Bidirectional Symbolic Analysis for Effective Branch Testing

Mauro Baluda, Giovanni Denaro and Mauro Pezzè

Abstract—Structural coverage metrics, and in particular branch coverage, are popular approaches to measure the thoroughness of test suites. Unfortunately, the presence of elements that are not executable in the program under test and the difficulty of generating test cases for rare conditions impact on the effectiveness of the coverage obtained with current approaches.
In this paper, we propose a new approach that combines symbolic execution and symbolic reachability analysis to improve the effectiveness of branch testing. Our approach embraces the ideal definition of branch coverage as the percentage of executable branches traversed with the test suite, and proposes a new bidirectional symbolic analysis for both testing rare execution conditions and eliminating infeasible branches from the set of test objectives. The approach is centered on a model of the analyzed execution space. The model identifies the *frontier* between symbolic execution and symbolic reachability analysis, to guide the alternation and the progress of bidirectional analysis towards the coverage targets.
The experimental results presented in the paper indicate that the proposed approach can both find test inputs that exercise rare execution conditions that are not identified with state-of-the-art approaches and eliminate many infeasible branches from the coverage measurement. It can thus produce a modified branch coverage metric that indicates the amount of feasible branches covered during testing, and helps team leaders and developers in estimating the amount of not-yet-covered feasible branches. The approach proposed in this paper suffers less than the other approaches from particular cases that may trap the analysis in unbounded loops.

Index Terms—Structural testing, Branch coverage, Program analysis, Symbolic execution, Symbolic reachability analysis.

1 INTRODUCTION

Structural testing has been studied since the early sixties and it is now commonly used in many industrial settings as a proxy measure of the thoroughness of test suites. Classical and recent empirical studies argue in favor of a relation between high coverage scores and high software quality levels [1], [2]. Unfortunately the existence of such a relation is not well supported experimentally yet, in particular when considering branch coverage, because of the difficulty of consistently achieving high branch coverage. In their study, Hutchins et al. argue that a relationship between coverage and effectiveness requires more than 90% coverage, and they confirm that such branch coverage levels are very difficult to reach [1]. Inozemtseva and Holmes argue that such correlation may not exist in general, but confirm the lack of data about a possible correlation in the presence of high branch coverage [3].

The problem of low branch coverage may depend on both the difficulty of covering feasible branches and the presence of many infeasible ones. In the first case a low branch coverage indicates the presence of branches that shall be exercised during testing, and suggests that the current test suite shall be augmented with further test cases, while in the second case a low branch coverage does not indicate a lack of test cases, but is simply due to the program structure.

Most classic and recent research on structural testing has focused mainly on increasing coverage, and in particular branch coverage, neglecting the problem that some code elements can be infeasible. Recent research studies have exploited symbolic and concolic execution [4], search based approaches [5] and random test case generation [6] to increase branch coverage. These approaches increase the amount of executed branches, but may miss some relevant corner cases, and may still result in low branch coverage due to the presence of loops and infeasible elements. The branch coverage obtained with all these techniques tends to grow fast initially and to stabilize while progressing with the test case generation. However, when the branch coverage stops growing, it is difficult to deduce to what extent the missing coverage depends on feasible, though not-yet-covered, or infeasible branches.

Many systems present special cases that depend on the execution of complex paths, loops and conditions, and that can be exercised only with particular input values. Such cases are not only difficult to execute, but may represent sinks that trap the analysis that underlies the test generation mechanisms, preventing progress. The heuristics proposed so far succeed in some cases, but do not deal with all cases [7], [8], [9].

- *M. Baluda is with the Secure Software Engineering Group, Fraunhofer SIT, Darmstadt, Germany.
E-mail: mauro.baluda@sit.fraunhofer.de*
- *G. Denaro is with the Department of Informatics, Systems and Communication, Università di Milano-Bicocca, Milano, Italy.
E-mail: denaro@disco.unimib.it*
- *M. Pezzè is with the Faculty of Informatics, Università della Svizzera italiana, Lugano, Switzerland and with the Department of Informatics, Systems and Communication, Università di Milano-Bicocca, Milano, Italy.
E-mail: pezze@disco.unimib.it*

While augmenting the test suite to cover not-yet-executed branches is extremely useful to thoroughly test the program, the presence of many infeasible branches may keep the classic branch coverage score quite low, and this does not help team leader and developers, who are seeking for metrics that can indicate if the program branches have been thoroughly tested or not. This does not help researchers either, who are seeking for evidence of the existence of correlation between high branch coverage and test effectiveness. In the presence of many infeasible branches, even a test suite that executes all feasible branches in a program may result in a low branch coverage. To tackle this problem we need to increase the amount of executed branches, identify infeasible branches, and define a measure that approximates well the percentage of feasible branches executed with the test suite, by taking into account executed as well as infeasible branches.

The problem of infeasible elements is currently addressed by either accepting incomplete coverage values, as in most industrial settings, or, more rarely, by manually inspecting the uncovered elements, especially when required by mandatory standards, like DO-178C [10]. Incomplete coverage measures may miss important corner cases that can induce severe problems after testing, while manually inspecting uncovered elements can be very expensive, and is thus justified only for safety critical applications.

The problem of identifying infeasible program elements has been studied in the more general context of verifying code assertions [11], [12], [13], [14], [15], [16], [17]. The infeasibility of a single or a small subset of branches can be demonstrated by showing the unsatisfiability of assertions suitably added to the target branches. Such approaches do not scale well in the presence of large numbers of branches to be verified.

Merging approaches to execute not-yet-covered elements with approaches to exclude infeasible ones is far from trivial. Simply combining complementary approaches leads to slow and inefficient solutions. Performance deteriorates both when being trapped in long sequences of attempts to execute elements that may be later revealed infeasible, and when failing multiple times to demonstrate the infeasibility of elements that may be later revealed executable. A good alternation of the different approaches depends on the analyzed programs and cannot be defined a-priory [18].

In this paper we present an original solution that targets high branch coverage, and produces a branch metric that better approximates the amount of feasible branches that are exercised with the test suite. Our solution combines approaches to execute non-yet-covered elements with approaches to exclude infeasible elements in a new way, overcoming the limitations that constrain heuristics solutions. The core new idea of our approach is to undertake a bidirectional exploration of the execution space by means of a technique that we call *bidirectional symbolic analysis*. Bidirectional symbolic analysis combines forward analysis in the form of symbolic execution to execute not-yet-covered elements and backward analysis in the form of symbolic reachability analysis to identify reachability conditions and reveal unreachable states.

Bidirectional symbolic analysis coordinates the forward and backward analysis through a model, the Generalized Control Flow Graph (GCFG), that captures the state of the analysis at each step, and indicates the ideal directions of progress for the next steps. The model avoids indefinite growth by removing information that is not needed any more through the progress of the analysis, thus supporting the scalability of the approach.

The GCFG models the exploration space of the program under analysis by integrating the program control flow graph with the state transition systems computed by both symbolic execution and symbolic reachability analysis. The model accounts for the reachability of states and transitions, and allows both forward and backward analysis to benefit from each other progress. The model identifies the unexplored reachability conditions that are contiguous to some already explored symbolic state, forming pairs of contiguous symbolic states and reachability conditions that we call the *frontier* of the bidirectional symbolic analysis. The frontier states represent the best candidates for increasing branch coverage by i) extending executed states towards reachability conditions of uncovered branches, ii) refining reachability conditions that cannot be satisfied from the currently executed states, iii) identifying unsatisfiable reachability conditions to reveal infeasible branches. The identification of the frontier states prevents forward and backward analyses to be trapped in indefinitely long explorations of states that do not specifically correlate with coverage increase, as it happens for heuristic based approaches. The experimental results presented in the paper indicate that the approach can indeed cover a high portion of feasible branches and produce a coverage indicator that well approximates the amount of executed feasible branches.

We introduced the idea of combining forward and backward analysis and an initial set of preliminary results in [19] and [18]. In both previous papers, the analysis is presented without a formalization of reference model, and the early version of the technique did not support pointer aliases and interprocedural analysis, and thus did allow an experimental evaluation on simple synthetic programs only.

This paper extends the previous work by defining the GCFG and formalizing its role in bidirectional symbolic analysis, clarifying the intertwining between forward and backward analysis by means of the formalized model, introducing the interprocedural extension of the analysis and its implementation, evaluating the approach with realistic programs that include both data structures and pointers, providing experimental evidence of the effectiveness of the approach, and comparing the approach with state-of-the-art techniques, namely KLEE [20] and CREST [7]. The extensions of the preliminary work towards interprocedural analysis and dynamic data structures allowed us to extend the experiments that in the previous work were limited to synthetic programs towards real programs. In this paper we report the results obtained on programs that are used as benchmark in the literature, and verify the applicability of the approach in the presence of interprocedural code and dynamic data structures, highlighting the different impact both of bidirectional analysis in identifying rare execution conditions and of considering infeasible branches when quantifying branch coverage.

The paper is organized as follows. Section 2 presents a motivating example that illustrates the limits of the state-of-the-art approaches and that we use as running example in the paper. Section 3 presents the GCFG model that we use to guide the progress of bidirectional symbolic analysis. Section 4 introduces the algorithms that comprise the bidirectional symbolic analysis. Section 5 describes our technique for automatic test case generation that exploits bidirectional symbolic analysis to pursue high branch coverage. Section 6 presents the interprocedural extension of the technique. Section 7 discusses our experiments on the effectiveness of the approach. Section 8 surveys the related work in the area of automatic test case generation and reachability analysis. Section 9 summarizes the results of this research and outlines future directions.

2 Motivating Example

In this section we illustrate the advantages and limits of symbolic execution, and introduce a running example that we use in the paper to illustrate the details of bidirectional symbolic analysis.

Figure 1 presents a simple controller of a set of valves. The C program `checkValves` checks the status of a hydraulic system by counting the number of valves that are out of order with the loop at the lines 22—29, and fires an alarm if the number of malfunctioning valves exceeds a predefined tolerance level (line 32). The status of each valve is accessed by calling the function `getStatusOfValve` at line 23. The function checks that the parameter `i` refers to a valid location of an array of sensor data, and aborts the program if this check fails (lines 7—12). Otherwise, it returns the status of the requested valve according to the sensor data (lines 14—15). The `while` loops at lines 20 and 34 mock other tasks that the program may be involved in.

The correctness of the program depends on the executions of the program in response to external inputs. The goal of testing is to exercise a finite sample of executions to improve the confidence on the correctness of the program behavior. In particular, branch testing measures the effectiveness of a test suite in terms of executed program branches, and identifies branches that have not been executed yet. Symbolic execution can be used to improve branch coverage, by identifying the

```
1  # define VALVE_KO(status) status==-1
2  # define TOLERANCE 2
3  extern int size;
4  extern int valvesStatus[];
5
6  int getStatusOfValve(int i){
7      if(i<0
8          ||
9         i>=size){
10         printf ("ERROR");
11         exit(EXIT_FAILURE);
12     }
13
14     int status=valvesStatus[i];
15     return status;
16 }
17
18 int checkValves(int wait1, int wait2) {
19     int count, i;
20     while(wait1 > 0) wait1--;
21     count=0, i=0;
22     while(i<size){
23         int status=getStatusOfValve(i);
24
25         if(VALVE_KO(status)) {
26             count++;
27         }
28         i++;
29     }
30
31     if(count>TOLERANCE)
32         printf ("ALARM");
33
34     while(wait2 > 0) wait2--;
35
36     return count;
37 }
```

Fig. 1. The C program `checkValves`

execution conditions of branches that have not been executed yet [21], [22]. Symbolic execution executes the program with symbolic input values and builds the condition on the input values for executing a path in the program.

Figure 2(a) exemplifies symbolic execution on a path of the program in Figure 1 executed with initial symbolic values `S1` for `size`, `W1` for `wait1`, `W2` for `wait2` and `[V1,V2,V3,...]` for `valvesStatus`. The figure indicates the sequence of symbolic states built during symbolic execution, starting from the entry of the program at line 18 and executing the loop at lines 22—29 three times before terminating. The nodes are labeled with the corresponding statement numbers, and represent the program state after executing the corresponding statement. The letters in the labels of the nodes incrementally distinguish nodes that correspond to different executions of the same statements. Nodes labeled with numbers of blank lines represent implicit statements, like a missing `else` statement (for instance line 33), a loop exit (for instance lines 30 and 35) and a return point (for instance line 24). The edges are labeled with the branch conditions that guard the transition between the corresponding symbolic states.

The symbolic execution first analyzes the statements 18 and 19 and builds the symbolic entry state S_{18a}, then executes three times the loop at lines 22—29. In the first iteration, it builds the symbolic states S_{20a}, S_{21a}, S_{22a}, S_{23a}, S_{24a}, S_{25a}, capturing the call to `getStatusOfValve(0)` with the symbolic states S_{23a} and S_{24a} that represent the call and return statement, respectively, and building the state S_{25a} reached when the first valve is out of order. The dotted line between S_{23a} and S_{24a} implicitly represents the states that correspond to the execution of the called function. The second and third iterations build similar sequences of states, but execute line 28 (states S_{28a} and S_{28b}) instead of line 25, since we assume that the symbolic execution follows the path along which the other two valves work correctly. The execution proceeds with the symbolic states S_{30a}, S_{33a}, S_{34a}, S_{35a} and S_{36a} that lead to the termination of the program. The states S_{20a} and S_{34a} represent the iterations of the loops at lines 20 and 34, respectively.

The bottom of the figure reports the complete symbolic representation of the states S_{22a}, S_{24b} and S_{30a}. The symbolic state S_{22a} represents the state of the program when entering the loop at line 22 for the first time: the variable `size` holds a value, denoted with the symbol $S1$, greater than 0, the parameter `wait1` holds a value, denoted with the symbol $W1$, equal to 1, and the other program variables are assigned as `i`$=0$ and `count`$=0$. In state S_{24b}, which represents the symbolic state after returning from `getStatusOfValve(1)` during the second iteration of the loop, the variable `size` holds a value greater than 1, the first value in the input array, denoted with the symbol $V1$, is equal to -1, and the other program variables are assigned as follows: `i`$=1$, `count`$=1$ and `status`$=V2$, where the symbol $V2$ denotes the second value in the array. S_{30a} represents the symbolic state after the three considered loop iterations.

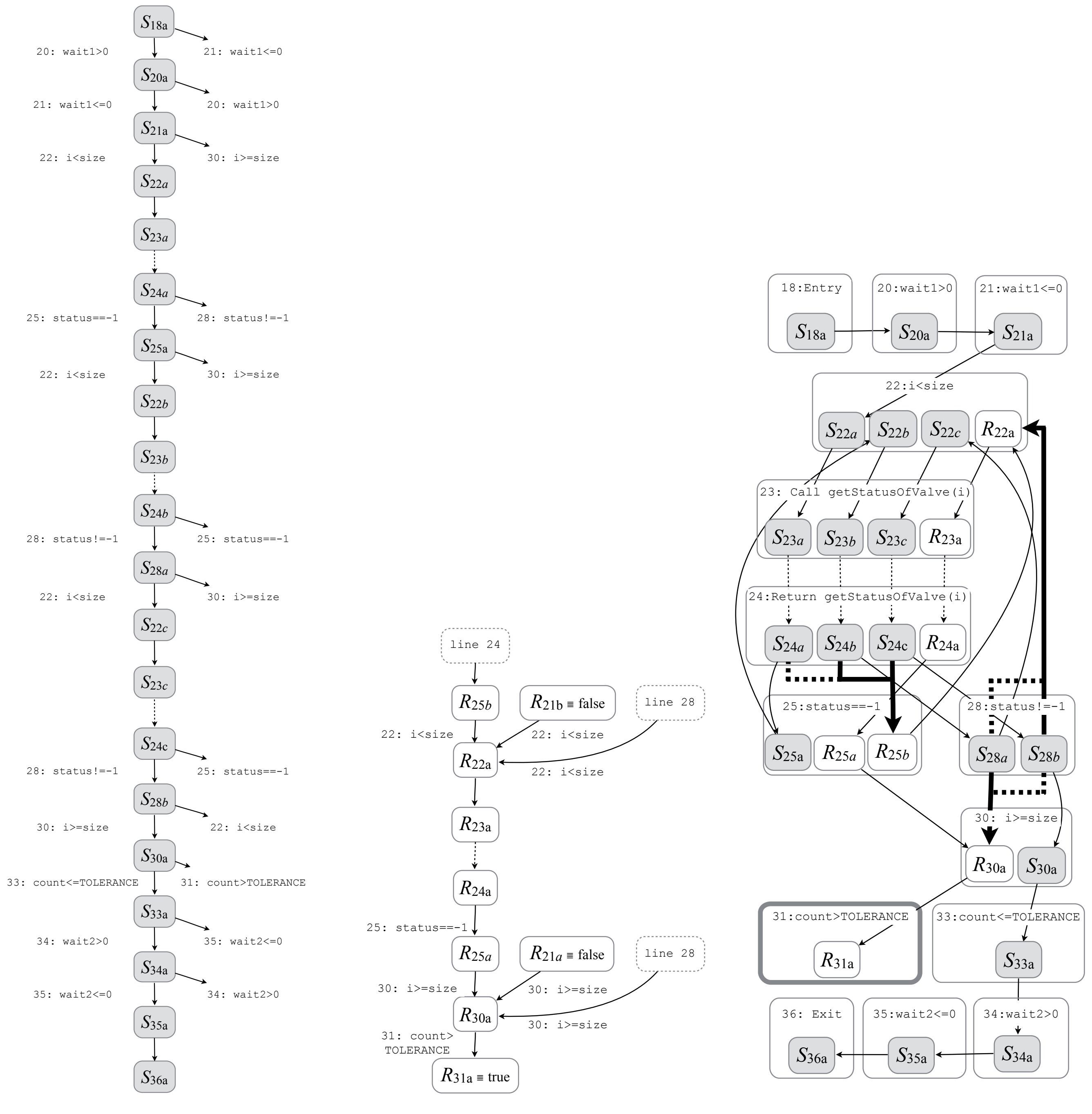
The symbolic execution illustrated in Figure 2(a) covers only a subset of the branches of the program `getStatusValve`. Symbolic execution can increase branch coverage by trying to execute branches still unexplored along the execution trace. Popular symbolic execution tools, like CREST [7] and KLEE [20], explore paths that share some prefix with already executed paths, like the ones corresponding to labels of dangling edges in the path of Figure 2(a), implementing different exploration strategies. We analyzed the program with both CREST and KLEE with random testing, bounded depth-first concolic execution and a heuristic strategy that aims to maximize branch coverage [7], and we have been able to execute all but two branches of the program, obtaining 83% branch coverage.[1] While this seems a good result, a closer look at the data suggests that the missed branches include critical cases. The uncovered branches correspond to the $true$ branches of the `if` statements at lines 31 and 7. While the $true$ branch of the `if` statement at line 7 is infeasible, since it repeats the boundary check performed at line 22, the $true$ branch of the `if` statement at line 31 (hereafter branch 31) represents a critical path in the program, since it corresponds to the behavior in the case of an alarm.

Determining the executability of branch 31 with symbolic execution is problematic because the execution condition of this branch depends on the number of executions of the loop at the lines 22—29 that rely on the value of variable `count`, which is not a direct function of the input values. Variable `count` counts how many times the program finds a valve that is out of order (line 26) while iterating though the loop at lines 22—29. The value of variable `count` is computed as the result of the assignments at lines 21 and 26 that always yield concrete values, and thus the evaluation of the condition `count>2` at line 31 always produces deterministic truth values during symbolic execution. Referring to the symbolic state S_{30a} of the considered example, variable `count` holds 1 and condition `count>2` evaluates to $false$, thus the symbolic execution cannot determine the number of iterations of the loop that would lead to the execution of the critical statement at line 31, but can only deduce that the `if` statement at line 31 cannot be ever taken along the program path that leads to S_{30a}.

To show the executability of branch 31, symbolic execution needs to exhaustively explore the feasible paths that reach line 31, until eventually finding a path for which the condition `count>2` evaluates to $true$. The loops in the program determine an unbounded number of paths, and thus the result ultimately depends on the path selection heuristics used in combination with symbolic execution. For example, when symbolically executing program `checkValves` with the common depth-first order strategy, symbolic execution is likely to iterate infinitely many times through the loops in the program, without ever executing any path with more than `TOLERANCE` valves in bad status. Other heuristics, for example the ones that prioritize paths through branches that are close to the not-yet-executed ones within a maximum budget of attempts [7], [8], [9], may be more successful in executing the `if` statement at line 31 in program `checkValves`, especially if `TOLERANCE` is set to a low value such as 2, but no heuristics can guarantee the solution in the general case.

The lack of direct dependency of branch conditions from input values is quite frequent, since it depends on common programming practices like summarizing intermediate computations as concrete values, later used to drive the execution, and the dependency of the values of the variables from loops. The problem of determining the feasibility of some branches with symbolic execution is complicated by the frequent occurrence of chains of implicit dependencies between the program branches, where a branch may depend on the concrete values assigned at some previous branch, which in turn depends

1. The interested readers can find a detailed discussion of the strategies used in the analysis with CREST and KLEE, and the results of symbolically executing this program in [18].



(a) (b) (c)

SYMBOLIC STATE PREDICATES:

S_{22a}: size=S1 ∧ wait1=W1 ∧ wait2=W2 ∧ valvesStatus=[V1,V2, V3, ...] ∧ S1>0 ∧ W1=1 ∧ i=0 ∧ count=0

S_{24b}: size=S1 ∧ wait1=W1 ∧ wait2=W2 ∧ valvesStatus=[V1,V2, V3, ...] ∧ S1>1 ∧ W1=1 ∧ V1=-1 ∧ i=1 ∧ count=1 ∧ status=V2

S_{30a}: size=S1 ∧ wait1=W1 ∧ wait2=W2 ∧ valvesStatus=[V1,V2, V3, ...] ∧ S1=3 ∧ W1=1 ∧ V1=-1 ∧ V2!=-1 ∧ V3!=-1 ∧ i=3 ∧ count=1

REACHABILITY STATE PREDICATES:

R_{25b}: valvesStatus[i+1]=-1 ∧ i>=-1 ∧ i<size-1 ∧ i>=size-2 ∧ count>0 ∧ status=-1

R_{30a}: count>2

Fig. 2. Intermediate results of the analysis of program *checkValves* in Figure 1: (a) a path of symbolic states built with symbolic symbolic execution, (b) a set of reachability states built with reachability analysis, (c) a GCFG model built with bidirectional symbolic analysis.

on concrete values assigned at earlier branches, and so forth. The presence of mutually dependent branches interleaved with other branches and loops further exacerbates the problem.

The importance of investigating rare execution conditions and infeasible branches as the ones exemplified above is witness by known severe failures like the *Apple's goto fail security bug*[2] that could have been easily revealed by combining the symbolic exploration of the execution space with the identification of infeasible branches, as done with the approach described in this paper. The conditions for executing a program branch can be determined with symbolic reachability analysis that consists in considering an abstract program state that models the execution of the target program branch, and progressively computing the (weakest pre-) conditions that must hold at the previous statements and eventually at the program entry, to reach the target branch [23].

Figure 2(b) exemplifies the symbolic reachability analysis of branch 31 that we discussed as difficult to cover with symbolic execution. The nodes in the figure indicate the symbolic reachability states computed during the analysis, and the edges indicate the relations between the symbolic reachability states according to the control structure of the program. We label the symbolic reachability states with the corresponding statement numbers. As before, the labels refer to the program state after executing the corresponding statement. We label the edges with the program conditions that guard the transition between the corresponding symbolic reachability states.

The analysis starts from a symbolic reachability state that models the execution of the target branch, and traverses the control flow graph backward towards the program entry. In the example of Figure 2(b), symbolic reachability analysis starts from the state R_{31a} that models the execution of the $true$ branch at line 31. This state is associated with the symbolic condition $true$ to indicate that symbolic reachability analysis has not yet identified any condition that must be satisfied to reach the target branch.

Symbolic reachability analysis proceeds traversing the program backward from line 31 reaching line 30 (the exit of the `while` loop) that corresponds to state R_{30a}, and identifies `count>2` as the condition to reach R_{31a}. Progressing backward, the analysis can move back to line 25 (from the loop body when the condition of the `if` statement evaluates to $true$), line 28 (from the loop body when the condition of the `if` statement evaluates to $false$) or line 21 (before entering the loop). The figure shows the reachability states computed for line 25 (R_{25a}) and 21 (R_{21a}), and indicates that the program branch that leads from line 28 to line 30 (when the condition of the `if` evaluates to $false$) has not been explored yet.

The symbolic condition of the state R_{21a} results in a contradiction ($false$) because the condition `count>2` contrasts with the assignment `count=0` at line 21, thus the symbolic reachability analysis concludes that the states R_{30a} and R_{31a} cannot be reached without iterating through the loop.

The backward exploration from R_{25a} towards the program entry traverses the states R_{24a}, R_{23a}, R_{22a}, R_{25b} and R_{21b}, where the dotted line between R_{24a} and R_{23a} implicitly represents the call to `getStatusOfValve`. The state R_{21b} results in another contradiction, indicating that a single iteration of the loop does not suffice to reach our target either. The state R_{25b} represents the conditions that the program variables must satisfy to reach branch 31 after executing line 25 at the second to last iteration of the loop, and is reported at the bottom of Figure 2, together with the condition of the symbolic reachability state R_{30a}.

To identify the conditions on the input values that lead to executing the target branch, symbolic reachability analysis shall proceed until the program entry. In the presence of loops, like in the example, this may lead the analysis through infinitely many broken paths, that is, paths that lead to contradictions before reaching the program entry, as the one from R_{30a} back to R_{21a}. However, the symbolic reachability state R_{25b} indicates a set of conditions on the input values that lead to the execution of the branch 31, thus providing the information that symbolic execution misses, as discussed above.

The example shows that forward analysis, instantiated as symbolic execution, can identify the conditions to reach program branches, but may suffer when dealing with branches whose executability does not depend directly from the input values. Backward analysis, instantiated as symbolic reachability analysis, may reveal implicit dependencies, thus complementing forward analysis towards increasing branch coverage.

The main contribution of this paper is a framework that combines forward and backward analysis to achieve high branch coverage and that we refer to as bidirectional symbolic analysis. Bidirectional symbolic analysis alternates symbolic execution and symbolic reachability analysis to both reach program branches that have not been executed yet and prune program branches that are proved infeasible. The core of bidirectional analysis is a model that integrates the results of symbolic execution and symbolic reachability analysis, thus improving over both kinds of analysis when applied independently. In the next sections, we first introduce the model, and then define bidirectional symbolic analysis in detail.

3 The Generalised CFG Model

In this section we define the GCFG model that is the core of bidirectional symbolic analysis, and discuss how the model allows us to identify the reachability frontier, which is used to guide the analysis.

3.1 The GCFG Model

The GCFG model integrates the states computed with symbolic execution, the states computed with sym-

2. See for instance http://avandeursen.com/2014/02/22/gotofail-security/

bolic reachability analysis and the control flow relations among them.

The GCFG represents the program branches executed during symbolic execution, and identifies the target branches that are program branches that have not been executed yet and have not been identified as unreachable yet with symbolic reachability analysis.

Figure 2(c) shows the GCFG obtained by integrating the symbolic execution states shown in Figure 2(a) and the symbolic reachability analysis states shown in Figure 2(b). In the figure we group the states that correspond to the same program statements with boxes labeled with both the line number of the statements and the branch condition that guards the execution of the statements. As discussed in the previous section, both symbolic execution states (S-states) and symbolic reachability states (R-states) represent the program states after executing the corresponding statements. The statements at lines 18 (program entry), 23 (function call site), 24 (function return site) and 36 (program exit) are not guarded by any specific branch condition, that is, they are necessarily executed as soon as they get reached; for these statements we use arbitrary labels that refer to the semantics of the statements: `Entry`, `Call`, `Return` and `Exit`.

The GCFG in Figure 2(c) contains all the S-states of Figure 2(a) and all the R-states of Figure 2(b), except for the states R_{21a} and R_{21b} that represent unreachable states. The GCFG nodes are connected with three types of edges that represent relations between pairs of S-states, relations between pairs of R-states and relations between S- and R-states. The edges that connect pairs of S-states and the edges that connect pairs R-states derive directly from symbolic computations, the edges that connect S-states to R-states represent program control flow relations that have not been analyzed yet. They are depicted in bold (either solid or dotted lines) in the figure, and represent the *frontier* between forward and backward analysis as discussed in detail below.

The source S-state of a frontier edge is a state that can be directly extended to satisfy the symbolic representation of the target R-state of the edge by means of symbolic execution, and can thus head to some not yet executed program branch, such as the branch 31 of program `getStatusOfValve`. The target state of a frontier edge is an R-state that can be analyzed by means of symbolic reachability analysis to identify new details that increase the chances to either satisfy uncovered branches or reveal infeasible branches in the sequel of the analysis. Thus, the frontier edges identify states that can can be analyzed to improve branch coverage or reachability information related to branch coverage. In Figure 2(c), the frontier edges visualized as solid lines indicate program branches whose reachability should be analyzed next, since they connect an already reached symbolic state with a symbolic reachability state still to be explored and thus can be efficiently analyzed. Correspondingly, the frontier edges visualized as dotted lines indicate R-states whose infeasibility should be analyzed next, since all the corresponding S-states have been already analyzed with symbolic execution and cannot be further extended.

In general, a GCFG is a connected graph that models the state of the symbolic evaluation of a program that we assume with a single entry point. GCFG nodes represent either S-states computed with symbolic execution or R-states that correspond to satisfiable symbolic reachability conditions. A GCFG satisfies the following properties that derive from the way GCFGs are constructed during the analysis: (i) The graph is rooted in a S-state that represents the program entry point, (ii) the R-states cannot be mutually satisfied together with some S-state that correspond to the same program statement, (iii) the R-states are always reachable from the entry node, (iv) the R-states either reach another R-state or are terminal nodes. The terminal R-states, which we refer to as target nodes, model the program state after not-yet-executed program branches. In Figure 2(c) the entry node is S_{18a}, and the only terminal node is R_{31a}.
The GCFG edges connect: (i) pairs of S-states to represent forward symbolic execution steps, (ii) pairs of R-states to represent backward reachability analysis steps, (iii) pairs $\langle S\text{-state}, R\text{-state}\rangle$, the frontier edges, to represent not-yet-fully-analyzed program control flow relations that may lead to cover not-yet-executed program branches.

The GCFG in Figure 2(c) includes 7 frontier edges that represent the static control flow relations of program `checkValves` that cannot be excluded from the GCFG yet according to the analysis done so far.

The frontier edges from the states S_{24b} and S_{24c} to the state R_{25b} indicate that the feasibility of the program branch from the return of the call of method `getStatusOfValve()` (states S_{24b} and S_{24c}) to the true branch of the following `if` statement (state R_{25b}) have not been fully investigated yet. The frontier edges $\langle S_{28a}, R_{30a}\rangle$ and $\langle S_{28b}, R_{22a}\rangle$ indicate that the feasibility of the program branches that either exit from the loop at the second iteration (from state S_{28a} to state R_{30a}) or continue with a fourth iteration (from state S_{28b} to state R_{22a}) requires further investigation. The frontier edges $\langle S_{24a}, R_{25b}\rangle$, $\langle S_{28a}, R_{22a}\rangle$ and $\langle S_{28b}, R_{30a}\rangle$ indicate R-states that require further investigation.

The model does not include a frontier edge from S_{21a} to R_{30a} because such branch has already been demonstrated to be non executable, as shown by the predicate R_{21a} in Figure 2(b).

We will further discuss the example in the next sections when incrementally introducing the analysis features. Below we formally define the GCFG.

Given a single entry program P, a GCFG is defined starting from the state transition systems $T_s = \langle N_s, E_s, S_s^0\rangle$ and $T_r = \langle N_r, E_r, B_r\rangle$ that represent information computed with symbolic execution and symbolic reachability analysis of P, respectively. N_s (N_r) and E_s (E_r) are the states and the transitions computed with symbolic execution (symbolic reachability analysis) of P. $S_s^0 \in N_s$ is the initial symbolic state corresponding to the program entry point, and $B_r \subseteq N_r$ are $true$-valued symbolic reachability states that model the execution of the program branches analyzed as reachability targets.

Given a single entry program P, a GCFG is a state transition system $\bar{T} = \langle \bar{N}, \bar{E}, \bar{S^0}, \bar{B} \rangle$, where $\bar{N}$ is a set of states, $\bar{E} \subseteq \bar{N} \times \bar{N}$ is a set of transitions, $\bar{S^0} \in \bar{N}$ is a state that corresponds to the entry point of P, and $\bar{B} \subseteq \bar{N}$ are states that model the execution of the unreached branches of P.

To facilitate the definition of the transition system $\bar{T}$, we introduce the following helper predicates on the states N_s, N_r and $\bar{N}$. The predicates identify the relation between states that refer to the same program blocks (the boxes in Figure 2(c), *sameCfgBlock*), the existence of control dependencies among states due to the program structure (*isCfgEdge*), the satisfiability of the symbolic formulas represented by the states (*sat*) and the existence of paths between GCFG nodes (*connected_$\bar{T}$*):

- $\text{sameCfgBlock} : (N_s \cup N_r) \times (N_s \cup N_r)$ relates the pairs of states that correspond to the execution of the same statements in the static control flow of P.
- $\text{isCfgEdge} : (N_s \cup N_r) \times (N_s \cup N_r)$ relates the pairs of states that correspond to the execution of statements that are connected by an edge in the static control flow of P.
- $\text{sat} : N_r \cup N_s$ identifies the states that correspond to satisfiable symbolic formulas.
- $\text{connected\_}\bar{\text{T}} : \bar{N} \times \bar{N}$ identifies the pairs of states connected with a path in $\bar{T}$.

Definition 3.1: Given a single entry program P and two state transition systems $T_s = \langle N_s, E_s, S_s^0 \rangle$ and $T_r = \langle N_r, E_r, B_r \rangle$ computed with symbolic execution and symbolic reachability analysis of P, respectively, the related GCFG is the state transition system $\bar{T} = \langle \bar{N}, \bar{E}, \bar{S^0}, \bar{B} \rangle$, such that:

- $\bar{S^0} = S_s^0$ is a state that denotes the entry point of P. It corresponds to the initial state of the symbolic execution of P.
- $\bar{B} = B_r$ is a set of states that model the execution of the not yet covered branches of P that correspond to the current targets of the symbolic reachability analysis.
- $\bar{N} = \bar{N_s} \cup \bar{N_r}$ is a set of states that includes both S-states and R-states, where $\bar{N_s} \subseteq N_s$ and $\bar{N_r} \subseteq N_r$ are the maximal subsets of N_s and N_r, respectively, such that the following invariants hold:

$$\forall s \in \bar{N_s} : \text{sat}(s) \tag{Inv1}$$

$$\forall r \in \bar{N_r} : \text{sat}(r) \tag{Inv2}$$

$$\forall r \in \bar{N_r} : \neg \exists s \in N_s : \text{sameCfgBlock}(s, r) \wedge \text{sat}(s \wedge r) \tag{Inv3}$$

$$\forall r \in \bar{N_r} : \text{connected\_}\bar{\text{T}}(\bar{S^0}, r) \wedge \exists b \in \bar{B} : \text{connected\_}\bar{\text{T}}(r, b) \tag{Inv4}$$

The GCFG includes only states that are computed during either symbolic execution or reachability analysis. It includes only satisfiable states (Inv1 and Inv2), does not include symbolic reachability conditions whose satisfiability can be already deduced from symbolic states at the same block of the control flow graph (Inv3), and does not include symbolic reachability conditions that either cannot be reached from the entry of $\bar{T}$ or cannot reach any not-yet-proved reachability target in $\bar{T}$ (Inv4). Inv4 excludes states that have been proven unreachable from the entry point and the ones whose target has been already reached from a different path, since such states do not carry information still useful for the analysis.

The definition guarantees that the initial state of symbolic execution belongs to $\bar{N}$, because $\bar{S^0} = S_s^0 \in \bar{N_s}$, and that all the (not yet excluded) targets of the reachability analysis belong to $\bar{N}$, because $\bar{B} = B_r \subseteq \bar{N_r}$.

- $\bar{E} = \bar{E_f} \cup \bar{E_s} \cup \bar{E_r}$ is a set of transitions, where $\bar{E_s} \subseteq E_s$ and $\bar{E_r} \subseteq E_r$ are the restrictions of E_s and E_r with respect to $\bar{N_s}$ and $\bar{N_r}$, respectively, and $\bar{E_f} \subseteq E_f$ is the maximal subset of $E_f = \{\langle s, r \rangle : s \in \bar{N_s} \wedge r \in \bar{N_r} \wedge \text{isCfgEdge}(s, r)\}$ such that:

$$\forall \langle s, r \rangle \in \bar{E_f} : (\neg \exists r' : \text{sameCfgBlock}(r', s) \wedge \langle r', r \rangle \in E_r) \tag{Inv5}$$

For the set of represented symbolic states and symbolic reachability conditions, the GCFG includes all transitions computed during the respective analysis and includes the frontier edges E_f. The set E_f represents the transitions from symbolic states to symbolic reachability conditions in the GCFG (recall that $\bar{N} = \bar{N_s} \cup \bar{N_r}$) that correspond to some edge of the control flow graph of P and that do not correspond to any transition already analyzed during the symbolic reachability analysis (Inv5). The invariants Inv5 and Inv2 guarantee that the GCFG does not include control flow transitions that have been proved to be infeasible with symbolic reachability analysis.

For the sake of improving the description of the algorithms, in the figures we graphically distinguish the frontier edges that start for S-states that can be further investigated with symbolic execution (solid lines) from frontier edges that cannot be further explored with symbolic execution but lead to R-states that can be further investigated with symbolic reachability analysis (dotted lines). However, this distinction is not formalized in the the GCFG.

3.2 The Frontier

The GCFG represents a snapshot of the progress of the bidirectional symbolic analysis of a program, and identifies the frontier that indicates analysis states from where we can extend the symbolic analysis in either forward or backward directions, to reach unexecuted branches or reveal infeasible branches. On one hand, the R-states summarize path suffixes that can reach not yet executed branches for some (symbolically represented) values of the program variables. On the other hand, the S-states summarize paths that certainly execute for some (symbolically represented) sets of inputs. Therefore, the frontier edges, which connect S-states to R-states, indicate (i) symbolic execution states where the forward analysis might be directly extended to satisfy the symbolic representation of a symbolic reachability state, and thus reach, or get closer to, some not yet executed branches, and (ii) symbolic reachability states where the backward reachability analysis can enrich the reachability information and identify new details that increase the chances to either satisfy these states or reveal infeasible branches in the sequel of the analysis.

Without the information provided in the frontier, symbolic execution and reachability analysis may converge slowly or, in some cases, may even diverge, as discussed in the previous section.

The frontier edges in the GCFG of Figure 2(c) convey information that boosts the chances of convergence of the symbolic analysis. These edges indicate states and paths that are promising to reach not yet executed branches. For instance, one of these promising paths is the path that traverses the frontier edge $\langle S_{24b}, R_{25b}\rangle$. Indeed, the symbolic execution of S_{24b} towards line 25 results in a symbolic state that can satisfy R_{25b}. For instance `valvesStatus=[`$-1,-1,-1,...$`]` is an assignment of the symbolic input vector that satisfies the symbolic representation of both states, and leads to execute the $true$ branch of the `if` statement at line 25 and subsequently the $true$ branch of the `if` statement at line 31, thus exercising the critical branch that can hardly be reached with either symbolic execution or reachability analysis alone, as discussed in Section 2. When the GCFG contains more than one frontier edge, the edge to be explored next can be selected according to different strategies. In the experiments reported in this paper, we explore the frontier edges in the order in which they are identified during the analysis.

In the state represented in Figure 2(c), bidirectional symbolic analysis selects one of the frontier edges, for instance the edge $\langle S_{24b}, R_{25b}\rangle$, and tries to advance the frontier by symbolically executing the branch identified by the edge, in this case the $true$ branch of the `if` statement at line 25. The branch can be executed and produces a new symbolic state, which is satisfiable jointly with the symbolic reachability condition R_{25b}. This new evidence opens new execution paths that we can explore with testing before the next iteration of bidirectional symbolic analysis. In the example, we can generate a test case that reaches the symbolic reachability condition R_{25b}, and thus leads to the uncovered program branch at line 31. We then symbolically execute the program along the path identified by the test case and update the GCFG by adding the new S-states, eliminating the R-states superseded by the new S-states, updating the edges, and recomputing the frontier. In the example, we successfully reach one of the critical branches (branch 31).

The analysis proceeds with a new iteration, selecting a new frontier edge, trying to either symbolically execute the corresponding program branch or refine the reachability information by means of symbolic reachability analysis. The symbolic execution can lead to yet a new state, as in the case illustrated above, or result in an unsatisfiable condition, thus leaving room for trying to either demonstrate the infeasibility of the frontier edge or reveal subtle indirect dependencies that may lead to execute the branch in the sequel of the analysis.

Generalizing from this example, the next section describes bidirectional analysis, while Section 5 presents the test generation approach based on bidirectional analysis.

4 Bidirectional Symbolic Analysis

Bidirectional symbolic analysis combines symbolic execution and symbolic reachability analysis to improve and refine branch coverage, by both covering not-yet covered branches and identifying infeasible branches to be pruned from the coverage domain. The approach consists of incrementally refining the GCFG with increasing reachability information.

Bidirectional symbolic analysis coordinates symbolic execution and symbolic reachability analysis steps through a GCFG model that identifies the target branches and drives the alternation of the different analysis steps.

Bidirectional symbolic analysis initializes the GCFG model referring to an initial set of target branches that include all the branches of the program. In the example of Figure 1 the initial set of target branches includes all the branches of both functions `CheckValves` and `getStatusOfValve`. In the figure we label these branches after the corresponding lines 20, 21, 22, 25, 28, 30, 31, 33, 34 and 35 of function `CheckValves` and 7, 8, 9 and 13 of function `getStatusOfValve` that correspond to the statements reached after the control decisions. For instance, line 20 refers to the statement `wait1--` that is reached through the $true$ branch `wait1>0` of the `while` statement, while line 21 refers to `count=0, i=0`, reached through the $false$ branch of the `while`.

Bidirectional symbolic analysis initializes the GCFG by symbolically executing the program from the entry statement following a set of paths that are arbitrarily chosen among the executable ones, and populates the GCFG with the set of explored symbolic S-states. The symbolic execution states are connected with solid edges in the

GCFG. For example, the sequence of connected states S_{18a}, S_{20a}, S_{21a}, S_{22a}, S_{23a}, S_{24a}, S_{25a}, S_{22b}, S_{23b}, S_{24b}, S_{28a}, S_{22c}, ..., S_{35a}, S_{36a} of Figure 2(c) represents the result of symbolically executing program `CheckValves` from the initial state (line 18) through three iterations of the `while` loop at lines 22–29 up to the exit (line 36), and corresponds to the symbolic execution states of Figure 2(a). In both Figure 2(a) and Figure 2(c), the dotted edges $\langle S_{23a}, S_{24a}\rangle$, $\langle S_{23b}, S_{24b}\rangle$ and $\langle S_{23c}, S_{24c}\rangle$ indicate the call-return of `getStatusOfValve(i)`, whose explored states, S_{6a}, S_{8a}, S_{13a}, S_{14a}, S_{6b}, S_{8b}, S_{13b}, S_{14b}, S_{6c}, S_{8c}, S_{13c} and S_{14c}, are not explicitly reported in the figures.

Symbolically executing a branch witnesses the feasibility of the branch, and thus the executed branches are removed from the set of targets. In the example, the initialization step reduces the set of targets to the branches corresponding to line 31 in function `CheckValves` and lines 7 and 9 in function `getStatusOfValve`.

Bidirectional symbolic analysis completes the initialization of the GCFG graph by adding an R-state with a $true$ predicate for each current target branch. In the working example, it adds the R-state R_{31a} explicitly represented in Figure 2(c) and the R-states R_{7a} and R_{9a} that are left implicit in the figure and that we will discuss later in this section. It connects the new R-states with the S-states currently in the GCFG according to the control flow relation in the program. For example, in the initialization step, it connects the R-state R_{31a} with the S-state S_{30a}. Such edges that connect R-states and S-states are frontier edges. We recall that the model in Figure 2(c) does not include the frontier edge $\langle S_{30a}, R_{31a}\rangle$ because it represents the frontier edges at a later point of analysis.

While the analysis progresses the frontier edges are updated according the new information computed with symbolic execution and symbolic reachability analysis. Figure 2(c) represents the state of the analysis after the symbolic reachability analysis steps described in Figure 2(b), and thus it does not include the frontier edge $\langle S_{30a}, R_{31a}\rangle$, but represents the frontier edges at that point of analysis. The frontier edges are represented with bold lines in the figure.

In the general step, bidirectional symbolic analysis selects a frontier edge, and uses symbolic execution to check if the R-state can be reached from the S-state while satisfying the computed reachability condition. In this case, the R-state is demonstrated to be reachable and is turned to a corresponding S-state. If the R-state cannot be reached from the S-state, the bidirectional analysis uses symbolic reachability analysis to update the frontier.

Bidirectional symbolic analysis progressively classifies the program branches as covered or infeasible. A branch is covered when the symbolic execution explores an S-state associated with a not-yet executed program branch. A branch is infeasible when the symbolic reachability analysis indicates that the branch is reachable only from $false$ R-states, that is, R-states with a condition that is a logical contradiction.

Bidirectional symbolic analysis iterates through three main steps: symbolic execution, symbolic reachability analysis and model coarsening. Symbolic execution identifies new symbolic states (S-states), trying to supersede some symbolic reachability states (R-states) of the current GCFG. Symbolic reachability analysis refines the R-states that symbolic execution cannot supersede. Model coarsening prunes the GCFG by eliminating the R-states that becomes obsolete throughout the progress of the analysis, thus supporting the scalability of the approach.

4.1 Symbolic Execution

A symbolic execution step targets a frontier edge $\langle S_i, R_j\rangle$ as described in Algorithm 1. First, it symbolically analyzes S_i towards R_j that represents a possible successor of S_i, since the frontier edge derives from a control flow edge (Algorithm 1, line 1).

It then evaluates the reachability condition of R_j in conjunction with the new state (line 2) and, if R_j and the new symbolic state are jointly satisfiable, it updates the GCFG adding the newly computed state (line 3) and new frontier edges that correspond to the new state if any (line 4). Next, it computes the set of R-states that share the same block of R_j and are jointly satisfiable with the new symbolic state (line 5), and invokes the coarsening step to remove these R-states (including R_j) from the GCFG (line 7). The invocation of `SymbolicExecution` refers to the classic symbolic execution approach that we informally surveyed in Section 2. The invocation of `ComputeFrontierEdges` updates the GCFG model as discussed in Section 3.1. We detail `ModelCoarsening` later in this section (Algorithm 3).

The algorithm requires to verify the satisfiability of the symbolic formulas at lines 2 and 5. As discussed in the next section, when used in the context of test generation, it verifies the satisfiability at line 2 by invoking a constraint solver, and deduces the satisfiability of the set of formulas at line 5 with the test case that is generated as the solution of the query at line 2. If the test case fails to identify the satisfiability of some R-states, those states remain in the model and their satisfiability will be further addressed in the subsequent iterations of symbolic execution.

For example, if we consider the $\langle S_{24b}, R_{25b}\rangle$ frontier edge of Figure 2, the symbolic execution step starts with the symbolic state S_{24b} that is reported in the bottom of the figure, and computes a new symbolic state that augments the path condition of S_{24b} with the conditional $status = -1$. The new symbolic state is indeed satisfiable jointly with the condition of state R_{25b}.

4.2 Symbolic Reachability Analysis

A symbolic reachability analysis step augments the GCFG with new reachability states, as illustrated in Algorithm 2. Symbolic reachability analysis considers the target R-state of a frontier edge $\langle S_i, R_j\rangle$, and analyzes R_j

Algorithm 1 SymbolicExecutionStep

```
Require:
    $model$, a GCFG
    $\langle S_i, R_j \rangle$, a frontier edge of the $model$
Ensure:
    A symbolic execution step on the input frontier edge
    Returns the resulting GCFG
 1: $S' \leftarrow$ SymbolicExecution($S_i \rightsquigarrow R_j$)
 2: if $sat(S' \wedge R_j)$ then
 3:     $model = model \cup S' \cup \langle S_i, S' \rangle$
 4:     $model = model \cup$ ComputeFrontierEdges($S', model$)
 5:     $reachedRNodes = \{R'\} | \; sameCfgBlock(R', R_j) \wedge sat(S' \wedge R')$
 6:     for all $R' \in reachedRNodes$ do
 7:         ModelCoarseningStep($model, R'$)                    # Invariant Inv3
 8:     end for
 9: end if
10: return $model$
```

backward along the control flow branch that corresponds to the frontier edge (Algorithm 2, line 1). It produces a new R-state by augmenting the predicate of R_j with the information of the control flow branch. If the new symbolic reachability state is a contradiction (line 2), the state R_j and all the corresponding frontier edges are pruned from the GCFG in the coarsening step (line 3). If the new symbolic reachability condition intersects some S-states corresponding to the same program branch (line 6), the considered frontier edge is indeed reachable. In this case the approach iterates with a symbolic execution step to detail the reachability of the frontier edge (line 8). Otherwise it updates the model to include the new state (lines 10—14).

The algorithm requires to solve the constraints at lines 2 and 5. As discussed in the next section, when used in the context of test generation, this requires only the call to a constraint solver at line 2, while the call at line 5 can be avoided, since the test generation algorithm guarantees that the frontier S-states have been already evaluated in the former iterations.

Algorithm 2 SymbolicReachabilityAnalysisStep

```
Require:
    $model$, a GCFG
    $\langle S_i, R_j \rangle$, a frontier edge included in $model$
Ensure:
    Accomplishes a symbolic reachability analysis on the input frontier edge
    Returns the resulting GCFG
 1: $R' \leftarrow$ WeakestPrecondition($R_j \rightsquigarrow S_i$)
 2: if $\neg sat(R')$ then
 3:     ModelCoarseningStep($model, R'$)                    # Invariant Inv2
 4: else
 5:     $reachingSNodes = \{S'\} | \; sameCfgBlock(S', S_i) \wedge sat(S' \wedge R')$
 6:     if $reachingSNodes \neq \emptyset$ then
 7:         $S' \leftarrow$ a node from $reachingNodes$
 8:         SymbolicExecutionStep($model, \langle S', R_j \rangle$)
 9:     else
10:         $E_f =$ the set of frontier edges in $model$
11:         $E_f^{delete} = \{\langle S'', R_j \rangle\} | \; sameCfgBlock(S'', S_i) \wedge \langle S'', R_j \rangle \in E_f$
12:         $model \leftarrow model \setminus E_f^{delete}$                    # Invariant Inv5
13:         $model \leftarrow model \cup R' \cup \langle R', R \rangle$
14:         $model \leftarrow model \cup$ ComputeFrontierEdges($R', model$)
15:     end if
16: end if
```

For example, let us consider the frontier edge $\langle S_{22a}, R_{23b} \rangle$ in the GCFG in Figure 3. The figure presents the GCFG of the program `checkValves` in Figure 1 after some steps of symbolic reachability analysis from R_{9a}. The state R_{9a} belongs to `getStatusOfValves` and is thus not represented explicitly in Figure 2(c). Figure 3 shows the details of the symbolic states corresponding to the call of `getStatusOfValves` that are abstracted away in Figure 2(c), and cuts the GCFG after the return from `getStatusOfValves`.

R_{9a} represents the execution of the $true$ branch of the `if` statement at line 7 of the program `checkValves` in Figure 1, and corresponds to an error state of the program. As mentioned in Section 2, this branch is infeasible, and the GCFG of Figure 3 shows some progress of bidirectional symbolic analysis towards the proof of the infeasibility of this branch. Specifically, the GCFG of Figure 3 extends the analysis steps represented by the GCFG of Figure 2(c) with the chain of symbolic reachability analysis states R_{23b}, R_{6b}, R_{8b} and R_{9a} that results from the symbolic reachability analysis from the target R_{9a} towards the program entry point.

The frontier $\langle S_{22a}, R_{23b} \rangle$ represents the next step towards the proof of the infeasibility of the $true$ branch of the `if` statement at line 7. The symbolic reachability analysis of R_{23b} produces the contradictory condition $i \geq size$ && $i \geq 0$ && $i < size$ that derives from the conjunction of R_{23b} (shown at the bottom of Figure 3) and the condition of the frontier edge. Thus the frontier edges $\langle S_{22a}, R_{23b} \rangle$, as well as the frontier edges $\langle S_{22b}, R_{23b} \rangle$ and $\langle S_{22c}, R_{23b} \rangle$ can be removed from the GCFG.

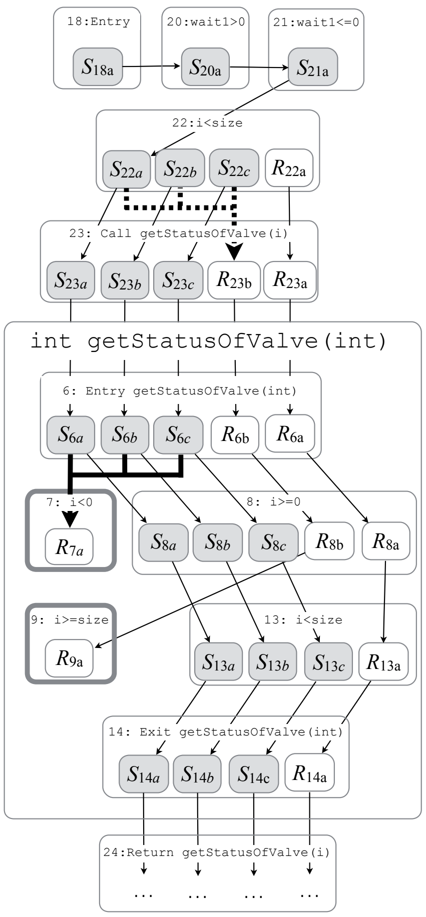
As an example of a refinement of a frontier edge that leads to new R-states, let us consider the frontier edge $\langle S_{6a}, R_{7a} \rangle$. The symbolic reachability analysis of R_{7a} propagates the non-contradictory condition $i < 0$ thus producing a new state R_{6c} and updating the model as shown in the zoom in of Figure 4.

The new symbolic reachability state may identify new frontier edges on the GCFG. In our example, augmenting the model with the new state R_{6c} produces new frontier edges from S_{23a}, S_{23b} and S_{23c} to R_{6c} traversing the call of function `getStatusOfValve` (Figure 4(b)).

The refinement process is conservative as no feasible path that might lead to not-yet-covered program branches is ever excluded from the model. Moreover the monotonic reduction in the number of program paths to be considered guarantees progress.

4.3 Model Coarsening

The *model coarsening* step described in Algorithm 3 consists of removing the symbolic reachability states that correspond to states that become obsolete after either symbolic execution or symbolic reachability analysis, or because of model coarsening itself (Algorithm 3, lines 1—6). Some R-states may become obsolete because of symbolic execution when a new S-state is satisfiable in conjunction with an R-state (Algorithm 1, line 2), as for instance in the case of S_{24b} and R_{25b} in the above example of symbolic execution, meaning that that R-state is indeed reachable. Some other R-states may become obsolete because of symbolic reachability analysis when

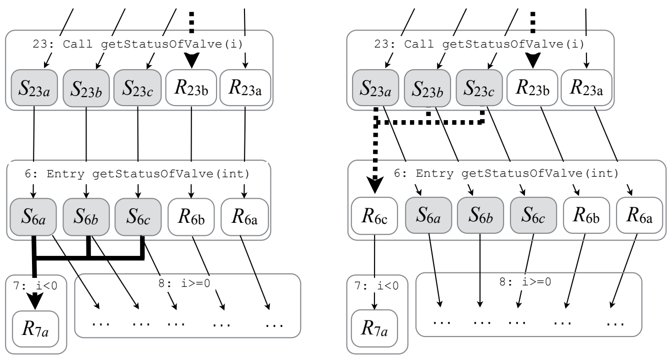


18:Entry
20:wait1>0
21:wait1<=0
S_{18a} S_{20a} S_{21a}
22:i<size
S_{22a} S_{22b} S_{22c} R_{22a}
23: Call getStatusOfValve(i)
S_{23a} S_{23b} S_{23c} R_{23b} R_{23a}
int getStatusOfValve(int)
6: Entry getStatusOfValve(int)
S_{6a} S_{6b} S_{6c} R_{6b} R_{6a}
7: i<0
R_{7a}
8: i>=0
S_{8a} S_{8b} S_{8c} R_{8b} R_{8a}
9: i>=size
R_{9a}
13: i<size
S_{13a} S_{13b} S_{13c} R_{13a}
14: Exit getStatusOfValve(int)
S_{14a} S_{14b} S_{14c} R_{14a}
24:Return getStatusOfValve(i)

REACHABILITY STATE PREDICATES:

R_{23b}: `i>=size` ∧ `i>=0`

Fig. 3. The GCFG obtained from the GCFG of Figure 2(c) after symbolic reachability analysis from R_{9a} that represents a not-yet-covered branch of `getStatusOfValve`.



23: Call getStatusOfValve(i)
S_{23a} S_{23b} S_{23c} R_{23b} R_{23a}
6: Entry getStatusOfValve(int)
S_{6a} S_{6b} S_{6c} R_{6b} R_{6a}
7: i<0
R_{7a}
8: i>=0
R_{6c}

(a): Excerpt of the GCFG of Figure 3

(b): Update of the GCFG after adding the new state R_{6c} computed with symbolic reachability analysis at the frontier $\langle S_{6a}, R_{7a}\rangle$

REACHABILITY STATE PREDICATES:

R_{7a}: `true`

R_{6c}: `i<0` — that is, WP(`i<0`, R_{7a})

Fig. 4. The symbolic reachability analysis step from R_7.

Algorithm 3 ModelCoarseningStep

```
Require:
    $model$, a GCFG modified after a forward/backward analysis step
    $R$, a reachability condition included in $model$
Ensure:
    Removes the node $R$ from $model$
    Removes other nodes and edges to restore the compliance with Definition 3.1
    Returns the resulting GCFG
 1: $E_r$ = the set of edges between reachability conditions in $model$
 2: $E_f$ = the set of frontier edges in $model$
 3: $edgesFromR = \{\langle R, R'\rangle\} | \langle R, R'\rangle \in E_r$
 4: $edgesToR = \{\langle N, R\rangle\} | \langle N, R\rangle \in (E_r \cup E_f)$
 5: $adjacentToR = \{R'\} | \langle R, R'\rangle \in E_r \vee \langle R', R\rangle \in E_r$
 6: $model = ((model \setminus \{R\}) \setminus edgesFromR) \setminus edgesToR$
 7: for all $R' \in adjacentToR$ do
 8:     $model$ = ModelCoarseningStep($model, R'$)        # Invariant Inv4
 9: end for
10: return $model$
```

the analysis proves that an R-state is disconnected from the entry point (Algorithm 2, line 3), as for instance R_{23b} in the example of symbolic reachability analysis, meaning that the R-state is indeed unreachable. Yet other R-states become obsolete because of model coarsening (Algorithm 3, line 7) either when the coarsening step removes the successor of an R-state, implying that the reachability of the target of the R-state has been decided at the current analysis step, or when it removes the predecessor node of an R-states, implying that the reachability condition is now disconnected from the entry point.

5 BIDIRECTIONAL TEST CASE GENERATION

Bidirectional Test Case Generation (BiTe) exploits bidirectional symbolic analysis to improve and refine branch coverage, by both covering not-yet-covered branches and identifying infeasible branches to be pruned from the coverage domain. As discussed in Section 2, increasing of branch coverage may exercise critical cases that may lead to severe failures, while pruning infeasible branches improves the effectiveness of the coverage metrics.

5.1 BiTe Initialization Step

BiTe starts with a program and a set of test cases, and initializes the GCFG by means of Dynamic Symbolic Execution (DSE) and referring to the program control structure [24], [4].

BiTe relies on DSE to execute the initial set of test cases with both symbolic and concrete values, and uses the concrete values to identify feasible paths and drive the symbolic exploration of the state-space through these paths, avoiding infeasible ones. The initial GCFG contains symbolic states that are computed with DSE, and whose satisfiability is guaranteed by construction.

BiTe completes the initial GCFG by adding the R-states that represent the not-yet-covered program branches. After coarsening, the initial GCFG contains only those R-states that are directly connected to some S-state with a frontier edge. The R-states eliminated with this initial coarsening step depend on the reachability of the R-states currently in the GCFG, and will be included in the GCFG in the next analysis steps.

5.2 BiTe Symbolic Execution Step

BiTe relies on DSE to divert the paths of the existing test cases to execute program paths that have not been explored yet. Differently from the standard embodiment of DSE, BiTe targets the paths that can be diverted along the frontier edges of the GCFG. If BiTe succeeds to traverse some frontier edges with a new test case, it extends the existing test suite towards the uncovered branches represented by the terminal R-states of the GCFG.

In detail, BiTe instantiates the symbolic execution step by symbolically executing a test case up to the frontier, computing the successor symbolic state beyond the frontier, and solving the condition obtained by appending the path condition to the frontier reachability predicate with a Satisfiability Modulo Theories (SMT) solver.

The condition produced with this process characterizes the program inputs whose execution follows the same path as the original test case up to the frontier, but then traverses the frontier and satisfies the frontier R-state.

If the condition is satisfied for a given input, BiTe executes the program both concretely and symbolically with the new input. By construction the test execution traverses the GCFG frontier and identifies new symbolic states and new frontier edges. BiTe adds the new (satisfiable) symbolic states to the GCFG, and uses the concrete states to evaluate the symbolic reachability conditions at the branches traversed during the execution. This limits the calls to the SMT solver in Algorithm 1 to one call per iteration.

Let us consider the frontier edge $\langle S_{24b}, R_{25b}\rangle$ of the GCFG in Figure 5(a) that reports the complete version of the GCFG whose excerpt is shown in Figure 3. The figure separates the states of the two functions of the program, using the graphical notation of dotted edges to indicate the function calls and indicating with a positional correspondence the states involved with the call flows across the two functions. The symbolic states represented in the GCFG have been produced with DSE by executing the (initial) test case T_1 reported at the bottom of Figure 5.

When analyzing the frontier $\langle S_{24b}, R_{25b}\rangle$, BiTe first computes the symbolic state S_{25b} as the successor of state S_{24b} towards the $true$ branch of the `if` statement at line 25, and then computes the path condition $S_{25b} \wedge R_{25b}$ by joining the symbolic formulas of the two states, both reported at the bottom of Figure 5. This path condition is satisfiable with an input array $valveStatus = [-1, -1, -1]$, and thus BiTe identifies the new test case T_2 reported at the bottom of the figure. Figure 5(b) presents the GCFG updated by BiTe after executing T_2. The new test case T_2 traverses the same symbolic states as T_1 up to the frontier state S_{24b}, and then traverses the frontier edge and explores the states S_{25b}, S_{22d}, S_{23d}, S_{6d}, S_{8d}, S_{13d}, S_{14d}, S_{24d}, S_{25c}, S_{30b}, S_{31a}, S_{34b}, S_{35b} and S_{36b}, covering the branch 31 with S_{31a} and thus leading the program to execute the alarm at line 32.

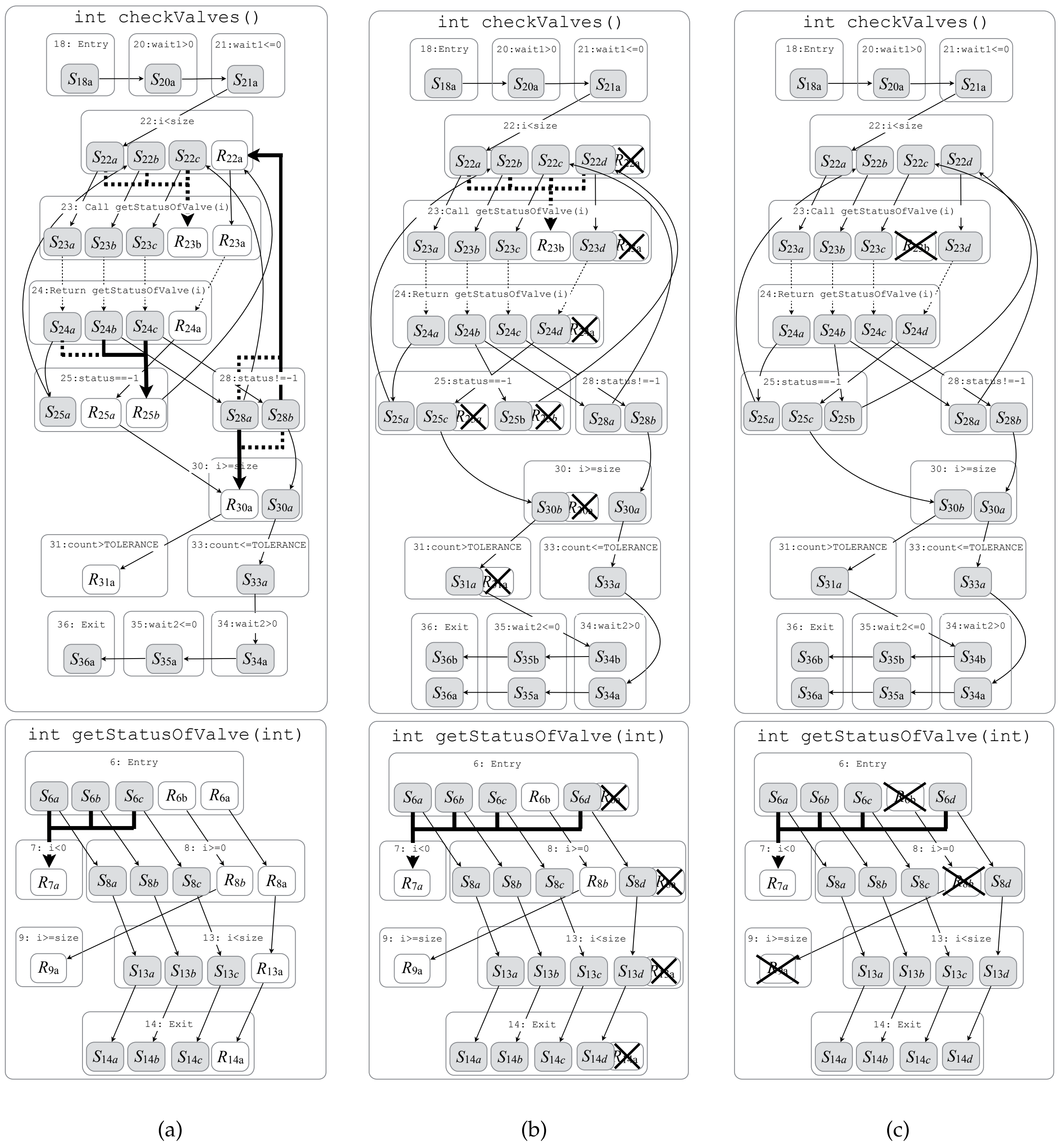
BiTe monitors the execution of the new test case, evaluates the R-states associated with the traversed branches, and removes (coarsening step) the R-states that are satisfied by concrete states traversed during the test case. Figure 5(b) partially overlaps the new S-states with the correspondingly satisfied R-states. These R-states (R_{25b}, R_{22a}, R_{23a}, R_{6a}, R_{8a}, R_{13a}, R_{14a}, R_{24a}, R_{25a}, R_{30a}, and R_{31a}) are marked with a cross, since they are satisfied in concrete states traversed during the execution of T_2 and are thus removed from the GCFG. The figure also indicates the new frontier edges after the update of the GCFG.

Bidirectional symbolic analysis requires evaluating the satisfiability of symbolic states and comparing states computed forward and backward. When exploring a symbolic state space, evaluating satisfiability and comparing states rely on SMT solvers, which can be computationally expensive. Being based on DSE, BiTe improves the efficiency of evaluating and comparing symbolic states, by limiting the calls to the solver only to the states that belong to the frontier, and taking advantage of the concrete states traversed by the test cases to evaluate the satisfiability of many symbolic reachability conditions with concrete evaluation.

5.3 BiTe Symbolic Reachability Analysis Step

BiTe instantiates the symbolic reachability analysis step by adapting the classic weakest precondition (WP) calculus [23] referring to the WPα predicate transformer proposed in DASH by Beckman et al. [15] that uses dynamic alias information that can be found in the GCFG frontier to guarantee efficient and conservative program abstractions in presence of pointers and pointer aliases. The WPα predicate transformer identifies a branch as infeasible in the absence of further aliases yet to be explored, otherwise it leaves the feasibility of the branch still open. The availability of more precise reachability analyzers may only improve on the current state of the prototype.

In the original proposal DASH addresses the problem of reaching an error state that represents the single target of the analysis. BiTe improves branch coverage by targeting all the program branches that have not been covered yet, thus extending DASH to the analysis of multiple branches. The GCFG model that BiTe incrementally maintains during the analysis supports the efficient coordination of multiple analyses à la DASH, mitigating the inefficiency of the independent analysis of the single targets. The coarsening step controls the space explosion that may derive from analyzing multiple targets by removing the states that become obsolete in the process.



SYMBOLIC STATE PREDICATES:

S_{25b}: `size=S1 ∧ wait1=W1 ∧ wait2=W2 ∧ valvesStatus=[V1,V2, V3, ...] ∧ S1>1 ∧ W1=1 ∧ V1=-1 ∧ V2=-1 ∧ i=1 ∧ count=1 ∧ status=V2`

REACHABILITY STATE PREDICATES:

R_{25b}: `valvesStatus[i+1]=-1 ∧ i>=-1 ∧ i<size-1 ∧ i>=size-2 ∧ count>0 ∧ status=-1`

TEST CASES:

T_1: `size=3 ∧ wait1=1 ∧ wait2=1 ∧ valvesStatus=[-1, 0, 0]`

T_2: `size=3 ∧ wait1=1 ∧ wait2=1 ∧ valvesStatus=[-1,-1,-1]`

Fig. 5. Updates of the GCFG model of the program `checkValves` after identifying both a new test case and an infeasible program branch with BiTe

The BiTe symbolic reachability analysis steps are triggered when the symbolic execution step fails in generating test cases that traverse the GCFG frontier. In this way we avoid wasting time in computing the weakest preconditions of elements that can be reached straightforwardly. When the symbolic execution steps fail on a given frontier, the symbolic reachability analysis steps progressively augment the reachability information in the model, up to possibly detecting infeasible branches. The new R-states drive the next iterations towards the analysis of program paths that are most likely to either reach uncovered branches or reveal infeasible branches in the program.

For instance, in the example of Figure 5(b) the forward analysis of the frontier from the states $S_{22a}, S_{22b}, S_{22c}, S_{22d}$ to the state R_{23b} has not identified any test case that can traverse the frontier. This triggers the symbolic reachability analysis step that eventually proves the frontier to be unreachable, as discussed in Section 4.2.

5.4 BiTe Coarsening Step

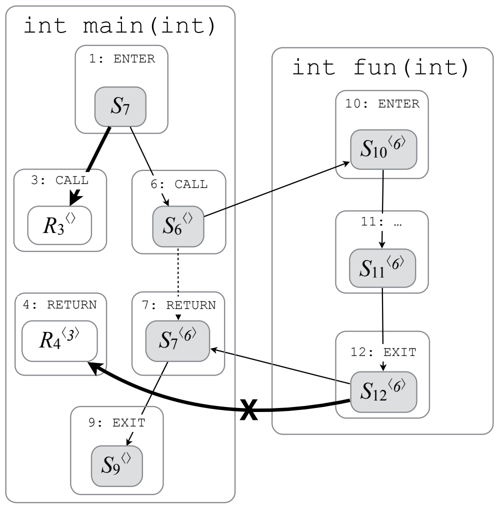
Symbolic reachability analysis leads to a monotonic growth of the model. The problem is further exacerbated because of the incremental growth of the GCFG and of the complex refinement predicates due to the simultaneous reachability analysis of several program branches. During the test generation process, branches are progressively classified as either covered or infeasible, thus rendering parts of the refined GCFG irrelevant for the goal of covering the still uncovered branches. The coarsening step controls the growth of the GCFG thus supporting scalability.

Coarsening removes nodes that become obsolete after the symbolic execution and the symbolic reachability analysis steps, or after the removal of other redundant nodes. We already exemplified the case of coarsening after a successful symbolic execution step in Figure 5(b). Figure 5(c) exemplifies the other cases. The symbolic reachability step illustrated in the previous subsection eliminates the frontier edge $\langle S_{22c}, R_{23b}\rangle$, and disconnects the node R_{23b} from the GCFG entry node. This implies that R_{23b} is recognized as unreachable and is pruned from the GCFG. Removing R_{23b}, which corresponds to a call to `getStatusOfValve`, disconnects node R_{6b}, which corresponds to the function entry point. Consequently the coarsening step recognizes also nodes R_{6b}, R_{8b} and R_{9a} as unreachable and removes them from the GCFG, thus proving that the program branch represented by the reachability condition R_{9a} is indeed infeasible.

6 Interprocedural analysis

The BiTe analysis is interprocedural, and can thus address realistic programs. The sample program of Figure 1 includes a procedure call that we have implicitly treated as a macro expansion in the examples so far, taking advantage of the presence of a single call. Here we present the interprocedural extension of BiTe that works with general programs.

Classical interprocedural analysis exploits either the *functional approach*, also called *compositional*, or the *call string approach* [25]. The functional approach combines intraprocedural summaries bottom-up for analyzing function calls. The call string approach maintains a description of the call sites traversed along each analyzed path, and excludes the interprocedural paths that do not satisfy the correct pairing between call and return sites [26]. These paths are called *non-realizable* paths in the literature on interprocedural analysis. Since bidirectional symbolic analysis requires pairing path prefixes and path suffixes incrementally analyzed during the forward and the backward analysis, respectively, we implement path sensitive interprocedural analysis according to the call string approach.



```
1:  prog(){                 10:  fun(){
2:    if(...){              11:     ...
3:      fun();              12:  }
4:
5:    } else {
6:      fun();
7:
8:    }
9:  }
```

Fig. 6. An example of interprocedural GCFG model.

Interprocedural GCFGs include interprocedural edges, and use call strings to store the interprocedural information of the symbolic states and the reachability conditions computed along paths that include interprocedural edges. The interprocedural edges connect the states at the call sites to the corresponding successor states at the entry point of the called procedures, and the states at the procedure exit points to the corresponding states at the return point of the calls. In the GCFG models of Figure 5 the interprocedural edges are the edges between the nodes that correspond to the lines 23 and 6 and between nodes 14 and 24 that correspond to the call to and return from function `getStatusOfValve`, respectively. Recall that in the figure these edge are indicated by the positional correspondence of the states involved with the call flows across the two functions. The call strings link the symbolic states and the reachability conditions in the GCFG to the call sites traversed and not yet returned along the reaching paths. In the GCFG models of Figure 5 we omit the call strings with no loss of generality, since the GCFG models deal with a single invocation.

Figure 6 illustrates the use of call strings in BiTe. The figure represents the GCFG obtained after symbolically executing the program `prog` (reported in the figure) along the branch that calls function `fun` at line 6, thus computing the states S_1, S_6, S_{10}, S_{11}, S_{12}, S_7, S_9. The states R_3 and R_4 represent the not yet reached call and return points related to the function call at line 3. The states include superscripts that represent the interprocedural calls strings. For instance, the call string $\langle 6 \rangle$ links the symbolic states S_{10}, S_{11}, S_{12} and S_7 to the call of function `fun` at line 6, indicating that these states have been explored in the context of that function call. The call strings of the states S_1, S_6 and S_9 are empty because no call site has been traversed yet. The call string $\langle 3 \rangle$ links the reachability condition R_4 to the call of function `fun` at line 3 that must be necessarily traversed to reach R_4. R_3 is associated to an empty call string because it can be reached from the entry point without going through any function call.

BiTe uses the call strings to exclude the frontier edges that depend on non-realizable paths. For example, with reference to the GCFG of Figure 6, the program control flow graph would suggest a frontier edge between nodes S_{12} and R_4, since the return point at line 4 can follow the function exit point at line 12. However, this frontier depends on the non-realizable path that reaches R_4 after traversing the call of function `fun` at line 6, and returning to line 4 when exiting from the function. The call strings associated to nodes $S_{12}^{\langle 6 \rangle}$ and $R_4^{\langle 3 \rangle}$ indicate the infeasibility of this frontier edge, which can thus be removed from the GCFG.

Generalizing from the example, BiTe computes frontier edges only between S-states and R-states that are associated with compatible call strings. The call string of a frontier S-state S is compatible with the call string of a frontier R-state R if the call string of S includes as a postfix the call string of R. This draws on the observation that the call string of the S-states always starts from the outermost program function, while the call string of the R-states starts from the program functions that contain the target branches.

7 Evaluation

Our research aims to provide a useful measure of branch coverage by both generating test cases that execute feasible branches, and identifying infeasible branches that can be excluded from the branch count. Executing feasible branches can increase the probability of revealing program failures, while excluding infeasible branches can improve the accuracy and stability of the produced coverage indicators. Both elements are important when considering the overall quality process where we need both to increase the likelihood of revealing subtle failures and to provide consistent data for managing the overall quality process.

7.1 Research Questions

We designed a set of experiments to address the following research questions:

- RQ_1 What is the coverage that BiTe test suites can achieve?
- RQ_2 What is the impact of the new measure of coverage that excludes infeasible branches from the coverage domain?
- RQ_3 How does BiTe compare with state-of-the-art symbolic approaches to test case generation?

The first research question, RQ_1, focuses on the usefulness of the coverage measured in terms of amount of executed feasible branches. BiTe aims to achieve high coverage by both generating test cases that execute feasible branches and excluding infeasible branches from the coverage domain. This leads to the second research question, RQ_2, that focuses on the relative weight of the two elements of the approach. We measure the impact of excluding infeasible branches from the coverage domain by comparing the new and classic branch coverage values. The third research question, RQ_3, addresses the validity of BiTe with respect to state-of-the-art techniques. We experimentally compare the classic branch coverage achieved with BiTe against the coverage obtained with state-of-the-art approaches, namely CREST [7] and KLEE [20].

7.2 Protoype Implementation

We ran our experiments with the *BiTe Prototype Tool for C* (BiTe$_c$) that implements the BiTe approach. The current implementation of BiTe$_c$ builds on top of CREST[3] for dynamic symbolic execution, CIL[4] for instrumenting and static analyzing C code, and on the Z3 SMT solver.[5] BiTe$_c$ supports most of the features of the C language including procedures, dynamic memory allocation, pointer arithmetics and reference aliasing. The high performance SMT solver Z3 provides partial support for nonlinear constraints. The test case generation procedure can be optionally seeded with an initial test suite.

The dynamic analysis components of BiTe$_c$ uses the GNU GDB debugger to monitor and access the state of running programs. The GDB API allows BiTe$_c$ to

3. https://github.com/jburnim/crest/
4. https://github.com/kerneis/cil/
5. http://z3.codeplex.com/

extract uniformly both concrete and symbolic data from a program execution as the symbolic memory is part of the concrete execution state of the instrumented program. BiTe$_c$ exploits the integration with GDB to identify the current alias conditions for building alias-aware refinement predicates. BiTe$_c$ computes the alias-aware refinement predicates in two steps. In the static initialization step, BiTe$_c$ records the sequences of consecutive assignments. In the dynamic execution step, BiTe$_c$ computes the alias predicates and finally the alias-aware refinement predicates.

7.3 Subject Programs

We conducted our experiments on a set of programs that are widely used as benchmarks in the literature and that are reported in Table 1. The first four subjects in the table correspond to four device drivers for the Windows NT kernel: `cdaudio`, `diskperf`, `floppy` and `kbfiltr`. These programs have a complex control flow that is reflected in a large number of paths, and have been proposed as benchmark in other scientific experiments on test case generation [27]. The next two programs implement the state machines that handle the communication at both the client and the server side in the the well known Secure Shell (SSH) protocol, which is ubiquitous on UNIX based systems [28]. The last program is `space`, a widely used benchmark that implements an antenna configuration system developed by the European Space Agency.

Figure 7 reports the code of function `GetKeyword` of program `space`, which illustrates the characteristics that complicate symbolic execution. The program `space` is an interpreter of a definition language for an array of antennas that we slightly adapted to obey some syntactic limitations of our current prototype. Function `GetKeyword` is invoked 107 times to parse the antenna system configuration file, and search for keywords (parameter `kw`) that shall appear in a given order in the configuration file (parameter `tp`). The loop at lines 12–18 checks for the validity of the input characters, while the call to function `strcmp` at line 22 checks whether the input keyword `kw` matches the string `word` identified in the loop.

To execute the many program branches that depend on the inputs that match the value of the keyword `kw`, the symbolic execution must solve the path condition that leads to executing the `then` branch of the `if` statement at line 22 after returning from the invocation of `strcmp(kw, word)`, for each invocation of function `GetKeyword`. Since all keywords are predefined strings and the length of the string `word` is determined at line 20 based on the value of variable `i` that counts the number of loop iterations, both the lengths of `kw` and `word` are always concrete constant values during symbolic execution, and thus solving the above path condition without additional information may require the exhaustive exploration of the execution space.

```
1  int GetKeyword(char *kw, struct charac** tp)
2  {
3      struct charac  *curr, **curr_ptr;
4      char word[KWDSLEN + 1], ch;
5      int i = 0;
6
7      curr_ptr = &curr;
8      *curr_ptr = *tp;
9
10     ch = TapeGet(curr_ptr);
11
12     while (((isalnum(ch)) || (ch == '_')) &&
13      (i < KWDSLEN) && ((*curr_ptr) != NULL))
14     {
15         word[i] = ch;
16         i = i + 1;
17         ch = TapeGet(curr_ptr);
18     };
19
20     word[i] = '\0';
21
22     if (strcmp(kw, word) == 0) {
23         *tp = *curr_ptr;
24         return 0;
25     } else
26         return 1;
27 }
```

Fig. 7. An excerpt of the program `space`

Moreover, an interesting execution is characterized by a proper sequence of characters that result in a combinatorial explosion of choices when executing function `GetKeyword` with classic symbolic execution. In fact at each iteration, the symbolic executor shall choose a set of characters out of all possible characters that satisfy the loop condition, that should also match the specified keywords. The variable length of the keywords introduces yet another combinatorial dimension to the problem. A classic symbolic executor that processes the input file forwardly learns the condition that shall be satisfied only after choosing the set of characters, thus leading to potential performance problems, as confirmed with the experiments discussed in this section.

For each of the subject programs, Table 1 reports the size expressed in lines of code (column *loc*) as counted by the GNU utility `wc`, and the number of branches (column *br*) as counted by BiTe$_c$. The number of branches in a programs reflects the complexity of the control structure. BiTe$_c$ counts the branches in a program statically, right after the CIL pre-compilation that unrolls decisions with multiple conditions as a cascade of single condition decisions, and eliminates the dead code that can be recognized by the compiler constant propagation analysis.

7.4 Experimental Methodology

To answer the research questions, we executed BiTe$_c$ on the subject programs, and studied the achieved coverage.

TABLE 1
Results of BiTe$_c$ execution on 6 subject programs.

Subjects			BiTe$_c$						
name	**loc**	**br**	**tc**	**cbr**	**ibr**	**cov$_b$**	**it$_b$**	**cov$_f$**	**it$_f$**
cdaudio	2,171	534	259	417	113	78%	20,965	99%	76,800
diskperf	1,104	194	59	166	14	86%	600	92%	13,200
floppy	1,144	234	75	206	14	88%	317	94%	14,700
kbfiltr	618	68	32	57	6	84%	129	92%	981
ssh client	760	160	39	135	24	84%	43,000	99%	85,000
ssh server	860	175	40	145	1	83%	49,400	83%	49,400
space	6,118	1,152	186	397	82	34%	5,610	66%	5,611

loc:	number of lines of code
br:	number of branches computed statically (after unrolling decisions with multiple conditions in equivalent cascade of single condition decisions)
tc:	number of generated test cases
cbr:	number of covered branches
ibr:	number of identified infeasible branches
cov$_b$:	branch coverage measured using the GNU `gcov` utility [as percentage]
it$_b$:	number of iterations to achieve cbr
cov$_f$:	cbr / (br - ibr) [as percentage]
it$_f$:	number of iterations to achieve both cbr and ibr

To answer RQ_1 and RQ_2 we compute the amount of program branches exercised with the test suites generated with BiTe$_c$, and the amount of branches that BiTe$_c$ identifies as infeasible. We measure the coverage of the BiTe$_c$ test cases referring both to the classic branch coverage, cov_b, and a new coverage, cov_f, that refers to feasible branches only. The cov_b coverage computes the portion of branches statically identified on the program control flow graph executed with the test suites generated with BiTe$_c$. The cov_f coverage only estimates the portion of feasible branches executed with the test suites generated with BiTe$_c$, and is defined as:

$$cov_f = \frac{br_{exec}}{br_{total} - br_{inf}}$$

where br_{exec}, br_{total} and br_{inf} indicate the number of program branches exercised with the BiTe$_c$ test suite, the number of program branches that belong to the static control flow graph of the program under test and the number of infeasible branches identified with BiTe$_c$, respectively.

The cov_f coverage metric interprets the intuitive and ideal concept of coverage that measures the amount of executed feasible branches and thus reaches 100% when all feasible branches are executed. Classic structural coverage metrics are easy to compute, but loose informative content when the amount of infeasible elements grows.

The cov_b and cov_f metrics obtained with BiTe$_c$ provide data to answer RQ_1, while their difference quantifies the contribution of eliminating the infeasible branches, thus answering RQ_2.

To answer RQ_3 we compare the coverage results of BiTe$_c$ with the ones produced with the state of the art test case generation tools KLEE and CREST. We instantiated CREST with three different strategies: plain random testing (Rand), bounded depth-first concolic execution[6] (CREST-dfs) and a heuristic strategy that aims to maximize branch coverage (CREST-cfg) [7]. KLEE supports many heuristics to drive the symbolic execution. We executed KLEE with all the available strategies, and compared with the best result for each program. We obtained the best results with the KLEE default strategy for all benchmark programs but *ssh server*, for which we achieved the best results with the *nurs:md2u* strategy, a heuristic that explores the programs paths in random order according to a non uniform distribution based on the minimum distance to an uncovered instruction.

To make the results comparable, we ran all the tools on the same code, obtained with the CIL code transformation that decomposes the decision statements that refer to the logic combination of atomic conditions in decision statements with a single atomic condition.

We ran the experiments on a laptop equipped with an Intel Core i7 2.7 GHz processor with 4 GB of memory and we repeated the executions with time budgets of both 60 and 180 minutes for each subject program.

We executed all the generated test suites with the GNU `gcov` tool to confirm that all the branches marked as covered are indeed traversed by the corresponding test cases. We also double-checked that none of the branches that BiTe$_c$ identify as infeasible were covered by other test generation tools. Moreover we manually confirmed the infeasibility of a sample of branches.

7.5 Results

Table 1 reports the results achieved when executing BiTe$_c$ on all the subject programs starting from an initially empty test suite and with a time budget of 180 minutes. The table reports the number of test cases that BiTe$_c$ generated for each program (column *tc*), the number of covered branches (column *cbr*) and the number of branches that BiTe$_c$ identified as infeasible (column *ibr*). Column cov_b shows the classic branch coverage computed using the GNU `gcov` utility. Column cov_f presents the new coverage obtained by removing infeasible branches from the coverage domain. Columns it_b and it_f report how many iterations were required to obtain cov_b and cov_f respectively.

BiTe$_c$ produced test suites of manageable size that cover most feasible branches (from 66% to 99%). The table shows clearly that cov_f is more accurate than cov_b. In most cases however, obtaining the most precise value requires more iterations (column it_f) than computing the approximated one (column it_b). The improved precision of coverage comes with a higher computation cost, but

6. We tried several bounds without observing significantly different results, thus we used the default bound.

the data about the relation between iterations and coverage suggest that completing the test case generation after investing a limited time budget impacts mostly on the precision of cov_f and less on the quality of the generated test cases.

At the same time, the improved accuracy of the results can greatly help technical staff in their decisions. Several standards require accurate metric data, for example, the DO-178C standard requires infeasible elements to be identified and removed from the final coverage metrics for critical software in in-flight systems. Normal production practice uses reachability analysis tools to identify most infeasible elements and alleviate the work of the specialists. BiTe is in line with this common practice. We can thus envision a process where BiTe is used with limited time budget to produce useful test cases, and additional time budget when accurate coverage data are required.

Table 2 presents the comparative analysis of BiTe$_c$ with Rand, CREST-dfs, CREST-cfg and KLEE. The first seven rows report the results with an analysis budget of 60 minutes for each program, as commonly used in literature, and indicate that (i) random testing achieves consistently low branch coverage for programs with a non trivial control structure, like the ones considered in our experiments, (ii) all the considered strategies largely improve over random testing, (iii) BiTe$_c$ (cov_b), KLEE and CREST achieve similar branch coverage rates for all programs but `ssh server`, (iv) BiTe$_c$ (cov_b) outperforms all other strategies for `ssh server`.

The last column of the table reports the branch coverage measured with BiTe$_c$ (cov_f) that corresponds to evaluate the coverage measured with BiTe by ignoring the branches proved infeasible with symbolic reachability analysis. The results clearly indicate that this novel measure of coverage better estimates the amount of covered feasible branches and thus provides more helpful results for developers and team managers.

We repeated the experiments with an analysis budget of 180 minutes. We obtain similar coverage values for all programs except for *space*. The last row of the table reports the results for the analysis of *space* with a time budget of 180 minutes. These results suggest that the time budget is not the key factor for the analysis and that the different approaches can effectively execute most feasible branches when the program structure is not too complex and does not include implicit dependencies that are difficult to reveal with classic approaches, but suffer when dealing with programs with implicit control dependencies as in *space*. The results with the higher time budget for *space* indicate that only BiTe can deal with complex control structures. In particular, the improvement of BiTe$_c$ (cov_b) from 11% to 34% suggests that BiTe can indeed improve on current approaches in the presence of implicit dependencies, and the improvement of BiTe$_c$ (cov_f) from 18% to 66% indicates that properly dealing with infeasible elements may indeed produce consistently realistic coverage information. Overall, the data reported in Table 2 indicate that the coverage obtained with BiTe is in line with the coverage obtained with the other approaches when the classic coverage is limited mostly by the presence of infeasible elements ($cov_f \geq 92\%$) and thus there is no much room for improvement, while BiTe increases significantly the classic branch coverage in the presence of uncovered but feasible elements ($cov_f \leq 83\%$).

These results confirm that the BiTe approach produces consistently better coverage data than state of the art techniques. The improvement is due both to the ability of identifying corner cases difficult to reach with classic heuristics and to the elimination of many branches identified as infeasible. The results for `ssh server` and `space` confirm that the presence of implicit dependencies between branches can largely hinder symbolic execution alone. In particular, in `space` many branches depend on specific values of the program variables that are assigned at previous branches and used at subsequent decision points in different functions of the program. Such implicit dependencies affect much less symbolic execution when combined with symbolic reachability analysis, regardless of the elimination of infeasible branches from the overall count.

In summary, our results indicate the advantages of bidirectional analysis and of the new branch coverage used in BiTe$_c$ (cov_f), and stimulate us to continue with the development of an optimized version of the prototype that may have a non trivial responsibility on the current performances. The current experiments confirm the research hypotheses encoded in the research questions: BiTe achieves a good branch coverage in all the experiments (between 66% and 98%) (RQ_1), BiTe largely improves the coverage data when dealing with infeasible elements (increasing the coverage from 34% to 66% in the best case of `space`) (RQ_2), BiTe achieves a classic branch coverage in line with state-of-the-art approaches in some experiments, and outperforms them in other experiments referring to both classic and new branch coverage (from 11% to 34% for classic coverage and to 66% for the new coverage, in the best case of `space`) (RQ_3).

7.6 Threats to Validity

The main threat to the *internal validity* of our experiments derives from possible faults in the prototype BiTe$_c$. We mitigated this threat by manually confirming the analysis results on the smaller test subjects and gradually moving to larger programs of increasing complexity. We further confirmed the correctness of the coverage results by cross checking the consistency with the outputs of other testing tools. Furthermore, we confirmed the coverage results with the GNU `gcov` tool, double-checked that no test generation tool executes the branches that BiTe$_c$ identified as infeasible, and manually confirmed the infeasibility of a sample of those branches.

We experimented with a dedicated machine to avoid

TABLE 2
Comparison of branch coverage scores of BiTe$_c$ and CREST and KLEE with an analysis budget of 60 minutes, and comparison of the results with analysis budget of 60 and 180 minutes for program *space*

	Branch coverage (%)					
Program	**Rand**	**CREST-dfs**	**CREST-cfg**	**KLEE**	**BiTe$_c$ (cov_b)**	**BiTe$_c$ (cov_f)**
cdaudio (60 min)	2%	78%	77%	79%	78%	**99%**
diskperf (60 min)	3%	75%	85%	76%	86%	**92%**
floppy (60 min)	2%	86%	86%	87%	88%	**94%**
kbfiltr (60 min)	38%	84%	84%	85%	84%	**92%**
ssh client (60 min)	2%	83%	83%	84%	84%	**99%**
ssh server (60 min)	34%	65%	77%	77%	83%	**83%**
space (60 min)	2%	11%	2%	9%	11%	**18%**
average	*12%*	*69%*	*71%*	*71%*	*73%*	**82%**
space (180 min)	2%	12%	2%	9%	34%	**66%**

interferences with the machine workload, and we computed the average coverage of repeated executions to further reduce the impact. Nevertheless, the background activity of the system and the random nature of the considered analysis techniques may have had a small impact on the results that shall be interpreted within a small accuracy interval. In particular, a difference of 1% in the coverage data reported in Table 2 shall be interpreted as a comparable coverage without a strict inclusion implication.

The *external validity* of our evaluation relates the extent to which our results are generalizable. Despite the high level of precision in both the symbolic execution and reachability analysis components, BiTe$_c$ might not handle completely some classes of software systems. The choice of industrially relevant case studies supports the industrial applicability of the results. We evaluated BiTe also in the context of a European project on a case study of four incremental versions of a real-time software component that controls a robot responsible for the maintenance of the ITER nuclear fusion plant [29]. Such industrial software systems display characteristics that are notoriously challenging for automated test data generation tools, such as the extensive use of nonlinear and floating point arithmetics. The study indicates that BiTe can expose subtle (previously unknown) bugs in this type of software.

Our comparison of BiTe with respect to current symbolic testing techniques may be threatened by the choices of the competitors. The publicly available versions of CREST and KLEE may not be sufficiently representative of all the symbolic techniques proposed in the recent years. The use of the four NT device drivers in the experiments reduces the severity of this limitation. These subjects have been used to evaluate other recent approaches to automated test generation, and thus the coverage obtained by BiTe$_c$ can be compared to the one reported in those papers, without requiring to repeat the experiments. In particular, the analysis of the data presented by Jaffar et al. [27] confirms that BiTe$_c$ obtains higher branch coverage on those subjects. This allows us to assert that the performance of BiTe$_c$ in terms of branch coverage exceeds the current state of the research.

8 Related Work

Automated Test Data Generation (ATDG) has been extensively studied in the past decades. The seminal work of the seventies has been extended with a rich variety of techniques and approaches [30], [31], [32]. In this section, we survey ATDG techniques with particular emphasis on approaches based on symbolic execution, present techniques to detect infeasible program elements, and discuss the contribution of this paper in relation to the existing work and the main open challenges.

Table 3 classifies the main approaches to ATDG according to the most popular taxonomies that identify three orthogonal dimensions: automation approach, test objective and analysis approach, independently from their technical background. In the table, columns map *test objectives* and *analysis approaches*, while rows correspond to the *automation approaches*. Gray cells pinpoint the unlikely combinations of path driven techniques and dynamic analysis, since purely dynamic analysis hardly suites for deriving inputs that make specific program paths execute. The remaining empty cells indicate open areas that could be investigated in the future.

Below we limit our survey to the ATDG approaches that represent the most relevant path driven (symbolic execution based testing) and review the research efforts on reachability analysis, verification and model checking approaches that relate to the detection of unreachable elements in BiTe.

8.1 Symbolic Execution Based Testing

Symbolic Execution (SE) is the most popular path based testing approach. SE was proposed in the seventies, but at that time its practical applicability to industrial scale software systems was limited by severe scalability issues. The relevant algorithmic improvements to SE proposed in the last decade, the availability of cheap computing power and the progresses in decision procedures fostered the recent emergence of many new SE tools [51], [53], [20], [62], [63].

The effectiveness of SE for generating test cases is challenged by the *path explosion*, the *constraint solving* and the *infeasible path* problems. As the number of paths

TABLE 3
Classification of the State of the Research according to: test objective, automation approach and analysis technique

		Test Objective (Analysis Approach)								
		Structural Coverage			Error State			Non Functional		
		Dyn	Stat	Hyb	Dyn	Stat	Hyb	Dyn	Stat	Hyb
Automation Aproach	Input Domain Driven	[33] [34] [35] [36]		[37]	[38] [39] [40] [41]	[42] [43] [11] [44] [45]	[46] [47] [48]	[49]		[50]
	Path Driven	▨	[51] [52] [53] [20] [54] [55] [56] [57] [58] [59] [60] [61] [62] [9] [63]	[64] [65] [7] [8] [66] [67] [27] [68]	▨	[69] [70] [71]	[4] [24] [37] [72] [73] [74] [75] [76] [77] [78] [79] [80] [15]	▨	[81] [82] [83]	[84]
	Goal Driven	[85] [86] [87] [88] [89] [90] [91] [92] [93] [94] [5] [95]		[96] [97] [18] [98] [99] [100] [101] [102]	[103] [104] [105] [106] [107] [108] [109] [110]			[103] [111] [112] [113] [114] [115] [116]		

grows exponentially with the number of conditions in the program, enumerating all paths is generally infeasible, and SE may diverge trying to explore an infinite number of paths, resulting in exploring only a subset of the program execution space. The incompleteness of constraint solvers reduces the set of successfully analyzable programs. Finally, SE may fail to decide that some path conditions are unsatisfiable, and may thus diverge in presence of infeasible program paths.

BiTe relies on *dynamic symbolic execution (DSE)*, also referred to as *concolic testing*, to mitigate the path feasibility and constraint solving issues by using dynamic information from concrete executions to guide SE [24], [4], [64]. DSE first symbolically analyzes the program along some concrete execution paths, which are feasible by construction and do not require constraint solver queries to validate the satisfiability of the corresponding path conditions, and then systematically explores alternative program paths by punctually modifying and solving the computed path conditions. DSE also replaces symbolic values with concrete ones every time the constraint solver cannot cope with them, thus further overcoming the constraint solving issues at the expense of precision.

BiTe addresses the path explosion problem by focusing DSE only on paths that may lead to execute not-yet-covered branches. Several studies investigate the effectiveness of SE when coupled with *heuristic* path exploration strategies that prioritize the analysis of promising program paths to execute not-yet-covered elements [7], [8], [67]. Papadakis and Malevris propose a path selection strategy to reduce the impact of infeasible paths while targeting branch coverage [57]. Li et al. identify less explored paths measuring the *length-n subpath program spectra* that generalize branch coverage to approximate full path information by profiling the execution of loop-free program paths of given length [9].

BiTe is based on bidirectional symbolic analysis, which is a universal search strategy and avoids ad hoc heuristics that can be defeated by specific program structures. Compared with heuristic path exploration strategies, the BiTe bidirectional analysis can identify and exclude from further exploration both the provably infeasible branches and the program paths that provably cannot reach uncovered branches, thus focusing the search strategy on the paths that can reach uncovered branches and may deserve deeper exploration. The experiments reported in this paper provide evidence that BiTe can be more effective then the heuristics strategies implemented in CREST and KLEE for generating test suites that achieve high branch coverage.

Other ATDG approaches prioritize the state space exploration or use DSE in combination with other techniques with the goal of maximizing program failures. EXE drives symbolic execution by monitoring the path constraints that are compatible with values that might cause a failure [69]. KATCH uses path selection heuristics to drive SE towards executing software patches by using a combination of static and dynamic techniques [68]. Predictive Testing and ZESTI use DSE to check runtime error conditions against the path conditions computed along the explored paths [74], [75]. Check 'n' Crash targets test case generation towards fault conditions computed with backward symbolic analysis [76], [79]. The Marple tool uses a backward path-sensitive symbolic analysis to detect buffer overflows [70]. X-force drives symbolic execution towards memory allocation sites [71]. Majumdar and Sen propose hybrid concolic testing that alternates random and concolic testing [37]: random testing guarantees a uniform exploration of the program input space, while concolic testing implements exhaustive local search. Păsăreanu et al. and Chipounov et al. tackle the path explosion problem with techniques that combine exhaustive SE at the code unit level with the concolic execution of selected paths at the system level [65], [66].

Other researchers use DSE to deal with dynamically generated code where static analysis is inapplicable, for example to generate test cases for database programs that include dynamically generated SQL queries, or for PHP programs that execute within dynamic Web pages [72], [73]. Xu et al. generalize concrete executions with the symbolic handling of buffer lengths to search for memory violations [84].

Differently from the approaches surveyed above, BiTe builds on the strengths of DSE to target branch coverage, instead of systematically exploring the program paths within regions of the input space. BiTe does not directly try to maximize a specific failure detection metric, but builds on the empirically validated hypothesis that high branch coverage, albeit hard to obtain, is correlated with high failure detection rates [1], [2]. By targeting branch coverage instead of systematically exploring the program execution space, BiTe avoids traps that hinder the state-of-the-art approaches and that can prevent high coverage for specific target programs.

The performance of BiTe could be further improved by reusing interprocedural computations compositionally, implementing optimizations at the decision procedure level, and exploiting parallel computation, as in the techniques that we survey below.

State matching and compositional interprocedural approaches can mitigate the path explosion problem in SE. *State matching* checks if a state visited during SE is subsumed by another symbolic state. This information is useful as it can prevent SE from analyzing the same states over and over again. However, as the number of symbolic states may be infinite, state matching becomes quickly impractical. Anand et al. propose abstract subsumption, a technique that enables state matching by exploiting specific program abstractions, in particular for lists and arrays [52]. Jaffar et al. propose to use interpolating theorem provers to mitigate the path-explosion problem, though they notice that the approach might be not cost-effective for branch coverage [27]. BiTe$_c$ builds the GCFG program model on demand by using the WPα predicate transformer that combines the precision of the classical weakest preconditions and the scalability of dynamic alias analysis. Abstract subsumption and interpolants could be integrated in the BiTe$_c$ architecture without major changes to the current implementation.

Another recent approach to improve the scalability of SE is to reason about program modules separately, deriving function *summaries* that can be reused compositionally [77]. Such approach is borrowed from classical interprocedural analysis design techniques [25]. The tool SMASH builds summaries on demand and distinguishes between *may* and *must* summaries that encode respectively over-approximated and under-approximated abstractions [58].

Invoking a decision procedure to solve constraints generated with SE is usually the most time consuming step of symbolic test case generation. SE tools normally use constraint solvers as black box tools, and take advantage of the improvements in the field by simply upgrading the solvers. Some ATDG approaches investigate the advantages of querying several solvers in parallel and make use of the results that are computed faster [117]. Other researchers investigate the effectiveness of caching and reusing the solutions produced with the constraint solver across the constraints that recur several times during the analysis [20], [118], [119], [120]. The *DomainReduce* approach aims to better integrate symbolic execution and constraint solving, eliminating potentially irrelevant constraints based on dynamically monitored program dependencies [121].

As multi-core processors and distributed systems are getting more and more common, researchers are investigating parallel computation strategies to accelerate symbolic execution, and reduce the performance issues that derive for the solvers and the path explosion problem. Staats and Păsăreanu apply a *static partitioning* technique to symbolic execution trees inspired by model checking approaches [56]. Static partitioning computes statically the preconditions that characterize distinct subtrees of the SE tree; such subtrees are then analyzed independently and in parallel obtaining speedups up to 90x. Starting from the observation that static balancing cannot predict precisely the actual workload of the worker nodes, Bucur et al. proposed Dynamic Distributed Exploration [60]. The design of BiTe suggests a natural parallelization strategy in which at every step all the elements in the reachability frontier are analyzed independently.

8.2 Reachability Analysis

In this paper, we advocate the need to tackle the problem of detecting infeasible code elements while generating test cases. Proving the infeasibility of a code element is the dual problem of finding a test case that executes it.

Many safety properties of programs can be expressed as reachability problems, and are addressed with automatic verification approaches that exploit the weakest precondition calculus [23]. Most state of the art approaches are grounded on both suitable logics that describe the safety properties and automatic decision procedures that handle satisfiability claims in (some decidable fragment of) those logics [122], [123], [124], [125], [126], [127], [128]. The BiTe symbolic reachability analysis steps are grounded on the weakest precondition calculus as well.

In its classic form, weakest precondition calculus requires loop invariants that cannot be automatically synthesized in general, and is challenged by the presence of aliases [23]. Recent approaches use abstractions to over approximate sets of program paths [129] and rely on techniques to automatically infer loop invariants [130]. BiTe uses dynamic alias information to improve the efficiency of computing the weakest preconditions, and takes advantage from the DSE steps to reduce the number of program paths and loops that must be analyzed with symbolic reachability analysis, and thus can be fully automated.

Software model checking can be naturally applied to reachability analysis. Traditional explicit state and symbolic model checking can scale to millions of states, but for general purpose software, with an unbounded state space, this is still not enough [131]. CEGAR (counter example based abstraction refinement) model checking contrasts the state explosion by refining a finite abstraction of the program behavior. CEGAR approaches based on satisfiability checking tools proved to be very effective and parallelizable for the sake of software model checking [16], [17]. Beyer et al. identify paths invariants to refine multiple infeasible paths at a time [132]. The SYNERGY approach of Gulvani et al. later enhanced by Beckman et al. within the DASH approach uses symbolic test case generation to generate counterexamples for the abstraction refinement steps [13], [15].

BiTe builds upon the DASH approach; differently from DASH, BiTe defines the GCFG model to efficiently coordinate multiple analyses à la DASH, thus mitigating the inefficiency of the independent analysis of the single targets, introduces the coarsening step to control the space explosion that may derive from tracking the preconditions of multiple targets via the WP_α predicate transformer, and exploits interprocedural call strings to pair the path prefixes and the path suffixes incrementally analyzed during symbolic execution and symbolic reachability analysis.

Optimizing compilers detect dead code using data flow analysis, a static technique that over approximates the set of possible values for variables at all program locations [133], [134]. Data flow analysis however is not path sensitive and can propagate data flow facts across infeasible paths, producing imprecise results. Symbolic execution can overcome data flow limitations as it integrates with constraint solvers to determine the feasibility of program paths. Delahaye et al. propose to combine the two approaches to generalize individual unreachability results to sets of program paths [135]. Unsound approaches aim to efficiently detect a large portion of infeasible code while accepting some false positives that are feasible paths wrongly classified as infeasible. Ngo and Tan use pattern matching to detect infeasible paths based on empirical rules, reporting high levels of precision and recall [136], [137]. BiTe implicitly relies on sound static data flow analysis to early detect and prune dead code.

9 Conclusions

Structural coverage, and in particular branch coverage, are popular techniques to measure the thoroughness of test cases. The difficulty of executing corner cases that represent rare execution conditions and the approximation that derives from the impossibility of identifying all infeasible elements limit the effectiveness of current approaches. The approaches proposed so far to improve structural coverage deal with either the problem of executing uncovered elements or the problem of demonstrating the infeasibility of other elements. The few straightforward combinations of the two classes of approaches improve coverage, but focus on specific elements, and may be trapped into infinitely long attempts to either include or exclude the target elements from coverage, and thus do not represent acceptable solutions.

In this paper we propose a novel solution that relies on a new metric that defines branch coverage referring to feasible branches only, thus excluding the branches that can be proven infeasible from the final metrics. The approach can indeed identify many feasible elements and at the same time exclude many infeasible ones, thus reaching consistently high coverage values that can greatly help developers and quality experts in their decisions. The approach is centered on a model, which identifies what we call a frontier that represents the set of elements that can be either most likely included in or excluded from the coverage domain. The frontier is the pivot for alternating symbolic execution that aims to execute not-yet-covered elements and symbolic reachability analysis that aims to identify infeasible elements to be excluded from the coverage domain. The frontier avoids both analyses to be trapped by singularities in the program execution space that may not be easily either reached or identified as infeasible, and thus overcomes the problems of current techniques that combine different types of analyses.

The results presented in this paper open new research frontiers. In particular, although the general technique is not restricted to a specific coverage criterion, so far we have targeted and experimented with branch coverage. We are currently investigating the extension of bidirectional symbolic analysis to other kinds of coverage. Moreover, the performance of the current prototype although acceptable may impact on the scalability of the technique. We are currently studying new techniques to overcome some technical problems of the current prototype to improve the performance and thus the scalability of the approach.

Acknowledgment

This work has been partially supported by the Swiss SNF project AVATAR (200021_132666) and by the Italian PRIN project IDEAS (2012E47TM2_006).

References

[1] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments of the effectiveness of dataflow-and controlflow-based test adequacy criteria," in *ICSE '94: Proceedings of the 16th international conference on Software engineering*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1994, pp. 191–200.

[2] A. S. Namin and J. H. Andrews, "The influence of size and coverage on test suite effectiveness," in *Proceedings of the eighteenth international symposium on Software testing and analysis (ISSTA '09)*. New York, NY, USA: ACM, 2009, pp. 57–68.

[3] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in *Proceedings of the International Conference on Software Engineering*, 2014.

[4] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE-13)*, 2005, pp. 263–272.

[5] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Trans. Softw. Eng.*, vol. 39, no. 2, pp. 276–291, Feb. 2013. [Online]. Available: http://dx.doi.org/10.1109/TSE.2012.14

[6] C. Pacheco and M. D. Ernst, "Randoop: Feedback-directed random testing for java," in *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion*, ser. OOPSLA '07. New York, NY, USA: ACM, 2007, pp. 815–816. [Online]. Available: http://doi.acm.org/10.1145/1297846.1297902

[7] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," in *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008)*, 2008, pp. 443–446.

[8] T. Xie, N. Tillmann, P. de Halleux, and W. Schulte, "Fitness-guided path exploration in dynamic symbolic execution," in *Proceedings of the 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2009)*, June-July 2009, pp. 359–368. [Online]. Available: http://www.csc.ncsu.edu/faculty/xie/publications/dsn09-fitnex.pdf

[9] Y. Li, Z. Su, L. Wang, and X. Li, "Steering symbolic execution to less traveled paths," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &#38; Applications*, ser. OOPSLA '13. New York, NY, USA: ACM, 2013, pp. 19–32. [Online]. Available: http://doi.acm.org/10.1145/2509136.2509553

[10] RTCA, Inc., "DO-178C/ED-12C: Software considerations in airborne systems and equipment certification," December 2011.

[11] T. Ball, B. Cook, V. Levin, and S. K. Rajamani, "SLAM and static driver verifier: Technology transfer of formal methods inside Microsoft," in *Proceedings of the 4th International Conference on Integrated Formal Methods (IFM 2004)*. Springer, 2004, pp. 1–20.

[12] L. A. Clarke and D. S. Rosenblum, "A historical perspective on runtime assertion checking in software development," *SIGSOFT Softw. Eng. Notes*, vol. 31, no. 3, pp. 25–37, May 2006. [Online]. Available: http://doi.acm.org/10.1145/1127878.1127900

[13] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani, "Synergy: A new algorithm for property checking," in *Proceedings of the 14th ACM SIGSOFT symposium on Foundations of Software Engineering (FSE-14)*, 2006, pp. 117–127.

[14] D. Beyer, T. Henzinger, R. Jhala, and R. Majumdar, "The software model checker BLAST," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 9, no. 5-6, pp. 505–525, Oct. 2007.

[15] N. E. Beckman, A. V. Nori, S. K. Rajamani, R. J. Simmons, S. D. Tetali, and A. V. Thakur, "Proofs from tests," *IEEE Transactions on Software Engineering*, vol. 36, no. 4, pp. 495–508, 2010.

[16] A. R. Bradley, "Sat-based model checking without unrolling," in *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation*, ser. VMCAI'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 70–87. [Online]. Available: http://dl.acm.org/citation.cfm?id=1946284.1946291

[17] A. Cimatti and A. Griggio, "Software model checking via IC3," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, P. Madhusudan and S. Seshia, Eds. Springer Berlin Heidelberg, 2012, vol. 7358, pp. 277–293.

[18] M. Baluda, P. Braione, G. Denaro, and M. Pezzè, "Enhancing structural software coverage by incrementally computing branch executability," *Software Quality Journal*, vol. 19, no. 4, pp. 725–751, 2011.

[19] M. Baluda, P. Braione, G. Denaro, and M. Pezzè, "Structural coverage of feasible code," in *Proceedings of the Fifth International Workshop on Automation of Software Test (AST 2010)*, 2010.

[20] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008)*, 2008.

[21] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976.

[22] L. A. Clarke, "A system to generate test data and symbolically execute programs," *IEEE Transactions on Software Engineering*, vol. 2, pp. 215–222, May 1976.

[23] E. W. Dijkstra, *A Discipline of Programming*. Upper Saddle River, NJ, USA: Englewood Cliffs: prentice-hall, 1976.

[24] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI 2005)*, 2005, pp. 213–223.

[25] A. Pnueli and M. Sharir, "Two approaches to interprocedural data flow analysis," *Program flow analysis: theory and applications*, pp. 189–234, 1981.

[26] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '95. New York, NY, USA: ACM, 1995, pp. 49–61.

[27] J. Jaffar, V. Murali, and J. A. Navas, "Boosting concolic testing via interpolation," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: ACM, 2013, pp. 48–58. [Online]. Available: http://dx.doi.org/10.1145/2491411.2491425

[28] D. Beyer and M. E. Keremoglu, "Cpachecker: A tool for configurable software verification," in *Proceedings of the 23rd International Conference on Computer Aided Verification*, ser. CAV'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 184–190. [Online]. Available: http://dl.acm.org/citation.cfm?id=2032305.2032321

[29] P. Braione, G. Denaro, A. Mattavelli, M. Vivanti, and A. Muhammad, "Software testing with code-based test generators: Data and lessons learned from a case study with an industrial software component," *Software Quality Control*, vol. 22, no. 2, pp. 311–333, Jun. 2014. [Online]. Available: http://dx.doi.org/10.1007/s11219-013-9207-1

[30] P. McMinn, "Search-based software test data generation: A survey: Research articles," *Software Testing, Verification and Reliability*, vol. 14, no. 2, pp. 105–156, Jun. 2004. [Online]. Available: http://dx.doi.org/10.1002/stvr.v14:2

[31] C. S. Păsăreanu and W. Visser, "A survey of new trends in symbolic execution for software testing and analysis," *International Journal on Software Tools for Technology Transfer*, vol. 11, no. 4, pp. 339–353, 2009.

[32] G. Fraser, F. Wotawa, and P. E. Ammann, "Testing with model checkers: a survey," *Software Testing, Verification and Reliability*, vol. 19, no. 3, pp. 215–261, Sep. 2009.

[33] S.-D. Gouraud, A. Denise, M.-C. el, and B. Marre, "A new way of automating statistical testing methods," in *Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, ser. ASE '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 5–. [Online]. Available: http://dl.acm.org/citation.cfm?id=872023.872550

[34] T. Y. Chen, H. Leung, and K. Mak, "Adaptive random testing," in *Advances in Computer Science - ASIAN 2004. Higher-Level Decision Making*, ser. Lecture Notes in Computer Science, M. Maher, Ed. Springer Berlin / Heidelberg, 2005, vol. 3321, pp. 3156–3157.

[35] S. H. Wu, S. Jandhyala, Y. K. Malaiya, and A. P. Jayasumana, "Antirandom testing: A distance-based approach," *VLSI Design*, vol. 2008, no. 2, pp. 2:1–2:9, Jan. 2008. [Online]. Available: http://dx.doi.org/10.1155/2008/165709

[36] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer, "Artoo: Adaptive random testing for object-oriented software," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 71–80.

[37] R. Majumdar and K. Sen, "Hybrid concolic testing," in *Proceedings of the 29th International Conference on Software Engineering*. IEEE Computer Society, 2007, pp. 416–426.

[38] F. Chan, T. Chen, I. Mak, and Y. Yu, "Proportional sampling strategy: Guidelines for software testing practitioners," *Information and Software Technology*, vol. 38, no. 12, pp. 775 – 782, 1996. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0950584996011032

[39] K. Claessen and J. Hughes, "Quickcheck: A lightweight tool for random testing of haskell programs," in *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '00. New York, NY, USA: ACM, 2000, pp. 268–279.

[40] Y. Wei, H. Roth, C. A. Furia, Y. Pei, A. Horton, M. Steindorfer, M. Nordio, and B. Meyer, "Stateful testing: Finding more errors in code and contracts," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 440–443. [Online]. Available: http://dx.doi.org/10.1109/ASE.2011.6100094

[41] R. Nokhbeh Zaeem and S. Khurshid, "Test input generation using dynamic programming," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY,

USA: ACM, 2012, pp. 34:1–34:11. [Online]. Available: http://doi.acm.org/10.1145/2393596.2393635

[42] T. Ball, "Abstraction-guided test generation: A case study," Microsoft Research, Tech. Rep. MSR-TR-2003-86, Nov. 2003.

[43] ——, "A theory of predicate-complete test coverage and generation," in *Proceedings of the Third International Symposium on Formal Methods for Components and Objects (FMCO 2004)*, ser. Lecture Notes in Computer Science, vol. 3657. Springer Berlin / Heidelberg, Nov. 2004, pp. 1–22.

[44] D. Beyer, A. J. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar, "Generating tests from counterexamples," in *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*. IEEE Computer Society, 2004, pp. 326–335.

[45] C. D. Nguyen, A. Marchetto, and P. Tonella, "Combining model-based and combinatorial testing for effective test case generation." in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ser. ISSTA 2012. New York, NY, USA: ACM, 2012, pp. 100–110. [Online]. Available: http://doi.acm.org/10.1145/2338965.2336765

[46] J. Callahan, F. Schneider, and S. Easterbrook, "Automated software testing using model-checking," in *Proceedings of the 1996 SPIN Workshop (SPIN 1996). Also WVU Technical Report NASA-IVV-96-022.*, 1996. [Online]. Available: citeseer.ist.psu.edu/article/callahan96automated.html

[47] C. S. Păsăreanu, R. Pelánek, and W. Visser, "Concrete model checking with abstract matching and refinement," in *Proceeding of the 17th International Conference on Computer Aided Verification (CAV 2005)*, ser. LNCS, no. 3576. Springer, 2005, pp. 52–66.

[48] G. Yorsh, T. Ball, and M. Sagiv, "Testing, abstraction, theorem proving: better together!" in *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*. New York, NY, USA: ACM, 2006, pp. 145–156.

[49] M. Grechanik, C. Fu, and Q. Xie, "Automatically finding performance problems with feedback-directed learning software testing," in *Proceedings of the 2012 International Conference on Software Engineering*, ser. ICSE 2012. Piscataway, NJ, USA: IEEE Press, 2012, pp. 156–166. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337242

[50] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *Proceedings of the 21st USENIX conference on Security symposium*, ser. Security'12. Piscataway, NJ, USA: USENIX Association, 2012, pp. 38–38. [Online]. Available: http://dl.acm.org/citation.cfm?id=2362793.2362831

[51] W. Visser, C. S. Păsăreanu, and S. Khurshid, "Test input generation with Java PathFinder," in *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2004)*. ACM, 2004, pp. 97–107.

[52] S. Anand, C. S. Păsăreanu, and W. Visser, "Symbolic execution with abstract subsumption checking," in *Proceedings of the 13th International Conference on Model Checking Software*, ser. SPIN'06. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 163–181.

[53] ——, "Jpf-Se: A symbolic execution extension to Java pathfinder," in *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 2007)*, Braga, Portugal, March 2007, pp. 134–138.

[54] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu, "Differential symbolic execution," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. SIGSOFT '08/FSE-16. New York, NY, USA: ACM, 2008, pp. 226–237. [Online]. Available: http://doi.acm.org/10.1145/1453101.1453131

[55] K. L. McMillan, "Lazy annotation for program testing and verification," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, T. Touili, B. Cook, and P. Jackson, Eds. Springer Berlin / Heidelberg, 2010, vol. 6174, pp. 104–118.

[56] M. Staats and C. S. Păsăreanu, "Parallel symbolic execution for structural test generation," in *Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA 2010)*. ACM, 2010, pp. 183–194.

[57] M. Papadakis and N. Malevris, "A symbolic execution tool based on the elimination of infeasible paths," in *Software Engineering Advances (ICSEA), 2010 Fifth International Conference on*, 2010, pp. 435–440.

[58] P. Godefroid, A. V. Nori, S. K. Rajamani, and S. D. Tetali, "Compositional may-must program analysis: Unleashing the power of alternation," in *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL 2010)*. ACM, 2010, pp. 43–56.

[59] G. Li, I. Ghosh, and S. P. Rajan, "Klover: A symbolic execution and automatic test generation tool for c++ programs," in *CAV*, 2011, pp. 609–615.

[60] S. Bucur, V. Ureche, C. Zamfir, and G. Candea, "Parallel symbolic execution for automated real-world software testing," in *Proceedings of EuroSys 2011*. ACM, 2011.

[61] J. Jaffar, V. Murali, J. A. Navas, and A. E. Santosa, "Tracer: A symbolic execution tool for verification," in *Proceedings of the 24th international conference on Computer Aided Verification*, ser. CAV'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 758–766.

[62] P. Braione, G. Denaro, and M. Pezzè, "Enhancing symbolic execution with built-in term rewriting and constrained lazy initialization," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: ACM, 2013, pp. 411–421. [Online]. Available: http://doi.acm.org/10.1145/2491411.2491433

[63] ——, "Symbolic execution of programs with heap inputs," in *Proceedings of the 2015 Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. ACM, 2015.

[64] N. Tillmann and J. de Halleux, "Pex: White box test generation for .NET," in *Proceedings of the 2nd International Conference on Tests and Proofs*, ser. TAP'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 134–153. [Online]. Available: http://portal.acm.org/citation.cfm?id=1792786.1792798

[65] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, and M. Pape, "Combining unit-level symbolic execution and system-level concrete execution for testing nasa software," in *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ser. ISSTA '08. New York, NY, USA: ACM, 2008, pp. 15–26.

[66] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A platform for in-vivo multi-path analysis of software systems," in *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2011)*. ACM, 2011, pp. 265–278.

[67] T. Su, G. Pu, B. Fang, J. He, J. Yan, S. Jiang, and J. Zhao, "Automated coverage-driven test data generation using dynamic symbolic execution," in *Software Security and Reliability (SERE), 2014 Eighth International Conference on*, June 2014, pp. 98–107.

[68] C. C. Paul Dan Marinescu, "Katch: High-coverage testing of software patches," in *European Software Engineering Conference / ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 8 2013, pp. 235–245.

[69] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: Automatically generating inputs of death," in *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS '06)*. New York, NY, USA: ACM, 2006, pp. 322–335.

[70] W. Le and M. L. Soffa, "Marple: A demand-driven path-sensitive buffer overflow detector," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. SIGSOFT '08/FSE-16. New York, NY, USA: ACM, 2008, pp. 272–282. [Online]. Available: http://doi.acm.org/10.1145/1453101.1453137

[71] F. Peng, Z. Deng, X. Zhang, D. Xu, Z. Lin, and Z. Su, "X-force: Force-executing binary programs for security applications," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, ser. SEC'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 829–844. [Online]. Available: http://dl.acm.org/citation.cfm?id=2671225.2671278

[72] M. Emmi, R. Majumdar, and K. Sen, "Dynamic test input generation for database applications," in *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ser. ISSTA '07. New York, NY, USA: ACM, 2007, pp. 151–162. [Online]. Available: http://doi.acm.org/10.1145/1273463.1273484

[73] S. Artzi, A. Kiezun, J. Dolby, F. Tip, D. Dig, A. Paradkar, and M. D. Ernst, "Finding bugs in dynamic web applications," in *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ser. ISSTA '08. New York, NY, USA: ACM, 2008, pp. 261–272. [Online]. Available: http://doi.acm.org/10.1145/1390630.1390662

[74] P. Joshi, K. Sen, and M. Shlimovich, "Predictive testing: Amplifying the effectiveness of software testing," in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations

of Software Engineering, ser. ESEC-FSE '07. New York, NY, USA: ACM, 2007, pp. 561–564. [Online]. Available: http://doi.acm.org/10.1145/1287624.1287710

[75] C. C. Paul Dan Marinescu, "Make test-zesti: A symbolic execution solution for improving regression testing," in *International Conference on Software Engineering (ICSE 2012)*, 6 2012.

[76] C. Csallner and Y. Smaragdakis, "Check'n'crash: combining static checking and testing," in *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*, 2005, pp. 422–431.

[77] P. Godefroid, "Compositional dynamic test generation," in *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '07. New York, NY, USA: ACM, 2007, pp. 47–54. [Online]. Available: http://doi.acm.org/10.1145/1190216.1190226

[78] P. Godefroid, M. Y. Levin, and D. Molnar, "Automated whitebox fuzz testing," in *Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS 2008)*, 2008, pp. 151–166.

[79] C. Csallner, Y. Smaragdakis, and T. Xie, "DSD-crasher: a hybrid analysis tool for bug finding," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 17, no. 2, pp. 1–37, Apr. 2008.

[80] B. Elkarablieh, P. Godefroid, and M. Y. Levin, "Precise pointer reasoning for dynamic test generation," in *ISSTA '09: Proceedings of the eighteenth international symposium on Software testing and analysis*. New York, NY, USA: ACM, 2009, pp. 129–140.

[81] J. Burnim, N. Jalbert, C. Stergiou, and K. Sen, "Looper: Lightweight detection of infinite loops at runtime," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 161–169. [Online]. Available: http://dx.doi.org/10.1109/ASE.2009.87

[82] J. Burnim, S. Juvekar, and K. Sen, "Wise: Automated test generation for worst-case complexity," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 463–473. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2009.5070545

[83] P. Zhang, S. Elbaum, and M. B. Dwyer, "Automatic generation of load tests," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 43–52. [Online]. Available: http://dx.doi.org/10.1109/ASE.2011.6100093

[84] R.-G. Xu, P. Godefroid, and R. Majumdar, "Testing for buffer overflows with length abstraction," in *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ser. ISSTA '08. New York, NY, USA: ACM, 2008, pp. 27–38. [Online]. Available: http://doi.acm.org/10.1145/1390630.1390636

[85] B. Korel, "Automated software test data generation," *Software Engineering, IEEE Transactions on*, vol. 16, no. 8, pp. 870–879, 1990.

[86] ——, "Dynamic method for software test data generation," *Software Testing, Verification and Reliability*, vol. 2, no. 4, pp. 203–213, Dec. 1992.

[87] R. Ferguson and B. Korel, "The chaining approach for software test data generation," *ACM Transactions on Software Engineering and Methodology*, vol. 5, pp. 63–86, Jan. 1996.

[88] C. Michael, G. McGraw, and M. Schatz, "Generating software test data by evolution," *IEEE Transactions on Software Engineering*, vol. 27, no. 12, pp. 1085–1110, Dec. 2001. [Online]. Available: http://dx.doi.org/10.1109/32.988709

[89] P. Tonella, "Evolutionary testing of classes," in *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, ser. ISSTA '04. New York, NY, USA: ACM, 2004, pp. 119–128. [Online]. Available: http://doi.acm.org/10.1145/1007512.1007528

[90] J. Wegener, A. Baresel, and H. Sthamer, "Evolutionary test environment for automatic structural testing," *Information and Software Technology*, vol. 43, no. 14, pp. 841 – 854, 2001. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584901001902

[91] P. McMinn, M. Harman, D. Binkley, and P. Tonella, "The species per path approach to searchbased test data generation," in *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, ser. ISSTA '06. New York, NY, USA: ACM, 2006, pp. 13–24. [Online]. Available: http://doi.acm.org/10.1145/1146238.1146241

[92] G. Fraser and A. Zeller, "Generating parameterized unit tests," in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ser. ISSTA '11. New York, NY, USA: ACM, 2011, pp. 364–374. [Online]. Available: http://doi.acm.org/10.1145/2001420.2001464

[93] F. M. Kifetew, A. Panichella, A. De Lucia, R. Oliveto, and P. Tonella, "Orthogonal exploration of the search space in evolutionary test case generation," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ser. ISSTA 2013. New York, NY, USA: ACM, 2013, pp. 257–267. [Online]. Available: http://doi.acm.org/10.1145/2483760.2483789

[94] M. Vivanti, A. Mis, A. Gorla, and G. Fraser, "Search-based data-flow test generation," in *ISSRE'13: Proceedings of the 24th IEEE International Symposium on Software Reliability Engineering*. IEEE Press, Nov. 2013.

[95] G. Denaro, A. Margara, M. Pezze, and M. Vivanti, "Dynamic data flow testing of object oriented systems," in *Proceedings of the International Conference on Software Engineering*, ser. ICSE 2015, 2015.

[96] K. Lakhotia, M. Harman, and P. McMinn, "Handling dynamic data structures in search based testing," in *Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '08. New York, NY, USA: ACM, 2008, pp. 1759–1766. [Online]. Available: http://doi.acm.org/10.1145/1389095.1389435

[97] K. Inkumsah and T. Xie, "Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution," in *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 297–306. [Online]. Available: http://dx.doi.org/10.1109/ASE.2008.40

[98] N. Alshahwan and M. Harman, "Automated web application testing using search based software engineering," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 3–12. [Online]. Available: http://dx.doi.org/10.1109/ASE.2011.6100082

[99] M. Harman, Y. Jia, and W. B. Langdon, "Strong higher order mutation-based test data generation," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 212–222. [Online]. Available: http://doi.acm.org/10.1145/2025113.2025144

[100] A. Baars, M. Harman, Y. Hassoun, K. Lakhotia, P. McMinn, P. Tonella, and T. Vos, "Symbolic search-based testing," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 53–62. [Online]. Available: http://dx.doi.org/10.1109/ASE.2011.6100119

[101] J. Malburg and G. Fraser, "Combining search-based and constraint-based testing," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 436–439. [Online]. Available: http://dx.doi.org/10.1109/ASE.2011.6100092

[102] J. P. Galeotti, G. Fraser, and A. Arcuri, "Improving search-based test suite generation with dynamic symbolic execution," in *IEEE International Symposium on Software Reliability Engineering*, 2013.

[103] N. Tracey, J. Clark, and K. Mander, "Automated program flaw finding using simulated annealing," in *Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*, ser. ISSTA '98. New York, NY, USA: ACM, 1998, pp. 73–81. [Online]. Available: http://doi.acm.org/10.1145/271771.271792

[104] A. Baresel, H. Pohlheim, and S. Sadeghipour, "Structural and functional sequence test of dynamic and state-based software with evolutionary algorithms," in *Proceedings of the 2003 international conference on Genetic and evolutionary computation: PartII*, ser. GECCO'03. Berlin, Heidelberg: Springer-Verlag, 2003, pp. 2428–2441. [Online]. Available: http://dl.acm.org/citation.cfm?id=1756582.1756738

[105] J. Wegener and O. Bühler, "Evaluation of different fitness functions for the evolutionary testing of an autonomous parking system," in *Genetic and Evolutionary Computation – GECCO 2004*, ser. Lecture Notes in Computer Science, K. Deb, Ed. Springer Berlin Heidelberg, 2004, vol. 3103, pp. 1400–1412.

[106] O. Bühler and J. Wegener, "Evolutionary functional testing," *Comput. Oper. Res.*, vol. 35, no. 10, pp. 3144–3160, Oct. 2008. [Online]. Available: http://dx.doi.org/10.1016/j.cor.2007.01.015

[107] T. E. Colanzi, W. K. G. Assunção, S. R. Vergilio, and A. Pozo, "Integration test of classes and aspects with a multi-evolutionary and coupling-based approach," in *Proceedings of the Third International Conference on Search Based Software Engineering*, ser. SSBSE'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 188–203. [Online]. Available: http://dl.acm.org/citation.cfm?id=2042243.2042268

[108] F. Gross, G. Fraser, and A. Zeller, "Search-based system testing: High coverage, no false alarms," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ser. ISSTA 2012. New York, NY, USA: ACM, 2012, pp. 67–77. [Online]. Available: http://doi.acm.org/10.1145/2338965.2336762

[109] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," *IEEE Transactions on Software Engineering*, vol. 38, no. 2, pp. 278–292, Mar. 2012. [Online]. Available: http://dx.doi.org/10.1109/TSE.2011.93

[110] T. E. Vos, F. F. Lindlar, B. Wilmes, A. Windisch, A. I. Baars, P. M. Kruse, H. Gross, and J. Wegener, "Evolutionary functional black-box testing in an industrial setting," *Software Quality Control*, vol. 21, no. 2, pp. 259–288, Jun. 2013. [Online]. Available: http://dx.doi.org/10.1007/s11219-012-9174-y

[111] H. Kayacik, A. Zincir-Heywood, and M. Heywood, "Evolving successful stack overflow attacks for vulnerability testing," in *Computer Security Applications Conference, 21st Annual*, 2005, pp. 8 pp.–234.

[112] M. Tlili, H. Sthamer, S. Wappler, and J. Wegener, "Improving evolutionary real-time testing by seeding structural test data," in *Evolutionary Computation, 2006. CEC 2006. IEEE Congress on*, 2006, pp. 885–891.

[113] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos, "Timeaware test suite prioritization," in *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, ser. ISSTA '06. New York, NY, USA: ACM, 2006, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/1146238.1146240

[114] M. Di Penta, G. Canfora, G. Esposito, V. Mazza, and M. Bruno, "Search-based testing of service level agreements," in *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '07. New York, NY, USA: ACM, 2007, pp. 1090–1097. [Online]. Available: http://doi.acm.org/10.1145/1276958.1277174

[115] R. C. Bryce and C. J. Colbourn, "One-test-at-a-time heuristic search for interaction test suites," in *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '07. New York, NY, USA: ACM, 2007, pp. 1082–1089. [Online]. Available: http://doi.acm.org/10.1145/1276958.1277173

[116] W. Afzal, R. Torkar, and R. Feldt, "A systematic review of search-based testing for non-functional system properties," *Journal of Information and Software Technology*, vol. 51, no. 6, pp. 957–976, Jun. 2009. [Online]. Available: http://dx.doi.org/10.1016/j.infsof.2008.12.005

[117] H. Palikareva and C. Cadar, "Multi-solver support in symbolic execution," in *Proceedings of the 25th international conference on Computer Aided Verification*, ser. CAV'13. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 53–68.

[118] W. Visser, J. Geldenhuys, and M. B. Dwyer, "Green: Reducing, reusing and recycling constraints in program analysis," in *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '12. ACM, 2012, pp. 1–11.

[119] A. Aquino, F. A. Bianchi, C. Meixian, G. Denaro, and M. Pezzè, "Reusing constraint proofs in program analysis," in *Proceedings of the International Symposium on Software Testing and Analysis*, ser. ISSTA '15. ACM, 2015, pp. 305–315.

[120] X. Jia, C. Ghezzi, and S. Ying, "Enhancing reuse of constraint solutions to improve symbolic execution," in *Proceedings of the International Symposium on Software Testing and Analysis*, ser. ISSTA '15. ACM, 2015, pp. 177–187.

[121] I. Erete and A. Orso, "Optimizing constraint solving to better support symbolic execution," in *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, ser. ICSTW '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 310–315.

[122] A. Møller and M. I. Schwartzbach, "The pointer assertion logic engine," in *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, ser. PLDI '01. New York, NY, USA: ACM, 2001, pp. 221–231.

[123] P. Madhusudan and X. Qiu, "Efficient decision procedures for heaps using strand," in *Proc. of the 18th International Conference on Static Analysis*, ser. SAS'11. Springer-Verlag, 2011, pp. 43–59.

[124] Z. Rakamarić, R. Bruttomesso, A. J. Hu, and A. Cimatti, "Verifying heap-manipulating programs in an smt framework," in *Proc. of the 5th International Conference on Automated Technology for Verification and Analysis*, ser. ATVA'07. Springer-Verlag, 2007, pp. 237–252.

[125] S. McPeak and G. C. Necula, "Data structure specifications via local equality axioms," in *Proc. of the 17th International Conference on Computer Aided Verification*, ser. CAV'05. Springer-Verlag, 2005.

[126] A. Bouajjani, C. Dr?goi, C. Enea, and M. Sighireanu, "Accurate invariant checking for programs manipulating lists and arrays with infinite data," in *Automated Technology for Verification and Analysis*, ser. LNCS, S. Chakraborty and M. Mukund, Eds. Springer, 2012, pp. 167–182.

[127] P. Madhusudan, X. Qiu, and A. Stefanescu, "Recursive proofs for inductive tree data-structures," in *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, NY, USA: ACM, 2012, pp. 123–136.

[128] C. Enea, V. Saveluc, and M. Sighireanu, "Compositional invariant checking for overlaid and nested linked lists," in *Proceedings of the European Conference on Programming Languages and Systems*, ser. ESOP'13. Springer-Verlag, 2013, pp. 129–148.

[129] M. Barnett and K. R. M. Leino, "Weakest-precondition of unstructured programs," *SIGSOFT Softw. Eng. Notes*, vol. 31, no. 1, pp. 82–87, Sep. 2005. [Online]. Available: http://doi.acm.org/10.1145/1108768.1108813

[130] C. Furia and B. Meyer, "Inferring loop invariants using postconditions," in *Fields of Logic and Computation*, ser. Lecture Notes in Computer Science, A. Blass, N. Dershowitz, and W. Reisig, Eds. Springer Berlin Heidelberg, 2010, vol. 6300, pp. 277–300.

[131] J. R. Burch, K. L. McMillan, D. L. Dill, and L. Hwang, "Symbolic model checking: 10 20 states and beyond," *Logic in Computer Science, 1990. LICS'90, Proceedings., Fifth Annual IEEE Symposium on e*, pp. 428–439, 1990.

[132] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko, "Path invariants," in *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007)*, ser. PLDI '07. New York, NY, USA: ACM, 2007, pp. 300–309.

[133] M. S. Hecht, *Flow Analysis of Computer Programs*. New York, NY, USA: Elsevier Science Inc., 1977.

[134] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 13, no. 4, pp. 451–490, October 1991.

[135] M. Delahaye, B. Botella, and A. Gotlieb, "Explanation-based generalization of infeasible path," in *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation*, ser. ICST '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 215–224.

[136] M. N. Ngo and H. B. K. Tan, "Detecting large number of infeasible paths through recognizing their patterns," in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ser. ESEC-FSE '07. New York, NY, USA: ACM, 2007, pp. 215–224.

[137] ——, "Heuristics-based infeasible path detection for dynamic test data generation," *Information and Software Technology*, vol. 50, no. 7-8, pp. 641–655, 2008.

Mauro Baluda is a Post-Doctoral Research Fellow in the Secure Software Engineering group at the Fraunhofer Institute for Secure Information Technology. He received his Ph.D. degree at the Università della Svizzera Italiana, Lugano, Swizerland. His scientific interests are in software testing and analysis, software security, software verification and rigorous software engineering.

Giovanni Denaro is researcher and assistant professor of software engineering at the University of Milano-Bicocca. He received his Ph.D. degree in computer science from Politecnico di Milano. His research interests include formal methods for software verification, software testing and analysis, distributed and service-oriented systems and software metrics. He has been co-investigator in several international research and development projects in close collaboration with leading European universities and companies.

Mauro Pezzè is a professor of software engineering at the University of Milano-Bicocca and at the Università della Svizzera italiana. He is associate editor of IEEE Transactions on Software Engineering, and has served as associate editor of ACM Transactions on Software Engineering, as general chair or the ACM International Symposium on Software Testing and Analysis in 2013, program chair of the International Conference on Software Engineering in 2012 and of the ACM International Symposium on Software Testing and Analysis in 2006. He is known for his work on software testing and program analysis and, more recently on self-healing and self-adaptive software systems.