking-based test generation is how to formulate test generation as a model checking problem, more specifically, how to encode a testing criterion as a temporal property accepted by a model checker. It has been shown that traditional structural coverage criteria such as Modified Condition/Decision Coverage (MCDC) can be encoded in Computational Tree Logic (CTL) (cf. [4]), which can be used by a model checker such as NuSMV for generating tests. In [2] Calvagna et al encoded several combinatorial testing criteria in Linear Temporal Logic (LTL), which was then used by the model checker SAL to generate tests. In [8] Hong et al expressed dataflow criteria in CTL. These previous works emphasize on applying model-checking-based test generation to existing testing criteria. In contrast, in [13] we considered specification-based with temporal logic requirements. We proposed a coverage metric that measures how well a linear temporal logic requirement was covered during a test. Our coverage metric in [13] is inspired by the notion of (non-) vacuity in [10]. Intuitively the vacuity-based coverage criterion requires that a test suite tests the relevancy of each subformula of a LTL property to a system. For a comparison of these techniques, interested readers may refer to [4].

This work can be seen as an extension of our previous work in [13]. Whereas our vacuity-based coverage metric helps define and develop test cases relevant to temporal specifications in LTL, a feature but also a critic of the coverage metric is that it heavily depends on syntactical structures of LTL formulae. For example, the LTL formula $f_0 : G(brake \Rightarrow F\ stop) \wedge (brake \Rightarrow F\ stop)$ is semantically equivalent to $f_1 : G(brake \Rightarrow F\ stop)$. Yet for vacuity-based coverage metric, the coverage of a test case for f_0 always subsumes its coverage for f_1. Defining coverage metrics based on Büchi automata will help alleviate this syntactical dependency. Moreover, there are several existing algorithms for minimizing Büchi automata, which can be used as a preprocess to further reduce syntactical variance of otherwise semantically equivalent Büchi automata. In addition, in this paper we will also discuss property refinement based on the proposed coverage metrics.

2 Preliminaries

2.1 Kripke Structures, Traces, and Tests

We model systems as *Kripke structures.* A Kripke structure is a finite transition system in which each state is labeled with a set of atomic propositions. Semantically atomic propositions represent primitive properties held at a state. Definition 1 formally defines Kripke structures.

Definition 1 (Kripke Structures). *Given a set of atomic proposition $\mathcal{A}$, a Kripke structure is a tuple $\langle V, v_0, \rightarrow, \mathcal{V} \rangle$, where V is the set of states, $v_0 \in V$ is* *the start state, $\rightarrow \subseteq V \times V$ is the transition relation, and $\mathcal{V} : V \rightarrow 2^{\mathcal{A}}$ labels each state with a set of atomic propositions.*

We write $v \rightarrow v'$ in lieu of $\langle v, v' \rangle \in \rightarrow$. We let $a, b, \cdots$ range over $\mathcal{A}$. We denote $\mathcal{A}_{\neg}$ for the set of negated atomic propositions. Together, $P = \mathcal{A} \cup \mathcal{A}_{\neg}$ defines the set of *literals*. We let $l_1, l_2, \cdots$ and $L_1, L_2, \cdots$ range over P and 2^P, respectively.

We use the following notations for sequences: let $\beta = v_0 v_1 \cdots$ be a sequence, we denote $\beta[i] = v_i$ for i-th element of β, $\beta[i, j]$ for the subsequence $v_i \cdots v_j$, and $\beta^{(i)} = v_i \cdots$ for the i-th suffix of β. A *trace* τ of the Kripke structure $\langle V, v_0, \rightarrow, \mathcal{V} \rangle$ is defined as a maximal sequence of states starting with v_0 and respecting the transition relation $\rightarrow$, i.e., $\tau[0] = v_0$ and $\tau[i-1] \rightarrow \tau[i]$ for every $i < |\tau|$. We also extend the labeling function $\mathcal{V}$ to traces: $\mathcal{V}(\tau) = \mathcal{V}(\tau[0])\mathcal{V}(\tau[1]) \cdots$.

Definition 2 (Lasso-Shaped Sequences). *A sequence τ is* lasso-shaped *if it has the form $\alpha(\beta)^\omega$, where α and β are finite sequences. $|\beta|$ is the* repetition factor *of τ. The length of τ is a tuple $\langle |\alpha|, |\beta| \rangle$.*

Definition 3 (Test and Test Suite). *A test is a word on $2^{\mathcal{A}}$, where $\mathcal{A}$ is a set of atomic propositions. A test suite ts is a finite set of test cases. A Kripke structure $K = \langle V, v_0, \rightarrow, \mathcal{V} \rangle$ passes a test case t if K has a trace τ such that $\mathcal{V}(\tau) = t$. K passes a test suite ts if and only if it passes every test in ts.*

2.2 Generalized Büchi Automata

Definition 4. *A generalized Büchi automaton is a tuple $\langle S, S_0, \Delta, \mathcal{F} \rangle$, in which S is a set of states, $S_0 \subseteq S$ is the set of start states, $\Delta \subseteq S \times S$ is a set of transitions, and the acceptance condition $\mathcal{F} \subseteq 2^S$ is a set of sets of states.*

We write $s \rightarrow s'$ in lieu of $\langle s, s' \rangle \in \Delta$. A generalized Büchi automaton is an ω-automaton. That is, it can accept the infinite version of regular languages. A run of a generalized Büchi automaton $B = \langle S, S_0, \Delta, \mathcal{F} \rangle$ is an infinite sequence $\rho = s_0 s_1 \cdots$ such that $s_0 \in S_0$ and $s_i \rightarrow s_{i+1}$ for every $i \geq 0$. We denote $\inf(\rho)$ for a set of states appearing for infinite times on ρ. A successful run of B is a run of B such that for every $F \in \mathcal{F}$, $\inf(\rho) \cap F \neq \emptyset$.

The plain version of generalized Büchi automata in Definition 4 defines successful runs as sequences of states. In order to accept infinite words, we need to extend a generalized Büchi automaton with an alphabet. The alphabet is a set of sets of atomic propositions. It shall be noted that there are more than one way to extend a generalized Büchi automaton with an alphabet. One may label states with a set of sets of atomic propositions, as in [5], or label transitions with a set of sets of atomic propositions, as in [6]. Just like Moore and Mealy versions of finite systems, these two representations are equivalent. In this paper we will use state labeling approach in [5] with one modification: instead of labeling a state with a set of sets of atomic propositions in [5], we label the state with a set of literals. A set of literals is a succinct representation of a set of sets of atomic propositions: let L be a set of literals labeling state s, then semantically s is labeled with a set of sets of atomic propositions $\Lambda(L)$, where

$\Lambda(L) = \{A \subseteq \mathcal{A} \mid (A \supseteq (L \cap \mathcal{A})) \wedge (A \cap (L \cap \mathcal{A}_\neg) = \emptyset)\}$, that is, every set of atomic propositions in $\Lambda(L)$ must contain all the atomic propositions in L but none of its negated atomic propositions. In the rest of the paper, generalized Büchi automata (GBA) refer to labeled generalized Büchi automata in Definition 5.

Definition 5. *A labeled generalized Büchi automaton is a tuple $\langle P, S, S_0, \Delta, \mathcal{L}, \mathcal{F}\rangle$, in which $\langle S, S_0, \Delta, \mathcal{F}\rangle$ is a generalized Büchi automaton, P is a set of literals, and the label function $\mathcal{L} : S \to 2^P$ maps each state to a set of literals.*

A GBA $B = \langle \mathcal{A} \cup \mathcal{A}_\neg, S, S_0, \Delta, \mathcal{L}, \mathcal{F}\rangle$ accepts infinite words over the alphabet $2^{\mathcal{A}}$. Let α be a word on $2^{\mathcal{A}}$, B has a run ρ induced by α, written as $\alpha \vdash \rho$, if and only if for every $i < |\alpha|$, $\alpha[i] \in \Lambda(\mathcal{L}(\rho[i]))$. B accepts α, written as $\alpha \models B$ if and only if B has a successful run ρ such that $\alpha \vdash \rho$.

Generalized Büchi automata are of special interests to the model checking community. Because a GBA is an ω-automaton, it can be used to describe temporal properties of a finite-state reactive system, whose executions are infinite words of an ω-language. Formally, a GBA accepts a Kripke structure $K = \langle V, v_0, \to, \mathcal{V}\rangle$ if for every trace τ of K, $\mathcal{V}(\tau) \models B$. Efficient Büchi automaton-based algorithms have been developed for linear temporal model checking. The process of linear temporal model checking generally involves translating the negation of a linear temporal logic property ϕ to a GBA $B_{\neg\phi}$, and then checking the emptiness of the product of the GBA and the Kripke structure K. If the product automaton is not empty, then a model checker usually outputs an accepting trace of the product automaton, which serves as a counterexample to $K \models \phi$.

3 State Coverage for Generalized Büchi Automata

Definition 6 (Covered States). *Given a generalized Büchi automaton $B = \langle P, S, S_0, \Delta, \mathcal{L}, \mathcal{F}\rangle$, a test t weakly covers a state s if B has a successful run ρ such that $t \vdash \rho$ and s is on ρ. A test t strongly covers a state s if B accepts t and for B's every successful run ρ such that $t \vdash \rho$, s is on ρ.*

Since a GBA B may have nondeterministic transitions, B may have more than one successful run induced by a test. A weakly covered state s shall appear on some successful run induced by t, whereas a strongly covered state s has to appear on every successful run induced by t. By imposing the additional requirement that B accepts t, Definition 6 also requires that at least one of such successful runs exists for a state s strongly covered by t.

Definition 7 (Weak State Coverage Metrics and Adequacy Criterion). *Given a generalized Büchi automaton $B = \langle P, S, S_0, \Delta, \mathcal{L}, \mathcal{F}\rangle$, let $\mathcal{S} \subseteq S$ be a set of states, the weak state coverage metric for a test suite ts on $\mathcal{S}$ is defined as $\frac{|\mathcal{S}'|}{|\mathcal{S}|}$, where $\mathcal{S}' = \{s \mid s \in \mathcal{S} \wedge \exists t \in ts.(t \text{ weakly covers } s)\}$. ts weakly covers $\mathcal{S}$ if and only if $\mathcal{S}' = \mathcal{S}$.*

Definition 8 (Strong State Coverage Metrics and Adequacy Criterion). *Given a generalized Büchi automaton $B = \langle P, S, S_0, \Delta, \mathcal{L}, \mathcal{F} \rangle$, let $\mathcal{S} \subseteq S$ be a set of states, the strong state coverage metric for a test suite ts on $\mathcal{S}$ is defined as $\frac{|\mathcal{S}'|}{|\mathcal{S}|}$, where $\mathcal{S}' = \{s \mid s \in \mathcal{S} \wedge \exists t \in ts.(t \text{ strongly covers } s)\}$. ts strongly covers $\mathcal{S}$ if and only if $\mathcal{S}' = \mathcal{S}$.*

Theorem 1 shows that the strong state coverage criterion subsumes the weak path coverage criterion.

Theorem 1. *Let $\mathcal{S}$ be a set of states of a GBA $B = \langle P, S, S_0, \Delta, \mathcal{L}, \mathcal{F} \rangle$, if a test suite ts strongly covers $\mathcal{S}$, then ts also weakly covers $\mathcal{S}$.*

Proof. Since ts strongly covers $\mathcal{S}$, by Definition 8, for every $s \in \mathcal{S}$, there exists a t such that (i) B accepts t; and (ii) for every B's successful run ρ such that $t \vdash \rho$, s is on ρ. By (i) and (ii), B has at least one successful run ρ such that $t \vdash \rho$ and s on ρ. Therefore, by Definition 8, ts also weakly covers $\mathcal{S}$. □

4 Model-Checking-Based Test Generation

Model checking via generalized Büchi automata is a well-studied subject and efficient algorithms have been developed over years (cf. [5]). We consider the approach that uses a Büchi automaton-based model checker to generate test cases for model-based testing. Model-based testing is an important component in the workflow of model-based design. In model-based design, engineers work on mathematical models of systems. These mathematical models are the abstractions of finish products. Model-based design helps improve the quality of finish products by supporting verification and validation at early design stage. Model-based testing extends this benefit by supporting efficient test generations from design models and then applying generated test suites to finish products.

The inputs to our test generation algorithms are (a model of) a transition system and a linear temporal property in GBA. A system model is a behaviorial abstraction of the system. In this paper we consider the path abstraction in Definition 9. We denote $K_s \prec K_m$, if K_m is a path abstraction of K_s. In model-based design, a system model exists as part of design artifact so we can use it directly as an input to our test generation algorithms.

Definition 9 (Path Abstraction). *Let $K_m = \langle \mathcal{S}_m, s_{0m}, \rightarrow_m, \mathcal{V}_m \rangle$ and $K_s = \langle \mathcal{S}_s, s_{0s}, \rightarrow_s, \mathcal{V}_s \rangle$ be two Kripke structures, K_m is a path abstraction of K_s, written as $K_s \prec K_m$ if and only if for every trace τ_s of K_s, there is a trace τ_m of K_m such that $\mathcal{V}_s(\tau_s) = \mathcal{V}_m(\tau_m)$.*

Theorem 2. *Given two Kripke structures K_m and K_s such that $K_s \prec K_m$, if K_m passes a test suite ts, then K_s also passes ts.*

Proof. Since K_m passes the test suite ts, for every $t \in ts$, there is a trace τ_m of K_m such that $\mathcal{V}_m(\tau_m) = t$, where $\mathcal{V}_m$ is K_m's labeling function. By Definition

9, there exists a trace τ_s of K_s such that $\mathcal{V}_s(\tau_s) = \mathcal{V}_m(\tau_m)$, therefore, K_s passes t and hence ts. $\square$

By Theorem 2, a test suite generated for a design model shall also be passed by its implementation, if the design model is a path abstraction of the implementation. In model-based testing a test suite is generated from a design model and then applied to the actual implementation. We will generate a test suite by utilizing the counterexample capability of a linear temporal model checker. As we discussed before, the first and foremost question in model-checking-based test generation is to formulate test generation as a model-checking problem. In our case, we need to translate state coverage criteria to temporal properties accepted by a linear temporal model checker. These temporal properties are also referred to as "trapping properties". Since we express temporal properties as GBAs, we will also formulate these "trapping properties" as GBAs. In our approach a model checker takes (the negation of) a trapping property in GBA and a system model, and produces a counterexample that is essentially a trace satisfying a given state coverage. The trapping property for generating a test case weakly covering a state s_m is given as a state marking generalized Büchi automaton (SM-GBA) in Definition 10.

Definition 10 (State Marking Generalized Büchi Automata (SM-GBA)). *Let $B = \langle P, S, S_0, \Delta, \mathcal{L}, \mathcal{F}\rangle$ be a GBA, a state marking generalized Büchi automaton for state $s_m \in S$ is a GBA $B(s_m) = \langle P, S \times \{0,1\}, S_0 \times \{0\}, \Delta', \mathcal{L}', \mathcal{F}'\rangle$, where,*

- $\Delta' = \left(\bigcup_{\langle s,s'\rangle \in \Delta} \{\langle (s,0),(s',0)\rangle, \langle (s,1),(s',1)\rangle\}\right) \cup \left(\bigcup_{\langle s_m,s'\rangle \in \Delta} \{\langle (s_m,0),(s',1)\rangle\}\right)$;
- *For every $s \in S$, $\mathcal{L}'((s,0)) = \mathcal{L}'((s,1)) = \mathcal{L}(s)$;*
- $\mathcal{F}' = \bigcup_{F \in \mathcal{F}} \{F \times \{1\}\}$.

A SM-GBA indexes each state of the original GBA with a number from $\{0,1\}$. The start states are always indexed with 0. The final states are always indexed with 1. The indexing number will be changed from 0 to 1 on the outgoing transitions from the marked state s_m. By the construction of the SM-GBA, the only way that a run can reach an acceptance state from a start state is through state s_m. Therefore, every successful run of the SM-GBA must have s_m on it.

Algorithm 1 generates a test suite that weakly covers all the states of a given GBA. A Büchi-automaton-based linear temporal model checker like SPIN [7] verifies the system model against a linear temporal property in two stages: first it builds a GBA for the negation of a given property, and then it checks the emptiness of the product of the GBA and the system model. If the system model satisfies the linear temporal property, then the product of the GBA and the system model shall be empty, that is, the product does not accept any word. The emptiness checking is at the core of many Büchi-automaton-based model checkers. Algorithm 1 uses the core emptiness checking algorithm of an existing model checker, which is represented by the function *MC_isEmpty*. By our definition function *MC_isEmpty* returns an empty word if the product of the GBA and the system model is empty, otherwise it returns a successful run

Algorithm 1 TestGen_WSC($B = \langle P, S, S_0, \Delta, \mathcal{L}, \mathcal{F}\rangle$, $K_m = \langle \mathcal{S}, s_0, \rightarrow, \mathcal{V}\rangle$)

```
Require: $B$ is GBA, $K_m$ is a system model, and $K_m$ satisfies $B$;
Ensure: Return the test suite $ts$ that weakly covers all the states of $B$ and $K_m$ passes
    $ts$. Return $\emptyset$ if such a test suite is not found;
 1: for every $s \in S$ do
 2:    Construct a SM-GBA $B(s)$ from $B$ that marks the state $s$;
 3:    $\tau = MC\_isEmpty(B(s), K_m)$;
 4:    if $|\tau| \neq 0$ then
 5:       $ts = ts \cup \{\mathcal{V}(\tau)\}$
 6:    else
 7:       return $\emptyset$;
 8:    end if
 9: end for
10: return $ts$;
```

of the product of the GBA and the system model K_m. The test case obtained from this successful run is added to the resulting test suite. Theorem 4 shows the correctness of Algorithm 1.

Theorem 3. *If the test suite ts returned by Algorithm 1 is not empty, then (i) K_m passes ts and (ii) ts weakly covers all the states of B.*

Proof. (i) For each $t \in ts$, there is a related state s, and $MC\_isEmpty(B(s), K_m)$ returns a successful run τ of the product of $B(s)$ and K_m such that $\mathcal{V}(\tau) = t$. Since any successful run of the product of $B(s)$ and K_m shall also be a trace of K_m, τ is also a trace of K_m. Therefore, K_m shall pass t. Furthermore, K_m passes every test case in ts.

(ii) As shown in (i), for each $t \in ts$, there is a related state s and a successful run τ of the product of $B(s)$ and K_m such that $\mathcal{V}(\tau) = t$. We will show that t weakly covers s. By Definition 7, we need to show that there is a successful τ' of B such that τ' goes through the state s. We obtain τ' by taking the projection of τ on the states of B as follows: since τ is a run of the product of $B(s)$ and K_m, each state on τ has the form of $\langle\langle s', i\rangle, v\rangle$, where s' is a state of B, i is an index number from $\{0, 1\}$, and v is a state of K_m. We project state $\langle\langle s', i\rangle, v\rangle$ to state s' on B, and let τ' be the resulting sequence. Clearly τ' is also a successful run of B because, by Definition 10, each transition in $B(s)$ is mapped to a transition in B and each acceptance state in $B(s)$ is mapped to an acceptance state in B. In addition, τ has to go through $\langle s, 0\rangle$ because, by Definition 10, acceptance states of a SM-GBA are indexed by 1, whereas start states are indexed by 0. The only way the index number is changed from 0 to 1 is to go through $\langle s, 0\rangle$. Therefore, τ' has to go through s, and we have proved (ii). $\square$

By Definition 8, a test case t strongly covers a state s of a GBA only if every successful run of the automaton accepting t has to visit state s. To generate such a test case, we will define an automaton whose successful runs are precisely those of the original automaton visiting state s. Such an automaton can be defined

Algorithm 2 TestGen_SSC(B, $K_m = \langle \mathcal{S}, s_0, \rightarrow, \mathcal{V} \rangle$)

```
Require: $B$ is a GBA, $K_m$ is a system model, and $K_m$ satisfies $B$;
Ensure: Return the test suite $ts$ that strongly covers all the states of $B$ and is passed
    by $K_m$. Return $\emptyset$ if such a test suite is not found;
 1: for every $s \in S$ do
 2:   Construct a SE-GBA $B_{\bar{s}}$ for $B$'s state $s$
 3:   $\tau = MC\_isEmpty(\neg B_{\bar{s}}, K_m)$;
 4:   if $|\tau| \neq 0$ then
 5:     $ts = ts \cup \{\mathcal{V}(\tau)\}$
 6:   else
 7:     return $\emptyset$;
 8:   end if
 9: end for
10: return $ts$;
```

as the negation of the state excluding generalized Büchi automaton (SE-GBA) in Definition 11. SE-GBA for state s removes s from the original automaton. Transitions, start states, and acceptance states are updated accordingly to reflect the removal of s. By the construction of the SE-GBA, its successful runs are exactly the subset of the original GBA's successful runs that do not visit s. Therefore, the test cases accepted by the SE-GBA do not strongly cover state s of the original GBA. Algorithm 2 uses the negation of the SE-GBA as an input to the core emptiness checking routine of a model checker to produce a test case with strong state coverage. Theorem 4 shows the correctness of Algorithm 2.

Definition 11 (State Excluding Generalized Büchi Automata (SM-GBA)). *Let $B = \langle P, S, S_0, \Delta, \mathcal{L}, \mathcal{F} \rangle$ be a GBA, a state excluding generalized Büchi automaton for $s \in S$ is a GBA $B_{\bar{s}} = \langle P, S - \{s\}, S_0 - \{s\}, \Delta - \{\langle s', s'' \rangle \in \Delta \mid s' = s \vee s'' = s\}, \mathcal{L}, \bigcup_{F \in \mathcal{F}} \{F - \{s\}\} \rangle$.*

Theorem 4. *If the test suite ts returned by Algorithm 2 is not empty, then (i) K_m passes ts; and (ii) ts strongly covers all the states of B.*

Proof. (i) For each $t \in ts$, there is a related state s, and $MC\_isEmpty(\neg(B_{\bar{s}}), K_m)$ returns a successful run τ of the product of $\neg B_{\bar{s}}$ and K_m such that $\mathcal{V}(\tau) = t$. Since any successful run of the product of $\neg B_{\bar{s}}$ and K_m shall also be a trace of K_m, τ is also a trace of K_m. Therefore, K_m shall pass t. That is, K_m passes every test case in ts.

(ii) As shown in (i), for each $t \in ts$, there is a related state s and a successful run τ of the product of $\neg B_{\bar{s}}$ and K_m such that $\mathcal{V}(\tau) = t$. We will show that t strongly covers s.

First, since τ is a trace of K_m and K_m satisfies B by the precondition of Algorithm 2, B accepts $t = \mathcal{V}(\tau)$.

Next, we will prove by contradiction that every successful run of B induced by the test case t shall visit s at least once: suppose not, and let ρ be a successful run of B induced by t and ρ does not visit s. It follows that ρ shall also be a successful run of $B_{\bar{s}}$, because, with the exception of missing state s, $B_{\bar{s}}$ is the same as B. Therefore, $B_{\bar{s}}$ shall accept t. By Algorithm 2, there is a τ such that $t = \mathcal{V}(\tau)$ and τ is a successful run of the product of $\neg B_{\bar{s}}$ and K_m. It follows that $t = \mathcal{V}(\tau)$ is accepted by $\neg B_{\bar{s}}$ and hence it cannot be accepted by $B_{\bar{s}}$. We reach a contradiction. Therefore, every successful run of B that accepts t shall visit s at least once. $\square$

A generalized Büchi automaton B can be translated to a Büchi automaton by indexing acceptance states. The resulting Büchi automaton has the size $O(|\mathcal{F}| \cdot |B|)$, where $|\mathcal{F}|$ is the number of acceptance state sets in B, and $|B|$ is the size of B. The emptiness checking for a Büchi automaton can be done in linear time (cf. [15]). Therefore, generating a test case weakly covering a state s can be done in $O(|K| \cdot |\mathcal{F}| \cdot |B|)$, where $|K|$ is the size of the model, and generating a test suite weakly covering all the states in B can be done in $O(|K| \cdot |\mathcal{F}| \cdot |B|^2)$. Algorithm 2 starts with the construction of a SE-GBA for a state, which can be done in linear time, and Algorithm 2 then negates SE-GBA. Michel [12] provided a lower bound of $2^{O(nlogn)}$ for negating a Büchi automaton of size n. Therefore, Algorithm 2 takes at least $O(|K| \cdot 2^{O(|\mathcal{F}|\cdot|B|log(|\mathcal{F}|\cdot|\mathcal{B}|)})$ to generate a test case strongly covering a state and at least $O(|K| \cdot 2^{O(|\mathcal{F}|\cdot|B|log(|\mathcal{F}|\cdot|\mathcal{B}|)})$ to generate a test suite strongly covering all the states of a GBA. The reason why generating test cases for strong state coverage is much more expensive than for weak state coverage can be traced back to Definition 6: to strongly cover a state s we need to examine all the successful runs and make sure that they visit s, whereas to weakly cover s we only need to find a single successful run visiting s.

5 Testing-Based Property Refinement

In specification-based testing, the correctness of a system is defined as the system's conformance to its specifications. Inadequate coverage on specification may suggest a problem in a system, but it may also indicate a problem in specification. For example, the specification may be imprecise and/or too general. Besides producing test suites, model-checking-based test generation algorithms also provide valuable information on how a property is related to a system model. Here we consider property refinement using feedbacks from test generation algorithms.

To facilitate our discussion, we need to formalize the notion of "refinement". Language inclusion is a natural candidate for defining a refinement preorder. Formally we define $B \sqsubseteq B'$ if and only if $L(B) \subseteq L(B')$, that is, B is a refinement of B' if the language accepted by B is a subset of the language accepted by B'. By Definition 3, we can infer that a test accepted by B will also be accepted by B'. We use state coverage metrics to guide the property refinement process. The purpose of the refinement is to fine-tune a property so it can more closely describe the behaviors of a system, measured by increased state coverage on the property.

Lemma 1. *Given a GBA $B = \langle P, S, S_0, \Delta, \mathcal{L}, \mathcal{F} \rangle$ with a state $s \in S$, let $B_{\bar{s}} = \langle P, S - \{s\}, S_0 - \{s\}, \Delta', \mathcal{L}, \mathcal{F}' \rangle$ be the state excluding GBA for s, then $B_{\bar{s}} \sqsubseteq B$.*

Proof. By Definition 11 the SE-GBA $B_{\bar{s}}$ misses state s. It follows that the successful runs of $B_{\bar{s}}$ are those that do not visit s in the original GBA. Therefore, $L(B_{\bar{s}}) \subseteq L(B)$ and hence $B_{\bar{s}} \sqsubseteq B$. $\square$

Theorem 5. *Given a GBA $B = \langle P, S, S_0, \Delta, \mathcal{L}, \mathcal{F} \rangle$ with a state $s \in S$ and a Kripke structure $K = \langle V, v_0, \rightarrow, \mathcal{V} \rangle$, let $B_{\bar{s}} = \langle P, S - \{s\}, S_0 - \{s\}, \Delta', \mathcal{L}', \mathcal{F}' \rangle$ be the state excluding GBA for s, if K passes a test case t strongly covering s and $K \models B$, then, $K \not\models B_{\bar{s}}$.*

Proof. We will prove by contradiction. Suppose that $K \models B_{\bar{s}}$. Since K passes t, K has a trace τ such that $\mathcal{V}(\tau) = t$. Since $K \models B_{\bar{s}}$, $t \models B_{\bar{s}}$. Let ρ be a successful runs of $B_{\bar{s}}$ induced by t, that is, $t \vdash \rho$. ρ is also a successful run of B because by Lemma 1 $B_{\bar{s}}$ is a refinement of B. Since ρ does not visit s, t does not strongly covers s, which contradicts to the condition of the theorem. Therefore, $K \not\models B_{\bar{s}}$. $\square$

Definition 12 (Vacuous States). *Given a generalized Büchi automaton $B = \langle P, S, S_0, \Delta, \mathcal{L}, \mathcal{F} \rangle$ and a Kripke structure K, a state s of B is vacuous with respect to K if and only if $K \models B$ implies $K \models B_{\bar{s}}$, where $B_{\bar{s}} = \langle P, S - \{s\}, S_0 - \{s\}, \Delta', \mathcal{L}, \mathcal{F}' \rangle$ is the SE-GBA for s.*

Definition 12 defines vacuous states. Since $B_{\bar{s}}$ is a refinement of B, $K \not\models B$ implies $K \not\models B_{\bar{s}}$. Therefore, Definition 12 indicates that a vacuous state s of a GBA B for a Kripke structure K does not affect whether K satisfies B. That is, if we remove the vacuous state s from B, the outcome of whether the system K satisfies GBA B will stay same. This observation prompts us to introduce the notion of state-coverage-induced refinement: *for a given system and a property in a GBA, if a state of the GBA is vacuous to the system, the state can be removed from the GBA, and the system still satisfies this refinement of the original GBA.*

Corollary 1. *Given a generalized Büchi automaton B and a Kripke structure $K = \langle V, v_0, \rightarrow, \mathcal{V} \rangle$, s is not vacuous with respect to K if and only if $K \models B$ and there exists a test t such that t strongly covers s and K passes t.*

Proof. Note that $K \not\models B$ implies $K \not\models B_{\bar{s}}$ since $L(B_{\bar{s}}) \subseteq L(B)$. By Definition 12, s is not vacuous with respect to K if and only if $K \models B$ and $K \not\models B_{\bar{s}}$. All we need to show that: if $K \models B$, then $K \not\models B_{\bar{s}}$ if and only if there exists a test t strongly covering s and K passes t.

($\Rightarrow$) Since $K \models B$ and $K \not\models B_{\bar{s}}$, there must be a trace τ of K such that (1) B has a successful run ρ such that $\mathcal{V}(\tau) \vdash \rho$, and (2) $B_{\bar{s}}$ does not have a successful run ρ' such that $\mathcal{V}(\tau) \vdash \rho'$. Since $B_{\bar{s}}$ is obtained by removing state s from B, it follows that B's every successful run ρ'' such that $\mathcal{V}(\tau) \vdash \rho''$ shall go through s, otherwise, ρ'' is also a successful run of $B_{\bar{s}}$, which contradicts to the condition of our selection of τ. Let $t = \mathcal{V}(\tau)$, by Definition 6, t strongly covers s and K passes t.

Algorithm 3 State_Refinement($B, K_m = \langle \mathcal{S}, s_0, \rightarrow, \mathcal{V} \rangle$)

.

```
Require: $B$ is a GBA, $K_m$ is a system model, and $K_m$ satisfies $B$;
Ensure: Return a GBA as a refinement of $B$, and a test suite $ts$ that strongly covers
    all the states of the new GBA;
 1: for every $s \in S$ do
 2:    Construct the SE-GBA $B_{\bar{s}}$ for $B$'s state $s$;
 3:    $\tau = MC\_isEmpty(\neg B_{\bar{s}}, K_m))$;
 4:    if $|\tau| \neq 0$ then
 5:       $ts = ts \cup \{\mathcal{V}(\tau)\}$
 6:    else
 7:       $B = B_{\bar{s}}$;
 8:    end if
 9: end for
10: return $\langle B, ts \rangle$;
```

($\Leftarrow$) Since K passes t, K has a trace τ such that $\mathcal{V}(\tau) = t$. Since t strongly covers s, we have (i) $t \models B$, and hence B has a successful run ρ such that $t \vdash \rho$; and (ii) for every successful run ρ' of B such that $t \vdash \rho'$, ρ' goes through s. We will prove by contradiction that $K \not\models B_{\bar{s}}$. Suppose that $K \models B_{\bar{s}}$. It follows that every trace of K shall be accepted by $B_{\bar{s}}$, and hence $B_{\bar{s}}$ has a successful run ρ'' such that $\mathcal{V}(\tau) \vdash \rho''$. Note that $B_{\bar{s}}$ is obtained by removing s from B, ρ'' is also a successful run of B but ρ'' does not visit s. It follows that t cannot strongly cover s because B has a successful run induced by t that does not visit s. We reach a contradiction. Therefore, $K \not\models B_{\bar{s}}$.

□

Corollary 1 shows the relation between the strong state coverage and non-vacuousness of a state in a GBA. It shall be noted that testing alone cannot prove the non-vacuousness of a state of a GBA. This is because the non-vacuousness of a state s of B for a system K requires that s affects the outcome of whether K satisfies B, that is, either $K \models B$ and $K \not\models B_{\bar{s}}$, or $K \not\models B$ and $K \models B_{\bar{s}}$. Since $B_{\bar{s}}$ is obtained by removing s from B, $K \not\models B$ implies $K \not\models B_{\bar{s}}$. The only option left is that $K \models B$ and $K \not\models B_{\bar{s}}$, but testing alone cannot show that K satisfies B. Nevertheless, lack of the strong coverage for a state s indicates that s is a vacuous state for K. If that happens, we can remove s from B without affecting the outcome of whether K satisfies B.

Algorithm 3 refines a GBA while generating a test suite strongly covering states of the new GBA. Algorithm 3 is a modification of Algorithm 2. The difference is at line 7. Instead of returning with a failed attempt in Algorithm 2 when strong state coverage cannot be obtained, Algorithm 2 refines the input GBA by removing vacuous states. The output will be a GBA refined by removing vacuous states with respect to the model, and a test suite for the refined GBA.

6 Experiment

To assess the feasibility and performance of our proposed coverage metrics and test generation algorithms, we test them on three examples using SPIN. The first example is the General Inter-ORB Protocol, a key component of Common Object Request Broker Architecture (CORBA) specification defined by Object Management Group (OMG). GIOP defines the inter-operability between Object Request Brokers (ORBs). The Promela model and the properties are provided by Kamel and Leue [9]. The second application is to generate tests for the Needham-Schroeder Public Key Protocol. The model and the properties are provided by Maggi and Sisto [11]. The last one is the Go-Back-N sliding window protocol as described by Tanenbaum [1]. The model and the properties come from the SPIN website [1]. In all three cases, the properties are initially provided in LTL.

We use GOAL [14] to generate a Büchi automaton from a LTL property, and we also use it to synthesize state marking and state excluding automata required for generating tests for weak and strong state coverage criteria. It shall be noted that SPIN takes an automaton in its negation form through its *never* claim form. For example, the GBA for a LTL property f in its *never* claim form is $\neg B(\neg f)$, where $B(\neg f)$ is the GBA for the LTL property $\neg f$. SPIN produces an error trace if it finds a violation of the property in *never* claim form. Using SPIN, we replace line 3 with $spin(\neg B(f)(s), K_m)$, where $B(f)$ is a Büchi automaton for the LTL property f and $B(f)(s)$ is its corresponding SM-GBA for state s. To generate test cases for strong state coverage, we replace line 3 in Algorithm 2 with $spin(\neg(\neg B(f)_{\bar{s}}), K_m)$. Note that the property is given as its negation in *never* claim form $never\{\neg B(f)_{\bar{s}}\}$. As we discussed in Section 4, negating a Büchi automaton is an expensive computation for test generation not just in its own right but also because the complemented automaton suffers an exponential blow-up [12]. Since SPIN produces an infinite error trace as a lasso-shaped sequence, we measure its length in a form defined in Definition 2. As a comparison, we also measure the coverage of generated test cases using a traditional structural coverage metric, which in our experiment, is branch coverage. We use SPIN version 6.0.1. All the experiments are run on a Dell server with one 2.33 GHz quadcore Xeon 5410 and 8 GB RAM. Table 1 shows the experiment results.

For each model, column 1 of Table 1 specifies the properties that we generate test cases for. We refer to these properties by their names originally used by their respective authors. For each property, column 2 specifies which state coverage metric, weak or strong, is used. Column 3 specifies which state a test case is to cover. Column 4 provides the length of each lasso-shaped test case, as defined in Definition 2 and column 5 is the time used to generate a test case. To compare the performance of the proposed metrics and a traditional coverage metric, we measure the branch coverage of each individual test case, which is given as the ratio of covered branches v.s. total branches. We also measure the accumulative

[1] `http://spinroot.com/spin/man/exercises.html`

Property	Coverage	State	Test case length		time (sec.)	Branch coverage	
			$\|\alpha\|$	$\|\beta\|$		Individual	Overall
General Inter-ORB Protocol							
`v6b`	Weak	s_1	577	1	0.01	46/70	51/70
		s_2*	577	1	0.01	46/70	
		s_3*	449	1	0.57	47/70	
	Strong	s_1	577	1	0.01	46/70	51/70
		s_2*	577	1	1.00	46/70	
		s_3*	449	1	1.00	47/70	
`v7`	Weak	s_2	577	1	0.01	46/70	54/70
		s_2*	577	1	0.01	46/70	
		s_3*	521	1	2.04	53/70	
	Strong	s_1	577	1	0.01	46/70	46/70
		s_2*	577	1	0.01	46/70	
		s_3*	(Exec. Time $>$ 300 min.)				
Needham-Schroeder Security Protocol							
`m_x1init_fixed`	Weak	s_1	22	1	0.01	15/59	15/59
		s_2*	22	1	0.01	15/59	
		s_3*	22	1	0.01	15/59	
	Strong	s_1	24	1	0.01	17/59	28/59
		s_2*	24	1	0.01	17/59	
		s_3*	23	1	0.01	15/59	
Sliding Window Protocol							
`ltl`$_3$	Weak	s_1	(Not covered)				14/23
		s_2	550	111	0.01	14/23	
		s_3*	185	111	0.01	7/23	
		s_4*	(Not covered)				
	Strong	s_1	(Not covered)				14/23
		s_2	550	111	0.01	14/23	
		s_3*	185	111	0.01	7/23	
		s_4*	(Not covered)				

Table 1. Experiment results. A test case $t = \alpha \cdot \beta^\omega$, where α is the prefix and β is a circular sequence. s_0 is the default and only start state for all the properties. Acceptance states are marked with $*$.

coverage of each test suite, which consists of all the test cases generated for a property and a particular (weak or strong) state coverage metric.

With only exception of property `v7` in GIOP model, the branch coverage achieved by a test suite for strong state coverage criterion is better than that for weak state coverage criterion. That is because strong state coverage criterion subsumes weak state coverage criterion. Generating a test case strongly covering state s_3 for property `v7` does not terminate in a reasonable time frame. Test suites generated for `v6b` and `v7` achieve a reasonable branch coverage (72.8% for `v6b` and 77.1% for `v7`), especially when taking into account that these two properties capture only a very limit requirement for the GIOP protocol. The model for Needham-Schroeder security protocol involves three parties: an initiator, a responder, and an intruder. The property `m_1init_fixed` is a liveness property requiring that the initiator sends a message only after the responder is up and running. It does not describes the security aspect of the protocol, that is, safety properties involving the intruder. Almost all of branches not covered by generated test suites are within the logic of the intruder. Lack of coverage in this case is related to the deficiency in the specification. In the model for the sliding window protocol, the branch points not covered by test suites for state coverage are within the logic of timeout mechanism. The timeout mechanism is put into place to handle packet loss. A close look at the model reveals that it does not contain a lossy channel. So in this case lack of coverage indicates the deficiency in the model. In these experiments, our proposed state coverage metrics do help reveal deficiency in specifications and/or models.

7 Conclusions

We considered specification-based testing for linear temporal properties expressed in generalized Büchi automata (GBAs). We proposed two variants of state coverage metrics for measuring how well test cases cover states of a GBA. The immediate application of these two metrics is to select test cases based on their relevancy to a GBA-based specification. For this application we provide model-checking-based test generation algorithms for proposed coverage criteria. This research extends our previous work on vacuity-based coverage metric for LTL formula. By using GBA as the underlying representation for linear temporal properties, we can use existing automaton minimization techniques to reduce syntactical variances of these temporal properties, and hence our state coverage metrics defined on GBAs are less susceptible to syntactical changes of properties. We argued that our property-based state coverage metrics also helped detect the deficiency in specification and one may use them to guide the refinement of requirement specifications. For this application, we defined the notion of vacuous states for a GBA and a system. Vacuous states are those states of the GBA that do not affect whether the system satisfies the GBA. Removing these vacuous states from the GBA yields a refined GBA that describes the behaviors of the system more closely. We provided a model-checking-based property refinement algorithm based on the notion of vacuous states and strong state coverage metric. Our experiment results demonstrated the feasibility and performance of our coverage metrics and test generation algorithms.

For the further research on the subject, we will study other GBA-based coverage metrics, and we will also consider a general framework for unifying temporal logic-based and GBA-based coverage metrics.

Acknowledgement The author would like to thank Bolong Zeng for his assistance in the experimental study.

References

1. Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall, 5th edition, 2010.
2. Andrea Calvagna and Angelo Gargantini. A Logic-Based Approach to Combinatorial Testing with Constraints. In *TAP'08*, volume 4966 of *LNCS*, pages 66–83. Springer, 2008.
3. Edmund Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.
4. Gordon Fraser and Angelo Gargantini. An Evaluation of Specification Based Test Generation Techniques Using Model Checkers. In *TAIC-PART'09*, pages 72–81. IEEE Press, 2009.
5. Rob Gerth, Doron Peled, Moshe Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *PSTV'95*, pages 3–18. Chapman and Hall, 1995.
6. Dimitra Giannakopoulou and Flavio Lerda. From States to Transitions: Improving Translation of LTL Formulae to Büchi Automata. In *FORTE'02*, volume 2529 of *LNCS*, pages 308–326. Springer, 2002.
7. Gerard J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
8. Hyoung Seok Hong, Insup Lee, Oleg Sokolsky, and Hasan Ural. A temporal logic based theory of test coverage and generation. In *TACAS'02*, volume 2280 of *LNCS*, pages 327–341, 2002.
9. Moataz Kamel and Stefan Leue. Formalization and validation of the General Inter-ORB Protocol (GIOP) using PROMELA and SPIN. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(4):394–409, March 2000.
10. Orna Kupferman and Moshe Y. Vardi. Vacuity detection in temporal model checking. *International Journal on Software Tools for Technology Transfer (STTT)*, 4(2):224–233, February 2003.
11. Paolo Maggi and Riccardo Sisto. Using SPIN to Verify Security Properties of Cryptographic Protocols. In *SPIN'02*, volume 2318 of *LNCS*, pages 187–204. Springer, 2002.
12. M. Michel. *Complementation is more difficult with automata on infinite words*. CNET, Paris, France, 1988.
13. Li Tan, Oleg Sokolsky, and Insup Lee. Specification-based Testing with Linear Temporal Logic. In *IRI'04*, pages 493–498. IEEE society, 2004.
14. Yih-kuen Tsay, Yu-fang Chen, Ming-hsien Tsai, Kang-nien Wu, and Wen-chin Chan. GOAL : A Graphical Tool for Manipulating Büchi Automata and Temporal Formulae. In *TACAS'07*, volume 4424 of *LNCS*, pages 466–471. Springer, 2007.
15. Moshe Vardi. Automata-theoretic model checking revisited. In *VMCAI'07*, volume 4349 of *LNCS*, pages 137–150. Springer, 2007.