Software Qual J (2015) 23:171–202
DOI 10.1007/s11219-014-9237-3

PBCOV: a property-based coverage criterion

Kassem Fawaz · Fadi Zaraket · Wes Masri · Hamza Harkous

Published online: 31 May 2014

Abstract Coverage criteria aim at satisfying test requirements and compute metrics values that quantify the adequacy of test suites at revealing defects in programs. Typically, a test requirement is a structural program element, and the coverage metric value represents the percentage of elements covered by a test suite. Empirical studies show that existing criteria might characterize a test suite as highly adequate, while it does not actually reveal some of the existing defects. In other words, existing structural coverage criteria are not always sensitive to the presence of defects. This paper presents PBCOV, a Property-Based COVerage criterion, and empirically demonstrates its effectiveness. Given a program with properties therein, static analysis techniques, such as model checking, leverage formal properties to find defects. PBCOV is a dynamic analysis technique that also leverages properties and is characterized by the following: (a) It considers the state space of first-order logic properties as the test requirements to be covered; (b) it uses logic synthesis to compute the state space; and (c) it is practical, i.e., computable, because it

Electronic supplementary material The online version of this article (doi:10.1007/s11219-014-9237-3) contains supplementary material, which is available to authorized users.

This paper builds on the short position paper (2 pages) entitled: "Property-based Coverage Criterion," presented at the DEFECTS Workshop, Chicago, IL, July 2009. This research was supported in part by NSF (Grant# 0819987) and by the AUB University Research Board.

K. Fawaz
University of Michigan, Ann Arbor, MI, USA
e-mail: kmfawaz@umich.edu

F. Zaraket (✉) · W. Masri
American University of Beirut, Beirut, Lebanon
e-mail: fz11@aub.edu.lb

W. Masri
e-mail: wm13@aub.edu.lb

H. Harkous
Swiss Federal Institute of Technology in Lausanne (EPFL), Lausanne, Switzerland
e-mail: hamza.harkous@epfl.ch

considers an over-approximation of the reachable state space using a cut-based abstraction.We evaluated PBCOV using programs with test suites comprising passing and failing test cases. First, we computed metrics values for PBCOV and structural coverage using the full test suites. Second, in order to quantify the sensitivity of the metrics to the absence of failing test cases, we computed the values for all considered metrics using only the passing test cases. In most cases, the structural metrics exhibited little or no decrease in their values, while PBCOV showed a considerable decrease. This suggests that PBCOV is more sensitive to the absence of failing test cases, i.e., it is more effective at characterizing test suite adequacy to detect defects, and at revealing deficiencies in test suites.

Keywords Software testing · Coverage criteria · Property-based coverage · State space coverage · Specification-based coverage · Test suite evaluation · Reachability analysis · Logic synthesis

1 Introduction

The number of potential test cases of most software programs is practically infinite, which makes exhaustive testing infeasible. Alternatively, the testing community believes that effective use of coverage criteria provides informal assurance that the software program is reliable. That is, coverage criteria provide practical rules for how to select tests and when to stop testing (Ammann and Offutt 2008). Testers leverage coverage criteria and configure their coverage requirements to maintain test suites for the purpose of (1) fully exercising the functionality of the system under test (validation testing), (2) guarding against previously detected defects (regression testing), and (3) increasing the likelihood of detecting undiscovered defects (defect testing). Researchers have proposed several techniques that use coverage criteria to augment, minimize, and build test suites (Harder et al. 2003; Ammann and Black 2001; Khurshid and Marinov 2004; Boyapati et al. 2002; Gligoric et al. 2010; Santelices et al. 2008). Most of those techniques incorporate a given test case in the test suite if its inclusion results in increasing the adopted coverage metric and excludes it otherwise.

Consider a test suite T with passing and failing test cases, T_{pass} and T_{fail}, respectively. This work deems a coverage metric that reports no increase in coverage between T_{pass} and $T = T_{full} = T_{pass} \cup T_{fail}$ as ineffective at measuring the adequacy of a test suite for regression and defect testing.[1] This is key to our experimental setup when comparatively evaluating the effectiveness of coverage criteria. This experimental approach is justified by the fact that it is possible for a coverage criterion to report high coverage, and even full coverage, for test suites that do not reveal existing defects.

Unlike testing, formal verification methods use static analysis and do not require the execution of a test suite (Holzmann 1997; Visser et al. 2003; Torlak and Jackson 2007; Clarke et al. 2004). For example, model checkers for decidable fragments of logics such as first-order logic (FOL), computational tree logic (CTL), or linear temporal logic (LTL) take as input a program and a set of formal properties therein and check whether the properties hold for the program. In general, model checkers either return with a proof that the properties hold for the program, return a counter example illustrating how the program violates the properties, or return an inconclusive result when they reach computational

[1] Please refer to Table 6 for a glossary of all the symbols used in this paper.

bounds such as memory or timeout limits. Such techniques can handle safety properties such as *null* pointer and array boundary checks, assume guarantee properties such as preconditions and postconditions, invariants such as data structure or loop invariants, as well as user assert statements. Existing dynamic analysis tools (Yang and Evans 2004; Ernst et al. 2007) can automatically infer such properties in case they were not specified.

In the presence of a set of properties P in a program S and a test suite T, our work aims at evaluating the quality of T and its adequacy at revealing defects in S. Specifically, this paper presents PBCOV, a new coverage approach that comprises a *property-based coverage criterion*, an associated metric, and a supporting tool. PBCOV that builds on the approach of the position paper (Zaraket and Masri 2009) is a dynamic analysis technique that leverages program properties and considers the state space of the properties as the test requirements to be covered. Our experiments show that PBCOV is more effective for regression and defect testing than existing structural coverage criteria.

The rest of this paper is organized as follows. In Sect. 2, we overview PBCOV and list our contributions. Section 3 walks through a motivating example that highlights the advantages of covering properties as opposed to structural elements. Section 4 provides a detailed description of PBCOV. Section 5 describes its implementation. Section 6 describes our experimental study and presents our results. We compare against related work in Sect. 7. Finally, we conclude and discuss future work in Sect. 8. Supplementary appendices are available online (PBCOV-APPENDICES 2013); Appendix A illustrates symbolic execution with an example, Appendix B details the PBCOV analysis of the motivating example, Appendix C describes the subject programs and their properties, and Appendices D and E list the results of the PBCOV and the structural coverage metrics in tabular form.

2 Overview and contributions

We consider a property P expressed as a *first-order* and *temporal logic* formula over a selected set of program and property variables, $x_1, x_2, \ldots, x_m$ where each variable x_i ranges over a domain $D_i, 1 \leq i \leq m$. Note that in our experiments, we did not need temporal operators to express the properties specified within the studied C programs, as first-order logic was enough to express those properties. For correctness, P must evaluate to `true` at the time of its execution for all test cases. We consider the smallest terms in P that evaluate to Boolean values to be *atomic predicate terms*, $p_1, p_2, \ldots, p_n$, and we consider the state space of P as all the 2^n possible valuations of the atomic predicate terms. The PBCOV metric measures the states of P covered by T against the reachable state space of P, knowing that many states may be infeasible.

Notice that $D = D_1 \times D_2 \ldots \times D_m$, the domain of the first-order variables occurring in P, is practically infinite since the variables may be scalars. Note also that the atomic predicate terms have a large, yet a finite number of valuations (2^n) that partition D into 2^n equivalence classes where each class maps to the same value under P. For example, consider the property $x<y \wedge y<z \wedge x<z$ and its atomic predicates $p_1 = x<y$,$p_2 = y<z$, and $p_3 = x<z$. The valuations $\langle 2,3,4\rangle$ and $\langle 3,4,5\rangle$ for variables x, y, and z are equivalent under the atomic predicate valuation $\langle true, true, true\rangle$ and are both mapped to `true`by the property. Thus, we consider the finite valuations of the atomic predicates as states when computing the state space coverage value of P. We still reason (with an SMT solver) about the domain of the first-order variables of P when considering the feasibility of a state

(valuation of atomic predicates). For example, the valuation $\langle true, true, false \rangle$ of $\langle p_1, p_2, p_3 \rangle$ is not feasible and thus should not count in the reachable state space.

We use *logic synthesis* techniques (ABC 2007) to compute a symbolic representation of P that comprises the state elements of P and the transition relation between its states. PBCOV computes an over-approximation of the reachable states of P using a *cut* abstraction of S and P. A cut C is a set of control points of S that split S into S_C and $S_{\overline{C}}$ such that S_C contains P and over-approximates S. The over-approximation is necessary since reachability analysis is expensive in nature.

We first instrument S and P and execute the instrumented program in a fashion similar to concolic execution (Burnim and Sen 2008; Godefroid et al. 2005) in order to identify the covered states. Second, we identify the states that were not covered (referred to as *missing states* hereafter) using an equivalence check (ABC 2007) between a symbolic definition of P and a truth table definition of the covered states. Many of the missing states could be infeasible; therefore, we compute an over-approximation of the reachable state space by performing feasibility checks on the missing states against a symbolic abstraction of S using a satisfiability modulo theory (SMT) solver (Dutertre 2006). We compute our property-based coverage metric and present to the user the feasible missing states based on which new test cases could be manually inferred. It should be noted that other researchers (Ammann and Black 2001; Ernst et al. 2007; Heimdahl et al. 2003) have previously proposed techniques based on coverage of specifications; we will compare our work to theirs in Sect. 7.

Logic coverage criteria such as *multiple condition* and *modified condition/decision coverage* (MC/DC) consider the Boolean predicates of a program and their valuations (Ammann and Offutt 2008). Considering the code equivalent to the synthesized property, PBCOV belongs to the family of logic coverage criteria. PBCOV differs from logic coverage criteria in that (1) it considers properties and not predicates in S, (2) the properties are not simple propositional formulae, but first-order and temporal logic formulae which PBCOV synthesizes into executable logic, and (3) it uses a *cut* abstraction to over-approximate the reachable state space of the properties. Specifically, we designed the metric associated with PBCOV to be similar to the *multiple condition* criterion modulo reachable states as opposed to the widely accepted MC/DC criterion for the following reason. MC/DC cannot be fully satisfied if the property can never evaluate to false, i.e., the program was correct. In other words, MC/DC may report the same partial coverage from when the program has a defect, and the test suite does not exercise it, to when the defect is fixed.

In order to evaluate PBCOV, we used several faulty versions of five C subject programs from different sources each annotated with properties and associated with a test suite (Do et al. 2005; Cormen et al. 2009; Baudin et al. 2009; Barr 2004; Coen-Porisini et al. 2001). Except for one program (RBBST), all the defects in the faulty versions of the programs were provided by the original sources as detailed in Table 1.

For each program, we computed the PBCOV metric and several existing structural coverage metrics once for $T_{pass} \cup T_{fail}$ and then for T_{pass} only. We observed that in most cases, structural coverage techniques exhibited little or no decrease in their associated metrics values, while the PBCOV metric showed considerable decrease. This indicates that PBCOV is more sensitive to the presence of defects, and thus, more effective at including test cases that exercise defects; hereafter, we refer to this as *sensitivity to the presence of defects*. Also, PBCOV identified missing states that suggested new test cases to augment the test suite. We considered existing standard tools, namely, GCOV (Gough and Stallman 2005) and ATAC (Horgan and London 1991), used as baseline in the recent literature

Table 1 Summary of subject programs

Program	Source of properties	Source of test suite	#Tests	Source of defects	LOC	#Properties	#APTerms in property	# of versions of interest
GZIP	Formalized comments	SIR	214	SIR	5,680	16	3–13	7
RBBST	Textbook (Cormen et al. 2009)	Auto	616	Authors	511	2	17–18	12 × 2
MMan	ACSL (Baudin et al. 2009)	Auto	1,124	FTB (Barr 2004	212	1	5	6
TCAS	Coen-Porisini et al. (2001)	SIR	1,608	SIR	173	5	4–8	41
SLL	Textbook (Cormen et al. 2009)	Auto	1,751	FTB (Barr 2004)	130	1	6	6 × 2

(Jaygarl et al. 2010), to measure structural coverage metrics for function calls, statements, basic blocks, branches, decisions, and defuses.

We identify four key advantages to deploying PBCOV as a coverage technique in the production of software programs.

Advantage 1 PBCOV is sensitive to program defects that might evade structural coverage techniques. This is mainly because PBCOV relies on the semantics of the program expressed in properties as opposed to code constructs.

Advantage 2 The quality of the PBCOV metric can be enhanced by modifying the properties to better describe the program. This is not possible with structural coverage metrics since modification of the structural elements of the program modifies the program itself.

Advantage 3 Adopting PBCOV as the means to evaluate test suites promotes the use of formal properties in code. In the long run, this will lead to quality programs with less ambiguous documentation and will enable a plethora of automated and interactive specification-based dynamic and static analysis tools that are not applicable in the absence of properties.

Advantage 4 In practice, test engineers look at coverage metrics after fixing defects induced by the test suite. The over-approximation of the reachable state space may still contain states that do not satisfy the properties. PBCOV provides a second metric that quantifies the level of confidence of a test engineer when he or she deems a test suite adequate by considering only states that satisfy the properties.

In this paper, we make the following contributions.

1. We present a coverage approach based on formal properties that describe program correctness.
2. We use an over-approximation of the reachable state space of the program properties to compute the PBCOV coverage metric. The over-approximation uses a cut abstraction of the program.
3. We present an implementation of PBCOV and demonstrate experimentally that it can detect deficiencies in test suites deemed effective with other traditional coverage techniques. The tool is available online at (PBCOV-TOOL 2013).

4. We took known benchmark programs with seeded defects, e.g., *TCAS* and *GZIP*, from several sources (Cormen et al. 2009; Baudin et al. 2009; Barr 2004; Coen-Porisini et al. 2001; Do et al. 2005) and augmented them with properties collected from the literature, textbooks, and existing semi-formal English comments. Table 1 provides details on the benchmarks including the source of the programs, the defects, and the properties. We provide the resulting code as useful annotated programs.

3 Motivation

We use the function `sort` in Fig. 1, a faulty implementation of *selection sort* (Barr 2004), to illustrate the advantages of covering properties as opposed to structural elements. The function takes as input an array '`a`' of size '`n`'; `current` and '`j`' are the iterators of the outer and inner loops, respectively; `lowestindex` holds the index of the minimum element so far in the array; and `temp` is used to perform the swap on Line 13. Line 9 has a defect as the inner loop does not always select the minimum and erroneously compares against `a[current]` instead of `a[lowestindex]`. But due to coincidental correctness (Masri 2010), the defect at Line 9 could be exercised without leading to failure. For example, the test cases in test suite T shown in Fig. 1 result in the sorted arrays in A and none of them leads to a failure. T apparently seems reasonable as it consists of non-sorted arrays of different sizes, a sorted array t_1, a reverse-sorted array t_2, and test cases that test boundary conditions such as t_7 and t_8.

We computed the structural coverage metrics resulting from executing T using GCOV and ATAC. GCOV computes four coverage metrics, the percentage of executed statements, executed branches, branches taken at least once, and invoked functions. ATAC measures basic block, decision, C-use, and P-use coverage. Basic block coverage is similar to statement coverage but might yield slightly different metric values. Decision coverage reports whether each condition in the program evaluates to both `true` and `false` at least once during test suite execution. Computational use (C-use) and predicate use (P-use) track the definitions and usages of variables. A C-use is a use of the variable in a computation such as an arithmetic expression, and a P-use is a use of the variable in a predicate

```
1 void sort ( int a[], int n) {
2   int current, j, lowestindex, temp;
3
4   for ( current = 0; current < n-1; current++ ) {
5     // find the minimum
6     lowestindex = current;
7     j = current+1;
8     while ( j < n ) {
9       if ( a[j] < a[current] )
10        lowestindex = j;
11      j++;
12    }
13    // swap
14    if ( lowestindex != current ) {
15      temp = a[current];
16      a[current] = a[lowestindex];
17      a[lowestindex] = temp;
18    } } }
19 // fix: replace Line 9 with "if (a[j] < a[lowestindex])"
```

```
Test suite T = {
t1 = ⟨1 2 3 4 5 6 7 8 9⟩,
t2 = ⟨9 8 7 6 5 4 3 2 1⟩,
t3 = ⟨2 1 3 4 5 6 7 8 9⟩,
t4 = ⟨9 2 3 4 5 6 7 1⟩,
t5 = ⟨9 8 7 4 5 6 7 8 3 2 1⟩,
t6 = ⟨1 2⟩,
t7 = ⟨1⟩,
t8 = ⟨⟩ }

After sort A = {
a1 = ⟨1 2 3 4 5 6 7 8 9 ⟩,
a2 = ⟨1 2 3 4 5 6 7 8 9 ⟩,
a3 = ⟨1 2 3 4 5 6 7 8 9 ⟩,
a4 = ⟨1 2 3 4 5 6 7 9 ⟩,
a5 = ⟨1 2 3 4 5 6 7 7 8 8 9 ⟩,
a6 = ⟨1 2 ⟩,
a7 = ⟨1 ⟩,
a8 = ⟨⟩ }
```

Fig. 1 Selection sort motivating example

expression that evaluates to a Boolean value. The C-use measure ensures that there is at least one path between the definition and a computational use of a variable. The P-use measure ensures that there is at least one path between the definition of the variable and both the `true` and `false` valuations of a predicate containing the variable (Rapps 1982).

T achieves full C-use coverage except for one infeasible def-use pair consisting of the definition `lowestindex = current` on Line 6 and the use `a[current] = a[lowestindex]` of `lowestindex` on Line 16. This def-use pair is not feasible because the execution of the use is in contradiction with the `if` condition predicate `lowestindex != current` on Line 14. T also achieves full P-use coverage except for three infeasible P-use pairs. The first is the definition `j = current + 1` on Line 7, and the `false` value of the loop predicates $j<n$ on Line 8. This is infeasible since `current` is bounded by $current<n-1$ on Line 4. The second infeasible pair is the definition `lowestindex = current` on Line 6 and the `true` value of predicate `lowestindex != current` on Line 14. The last infeasible pair is the definition `lowestindex = j` on Line 10 and the `false` value of the predicate `lowestindex != current` on Line 14. This is infeasible since '`j`' is guaranteed to be different than `current` as it starts at `current+1` on Line 7 and only gets incremented later. T achieves full coverage for all the other GCOV and ATAC metrics. We conclude that T is a deficient test suite that attained full coverage using traditional structural techniques, which motivates our work on property-based coverage.

We introduce a property P in `sort` specifying that at the end of execution every two arbitrary neighboring elements `a[k]` and `a[k+1]` within the bounds of the array must be in order.

$$P = (0 \leq k) \wedge (k < n-1) \rightarrow a[k] \leq a[k+1]$$

Formally, $k \in \mathbb{Z}, n \in \mathbb{Z}$ and the array $a : \mathbb{Z} \mapsto \mathbb{Z}$ maps an index to a value.

The property P has three atomic predicate terms $p_1 = (0 \leq k)$, $p_2 = (k < n-1)$ and $p_3 = (a[k] \leq a[k+1])$ where $p_i(k,n,a) : \mathbb{Z} \times \mathbb{Z} \times (\mathbb{Z} \mapsto \mathbb{Z}) \mapsto \mathbb{B}$ for $1 \leq i \leq 3$ and $P = p_1 \wedge p_2 \rightarrow p_3$.

Let $a_i \in A$ be the array resulting from executing `sort` with test case $t_i \in T$.

Consider $P_{cover} = \{\langle b_1, b_2, b_3 \rangle, b_1 = p_1(k,n,a_i), b_2 = p_2(k,n,a_i), b_3 = p3(k,n,a_i), k \in \mathbb{Z}, a_i \in A, n = |a_i|\}$ the set of all valuations of $\langle p_1, p_2, p_3 \rangle \in \mathbb{B}^3$ over all test cases in T. Note that a test case $t \in T$ assigns values for `a` and `n`, while k remains a free variable in P. We compute the set of all feasible states from a test case t using a satisfiability check on each state, or by covering the full range of the free variable k when k is bounded.

The set P_{cover} contains all feasible valuations of $\langle p_1, p_2, p_3 \rangle$ except for the valuation $e = \langle$ true , true , false $\rangle$ which describes two valid array elements that are not in order. Executing `sort` with test input $\langle 3\,1\,2\,4 \rangle$ results in the array $\langle 2\,1\,3\,4 \rangle$ (`a[0] = 2, a[1] = 1, a[2]= 3, a[3] = 4`) where e is satisfied for $k = 0$ since `a[0] > a[1]`. In this example, PBCOV reports that T did not achieve full coverage and that, specifically, e is missing. This suggests that T must be augmented until e is covered, which will help induce a failure and thus reveal the defect. Appendix B elaborates more on PBCOV using this same example.

One could argue that using MC/DC on the *synthesized property*[2] would be as effective as PBCOV. In fact, MC/DC for the synthesized property will require the inclusion of e and

[2] Typically MC/DC considers the coverage of a code decision predicate in terms of its clauses. Here the property and its atomic predicates are presented to MC/DC as the predicate and its clauses, respectively.

will report partial coverage. However, when the defect is fixed, MC/DC will still report the same partial coverage. On the other hand, coverage of the reachable states exhibits a difference from when the property is violated to when it is not. This motivates the design decision for the PBCOV metric that considers the covered reachable state space.

4 PBCOV

We now describe the PBCOV approach, its mechanism to over-approximate the reachable property state space, its cut-based abstraction, and its metrics. An illustration of symbolic execution which we use in over-approximating the reachable property state space is presented in Appendix A.

The flow diagram in Fig. 2 and the algorithm in Fig. 3 illustrate the PBCOV process. PBCOV takes as input the source code of the program S with a set of properties therein, P, as well as a test suite T and reports the adequacy of T in assessing the behavior of S as formally specified by P. The instrumentation generates an instrumented program S_i. The analysis generates the following: **(a)** a symbolic representation P_{sym} of P, and **(b)** a symbolic representation S_{sym} of S or part of S.

Line 3 of the algorithm computes n atomic predicates $\langle p_1, p_2, \ldots, p_n \rangle$ from the property P. Line 4 returns a symbolic representation of P in terms of the atomic predicates. If P has temporal components, then P is translated into a finite state machine using textbook transformations (Linz 2012), and the states of the machine are considered for coverage and feasibility as follows in the paper. For example, the property "o;r*;c" specifying that a file open 'o' must be followed by zero or more file read 'r', then followed by a file close 'c' can be translated to the state machine in Fig. 4.

Line 8 builds the instrumented program S_i that takes a test case t and computes and returns values for the atomic predicates and program variables.

Line 12 runs S_i with a test case $t \in T$ and returns/saves the values of the program variables in a corresponding formula V. The formula V is a conjunction of equivalence statements constraining each variable to its assigned value. For example, the formula $a[0] = 1 \wedge a[1] = 2 \wedge n = 2$ corresponds to an array a with size n equal to 2 and with

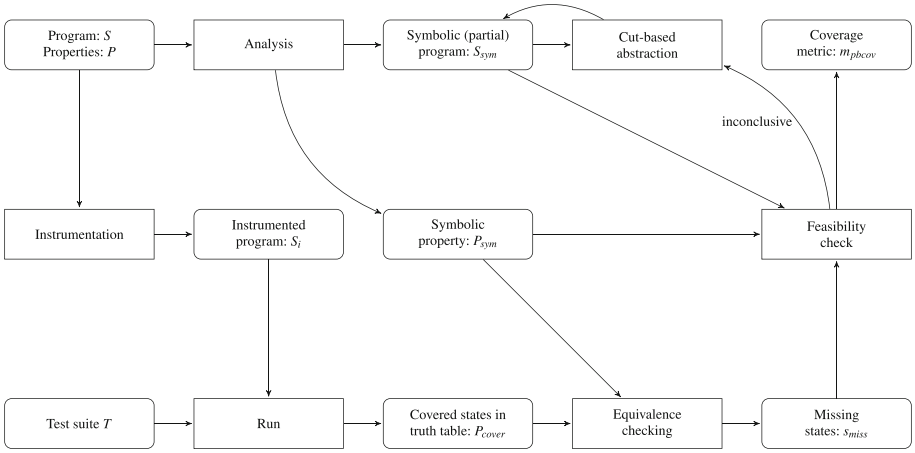


Fig. 2 Flow diagram of PBCOV

```
1 Algorithm PBCOV ( S, P, T )
2
3   ⟨ p1, p2, ..., pn ⟩ = getAtomicPredicates(P)
4   Psym = analyze(P, ⟨ p1, p2, ..., pn ⟩)
5
6   Declare Ppass, Pfail, Pmiss, Pcover, Pinconclusive ⊆ B^n // sets of property states
7
8   Si = instrument (S)
9
10  foreach t ∈ T
11    // Run the Si with t and return program and property variable values
12    V = run(Si, t)
13
14    // add the feasible states that satisfy the property to Ppass
15    Ppass += { s: SAT(s ∧ P ∧ V) }
16
17    // add the feasible states that fail the property to Pfail
18    Pfail += { s: SAT(s ∧ ¬ P ∧ V) }
19
20  // compute the observed property by the test suite and the propram
21  // Pcover is true when pass, false when fail, free when not covered
22  Pcover = Ppass ∨ ( free ∧ ¬ Pfail)
23
24  Pmiss = (Psym ≠ Pcover) // when Pcover = free
25
26  Ssym = analyze(S)
27  iter = 0;
28  while ( iter <= LIMIT )
29    ⟨ n^true_feas, n^false_feas, Pincoclusive ⟩ = checkFeasibility (Pmiss, Ssym)
30    if (Pinconclusive = {})  // no inconclusive results
31      break; // done
32    Pmiss = Pinconclusive
33    Scut = cutBasedAbstraction(S, iter++)
34    Ssym = analyze(Scut)
35
36  mpbcov = computeMetric(|Ppass|, |Pfail|, n^true_feas, n^false_feas)
```

Fig. 3 The **PBCOV** algorithm

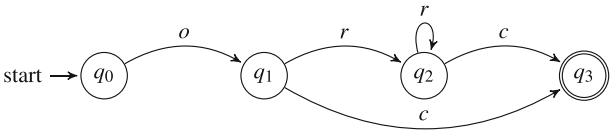


Fig. 4 Finite state machine corresponding to temporal property "o;r*;c"

values $\langle 1, 2 \rangle$. Lines 15 and 18 compute the feasible states with the formula V that satisfy and fail P and add them to P_{pass} and P_{fail}, respectively. PBCOV uses an SMT solver to compute whether a state s is feasible in case the free variables in $s \wedge P \wedge V$ were not bounded.

Line 22 computes the states of P that are observed by S and T and represents the observed values faithfully in P_{cover}. For the states that are not observed by S and T, Line 22 leaves P_{cover} undetermined by setting it to a *free* Boolean variable. Intuitively, P_{cover} is `true` when P_{pass} is `true`, `false` when P_{fail} is `true`, and *free* otherwise, where *free* is a nondeterministic variable.

Line 24 computes the missing states as the difference between P_{sym} and the covered states. runs an equivalence check $P_{sym} = P_{cover}$ to compute the missing states P_{miss}.

The missing states may be:

- states where P evaluates to `true`, and in that case, it is likely that T may not be executing all the specified behavior of S.
- states where P evaluates to `false`, and in that case, it is likely that T may not be inducing failures that are due to defects in S.
- unreachable states due to the dependencies among the atomic predicate terms or due to the details of the implementation and the structure of S. In this case, they should not be considered by the coverage metric.

PBCOV uses an SMT solver to check the feasibility of the missing states. We express the conjunction of S_{sym} and each missing state $s_{miss} \in P_{miss}$ as an SMT formula and pass that to the SMT solver with a satisfiability check. An SMT solver returns either a satisfiable answer in case the SMT formula is satisfiable, an unsatisfiable answer in case the SMT formula is unsatisfiable, or an inconclusive answer in case the solver exhausted its computational resources before reaching an answer. The following should be noted about the outcome of the reachability analysis.

a. It computes the exact reachable state space if it provides conclusive results for the complete program.
b. In case the solver returns a satisfiable answer, it also returns a model with input values that induce the missing state. PBCOV can use these input values to augment T with a new test case that covers the missing state in question. Concolic testing tools such as that in Burnim and Sen (2008) can be readily used for test generation given the valuation of the program variables that satisfy the missing state.
c. When the solver returns an inconclusive results, PBCOV over-approximates the reachable state space via assuming that the state is reachable. This is one source of the reachable state space over-approximation, and the second source is the cut abstraction discussed in Sect. 4.2. If the number of inconclusive results is high, PBCOV computes a cut abstraction of the program and performs the inconclusive feasibility checks again.
d. Finally, PBCOV computes a metric, m_{pbcov}, which compares the covered states to the reachable states of the program, as described in Sect. 4.3.

4.1 Over-approximation of the reachable property state space

The diagram in Fig. 5 shows a program S with n statements l_1 to l_n. Each statement l_i, $1 \leq i \leq n$, is paired with pc_i, its associated path condition that we compute using symbolic execution as illustrated in Appendix A. A path condition associated with a line of code is the condition necessary for the program to execute that line of code. For instance, the path condition resulting from an `if (b) { I } else { E }` statement is a disjunction of the path conditions of both branches and is of the form $\left(pc_b \wedge pc_I \vee pc_{\neg b} \wedge pc_E\right)$. The path condition resulting from a loop statement `while (b) { W }` is of the form $(pc_b \wedge pc_W)^* \vee pc_{\neg b}$, where the Kleene star operator $*$ stands for zero or more iterations of the loop.

The path conditions form S_{sym}, the symbolic SMT representation of S. The box on the left represents the state space of the property P and the horizontal ellipse labeled $[P]$ is the

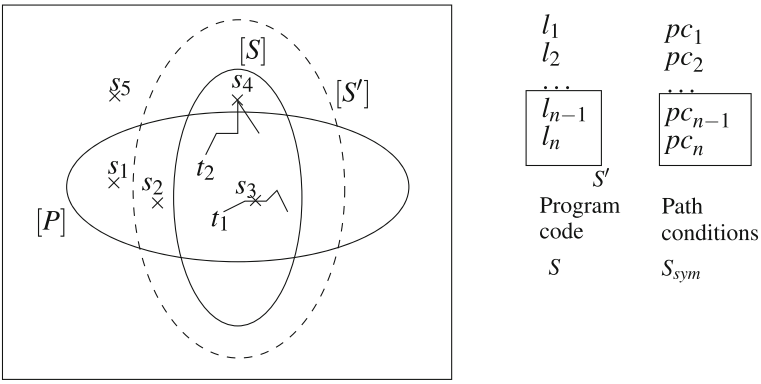


Fig. 5 Over-approximation of reach/able state space

subset of states where P holds. The solid (inner) vertical ellipse labeled $[S]$ is the subset of states reachable within the program S. Consequently, s_3 is a reachable state where P holds, while s_4 is a reachable state where P does not hold, and thus, s_4 is a bad state associated with a defect. The test case t_1 passes P since P holds in all of its states, and test case t_2 fails P since s_4 belongs to it.

Test suite $T = \{t_1, t_2\}$ does not cover states s_1, s_2, and s_5 among other states. These states are not reachable. States s_1 and s_2 are passing states, and state s_5 is a failing state. PBCOV checks these states against S_{sym} with the SMT solver. If the solver returns a satisfiable solution for a state, then the state is considered in the computation of the coverage metric, but if it returns an unsatisfiable result, then the state will not be considered.

In case the solver returns an inconclusive result, PBCOV computes a *cut* of the program with corresponding path conditions S' as an approximation of the program. The over-approximation of the program is defined by the following: **(a)** selecting a boundary in the program and considering the path conditions between the boundary and the property, thus defining a partial program, and **(b)** treating the variables that are not defined in the partial program as free unconstrained variables. *This is an over-approximation of the reachable states of S since the free variables can assume all possible values while the removed clauses assume only a subset of these values*. Figure 5 illustrates the partial program statements l_{n-1} and l_n, and path conditions pc_{n-1} and pc_n.

The dashed ellipse labeled $[S']$ shows the state space reachable by S' and contains $[S]$ as it is an over-approximation of it. As a result, s_2 is now considered when computing the PBCOV metric as it falls in the over-approximation of the state space, while s_1 and s_5 are still not considered. This example shows how over-approximation might lead to over-estimating state space coverage.

4.2 Computing the cut abstraction

Let X be the set of variables of S. Let $G = \langle V, E \rangle$ be the control flow graph of S where a node in V represents a statement or a function call and $E \subset V \times V \times 2^X$ is the transition relation from one node to the other labeled with the set of variables used or defined in the source node. For example, the edge $(u, v, \{x, y, z\}) \in E$ represents the transition from u to v, the two nodes corresponding to the two consecutive statements `x = y + z; w++`, respectively.

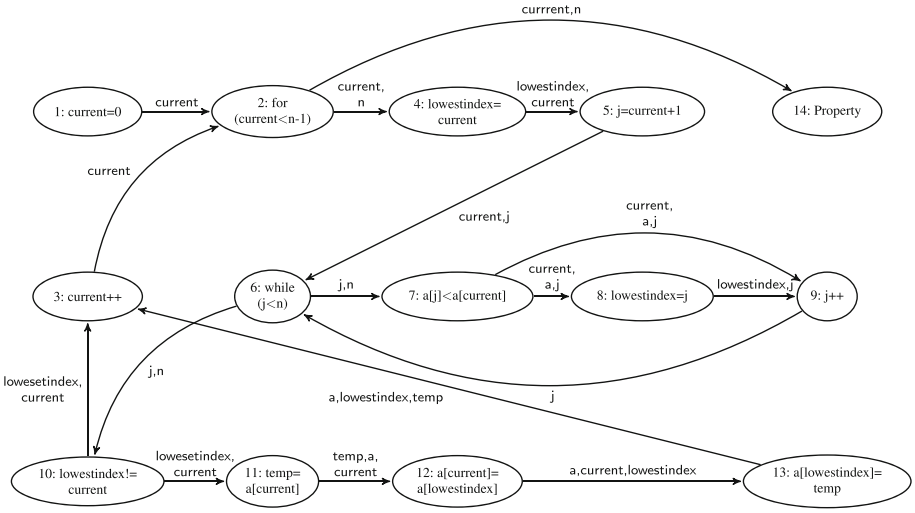


Fig. 6 Control flow graph of the `sort` example of Fig. 1

A *cut of a graph* is a partition of V into two sets: $\mathcal{C}$ and $\overline{\mathcal{C}} = V \setminus \mathcal{C}$. A cut induces two sets of *cut nodes* $V_{\mathcal{C}} = \{u \in \mathcal{C} : \exists v \in \overline{\mathcal{C}}.\exists x \in 2^X.\big((u,v,x) \in E\big)\}$ and $V_{\overline{\mathcal{C}}} = \{v \in \overline{\mathcal{C}} : \exists u \in \mathcal{C}.\ \exists x \in 2^X.\big((u,v,x) \in E\big)\}$. For example, Fig. 6 shows the control flow graph of the `sort` example from Fig. 1 assuming the correctness property is inserted at the end. A cut between nodes 1 and 2 separates the graph into $\mathcal{C} = \{1\}$ and $\overline{\mathcal{C}} = \{2,3,4,5,6,7,8,9,10,11,12,13,14\}$ where $V_{\mathcal{C}} = \{1\}$ and $V_{\overline{\mathcal{C}}} = \{2\}$. Another cut where $V_{\mathcal{C}} = \{2\}$ and $V_{\overline{\mathcal{C}}} = \{3,4,14\}$ separates the graph into $\mathcal{C} = \{1,2\}$ and $\overline{\mathcal{C}} = \{3,4,5,6,7,8,9,10,11,12,13,14\}$.

An *s-t cut* (source-target cut) is a cut seeded with sets $s \subseteq \mathcal{C}$ and $t \subseteq \overline{\mathcal{C}}$. An *s-t mincut* refers to an *s-t cut* where the set of variables referred in the transitions from $V_{\mathcal{C}}$ to $V_{\overline{\mathcal{C}}}$ is of minimal cardinality as follows.

$$\text{argmin}_{V_{\mathcal{C}}} |\{x : x \in X, (u,v,X) \in E, u \in V_{\mathcal{C}}, v \in V_{\overline{\mathcal{C}}}\}|$$

PBCOV computes S' to be the $V_{\overline{\mathcal{C}}}$ resulting from an *s-t mincut* of G where s is the node in V referring to the first statement of S and t refers to the node in V representing the property.

For example, the cut between 1 and 2 is the *s-t mincut* of the graph in Fig. 6 where $s = \{1\}$ and $t = \{14\}$. We proceed by computing the symbolic representation of the sorting routine less the initialization `current=0` and use that to check for the feasibility of the missing states. This obviously over-approximates the reachable state space. We use the *augmenting-path* algorithm to compute the *s-t mincut*, which yields practically linear runtimes even on large graphs (Ford and Fulkerson 1956).

In case the feasibility solver returns an inconclusive result for S', PBCOV iteratively computes the *s-t mincut* of S' considering $V_{\overline{\mathcal{C}}}$ as s this time. This terminates when the solver returns a conclusive result, or when the *s-t mincut* returns $S' = V_{\overline{\mathcal{C}}} = t$.

In cases where loop boundaries may separate $V_{\overline{\mathcal{C}}}$ from the property, we unroll the last K loop iteration where K is an unrolling bound and consider that the behavior of the previous

iterations is nondeterministic. This is similar to the abstraction of weakest precondition computations (Ball et al. 2011, 2001; Yang et al. 2010). We handle recursion similarly.

4.3 The PBCOV metric and the confident PBCOV metric

Let n_{cov}^{true} be the number of covered states that evaluate P to `true`, n_{cov}^{false} the number of covered states that evaluate P to `false`, n_{feas}^{true} the number of feasible states that evaluate P to `true`, and n_{feas}^{false} the number of feasible states that evaluate P to `false`. We define the PBCOV metric to be:

$$m_{pbcov} = \log(1 + n_{cov}^{true} + n_{cov}^{false}) / \log(1 + n_{feas}^{true} + n_{feas}^{false}) \quad (1)$$

We use a logarithmic scale for practical reasons to yield metrics that are in the same order of magnitude as of the traditional structural metrics, as illustrated in Table 4 for example. We justify that by the fact that the number of over-approximated feasible states is of exponential nature because of the state explosion problem that characterizes static analysis. Meanwhile, the number of covered states is of polynomial nature since test suites are designed such that the program terminates within a reasonable time.

In practice, programmers fix the defects revealed by T and P, and run T again on the fixed program to compute coverage. Consequently, all the covered states evaluate P to `true`. The feasible states however may still contain states that are not induced by the test suite and evaluate the properties to `false`. Nevertheless, the programmer may be interested in evaluating T in terms of property coverage assuming that the code is correct. We define an confident version of PBCOV that measures the covered states against the feasible states that evaluate P to `true` as:

$$m_{pbcov}^{con} = \log(1 + n_{cov}^{true}) / \log(1 + n_{feas}^{true}) \quad (2)$$

The difference between the confident and the actual PBCOV metrics, $m_{pbcov} - m_{pbcov}^{con}$, quantifies the level of confidence of a test engineer when he or she deems a test suite to be adequate. A large difference means that he or she is too confident.

The two metrics are good indicators in programs where reachability analysis works well and concludes on significant cuts of the program. However, the denominator grows exponentially where the reachability analysis does not conclude except on small parts of the program, and thus, the magnitude values of the metrics may mislead the user to doubt the test suite. In such cases, programmers should not use values of m_{pbcov} as absolute indicators; rather they should consider them relative to other instances of m_{pbcov} computed with different test suites, as done in our experiments in Sect. 6.

5 Implementation and tools utilized

The current implementation of PBCOV supports C programs with user assertions. PBCOV makes use of the concolic testing tool Crest (Burnim and Sen 2008) to instrument and build the symbolic representation of the program, the ABC (ABC 2007) model checker to perform the equivalence check, and the SMT version of the tool CBMC (Clarke et al. 2004) powered by the Yices (Dutertre 2006) solver to compute the reachability analysis.

5.1 Instrumentation

Crest takes a program written in the C programming language and instruments it using the C intermediate language (CIL) (Necula et al. 2002) platform. The instrumented code follows the execution of the program over a concrete input while constructing a Boolean expression Φ representing the path condition of the program at every executed statement. Once the execution is done, Crest modifies Φ to represent a path that the program has not taken yet and stores that in Ψ. It then passes Ψ to Yices, to compute a valuation e of the program input variables that can take the program to the desired path. If the query is satisfiable, Crest executes again with e as input. Crest repeats this procedure until all paths have been covered.

PBCOV modifies and leverages Crest to instrument the atomic predicate terms of the specified properties and computes the path conditions of the code.

5.2 Missing states and reachability analysis

PBCOV runs the test suite T against the instrumented program S_i as computed in Sect. 5.1 and collects the covered states in a truth table to build the equivalence check $P_{cover} = P_{sym}$ as shown in Fig. 3 and discussed in Sect. 4. PBCOV passes the $P_{cover} = P_{sym}$ as a circuit and passes that to the ABC model checker. ABC uses transformation-based verification techniques to check properties of sequential circuits. A sequential circuit is a Boolean formula with memory elements that can simulate the execution of transition systems and is thus well suited to represent temporal properties. The memory elements arise from quantifiers and temporal operators that might be used in the property. The ABC model checker detects the states where the two formulas P_{cover} and P_{sym} differ; these constitute the missing states P_{miss}.

PBCOV uses CBMC and Yices to check whether these missing states are feasible by running a satisfiability check of each state against the symbolic representation of the program S_{sym}: the path conditions that describe the program. PBCOV discards the missing states reported as not satisfiable by Yices from the reachable state space and then computes the coverage metric. Satisfiability is an intractable problem in theory. In practice, and for formulas arising from logic design and software contexts, satisfiability solvers emerged lately that use the structure of the formula in question to answer the satisfiability query in reasonable time. To guarantee polynomial running time, PBCOV sets a timeout when it makes a call to the satisfiability solver. In case the satisfiability solver does not return a conclusive result before the timeout, PBCOV computes an abstraction of the formula that over-approximates the reachable state space and passes that to the solver for at most *LIMIT* times, where *LIMIT* was set to 5 in our experiments.

6 Experimental study

In this section, we empirically demonstrate that PBCOV is (1) more sensitive to the presence of defects than other structural coverage techniques, and we show that (2) there is a correlation between the PBCOV metrics computed for a test suite and the defect detection capabilities of the test suite. We describe the setup of our experiments and present the subject programs and the results. We comment on the results, and we finally discuss the validity of the PBCOV metric.

6.1 Experimental setup

In order to demonstrate the potential of PBCOV, we applied it along with several structural coverage techniques to five subject programs with property annotations as described in Table 1. Each program is associated with a test suite exhibiting high structural coverage and one or more program versions that are seeded with defects. The diagram in Fig. 7 illustrates a unit experiment. A version of one of the subject programs is selected and executed against the full test suite T_{full}. The passing tests are identified and collected in a subset of T_{full} denoted by T_{pass}. A test is considered passing if it produces the expected output while noting that a passing test may still violate a property. Where appropriate, e.g., with *TCAS*, we form $T_{pass-assert}$ to contain those test cases that passed and did not violate the properties.

Consequently, the unit experiment is setup such that (a) each subject program is associated with a single T_{full}, (b) each seeded version is associated with its own T_{pass}, (c) the failing tests

($T_{fail} = T_{full} \setminus T_{pass}$) are capable of revealing the seeded defects, and **(d)** T_{pass} does not reveal the defect in the seeded version.

We compute the structural coverage attained for each seeded version using GCOV and ATAC. To comparatively evaluate PBCOV and the structural coverage techniques, we compute the percent decrease in coverage for each coverage metric, denoted by $\%\delta_{cov}$, from when T_{full} was applied to when only T_{pass} was applied. For a specific metric, $\%\delta_{cov}$ assesses whether the metric is of utility to uncover the seeded defects. For PBCOV, $\%\delta_{cov}$ expresses the total number of states induced by T_{full} versus the total number of states induced by T_{pass} and is defined as the following expression:

$$\%\delta_{cov}^{PBCOV} = \left(n_{cov}^{true} + n_{cov}^{false}\right)\big|_{T_{full}} - \left(\left(n_{cov}^{true} + n_{cov}^{false}\right)\big|_{T_{pass}} / \left(n_{cov}^{true} + n_{cov}^{false}\right)\big|_{T_{full}}\right) \quad (3)$$

We computed $\%\delta_{cov}$ for the structural coverage metrics in a similar fashion. For a structural metric K, let K_{full} and K_{pass} be the number of structural elements covered by T_{full} and T_{pass}, respectively, and let K_{total} be the total number of structural elements. The structural coverage metric is the ratio of the covered elements over the total elements, i.e., $m_{full}^{K} = K_{full}/K_{total}$ and $m_{pass}^{K} = K_{pass}/K_{total}$. The percent decrease is defined as follows:

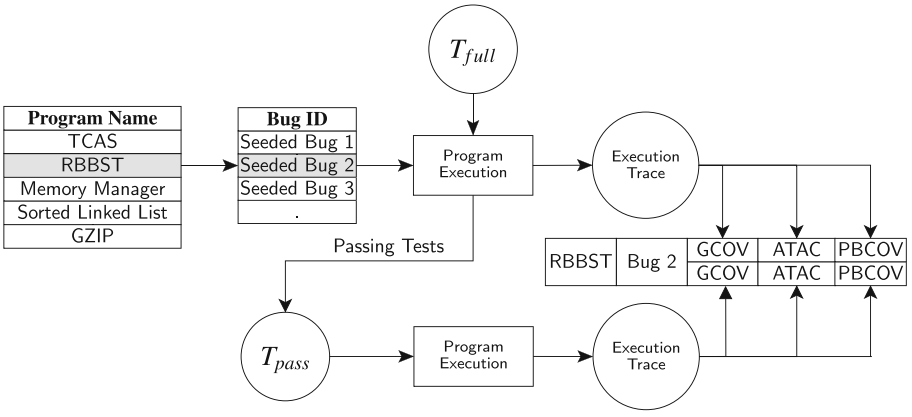


Fig. 7 Experimental setup flow diagram

$$\%\delta_{cov}^{K} = \left(K_{full} - K_{pass}\right)/K_{full} = \left(m_{full}^{K} - m_{pass}^{K}\right)/m_{full}^{K} \quad (4)$$

6.2 Subject programs, properties, and test suites

Table 1 lists the subject programs, their sizes in terms of lines of code (LOC), the sizes of the test suites associated with them, and the sources of the properties, test suites, and defects. The table also provides the number of atomic predicate terms (APTerms) in the properties. TCAS and GZIP have several properties with a varying number of terms, and the property of RBBSTInsert has one term more than that of RBBSTRemove.

Appendix C describes the programs and the properties in detail. Briefly, the *TCAS* and *GZIP* programs along with their test suites and seeded defects were downloaded from the SIR repository (Do et al. 2005). The properties of *TCAS* originated from the work in Coen-Porisini et al. (2001). The properties for *GZIP* are user assert statements already embedded in the code and selected English pre-/postcondition comments for functions that are two levels deep in the call chain that the authors formalized verbatim. The memory manager (*MMan*) program and its properties are provided with the Frama-C ANSI C Specification Language (ACSL) (Baudin et al. 2009). The Red Black Binary Search Tree (*RBBST*) originates from (Martinian 2010), and its properties along with the Sorted Linked List (*SLL*) and its properties originate from textbook implementations.

The defects seeded in *SLL* and *MMan* are equivalent to defects found in several buggy programs described in (Barr 2004), which involved linked lists and memory allocators. For *SLL*, *RBBST*, and *MMan*, we automatically generated test suites using available tools such as UDITA (Gligoric et al. 2010) and Crest. Note that we used UDITA when Crest was not applicable. The two tools differ in the techniques they use to select concrete inputs that under-approximate the symbolic representation of a program. Notice also that test case generation and over-approximation reachability analysis using symbolic execution differ in that the first tries to under-approximate the symbolic representation of a program. In all cases, we specified the properties independent from the seeded defects. Table 2 presents the rates of structural and PBCOV coverage for the subject programs; clearly, structural is higher than the more conservative PBCOV coverage except for the *GZIP* program. This is due to (1) legacy dead code that is not covered, and (2) we only formalized functions that are two levels deep in the call chain. This is evidence that for realistic size programs like *GZIP*, more specifications need to be added. Table 2 also shows the average running time of the structural coverage metrics, versus the running time for PBCOV. PBCOV was at worst three times slower than the structural coverage metrics. This is due to the several calls to the SMT solver that timed out. This is acceptable in most cases since regression testing typically takes place overnight or on weekends. Further work to reduce the running time of PBCOV could make use of computing the unsatisfiable core of an unsatisfiable formula and using it to eliminate SAT solver calls.

6.3 Percent decrease of coverage results

For each seeded version, we report $\%\delta_{cov}$ from when T_{full} is used to when T_{pass} is used. Here, we provide a summary of our results, and the full set of data is available in Appendices D and E. As shown in Table 1, there are a total of 90 versions which consist of 41 *TCAS* versions, 12 *RBBST* versions studied once for the *Insert* function and once for the *Remove* function, 6 *SLL* versions studied once for the *Insert* function and once for the

Table 2 Structural coverage and PBCOV results with T_{full} and without the seeded defects

Program	GCOV (%)				ATAC (%)				Time (h)	m_{pbcov}	m_{pbcov}^{con}	Time (h)
	Line	Branch	Branch once	Call	Block	Decision	C-use	P-use				
TCAS	100	100	94	100	100	91	100	93	0.42	91.57	N/A	0.57
RBBSTInsert	100	100	92	100	100	96	85	80	2.7	33.6	56.59	5.9
RBBSTRemove	92	93	84	100	91	89	67	53	3.1	35.98	56.65	7.8
MMan	100	100	100	100	100	100	100	100	1.6	28.23	30.21	3.8
SLLInsert	100	100	100	–	100	100	100	100	2.3	86.14	100	4.8
SLLRemove	100	100	100	–	100	100	100	100	2.8	69.9	82.71	7.3
GZIP	74.69	75.81	59.01	50.1	58	50	56	42	4.5	68.36	58.35	10.2

There were no function calls in either of the *SLLInsert* or *SLLRemove* functions

Remove function, 7 *GZIP* versions of interest, and 6 *MMan* versions. Based on the $\%\delta_{cov}$ results, we recognize six categories that divide the 90 versions.

Category 1 The first category consists of 13 versions where no change occurs in coverage for both the structural and PBCOV metrics. We refined the properties to include terms related to the seeded defects. We reran our experiments and obtained significant PBCOV decrease for 10 out of the 13 versions. This shows that the quality of PBCOV can be enhanced while that is not possible for the structural coverage metrics. In future work, we will investigate methods to refine properties by seeding defects and observing the effect of that on the PBCOV metrics.

Category 2 The second category contains 19 versions where no change occurred with the structural coverage metrics but significant change occurred with PBCOV.

Figure 8 shows $\%\delta_{cov}^{PBCOV}$ as a function of the version ID and the maximum $\%\delta_{cov}^{K}$ among the structural coverage metrics. Figures 8, 9, 10, and 11 use the legends of the maximum $\%\delta_{cov}^{K}$ to distinguish the subject programs. For readability, the horizontal axis is sorted by the subject program then by $\%\delta_{cov}^{PBCOV}$. Note that structural coverage is high for all the 19 versions as shown in Appendix E. This category shows that PBCOV can detect deficient test suites where structural coverage metrics fail to do so.

Category 3 Category 3 shown in Fig. 9 has 8 versions. They exhibit little structural coverage decrease, but significant PBCOV decreases. Along with the category 2 versions,

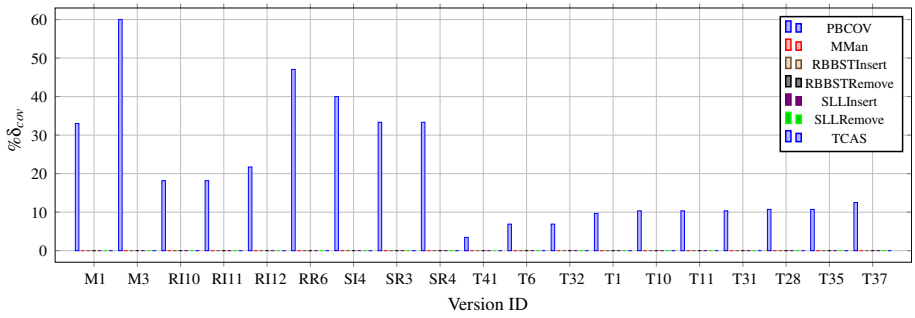


Fig. 8 Category 2 versions with no structural and with significant PBCOV decrease

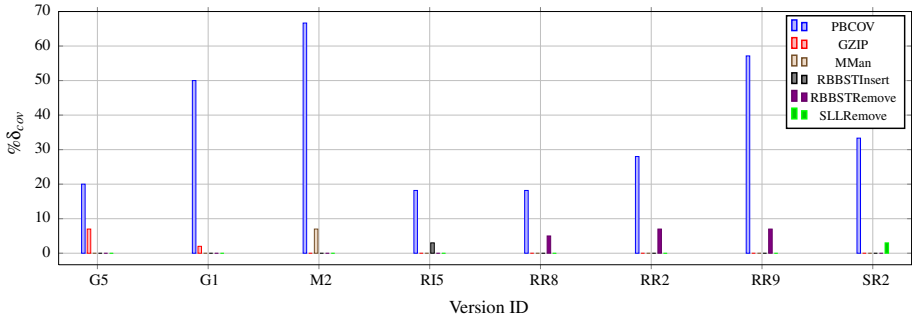


Fig. 9 Category 3 with little structural and significant PBCOV coverage decrease

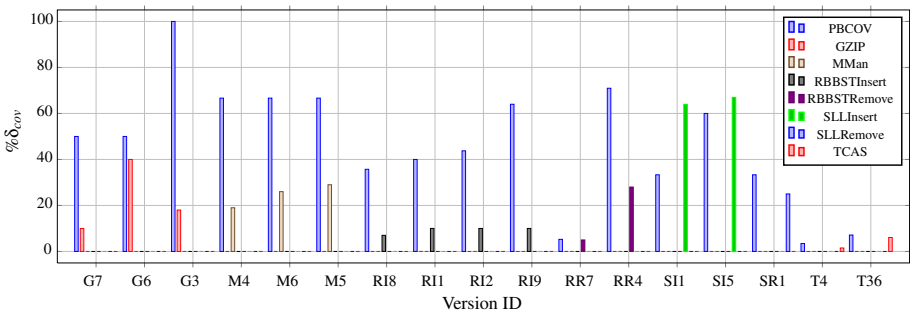


Fig. 10 Category 4 versions with significant structural and PBCOV coverage decrease

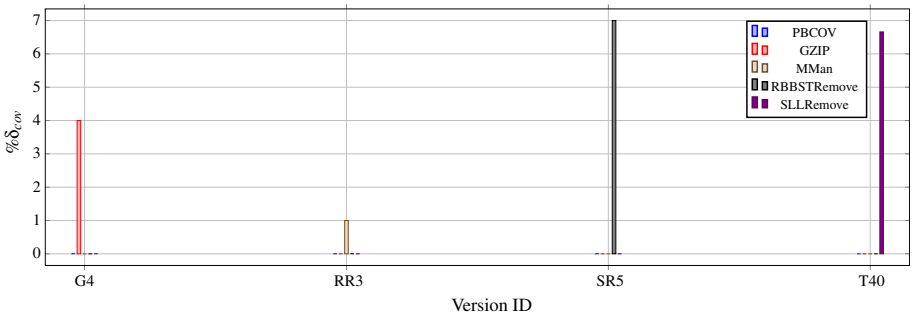


Fig. 11 Category 5 versions with no PBCOV and little structural coverage decrease

these versions show the utility of PBCOV to reveal defects that have little or no effect on structural elements.

Category 4 This category includes 17 versions, shown in Fig. 10, that exhibit significant decrease of structural coverage metrics, i.e., structural coverage is sensitive to the defects in these versions. Nevertheless, Fig. 10 shows that PBCOV is more sensitive in most cases.

This category describes defects where the missing states correspond to predicates in the programs that directly affect the structural elements.

Category 5 Category 5 shown in Fig. 11 has 4 versions exhibiting no change in PBCOV and a little change with structural coverage. T_{pass} violates properties with versions *GZIP-4* and *TCAS-40* which is sufficient to alarm the test engineer to fix the program before checking coverage results. Note that $\%\delta_{cov}^{K}$ for both versions is small (4 and 7 %) and the high structural coverage values of *TCAS* may still mislead the test engineer. *GZIP-4* computes the header length of the output file in an erroneous manner. All test cases compute the length, but only failing ones display it. So, even though T_{pass} exhibits a slight 4 % reduction in structural coverage, it does not reveal the defect. On the other hand, PBCOV exhibits no reduction but alarms the user due to a property (which checks the header length) that gets violated by all test cases.

SLLRemove-5 and *SLLInsert-5* share the same code, and the seeded defect therein is not exercised in the *Remove* function but in the *Insert* function where $\%\delta_{cov}^{PBCOV}$=60 % (see Fig.

10). *RBBSTRemove-3* shows a 1.3 % decrease in decision coverage. This version is the only case we encountered in our study in which structural coverage was sensitive to the defect, whereas PBCOV was not. Even though the decrease is minor, it is evidence that structural coverage metrics should still be used to reveal defects that correlate with structural features and that do not affect the state space of the current property annotations.

Category 6 Category 6 contains 29 versions where no structural or PBCOV change occured but where T_{pass} violated the properties, which is sufficient to alarm the test engineer. These 29 versions show the utility of property annotations to uncover defects.

In all, we observe that PBCOV was very sensitive to 27 defects (Categories 2 and 3) where structural coverage exhibited no or little decrease and was more sensitive to the rest of the defects in most cases.

The results for *SLL* differ from the trend we have seen with other programs:

1. For example, structural coverage yields better results for versions 1 and 5 as shown in Fig. 10, but still PBCOV decreased significantly. The defects in versions 1 and 5 are trivial and can be revealed by a majority of the test cases. As a result, T_{pass} included only 73 out of 1,751 test cases, as shown in Table E.4 in Appendix E, that did not execute most of the code. The defects in versions 2, 3, and 6 are not accessible from *SLLInsert,* and thus, they did not cause any reduction in coverage.
2. The defect in version 6 of *SLL* is seeded in the pointer of the removed element, and since failure is determined by comparing the resulting lists only, this defect goes undetected by all test cases. Nevertheless, the properties annotating *SSLRemove* check the return pointer. Therefore, on the one hand, PBCOV does not decrease because T_{pass} and T_{full} are identical, but on the other hand, one of the properties evaluates to `false` on two of the test cases, which alarms the tester to either fix the code or check the test suite.

TCAS is the only program where a considerable number of passing test cases violated the properties. Therefore, for each seeded version of *TCAS*, we also considered $T_{pass-assert}$, which is a subset of T_{pass} that excludes the test cases that violate properties. While $T_{pass-assert}$ is designed to have less state space coverage, we are interested in evaluating it using the structural coverage techniques. Our experiments show that compared to both T_{full} and T_{pass}, in most cases $T_{pass-assert}$ exhibits the same structural coverage. Figure 12 presents the results for the *Combined-Property*, i.e., the conjunction of the five *TCAS* properties. In the rest of the cases, $T_{pass-assert}$ exhibits sharper decrease in PBCOV relative to structural coverage.

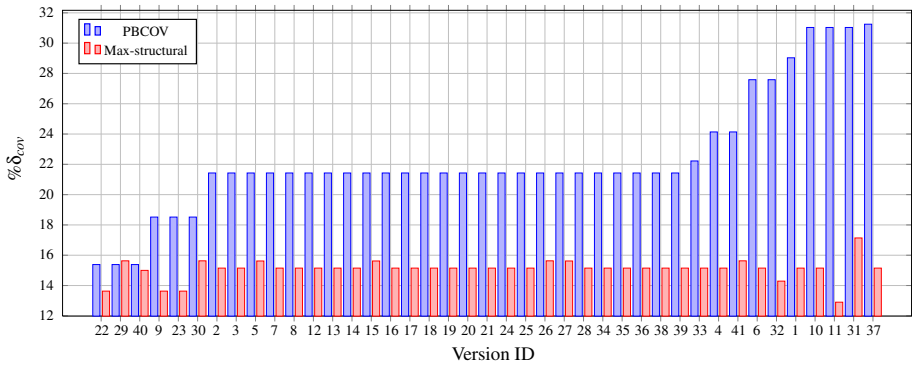


Fig. 12 *TCAS*–$\%\delta_{cov}$–Combined-Property: T_{full} vs. $T_{pass-assert}$

This is evidence of the utility of PBCOV in detecting deficiencies of test suites. The results for the five *TCAS* properties are detailed in Tables D.7–11 in Appendix D.

As expected, for each version where PBCOV decreased for any of the individual five properties, PBCOV also decreased for the *Combined-Property*. Interestingly, PBCOV decreased significantly for the *Combined-Property* in versions where it decreased slightly for only one of the individual five properties. This is due to the fact that the *Combined-Property* annotates the same position of code as the individual five properties, and thus, its state space is a product rather than a simple union of the state spaces of the five individual properties. To clarify, consider properties P_1 and P_2 each comprising two terms and four states. Consider property P that combines P_1 and P_2 and has four terms and 16 states. Given a test suite $T = \{t_1, t_2, t_3, t_4\}$ that covers the states $\{0000, 0001, 0100, 0101\}$ over P and the states $\{00, 01\}$ over each of P_1 and P_2. A reduced test suite $T' = \{t_1, t_4\}$ that covers only $\{0000, 0101\}$ over P also covers $\{00, 01\}$ over each of P_1 and P_2. Note how when going from T to T' state coverage decreased for the combined property P but it did not decrease for P_1 and P_2. Consequently, many states of the *Combined-Property* that can be excited with T_{full} cannot be excited with T_{pass}. This shows that by adding more properties to a program, one can scrutinize the efficiency of a test suite in an exponential fashion. This is a qualification that has no counterpart with structural coverage metrics.

6.4 PBCOV metric results

We used the SMT version of the tool CBMC (Clarke et al. 2004) to perform the feasibility checks as described in Sect. 3. We inserted the formula expressing the state s_{miss} as an assertion statement in S and asked the SMT version of CBMC to check for satisfiability. We allowed the CBMC tool to run for 30 min for each missing state and considered it to return an inconclusive result in case it timed out. We also set the *LIMIT* parameter from the algorithm in Figs. 3, 4 and 5. In practice, the cut-based abstraction ignored the whole program and only kept the property included in the formula before the fourth iteration of the cut-based abstraction. As for the timeout parameter, almost every inconclusive result from the checker took between 3 and 12 min and the few checks that took more than 12 min all timed out. So, the same PBCOV metric values would have been returned whether *LIMIT* was set to 3 or 5, and timeout was set to 12 or 30 min.

In this section, we present the results of the *RBBST* program in Tables 3 and 4. We summarize the rest of the results and discuss interesting cases. The rest of the results is described in Appendices D and E. The rows in the tables are merged when distinct versions

Table 3 PBCOV results for *RBBSTInsert*

Version	n_{cov}^{true}	n_{cov}^{false}	n_{feas}^{true}	n_{feas}^{false}	m_{pbcov}	m_{pbcov}^{con}
Original	18	0	181	6,219	33.6	56.59
1	18	12	181	6,219	39.19	N/A
2	18	14	181	6,219	39.9	N/A
3, 4, 6, 7	18	0	181	6,219	33.6	56.59
5	19	3	181	6,219	35.78	N/A
8	18	10	181	6,219	38.43	N/A
9	24	26	181	6,219	44.87	N/A
10, 11	18	4	181	6,219	35.78	N/A
12	18	5	181	6,219	36.27	N/A

Table 4 PBCOV results for *RBBSTremove*

Version	n_{cov}^{true}	n_{cov}^{false}	n_{feas}^{true}	n_{feas}^{false}	m_{pbcov}	m_{pbcov}^{con}
Original	18	0	180	3,404	35.98	56.65
1	19	0	180	3,404	36.61	57.63
2	18	7	180	3,404	39.81	N/A
3, 5	18	0	180	3,404	35.98	56.65
4	20	42	180	3,404	50.63	N/A
6	18	16	180	3,404	43.44	N/A
7	18	1	180	3,404	36.61	N/A
8	18	4	180	3,404	38.32	N/A

of a subject program yield similar results, as in the case of versions 3, 4, 6, and 7 in Table 3. We also do not compute the confident metric $\%\delta_{cov}^{con}$ where it is not defined such as for version 1 in Table 3 where the property is violated. The user needs to fix the code to satisfy the property before evaluating the confident metric.

The results in Table 3 shows the PBCOV metric results for the *RBBSTInsert* function. The *RBBST* property included 18 distinct terms, and thus, the state space comprises $2^{18} = 256$K states. The reachability analysis did not provide conclusive results on the full program, and thus, PBCOV used a cut of the program that included the property with the last two calls to the recursive *Insert* function to over-approximate the reachable state space of the program. This cut did not include any of the seeded defects, and thus, the number of feasible states was the same across all versions. This yielded a significant reduction from the possible state space (6,400 out of 256K states). The over-approximation with *Insert* was also due to some features of the CBMC tool. For example, CBMC considers pointer dereferencing as uninterpreted functions to account for pointer arithmetic. Thus, CBMC translates the expression `n->left->color`, where 'n', `left`, and `color` denote the node, a field therein pointing to the left *RBBST*, and the color of the node, respectively, to `color1(left1(n))`. It translates another occurrence of the expression `n->left->color` in the same property to `color2(left2(n))`. We did not attempt to fix this as we wanted our PBCOV results to reflect the existing state of the art tools. The fix could have been to store the result `n->left->color` in a temporary variable and use the variable in the property instead.

The PBCOV results of *RBBSTInsert* illustrate the utility of the logarithmic scale as a pedagogical tool to compensate for the notorious exponential explosion in the over-approximation of the reachable state space. For example, compared to the reported 33.6 % in Table 3, the ratio without the logarithmic scale for the original program would have reported 18/6,400 = 0.28 %, which might be misleading to a test engineer accustomed to existing coverage metrics.

The structural coverage metrics for the *RBBSTInsert* function in Table 2 show a high confidence in the test suite as they range between 80 and 100 %, while PBCOV shows little confidence with a metric value of 33.6 %. This is mainly due to the complex and pointer operations used in *Insert* function where reachability analysis does not scale well. PBCOV in here is a good indicator of the complexity of the subject program since test suites for complex programs should require continuous maintenance and PBCOV suggests exactly that in this case.

Table 4 shows the PBCOV metric results for the *RBBSTRemove* function. The state space includes $2^{17} = 128$K states. Similar to *RBBSTInsert*, the reachability analysis did not

provide conclusive results on the full program, and thus, PBCOV used a cut of the program that included the property with the last two calls to the recursive *Remove* function to over-approximate the reachable state space. This cut did not include any of the seeded defects, and thus, the number of feasible states was the same across all versions. Similar to *Insert*, the PBCOV coverage numbers in Table 4 show more conservative results compared to the structural coverage metrics in Table 2.

Similar to *RBBST*, the reachability analysis did not provide conclusive results for *GZIP*, *MMan*, and *SLL*. PBCOV used an abstraction of each program to over-approximate the reachable state space. The *GZIP* ^{cut} and *MMan* ^{cut} did not include the seeded defects, and thus, the number of feasible states was the same across all versions. *SLL* ^{cut} included the seeded defects and produced different numbers of feasible states across versions. The reachability analysis returned conclusive results on the *TCAS* program, and no abstraction was needed. The over-approximation of the reachable state space computed the number of feasible states to be up to: 8 out of 8K states for *GZIP*, 48 out of 256 states for *MMan*, 10 out of 64 states for *SLL*, and 20 out of 256 states for *TCAS*. This shows that the over-approximation can produce tight results even on programs with complex operations such as *GZIP*, unlike what we have seen with *RBBST*.

Property-15 for *GZIP* is interesting as it evaluated to `false` for all versions including the original version, i.e., the program originally violates *Property-15*. We checked the original version of the subsequent releases of *GZIP* and found out that release 1.3 available at SIR does not violate *Property-15* and shows a higher coverage with a higher n_{cov}^{true} and a zero n_{cov}^{false}. This shows the utility of property-based testing in finding defects and of PBCOV as an indicator of the adequacy of the test suite.

Finally, the original 8 user assertions of *GZIP* guard against erroneous inputs and boundary conditions and do not describe the general behavior of the program. The PBCOV metrics related to those assertions did not exhibit changes, whereas the 8 properties that we added by formalizing the English pre- and postconditions from comments that described the general behavior of *GZIP* better described the seeded defects. This shows that the quality of PBCOV can be enhanced by modifying the properties, which cannot be done for other coverage metrics.

6.5 Discussion of the results

We make the following additional observations in regard to Sects. 6.3 and 6.4.

1. In the presence of complex defects, PBCOV is likely to perform better than GCOV and ATAC. This is because these two structural coverage tools monitor relatively simple structural elements that might not be able to characterize complex defects (Masri et al. 2007; Masri 2010), whereas PBCOV monitors properties that typically describe the values and relationships of multiple variables and might lead to a better characterization of complex defects. This is illustrated in version 1 of *GZIP* shown in Fig. 9.
2. In most cases, the structural coverage metrics behave similarly, i.e., they either all decrease or they all remain unchanged. This could be explained by the subsumption relationships (Ammann and Offutt 2008) that exist between them, e.g., given that branch coverage subsumes statement coverage, a decrease in branch coverage should be accompanied by a decrease in statement coverage.
3. The PBCOV metric is very conservative compared to the GCOV and ATAC metrics. ATAC and GCOV predominantly reported structural coverage that exceeded 90%,

while PBCOV reported coverage that ranged between 30% and 98% while on the lower side in most cases.

4. Whenever a version seeded with defects reported higher coverage than the original version, and the seeded version did not violate the property, the difference between the confident and the actual coverage metrics either remained the same or increased indicating that the user is too confident to accept T_{full} (check version 1 in Table 4).
5. Whenever the reachability analysis reached a conclusive result on a missing state, it produced an input valuation that works well as a test case. This means that the PBCOV technique can be extended in future work to augment test suites.

6.6 Threats to validity

The first external threat to the validity of PBCOV is that it is not applicable in the absence of properties in source code. We admit that it was a hard task to find public programs already adequately annotated with meaningful properties. We think that with the emergence of formal verification tools, annotating code with properties will be a more common practice. In fact, Microsoft reports that in some of its most successful projects, the ratio of annotations to code is 1–10 (Woodcock et al. 2009).

The internal validity of PBCOV may be in question under the following conditions.

1. PBCOV is as good as the properties embedded in the code. The properties must describe the general behavior of the program and not only guard against illegal boundary behaviors. It is not trivial to write such properties, and PBCOV does not provide the means to assess the quality of the claimed properties, for example, in the form of a metric. We will explore providing such a metric in future work. Intuitively, we suggest considering a property to completely define the behavior of an output variable if it deterministically defines its value for each acceptable input.
2. Although the over-approximation of reachable states using symbolic analysis yielded significant reductions in the considered state space, it did not provide a tight approximation when the program had complex constructs, e.g., the *RBBST* program. This is expected to persist even with advances in static analysis research because of the nature of the reachability analysis notorious *state explosion* problem. A loose over-approximation of the feasible state space leaves the PBCOV metric with values lower than what testers are accustomed to with other coverage metrics. On the other hand, this can be viewed positively as an indication that the program is complex and requires continuous maintenance.
3. Satisfiability analysis often takes long before returning a non-satisfiable result, especially when dealing with large state spaces such as in the case of *RBBSTInsert*. This can be remedied by checking for the feasibility of sets of states that can be encoded with simpler formulae instead of checking one state at a time. For example, if a check returned that term p_1 is not satisfiable, then all the missing states containing p_1 as a factor would be dismissed.

7 Related work

Given a program with properties therein, PBCOV evaluates a test suite by studying the state space it covers. Below, we review and compare against several related approaches,

namely, *specification-based test generation*, *state-based coverage*, *specification-based coverage*, and *operational specifications*. PBCOV computes the sensitivity of a test suite to the presence of defects to compare against existing techniques. This is similar to *checked coverage* that assesses oracle quality based on the ratio of program elements covered by oracle checks (Schuler and Zeller 2011). Table 5 provides a comparative summary to related techniques.

Specification-based test generation Previous work on coverage and testing that involves specifications focused on methods and techniques to automatically generate test suites from specifications and properties (Khurshid and Marinov 2004; Boyapati et al. 2002; Gligoric et al. 2010). These techniques consider a precondition P as a conditional statement C and compute a test suite that provides full branch coverage to the conditions in C. TestEra (Khurshid and Marinov 2004) takes a program with a precondition and a postcondition and generates a test suite with all nonisomorphic test cases with respect to the structure of the precondition; the postcondition is used as an oracle. This test suite may produce high state coverage for the precondition, but does not deal with the state space of the postcondition. Also, TestEra does not generate test cases that violate the precondition. In practice, programs should return an error code on such test cases.

Korat (Boyapati et al. 2002) improves on TestEra by synthesizing the precondition into a Java predicate that can be executed to select test cases. UDITA (Gligoric et al. 2010) is a nondeterministic input specification language that specifies input descriptions. It is used to automatically generate complex input test cases that meet the UDITA description.

PBCOV differs in that it considers the reachable state space of all properties in the code including preconditions, postconditions, and invariants.

State-based coverage The closest work to PBCOV in state-based coverage is that of Ball (Ball 2004) as it introduces a theory for predicate-complete test coverage and generation. Given a program with n statements including m predicates, the full predicate state space of the program is $n \times 2^m$ states including all valuations of the predicates for each statement. The coverage metric proposed by Ball is the ratio of the covered predicate states against an approximation of the reachable observable predicate states. PBCOV differs in that (1) it considers only the atomic predicates in the properties and not in the code, as the code is suspect; (2) it computes the covered states in the context of the property, i.e., when the program counter is referring to the property, as opposed to program statements; and finally, (3) it uses symbolic execution with SMT and a cut-based abstraction to provide an over-approximation of the reachable property state space as opposed to the predicate abstraction and *modal* transitions used in (Ball 2004).

The work of Santelices et al. (Santelices et al. 2008) performs *test suite augmentation for evolving software* (TSAES). It takes the latest version of a program P, its previous version Q the set of changes c between P and Q and a distance d. It computes a slice of P that includes c and the statements in P which c depends on such that they are at a distance d from c. Then, TSAES performs symbolic execution on the slice to extract path conditions necessary for c to be exercised. TSAES checks whether the test suite covers these conditions using *dependence chain* and *state difference* coverage. This is similar to computing a precondition that is necessary to exercise c and then applying the precondition specification-based techniques discussed above (Khurshid and Marinov 2004; Boyapati et al. 2002; Gligoric et al. 2010). The dependence chain coverage requires that all dependency chains in c are covered. The state difference coverage computes the difference between the state space of the slice in P and that of the slice in Q and requires that all the difference states be covered.

Table 5 Summary of comparison to related work

Technique	Description	Comparison
Specification-based test generation	Khurshid and Marinov (2004), Boyapati et al. (2002), Gligoric et al. (2010) Use a precondition predicate that constrains the inputs of a program to generate test cases and attain acceptable coverage metrics	PBCOV considers the reachable state space of the property whether it was a precondition, a postcondition, an invariant, or an assertion to compute a coverage metric
Predicate-complete test coverage and generation	Ball (2004) Considers all valuations of all the predicates of the program across all its statements. The coverage metric proposed by Ball is the ratio of the covered predicate states against an approximation of the reachable observable predicate states. The approximation is based on predicate abstraction and modal transitions	PBCOV considers only the atomic predicates in the properties and not in the code, as the code is suspect PBCOV computes the covered states in the context of the property only and not all the statements of the code. PBCOV uses symbolic execution with SMT and a cut-based abstraction to provide an over-approximation of the reachable state space of the property
Test suite augmentation for evolving software (TSAES)	Santelices et al. (2008) Augments existing test suite with additional test cases to cover the changes between two versions of a program. It measures the coverage of path conditions necesary to exercise the code changes using dependence chainstate difference coverage. This requires that the difference state space incurred by the changes be all covered	PBCOV differs in that it uses symbolic execution to over-approximate the reachable state space while TSAES uses it to compute conditions for coverage PBCOV uses state differencing to compute the missing states and report the coverage metric, whereas TSAES uses it to compute state coverage requirements PBCOV checks the adequacy of the test suite against the whole program on existing requirements that express the intended behavior whereas TSAES checks whether the test suite is adequate for testing the new behavior in P
Specification-based coverage (SBC)	Ammann and Black (2001) Computes execution traces from test cases and computes mutated specifications by altering specification elements, and then uses model checking to check whether the test cases can *kill* the mutant specifications. The more mutants the test cases can kill, the better they are	PBCOV subsumes SBC coverage since SBC requires test cases that cover feasible states in the specification PBCOV requires an overapproximation of the reachable states, wheras SBC mutants are based on syntactic rules which may include mutants that can only be killed through unreachable states PBCOV requries coverage of all the state space of the specification, whereas the mutants generated by SBC might all be killed by test cases that all correspond to one state
Operational specifications	Harder et al. (2003) Uses Daikon and the existing test suite to generate an operational abstraction of the program, requires test cases that modify the abstraction, and does not provide a quantifiable metric	PBCOV requires a set of properties, measures the ability of the test suite to cover the reachable state space of the properties, and provides a metric

Similar to PBCOV, TSAES uses partial symbolic execution and symbolic state differencing. However, TSAES uses partial symbolic execution to compute conditions for coverage, while PBCOV differs in that it symbolically over-approximates the reachable state space. TSAES uses state differencing to compute state coverage requirements, whereas PBCOV uses state differencing to compute the missing states. TSAES computes full state coverage for the difference states obtained from the generated transient requirements, while PBCOV computes coverage of an over-approximation of the reachable state space.

At a higher level, TSAES checks whether the test suite is adequate for testing the new behavior in P while PBCOV differs in that it checks the adequacy of the test suite against the whole program on existing requirements that express the intended behavior of P.

Specification-based coverage Ammann and Black (Ammann and Black 2001) introduced specification-based coverage (SBC) using specification mutation analysis. SBC measures the adequacy of a test suite against a set of specifications expressed in computational tree logic (CTL) (Clarke et al. 1999). It uses mutation analysis (DeMillo et al. 1978) where it mutates specification elements such as variables and operators to compute several inaccurate specifications. It computes execution traces from the test cases and passes the specification mutants, and the execution traces to a model checker (McMillan 1998) where the execution traces are the reference. The model checker reports the mutant specifications that fail as *killed* by the test cases. The more mutants the test cases can kill, the better they are. The mutant space is infinite, and SBC can only generate a finite set of mutants.

Consider Q, a mutant of a specification P that may introduce new atomic predicates to P, and consider the state space that includes all valuations to the atomic predicates $\{p_1, p_2, \ldots, p_n\}$ in P and Q. SBC deems a test suite inadequate if it contains no test case that kills/violates Q. Requiring such a test case is equivalent to requiring a test case that covers a state where P is met and Q is violated, which implies covering some feasible state of P. Thus PBCOV subsumes killing the specification mutants without the need to explicitly generate those mutants. In fact, PBCOV considers only an over-approximation of the reachable states, while SBC generate mutants based on syntactic rules which may include mutants that can only be killed through unreachable states. For example, in a correct program, a specification P will not be violated and a trivial mutant ($\neg P$) could never be killed.

On the other hand, one may fulfill the SBC requirements without fulfilling the PBCOV requirements. Given a specification P, let $S(T)$ be the set of states covered by the test suite T, and let $S(Q_1), S(Q_2), \ldots,$ and $S(Q_m)$ be the sets of states that satisfy each of the m mutants of P generated by SBC, respectively. SBC requires that the set $S(T) \cap S(\neg Q_i)$ be not empty for all $1 \leq i \leq m$. One possibility to meet this condition is for the set $\{S(T) \cap S(\neg Q_1) \cap (\neg Q_2) \cap \ldots \cap S(\neg Q_m)\}$ to contain only one state e that can kill all the mutants. PBCOV requires all the reachable state space to be covered, including e, which suggests that PBCOV subsumes SBC. On the other hand, given that $S(T)$ satisfies SBC does not imply that PBCOV is also satisfied, because $S(T)$ might not include a number of reachable states of P.

Earlier work introduced ADLscope (Chang and Richardson 1999), a tool that measures the adequacy of test suites with respect to specifications written in the ADL specification language. The ADLscope tool computes coverage metrics that are associated with the expression syntax of the ADL language. The multiple condition strategy is used with logical expressions; weak mutation testing is used with relational expressions, and other ADLscope-specific metrics are used with the rest of the expressions such as conditionals,

Table 6 Glossary table

Term	Description
P	First-order and temporal logic property
S	Program under test
T	Full test suite used used interchageably with T_{full}
T_{full}	Full test suite used to stress the fact that the full test suite is used
T_{pass}	Subset of T_{full} that comprises only passing test cases
$T_{pass-assert}$	Subset of T_{pass} that comprises only passing test cases that did not violate the property
T_{fail}	Subset of T_{full} that comprises only failing test cases
m_{pbcov}	Property-based coverage metric
m^{con}_{pbcov}	Confident property-based coverage metric
K	Structural coverage metric
K_{full}	The number of structural elements covered by T_{full}
K_{pass}	The number of structural elements covered by T_{pass}
K_{total}	The total number of structural elements
$\%\delta_{cov}$	Percentage decrease in coverage for coverage metrics
P_{sym}	Symbolic representation of P
Term	Description
P_{pass}	The set of feasible states that *satisfy* the property P
P_{fail}	The set of feasible states that *fail* the property P
P_{miss}	The missing states that are not covered and are deemed in the over-approximation of the reachable state space
P_{cover}	Symbolic representation of covered states that is deterministic for states observed by T and S, and nondeterministic for the rest
C	A cut in the program which is a set of control points that split S into S_C and $S_{\overline{C}}$
S_C	A cut of S along C that contains the property P
$S_{\overline{C}}$	A cut of S along C that does not contain the property P
S_i	Instrumented version of S
x_i	Program and property variable where $1 \leq i \leq m$ and m is the number of variables
D_i	Domain of variable x_i
D	Domain of property variables $D = D_1 \times D_2 \times \ldots D_m$
p_i	Atomic predicate in property P, $1 \leq i \leq n$ and n is the number of atomic predicates in P
t_i	A test case in T such that $1 \leq i \leq \|T\|$

quantifiers, and others. The paper omits the details about these metrics and refers to earlier work that in turn refers to a technical report that we were not able to locate.

Operational specifications Harder *et al* (Harder et al. 2003) present the *operational difference* (OD) technique for generating, augmenting, and minimizing test suites. Given a program S and a test suite T, the technique uses Daikon (Ernst et al. 2007), an automatic invariant detection tool, to generate a set of formal specifications and calls them an *operational abstraction* of the program. If a test case modifies the operational abstraction of the program, then it is added to the test suite; otherwise, it is discarded if appropriate. The OD technique measures the ability of the test suite to generate an ideal operational

abstraction that is not available at hand; thus, it does not provide a quantifiable metric. In contrast, PBCOV requires a set of properties, measures the ability of the test suite to cover the reachable state space of the properties, and provides a metric.

8 Conclusion and future work

Verification engineers leverage coverage metrics to acquire informal assurance of the adequacy of test suites so that testing could be stopped. In recent years, annotating code with formal properties in the form of preconditions, postconditions, and invariants has become more practiced. Formal properties describe the behavior of a program and act as a reference of its correctness. Given a program with properties therein and a test suite, this paper presented PBCOV, a property-based coverage metric that measures the adequacy of the test suite at revealing the defects in the program. PBCOV measures the covered state space of the properties against an over-approximation of the reachable state space of the properties. The paper also compared the PBCOV metric to traditional structural coverage metrics, and the experimental results showed that PBCOV was in most cases more sensitive to the presence of defects than the structural coverage metrics are. PBCOV reports an additional confident coverage metric that serves as an indicator of how much a test engineer is confident when he or she deems a test suite adequate.

The quality of the PBCOV metric depends on the quality of the properties embedded in the code. In the future, we will explore techniques to check how complete these properties are with respect to the program.

When PBCOV reports missing states, it also reports values for program variables that may be at the input level, and in that case, those values can be used directly to augment the test suite. More often, those values are for internal program variables. In the future, we will enhance PBCOV to compute test cases that induce the values of the reported internal variables using SAT techniques.

References

ABC. (2007). ABC: Berkeley logic synthesis and verification group. a system for sequential synthesis and verification, release 70930. http://www.eecs.berkeley.edu/alanmi/abc/.

Ammann, P., & Black, P. E. (2001). A specification-based coverage metric to evaluate test sets. *International Journal of Reliability, Quality and Safety Engineering*, *8*(4), 239–248.

Ammann, P., & Offutt, J. (2008). *Introduction to software testing* (1st ed.). New York, NY: Cambridge University Press.

Ball, T. (2004). A theory of predicate-complete test coverage and generation. In *In FMCO 2004: Symposium on formal methods for components and objects*, pp 1–22.

Ball, T., Majumdar, R., Millstein, T., & Rajamani, SK. (2001). Automatic predicate abstraction of c programs. In *Programming language design and implementation, ACM*, New York, NY, USA, PLDI '01, pp. 203–213.

Ball, T., Levin, V., & Rajamani, S. K. (2011). A decade of software model checking with slam. *Communications of the ACM*, *54*, 68–76.

Barr, A. (2004). *Find the bug: A iook of incorrect programs*. Reading: Addison-Wesley Professional.

Baudin, P., Filliâtre, J.C., Hubert, T., Marché, C., Monate, B., Moy, Y., et al. (2009). ACSL: ANSI C specification language (preliminary design V1.9). http://www.frama-c.cea.fr/acsl.html.

Boyapati, C., Khurshid, S., & Marinov, D. (2002). Korat: Automated testing based on java predicates. In *International symposium on software testing and analysis (ISSTA)*, pp. 123–133.

Burnim, J., & Sen, K. (2008). Heuristics for scalable dynamic test generation. In *International conference on automated software engineering*, pp. 443–446.

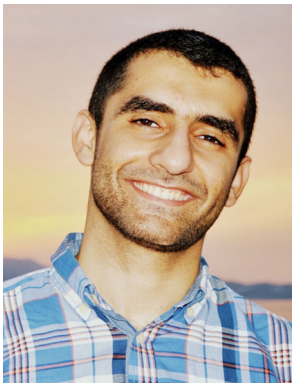
Chang, J., & Richardson, DJ. (1999). Structural specification-based testing: Automated support and experimental evaluation. In *European, software engineering conference*, ESEC/FSE-7, pp. 285–302.
Clarke, E., Brumberg, J. O., & Peled, D. A. (1999). *Model checking*. Cambridge: MIT Press.
Clarke, E., Kroening, D., & Lerda, F. (2004). A tool for checking ansi-c programs. In *Tools and algorithms for the construction and analysis of systems*, pp. 168–176.
Coen-Porisini, A., Denaro, G., Ghezzi, C., & Pezzé, M. (2001). Using symbolic execution for verifying safety-critical systems. *ACM SIGSOFT Software Engineering Notes*, *26*, 142–151.
Cormen, T. H., Stein, C., Rivest, R. L., & Leiserson, C. E. (2009). *Introduction to algorithms* (3rd ed.). Cambridge: MIT Press.
DeMillo, R. A., Lipton, R. J., & Sayward, F. G. (1978). Hints on test data selection: Help for the practicing programmer. *Computer*, *11*, 34–41.
Do, H., Elbaum, S., & Rothermel, G. (2005). Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, *10*, 405–435.
Dutertre, B., & Moura, L. M. D. (2006). A fast linear-arithmetic solver for dpll(t). *Computer Aided Verification*.
Ernst, M. D., Perkins, J. H., Guo, P. J., McCamant, S., Pacheco, C., Tschantz, M. S., & Xiao, C. (2007). The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, *69*, 35–45.
Ford, L. R., & Fulkerson, D. R. (1956). Maximal flow through a network. *Canadian Journal of Mathematics, 8*, 399–404.
Gligoric, M., Gvero, T., Jagannath, V., Khurshid, S., Kuncak, V., & Marinov, D. (2010). Test generation through programming in udita. In *International conference on software engineering, ACM*, ICSE '10.
Godefroid, P., Klarlund, N., & Sen, K. (2005). Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on programming language design and implementation, ACM*, PLDI '05.
Gough, B. J., & Stallman, R. M. (2005). *An introduction to GCC*. Network Theory Ltd
Harder, M., Mellen, J., & Ernst, M. D. (2003). Improving test suites via operational abstraction. In *International conference on software engineering*, pp. 60–71.
Heimdahl, M. P. E., Rayadurgam, S., Visser, W., Devaraj, G., & Gao, J. (2003). Auto-generating test sequences using model checkers: A case study. In Workshop on formal approaches to testing of software, pp 42–59.
Holzmann, G. J. (1997). The model checker spin. *IEEE Transactions on Software Engineering*, *23*, 279–295.
Horgan, J. R., & London, S. (1991). Data flow coverage and the c language. In *Proceedings of the symposium on testing, analysis, and verification*, TAV4, pp. 87–97.
Jaygarl, H., Lu, K. S., & Chang, C. K. (2010). Genred: A tool for generating and reducing object-oriented test cases. In *IEEE annual computer software and applications conference*, pp. 127–136.
Khurshid, S., & Marinov, D. (2004). Testera: Specification-based testing of java programs using sat. *Automated Software Engineering*, *11*, 403–434.
Linz, P. (2012). *An introduction to formal languages and automata* (5th ed.). Burlington: Jones and Bartlett Learning.
Martinian, E. (2010). Red-black tree c code. http://www.mit.edu/emin/source_code/red_black_tree.
Masri, W. (2010). Fault localization based on information flow coverage. *Software Testing, Verification and Reliability*, *20*, 121–147.
Masri, W., & Abou-Assi, R. (2010). Cleansing test suites from coincidental correctness to enhance fault-localization. In *International conference on software testing, verification and validation*, ICST '10.
Masri, W., Podgurski, A., & Leon, D. (2007). An empirical study of test case filtering techniques based on exercising information flows. *IEEE Transactions on Software Engineering*, *33*, 454–477.
McMillan, K. L. (1998). The smv language: Cadence berkeley labs. Technical report.
Necula, G. C., Mcpeak, S., Rahul, S. P., & Weimer, W. (2002). Cil: Intermediate language and tools for analysis and transformation of c programs. In *International conference on compiler, construction*, pp. 213–228.
PBCOV-APPENDICES. (2013). PBCOV-APPENDICES. http://webfea.fea.aub.edu.lb/fadi/dkwk/doku.php?id=pbcov.
PBCOV-TOOL. (2013). PBCOV-TOOL. http://webfea.fea.aub.edu.lb/fadi/dkwk/doku.php?id=pbcov.
Rapps, S., & Weyuker, E. J. (1982). Data flow analysis techniques for test data selection. In *International conference on Software engineering* (pp. 272–278). CA, USA: Los Alamitos.
Santelices, R. A., Chittimalli, P. K., Apiwattanapong, T., Orso, A., & Harrold, M. J. (2008). Test-suite augmentation for evolving software. In *ASE*, pp 218–227.
Schuler, D., & Zeller, A. (2011). Assessing oracle quality with checked coverage. In *International conference on software testing, verification and validation*, pp. 90–99.

Torlak, E., & Jackson, D. (2007). Kodkod: A relational model finder. In *Proceedings of the 13th international conference on tools and algorithms for the construction and analysis of systems*, TACAS'07.
Visser, W., Havelund, K., Brat, G. P., Park, S., & Lerda, F. (2003). Model checking programs. *Automated Software Engineering Journal*, *10*(2), 203–232.
Woodcock, J., Larsen, P. G., Bicarregui, J., & Fitzgerald, J. (2009). Formal methods: Practice and experience. *ACM Computing Surveys*, *41*, 19:1–19:36.
Yang, J., & Evans, D. (2004). Dynamically inferring temporal properties. In *SIGPLAN-SIGSOFT workshop on program analysis for software tools and engineering*, PASTE '04, pp 23–28.
Yang, X., Wang, J., & Yi, X. (2010). Slicing execution with partial weakest precondition for model abstraction of c programs. *The Computer Journal*, *53*(1), 37–49.
Zaraket, F., & Masri, W. (2009). Property based coverage criterion. In *International workshop on defects in large software systems*, DEFECTS '09, pp. 27–28.

Kassem Fawaz received the BE degree with high distinction and the ME degree in computer and communications engineering from the American University of Beirut in 2009 and 2011, respectively. Currently, he is a PhD candidate at the University of Michigan, where he is doing work in the areas of mobile computing and privacy. He received the Distinguished Graduate Award upon graduation in 2009 and has published 15 papers in web systems and pervasive computing.

Fadi Zaraket is an Assistant Professor in the Electrical and Computer Engineering Department at the American University of Beirut (AUB). His research interests are logic synthesis and verification, and natural language processing. He received his PhD in computer engineering from the University of Texas at Austin, and his Masters and Bachelor of Engineering degrees from AUB. From 2001 until 2009, Fadi worked in the design automation technology and logic verification field for IBM. Before that, he worked for several companies including Sun Microsystems and SCO on building kernel modules.

Wes Masri is an Associate Professor in the Electrical and Computer Engineering Department at the American University of Beirut. He received his PhD in computer engineering from Case Western Reserve University in 2005, his MS in electrical engineering from Penn State in 1988, and BS in electrical engineering also from CWRU in 1986. His research interest is in devising software analysis techniques that enhance software testing, software security, and fault localization. He also spent over fifteen years in the software industry mainly as a software architect and developer. Some of the industries he was involved in include medical imaging, middleware, telecom, genomics, semiconductor, document imaging, and financial.



Hamza Harkous is a PhD student in the School of Computer and Communication Sciences at École Polytechnique Fédérale de Lausanne (EPFL). His current research revolves around privacy of information sharing, especially in personal cloud computing systems. He completed his masters studies in Communication Systems at EPFL, conducting his thesis project in collaboration with Nokia Research Center, and received his Bachelor of Computer and Communications Engineering from the American University of Beirut. He previously joined research projects on adhoc networks privacy, software verification, and natural language processing.