Software Quality Journal manuscript No.
(will be inserted by the editor)

Enhancing Structural Software Coverage by Incrementally Computing Branch Executability*

Mauro Baluda · Pietro Braione · Giovanni Denaro · Mauro Pezzè

Received: TODO / Accepted: TODO

Abstract Structural code coverage criteria have been studied since the early seventies, and now they are well supported by commercial and open source tools, and are commonly embedded in several advanced industrial processes. Most industrial applications still refer to simple criteria, like statement and branch coverage, and consider more complex criteria, like modified condition decision coverage, only rarely and often driven by the requirements of certification agencies. The industrial value of structural criteria is limited by the difficulty of achieving high coverage, due to both the complexity of deriving test cases that execute specific uncovered elements and the presence of many infeasible elements in the code.

In this paper, we propose a technique that both generates test cases that execute yet uncovered branches and identifies infeasible branches that can be eliminated from the computation of the branch coverage. In this way, we can increase branch coverage up to closely approximate full coverage, thus improving its industrial value. The algorithm combines symbolic analysis, abstraction refinement, and a novel technique named *coarsening*, to execute unexplored branches, identify infeasible ones, and mitigate the state space explosion problem. In the paper, we present the technique, and illustrate its effectiveness through a set of experimental results obtained with a prototype implementation.

* This paper is an extended version of Baluda et al (2010).

Mauro Baluda
Università della Svizzera Italiana, via Buffi 13, 6900 Lugano, Switzerland
E-mail: mauro.baluda@usi.ch

Pietro Braione
Università degli Studi di Milano-Bicocca, viale Sarca 336, 20126 Milano, Italy
E-mail: braione@disco.unimib.it

Giovanni Denaro
Università degli Studi di Milano-Bicocca, viale Sarca 336, 20126 Milano, Italy
E-mail: denaro@disco.unimib.it

Mauro Pezzè
Università della Svizzera Italiana, via Buffi 13, 6900 Lugano, Switzerland and
Università degli Studi di Milano-Bicocca, viale Sarca 336, 20126 Milano, Italy
E-mail: mauro.pezze@usi.ch

Keywords Structural testing · concolic execution · automatic test generation

CR Subject Classification D.2.5

1 Introduction

High code coverage has been longly advocated as a convenient way to assess test adequacy (Weyuker, 1988; Frankl and Weyuker, 1988; RTCA, Inc., 1993). Over time, researchers have defined several testing criteria based on code coverage, referring to code elements (coverage targets) at increasing levels of granularity, from program statements, to decisions, paths and data-flow associations (Pezzè and Young, 2007). Recently, code coverage is experiencing a renewed interest, thanks to the availability of both new tools that efficiently compute accurate coverage measures, and new studies that provide additional empirical evidence of a strong correlation between the code coverage and the ability to expose faults of test suites (Namin and Andrews, 2009).

Despite the encouraging experimental data and the many available criteria, a closer look at the current industrial practice indicates an adoption of code coverage limited to simple criteria, usually only statement coverage and, more rarely, branch coverage. The gap between expected effectiveness and practical usage depends on two main factors. First, generating test suites that execute many code elements is in general extremely demanding, since the number of coverage targets rapidly increases with the size of the program under test and the complexity of the criteria. Testing large programs referring to sophisticated criteria can be often impractical within the limits of a typical testing budget.

Second, covering all elements according to a coverage criterion (achieving 100% code coverage) is impossible in general, since some coverage targets can be infeasible, that is non executable under any possible condition. Typical causes of infeasible code elements are, for example, changes in the source code that affect the executability of some elements, reuse of components integrated by specialization that limits the set of executed functionality, redundant code inserted for the sake of defensive programming, and reachability dependencies between code regions. Infeasible targets impact both on the testing effort and on the achievable coverage. They divert the testing effort, since test designers may waste time trying to identify test cases that execute infeasible elements, and produce bad approximations of the set of target elements that can be statically identified, thus resulting in highly variable coverage. The amount of infeasible elements grows with the complexity of the criteria, and quickly becomes a big hurdle to the practical applicability of sophisticated criteria.

Both the problem of finding the inputs that exercise specific targets and the problem of identifying infeasible elements are undecidable in general and hard to solve in practice. As a consequence mature practical processes refer mostly to statement coverage, and use more sophisticated coverage criteria only when required by domain regulations. For example the standard DO-178B for safety-critical avionic applications requires the modified condition decision coverage (RTCA, Inc., 1993).

The problem of generating test cases that increase code coverage is being recently tackled by approaches that generate test cases using symbolic and concolic (that is, interwoven concrete and symbolic) execution (Visser et al, 2004; Godefroid et al, 2005; Sen et al, 2005). These approaches explore the executable space of a program, typically in depth-first order, and generate test cases accordingly. Since most programs have

```
1  void scan1(int* array, int size, int startAt){
2          while(startAt >= 0 && startAt < size){
3                  do_something(array, size, startAt);
4                  startAt = startAt + 1;
5          }
6  }
7
8  void scan2(int* array1, int size1, int startAt1,
9          int* array2, int size2, int startAt2){
10         while(startAt1 >= 0 && startAt1 < size1){
11                 do_something(array1, size1, startAt1);
12                 startAt1 = startAt1 + 1;
13         }
14         while(startAt2 >= 0 && startAt2 < size2){
15                 do_something(array2, size2, startAt2);
16                 startAt2 = startAt2 + 1;
17         }
18 }
19
20 inline void do_something(int *array, int size, int itemAt){
21         if(itemAt < 0 || itemAt >= size){
22                 printf("infeasible");
23                 exit(-1);
24         }
25         printf("Doing␣something␣with␣%d...␣done!", array[itemAt]);
26 }
```

Fig. 1 The source code of programs `scan1` and `scan2`

infinitely many paths, a depth-first search is in general ill-suited for the goal of covering a finite domain: It leads to a fine-grained exploration of only small portions of the program state space, easily diverges, and often identifies many test cases that increase the coverage of the program structure only marginally. Other search strategies use heuristics to select paths that lead to uncovered elements in the control-flow graph (Godefroid et al, 2008; Burnim and Sen, 2008). Heuristics can increase coverage, but do not prevent the search to be stuck in attempting to execute infeasible targets.

The sample programs in Figure 1 exemplify the case of infeasible code due defensive programming, and the limits of approaches that try to cover all code elements by means of symbolic/concolic execution. Program `scan1` implements a simple loop that scans an array starting from a specified item (parameter `startAt`) until the end of the array, and executes the same action `do_something` on the scanned elements. Program `scan2` is a variant of program `scan1`, and implements a sequence of two scanning loops on two different arrays. Procedure `do_something` is invoked by both `scan1` and `scan2`, and accesses the arrays at the specified index (parameter `itemAt`) after a safety check in the typical style of defensive programming to ensure that the index is within the array bounds. The safety check is redundant with respect to the conditions of the loops that invoke `do_something` (for instance, line 2). As a result, the code block at line 22 is infeasible in the context of both `scan1` and `scan2`.

Trying to cover the code by means of concolic (or symbolic) execution yields sub-optimal results for these sample programs. Concolic execution evaluates the program symbolically by following the path executed by a concrete test case, computes the path condition up to a selected branching point along this path, flips the last clause of this path condition by prefixing it with a negation operator, and solves the resulting pred-

```
1  #define VALVE_NOT_WORKING(v) v == 0
2  #define TOLERANCE 3
3  int valves1(int valves[], unsigned int size) {
4          int count = 0, index = size;
5          while(index != 0){
6                  if(VALVE_NOT_WORKING(valves[index])) count++;
7                  index--;
8          }
9          if(count > TOLERANCE) printf("alarm\n");
10         return count;
11 }
```

Fig. 2 The source code of program `valves1`

icate to compute new input values that, once executed, steer the execution through the not-yet-executed decision of the branching point under concern. For instance, let us feed a concolic execution of program `scan1` with a test case that calls the program with a negative value of the parameter `startAt`, thus determinining an execution that does not enter the loop. In the first iteration, based on a depth-first exploration of the program execution space, concolic execution generates a new test case that enters the loop. The new test case then covers all statements within the loop and executes `do_something`, but does not cover the infeasible block within `do_something`. In the next steps, concolic execution iterates forever through a complete depth-first exploration of the (infinitely many) program paths through the loop, and eventually terminates for resource shortage. In the case of bounded depth-first exploration, concolic execution terminates after experiencing a few failures to generate a test case that covers the infeasible block. Unbounded concolic execution might achieve even worse results on `scan2`. When the initial test case does not enter either loops, concolic execution will be stuck in the depth-first exploration of the first loop, without even attempting to cover the second one. For all above cases, concolic execution reports that the code has been only partially covered and can by no means conclude that the missed block is indeed infeasible.

Heuristic path selection strategies, for example those proposed by Burnim and Sen (2008) and by Xie et al (2009), can mitigate but not solve the problem. Such strategies define heuristic metrics that determine, at each step of the exploration, the program path with the highest chance of approaching an element not yet covered. These strategies update the metric after exploring paths that do not increase coverage, to avoid being trapped in unbounded attempts to explore paths in fruitless directions. For example, the heuristic search defined by Burnim and Sen (2008) iteratively explores paths where the condition holds true at most few times. In the case of `scan2` it quickly covers all feasible paths, and steadily reports the coverage of three out of four (75%) branches per loop, since each loop contains an infeasible branch. The complete results are discussed in Section 4. Heuristic strategies may discover feasible branches quicker than simple depth-first symbolic and concolic executions, but they cannot identify infeasible branches, and thus cannot eliminate them from the new execution attempts nor from the final coverage measure.

In our experiments, we noticed that heuristic strategies often miss feasible branches that belong to paths with low heuristic score. This is for example the case of branches that are executed only under the combination of several decisions along a path. Let

us consider for instance the program `valves1` that scans an array of integer values, tracks the count of all values equals to zero, and signals an alarm if the count is above a given threshold (Figure 2). The branch at line 9 is hardly covered by the heuristic concolic exploration proposed by Burnim and Sen (2008), because it is executed only when the condition at line 6 holds true for several iterations of the loop. Since trying to execute paths on which the condition at line 6 holds true for several but not enough iterations fails in covering the target branch many times, the search quickly exceeds the number of failed attempts that can be tolerated by the heuristic metric. Section 4 reports additional experiments with the open source testing tool CREST and KLEE, and confirms the limits of both depth-first and heuristic path selection strategies.

The approach presented in this paper combines the advantages of using concolic execution to steer the generation of new test cases towards uncovered branches, with a program analysis approach that proves the infeasibility of branches that cannot be covered. Our approach explores the control flow graph looking for paths that lead to uncovered elements, and concolically executes one of such paths to increase coverage. In this way, it avoids being trapped in infinite iterations of unbounded loops as depth-first explorations. When it does not find a test case that covers a target branch, it investigates the feasibility of the branch using an analysis based on abstraction refinement of the control flow graph and backward propagation of preconditions for executing the uncovered branches. In this way, it avoids being stuck in infinite unsuccessful attempts to find test cases that execute infeasible elements, and increases the chances to cover in next iterations those branches that require difficult combination of many decisions[1]. It finally adjusts the coverage measurements according to the identified infeasible elements, thus increasing the precision of the computed coverage. For the sample codes in Figures 1 and 2, our approach generates test suites that cover all feasible branches, and correctly identifies all infeasible ones, thus reporting a 100% branch coverage of the code. As shown in the experiments reported in Section 4, for increasingly complex variants of the programs in Figures 1 and 2, our approach steadily converges to 100% branch coverage, while concolic approaches degrade.

The approach presented in this paper leverages and extends recent results on combining static and dynamic analysis to decide reachability of program states (Gulavani et al, 2006; Beckman et al, 2010), and introduces a technique called coarsening to control the growth of the space to be explored by pruning useless details, while analyzing increasingly large programs.

The paper is organized as follows. Section 2 presents the approach, and defines the algorithm in details. Section 3 discusses the architecture of an analysis environment based on the approach. Section 4 illustrates empirical results that show the effectiveness of the approach comparing it with random testing, symbolic and concolic execution approaches. Section 5 surveys the related work. Section 6 summarizes the results of this paper.

[1] Notice that both the problem of covering all feasible elements and the problem of revealing all infeasible elements of a program are undecidable in general, and thus our approach may not terminate on some programs. When this is the case, our approach stops after a timeout and reports the elements that have been neither covered nor identified as infeasible.

2 Abstraction refinement and coarsening

This section presents in details *abstraction refinement and coarsening* (ARC), our procedure that automatically generates test suites with high structural coverage and computes precise coverage information. ARC systematically explores uncovered program elements, accounts for infeasible elements, and abstracts from useless details to improve scalability. As described in this section, ARC applies to any structural coverage criterion. Both the prototype implementation described in Section 3 and the experimental evaluation reported in Section 4 illustrate the use of ARC for branch coverage.

2.1 Rationale

ARC extends an initial test suite with new test inputs that execute uncovered code elements, and identifies infeasible code elements. Identifying infeasible elements allows ARC to drop these elements out of the coverage domain where it measures the code coverage. In this way, ARC can produce precise coverage measurements, up to 100%, also in presence of non-anticipated infeasible elements.

ARC works by integrating reachability information from concrete execution of tests with the static analysis of a finite (abstract) model of the program state space. On one hand, ARC refers to the abstract model to identify code elements that are not covered yet and are the most promising next targets, and then builds new test inputs that traverse program paths increasingly closer to these code elements along the lines of the approach of Godefroid et al (2005) and Sen et al (2005). On the other hand, ARC exploits the intuition that failing to build test inputs that cover a given element spots elements that may be unreachable along some control paths. In this case, ARC considers the possibility that these elements are infeasible, and tries to prove their infeasibility by iteratively refining the abstract model, along the lines of Ball et al (2004), Beyer et al (2007), and Gulavani et al (2006). ARC progressively refines the model by excluding unreachable paths, thus reducing the number of paths that reach code elements. A code element is infeasible when it is no more reachable in the refined model.

Reasoning over formal models of complex software systems does not scale due to space explosion problems that are exacerbated when targeting many elements as in the case of code coverage. Compared to previous techniques, ARC stands out specifically for its approach to mitigate the state space explosion problem: It introduces the novel idea of *coarsening* into the basic abstraction refinement loop. Coarsening is the process of partially re-aggregating the abstract states generated by refinement as the analysis of the program progresses. Coarsening elaborates on the observation that every refinement step aims to decide about the reachability of a code element. When ARC meets a specific goal by either covering the target element or proving its infeasibility, ARC drops the refinements generated throughout the decision process involved with this element, and thus reduces the number of states of the abstract model. By means of coarsening ARC eliminates the states produced during the refinement process as soon as they become useless for the analysis, and thus updates the reference abstract model, differently from previous approaches, where models grow monotonically as refinement progresses.

```
1  ARC(P, I, χ):
2    M := M_χ
3    T := T_χ
4    U := {}
5    split_for[nodes(M)] := {}
6    loop:
7      C := {n ∈ nodes(M) | n covered by run(P, I)}
8      coarsen(M, C, split_for)
9      T := T - (C ∪ U)
10     if T = {}:
11       return ⟨I,U⟩
12     e := choose_frontier(M, T, C, I)
13     i′ := try_generate(P, I, e)
14     if i′ = ϵ:
15       RP := refine(M, e)
16       if RP = false:
17         N := { n ∈ nodes(M) | no root(M) --stmt_1--> n_1 ... --stmt_n--> n_n ∈ paths(M) }
18         coarsen(M, N, split_for)
19         nodes(M) := nodes(M) - N
20         edges(M) := edges(M) -
21                     { n_0 --stmt--> n_1 ∈ edges(M) | n_0 ∈ N ∨ n_1 ∈ N}
22         U := U ∪ (T - nodes(M))
23       else:
24         split_for[n_post] := split_for[n_post] ∪ {⟨n_pre, RP⟩}
25                              where e = n_pre --stmt--> n_post
26     else:
27       I := I ∪ {i′}
28   forever
```

Fig. 3 The ARC algorithm

2.2 ARC

The ARC algorithm takes a program P, a nonempty set I of program inputs (the initial test suite) and a structural coverage criterion χ as inputs, and returns both a test suite that extends I for P and a set U of unreachable targets in P. ARC deals with imperative, sequential and deterministic programs composed of single procedures, and analyzes multi-procedural programs without recursive calls via inlining. The input programs are written in a procedural language, like Java or C, with assignment, sequencing, if-conditionals and while-loops statements, and are represented as control flow graphs that contain a node for each program *location* that represents a value of the program counter, and an edge for each statement. Nodes have at most two outgoing edges. Edges are labelled with assignment statements `X := expr`. Pairs of edges exiting the same node are also labelled with complementary test conditions, `cond`? and ¬`cond`?. We assume, without loss of generality, that program control flow graphs have exactly one entry node, which is connected to any other node in the graph by at least a path.

The ARC algorithm outlined in Figure 3 works on a model M, a set of targets T and a set of unreachable elements U. The model M is derived from the control flow graph of the program under analysis (line 2 in Figure 3). The derivation of the model from the control flow graph is discussed in details later in this section. in a nutshell, the model defines the elements to be covered as abstract states. The set T of targets represents

the code elements not yet proved to be either feasible or infeasible, and is initialized at line 3 in Figure 3 with all the targets identified by the coverage criterion. ARC records the set `U` of unreachable code elements discovered during the analysis. The set `U` is initialized to the empty set (line 4 in Figure 3). ARC works as follows:

1. (lines 7–11) ARC executes the test suite, and coarsens the model to eliminate all the refinements associated with the nodes covered by the tests.
2. (line 12) ARC identifies a *frontier*, that is an edge of the model that connects a covered node $\mathtt{n}_{pre}$ to an uncovered node $\mathtt{n}_{post}$. If there exists no frontier, then the program is completely covered according to the input criterion, and ARC terminates, otherwise ARC continues with the next step.
3. (line 13) ARC tries to generate a test input that covers $\mathtt{n}_{post}$ by extending some of the covered paths that reach $\mathtt{n}_{pre}$.
4. (line 27) If it can generate a proper test input, ARC adds it to the test suite.
5. (lines 15–25) Otherwise, it *conservatively* refines the model between $\mathtt{n}_{pre}$ and $\mathtt{n}_{post}$. After refining the model, it removes all the unreachable nodes, coarsens the model by dropping all the refinements associated with the removed nodes, and updates the set of infeasible elements according to the removed nodes.
6. It repeats all the steps 1–5.

As discussed above, ARC alternates generation of test inputs with model refinement, progressing with either one of these activities. Thus at each iteration, ARC either generates a new test that covers a node that was not yet covered, or refines the model by eliminating a path identified as unreachable. Since the problem of covering all feasible elements of a program is undecidable in general, ARC may not terminate on some inputs.

The rest of this section discusses ARC in details.

Data structures and initialization ARC operates on a model `M` (line 2), and keeps a set of targets `T` that represents the code elements not yet proved to be either feasible or infeasible (line 3). ARC also records the set `U` of unreachable code elements discovered during the analysis (line 4). We discuss `split_for` (line 5) below, when presenting coarsening.

`M` is a rooted directed graph with labelled nodes and edges. Each node `n` corresponds to a program location, and is annotated with a predicate over the program variables. Edges are labelled with (blocks of) program statements. A node represents a *region* of the concrete state space, i.e., a set of concrete states. The predicates associated with the nodes identify the subsets of concrete states represented by the nodes. We say that a concrete state *covers* a node when it satisfies the predicate associated with the node, and thus belongs to the region represented by the node. An edge $\mathtt{n} \xrightarrow{\mathtt{stmt}} \mathtt{n}'$ indicates that the execution of `stmt` from a state that covers `n` may lead only to states that cover $\mathtt{n}'$.

ARC derives the initial model $\mathtt{M}_\chi$ and the initial set of targets $\mathtt{T}_\chi$ from the control flow graph of `P` according to the coverage criterion χ to be satisfied, as follows. ARC instruments `P` to ensure that every code element for χ is associated to a program location such that the code element is covered for the criterion when the location is covered in the model. This reduces the problem of deciding the feasibility of a set of structural code elements, to the problem of deciding the reachability of a set of program locations, and allows ARC to initialize $\mathtt{T}_\chi$ to the corresponding set of nodes

of the model. Similarly, `U` is a set of nodes of the model. The model may vary to take into account the needs of specific coverage criteria. For example, when dealing with branch coverage, ARC adds a `skip` statement to each branch that does not contain a statement. Slightly more complex instrumentations may be required to allow ARC to operate with other control-flow-based criteria. Finally, ARC annotates each node of $\mathtt{M}_\chi$ with a predicate satisfied by all the states at the corresponding program location.

We refer to the ARC data structures introduced above with the following notation:

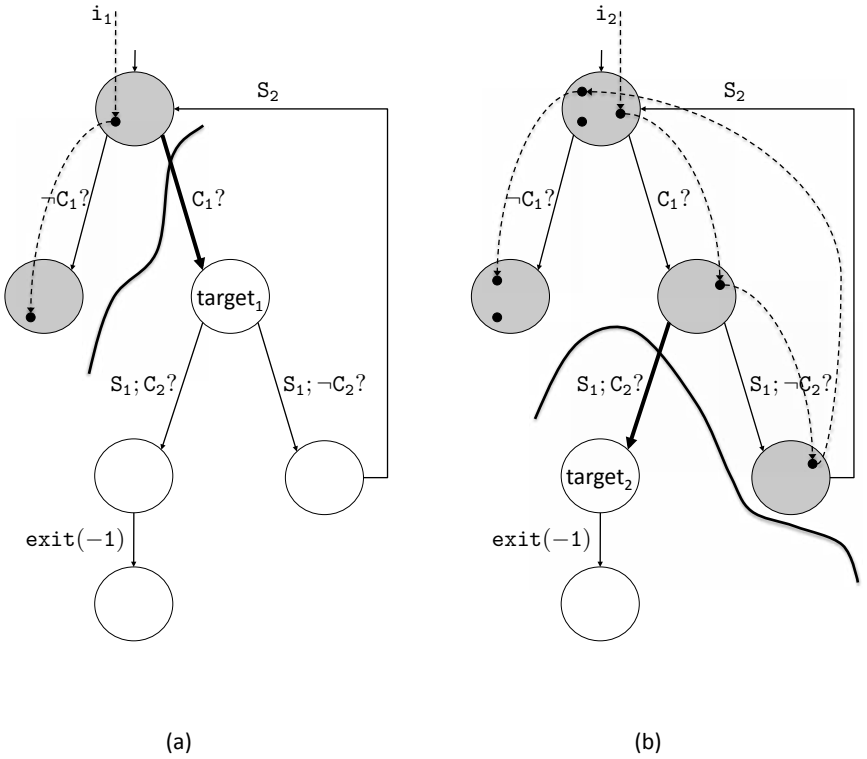
- `nodes(M)` is the set of all the nodes `n` of `M`;
- `root(M)` is the entry node of `M`;
- `edges(M)` is the set of all the edges `e` of `M`, $\mathtt{e} = \mathtt{n}_0 \xrightarrow{\mathtt{stmt}_1} \mathtt{n}_1$;
- `paths(M)` is the set of all the (finite) paths π between two nodes in `M`, $\pi = \mathtt{n}_0 \xrightarrow{\mathtt{stmt}_1} \mathtt{n}_1 \ldots \xrightarrow{\mathtt{stmt}_n} \mathtt{n}_n$;
- `predicate(n)` is the predicate associated to the node `n`.

Execution of tests and target update ARC starts each iteration by executing the test suite `I`, and computes the set `C` of the nodes covered by at least one test (line 7). Then, it simplifies the model by invoking function `coarsen` described below (line 8), and updates the set of target elements by removing all the nodes covered and identified as unreachable (line 9). If ARC exhausts the set of targets, it terminates (lines 10–11).

ARC executes the current test suite `I` by invoking function `run(P, I)` that returns the set of all the computations $c = \mathtt{s}_0 \xrightarrow{\mathtt{stmt}_1} \mathtt{s}_1 \ldots \xrightarrow{\mathtt{stmt}_n} \mathtt{s}_n$ produced by executing the tests in `I` (since `P` is sequential and deterministic, `run(P, I)` returns one computation for each test input).

Test generation After having executed the tests and updated the set of targets, ARC tries to generate a new test input to cover at least one uncovered node on a path to a target element (lines 12–13). First, ARC invokes function `choose_frontier` that returns a frontier edge of the model (line 12), that is an edge from a covered node $\mathtt{n}_{pre}$ to the first node $\mathtt{n}_{post}$ of an uncovered path to a target. Since we assume that every node of the model is reachable from the entry node, there is always a frontier for each target node. Next, ARC invokes the function `try_generate` that tries to generate a new test that traverses the selected frontier and covers $\mathtt{n}_{post}$ (line 13). Upon success, the execution of the new test will move the frontier at least one step forward towards an uncovered target.

Figure 4 illustrates a typical iteration of ARC on the example function `scan1` whose code is given in Figure 1, considering branch coverage as target criterion. The initial model of function `scan1` is a control flow graph with six nodes: the entry point (top node), the exit point reached by the explicit call to `exit(-1)` (bottom node) and the four static branches in the code. The edges are labelled with the sequences of statements that correspond to the sequences of assignments and conditional checks in the original C code. Let us consider an initial test suite that includes only one test input $\mathtt{i}_1$ (empty array, start at item 0). This test covers only the entry and the exit nodes that are shown with a gray background in Figure 4 (a). The black dots in the nodes represent the concrete states executed by the test suite. In this case, the frontier includes only one edge in bold in the figure, and is represented as a curved line in bold that separates the set of nodes covered by some test from the set of nodes not yet covered. After having executed the initial test suite and computed the frontier,



Program conditionals:
$\texttt{C}_1 \stackrel{\text{def}}{=} \texttt{startAt} >= 0 \;\&\&\; \texttt{startAt} < \texttt{size}$
$\texttt{C}_2 \stackrel{\text{def}}{=} \texttt{itemAt} < 0 \;||\; \texttt{itemAt} >= \texttt{size}$
Program statements:
$\texttt{S}_1 \stackrel{\text{def}}{=} \texttt{itemAt} = \texttt{startAt}$
$\texttt{S}_2 \stackrel{\text{def}}{=} \texttt{startAt} = \texttt{startAt} + 1$
Test inputs:
$\texttt{i}_1 \stackrel{\text{def}}{=} \langle [], 0, 0 \rangle$
$\texttt{i}_2 \stackrel{\text{def}}{=} \langle [0], 1, 0 \rangle$

Fig. 4 A possible execution of ARC on `scan1`: (a) first iteration, (b) second iteration

ARC tries to cross it by symbolically executing the code along a test execution up to a node beyond the frontier. In this case, there is only one node immediately beyond the frontier (node $\texttt{target}_1$ in Figure 4 (a)). By symbolically executing the one-edge path from the entry node to node $\texttt{target}_1$, ARC generates a new test input $\texttt{i}_2$ (array with one item with value 0, start at item 0) that reaches the node and extends the frontier as shown in Figure 4 (b). ARC then tries to further extend the frontier to a new target $\texttt{target}_2$ by symbolically executing the only feasible path to its pre-frontier, i.e., the path $\texttt{root}(\texttt{M}) \xrightarrow{\texttt{C}_1?} \texttt{target}_1 \xrightarrow{\texttt{S}_1;\texttt{C}_2?} \texttt{target}_2$. In this case, ARC does not find a test that reaches $\texttt{target}_2$.

Figure 5 shows the algorithms `choose_frontier` and `try_generate`. The algorithm `choose_frontier` looks for a path π in the model from a covered node $\texttt{n}_{pre}$ to an

```
1  choose_frontier(M, T, C, I):
2    π := some π ∈ paths(M) s.t.
3          π = n_pre --stmt--> n_post(≝ n_0) --stmt_1--> n_1 ... --stmt_n--> n_n ∧
4          n_pre ∈ C ∧ n_k ∉ C for all 0 ≤ k ≤ n ∧ n_n ∈ T
5    return e = n_pre --stmt--> n_post
6
7  try_generate(P, I, e = n_pre --stmt--> n_post):
8    i := some i ∈ I s.t. run(P, i)|_n ends in n_pre for some n
9    π := the π ∈ paths(M) s.t. run(P, i)|_n covers π
10   π' := π ⊕ e
11   ⟨σ,PC⟩ := run_symbolic_lightweight(P, π')
12   TC := PC ∧ eval_symbolic(predicate(n_post), σ)
13   if exists i' s.t. i' ⊨TC:
14     return i'
15   else:
16     return ϵ
```

LEGEND: Given a computation c, $c|_n$ indicates the subcomputation that involves the first n steps of c.

Fig. 5 Test input generation in ARC

uncovered target $\mathtt{n}_n$, such that all the successors of $\mathtt{n}_{pre}$ along π are uncovered (lines 3–4), and returns the first edge in π (line 5). When there exist several candidate paths, the choice can be arbitrary.

The algorithm `try_generate` chooses a test input `i` that reaches $\mathtt{n}_{pre}$ (line 8), and appends the frontier `e` to the concrete path corresponding to the execution of `i` up to $\mathtt{n}_{pre}$ (lines 9–10), to build a path π' to $\mathtt{n}_{post}$. Then, it executes π' symbolically to build a predicate `TC` whose solutions, if exist, are test inputs that reach $\mathtt{n}_{post}$ along π' (line 11.) Function `run_symbolic_lightweight` is similar to the ones described in Godefroid et al (2005), Sen et al (2005), and Beckman et al (2010). It returns both the final symbolic state σ and a *path constraint* `PC`, a predicate that is satisfied by all and only the test inputs whose computations reach the *location* of $\mathtt{n}_{post}$ along π'. For more details about lightweight symbolic execution, the reader may refer to Beckman et al (2010). The algorithm uses the path condition `PC` returned by `run_symbolic_lightweight` to build the predicate `TC` that characterizes the inputs that reach $\mathtt{n}_{post}$ from the path π'. The predicate `TC` is the logical-and of `PC` and the region predicate of $\mathtt{n}_{post}$, evaluated on the symbolic state σ (line 12). This corresponds to selecting the test inputs that satisfy both `PC` and the region predicate of $\mathtt{n}_{post}$ after the execution of the statement at the frontier edge. The algorithm `try_generate` returns either a solution $\mathtt{i}'$ of `TC` (line 14) or a failure value ϵ (line 16).

Refinement If the algorithm `try_generate` fails to generate a test input, then the uncovered node ($\mathtt{n}_{post}$) is unreachable along the control flow path π'. In this case, ARC conservatively refines the model by invoking the function `refine`, whose code is reported in Figure 6. The function `refine` computes a predicate `RP` that identifies a subregion of $\mathtt{n}_{pre}$ that cannot reach $\mathtt{n}_{post}$ (line 6), and updates the model so it represents it. ARC computes `RP` as the weakest precondition of $\neg$`predicate`($\mathtt{n}_{post}$) through the frontier statement. According to this definition, `RP` is the *largest* subregion of $\mathtt{n}_{pre}$ that cannot reach $\mathtt{n}_{post}$. With this approach, the refinement predicate can be computed syntactically, without invoking a decision procedure.

```
1  refine(M, e = n_pre --stmt--> n_post):
2    if n_pre = root(M):
3      edges(M) := edges(M) - e
4      return false
5    else:
6      RP := wp(stmt, ¬predicate(n_post))
7      nodes(M) := nodes(M) + {n'_pre}
8      predicate(n_pre) := predicate(n_pre) ∧ ¬RP
9      predicate(n'_pre) := predicate(n_pre) ∧ RP
10     for all n_pre --stmt'--> n'_post s.t. n'_post ≠ n_post:
11       edges(M) := edges(M) ∪ {n'_pre --stmt'--> n'_post}
12     return RP
```

Fig. 6 Model refinement in ARC

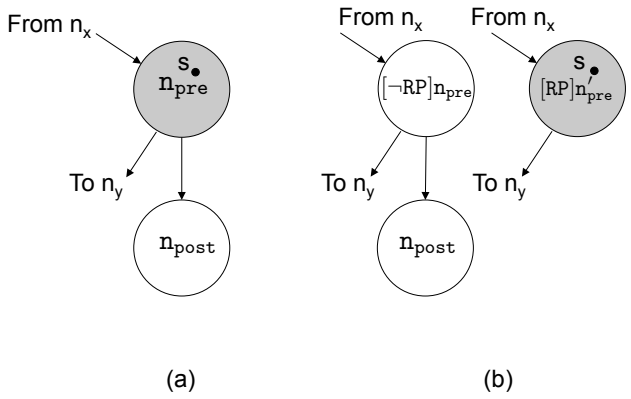
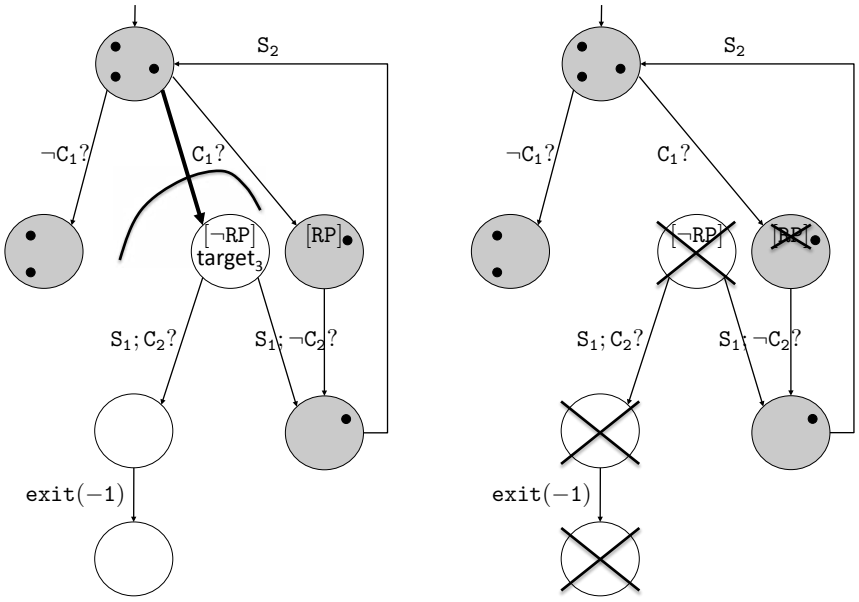


Fig. 7 Refinement of an infeasible transition: (a) before and (b) after refinement

After having computed the refinement predicate RP, ARC splits $\mathtt{n}_{pre}$ along the computed predicate, and removes the infeasible frontier edge from the model (lines 7–11). Figure 7 illustrates visually the refining procedure: ARC adds a clone of node $\mathtt{n}_{pre}$ ($\mathtt{n}'_{pre}$) to the model, adds the clauses ¬RP and RP to the region predicates of $\mathtt{n}_{pre}$ and $\mathtt{n}'_{pre}$, respectively, and removes the edge from node $\mathtt{n}'_{pre}$ to node $\mathtt{n}_{post}$, while preserving all other edges. Figure 7 (b) shows the splits of $\mathtt{n}_{pre}$ as $[\neg\mathtt{RP}]\mathtt{n}_{pre}$ and $[\mathtt{RP}]\mathtt{n}'_{pre}$. The coverages of the nodes of the models after the splitting can be computed easily: Since RP is the largest subregion of $\mathtt{n}_{pre}$ that cannot reach node $\mathtt{n}_{post}$, and ARC assumes deterministic programs, all the tests that do not reach $\mathtt{n}_{post}$—i.e., all the tests—reach $[\mathtt{RP}]\mathtt{n}_{pre}$, and do no reach $[\neg\mathtt{RP}]\mathtt{n}_{pre}$.

The refinement step sets the frontier one step backwards, since it reduces the reachability of $\mathtt{n}_{post}$ to the reachability of $[\neg\mathtt{RP}]\mathtt{n}_{pre}$. If successive refinements push the frontier back to the entry node of the model, the function `refine` can safely conclude that the whole frontier is infeasible, and can thus remove the corresponding edge from the model (lines 3–4). The function `refine` returns either the refinement predicate or $false$ if the whole frontier is infeasible.



Program conditionals:
$\mathtt{C}_1 \stackrel{\text{def}}{=} \mathtt{startAt} >= 0 \;\&\&\; \mathtt{startAt} < \mathtt{size}$
$\mathtt{C}_2 \stackrel{\text{def}}{=} \mathtt{itemAt} < 0 \;||\; \mathtt{itemAt} >= \mathtt{size}$
Program statements:
$\mathtt{S}_1 \stackrel{\text{def}}{=} \mathtt{itemAt} = \mathtt{startAt}$
$\mathtt{S}_2 \stackrel{\text{def}}{=} \mathtt{startAt} = \mathtt{startAt} + 1$
Refinement predicates:
$\mathtt{RP} \stackrel{\text{def}}{=} \mathtt{startAt} >= 0 \;\&\&\; \mathtt{startAt} < \mathtt{size}$

Fig. 8 A possible execution of ARC on `scan1`: (a) third iteration, (b) fourth iteration

Pruning and detecting infeasible targets Whenever a frontier is removed from the entry node, ARC prunes the portion of the model that is unreachable from the entry node. The pruning procedure is shown in the algorithm in Figure 3 at page 7: ARC detects the unreachable portion (line 17), coarsens the model (line 18), eliminates the unreachable portion from the model (lines 19–21), and marks as unreachable all the targets that do not exist anymore in the refined model (lines 22).

Figure 8 illustrates the refining process referring to the example `scan1` presented in Figure 1 at page 3, and discussed in Figure 4. Figure 8 (a) illustrates the results of the third iteration of ARC, which refines the model shown in Figure 4 (b). ARC splits the pre-frontier node in two nodes, one labelled with predicate `RP` that may not reach $\mathtt{target}_2$, and one labelled with predicate ¬`RP` that may reach $\mathtt{target}_2$. The node

```
1  coarsen(M, N, split_for):
2    for all n_post ∈ N:
3      for all ⟨n_pre, RP⟩ ∈ split_for[n_post]:
4        for all n'_pre ∈ companions(M, n_pre):
5          remove RP (or ¬RP) from predicate(n'_pre)
6          stmt := the statement between loc(n_pre) and loc(n_post)
7          edges(M) := edges(M) ∪ {n'_pre --stmt--> n_post}
8        for all n'_pre ∈ companions(M, n_pre):
9          if exists n''_pre ≠ n'_pre ∈ companions(M, n_pre) s.t.
10         predicate(n''_pre) ⟹ predicate(n'_pre) is logically valid:
11           nodes(M) := nodes(M) - {n'_pre}
12           edges(M) := edges(M) -
13                       { n_0 --stmt--> n_1 ∈ edges(M) | n_0 = n'_pre ∨ n_1 = n'_pre }
14     split_for[n_post] := {}
```

Fig. 9 Model coarsening in ARC

labeled with predicate P is not covered, and becomes the new target for test input generation. At the fourth and last iteration (Figure 8 (b)), ARC discovers that $\texttt{target}_3$ cannot be reached from the entry point of the program, and thus refines the model by simply eliminating the edge from the entry to this node. Then, it prunes the model by deleting the three unreachable nodes that result from the refinement steps. ARC has therefore proved that the generated tests cover all the reachable branches of the program, and terminates the analysis.

Coarsening The core contribution of ARC is the coarsening step described by the procedure `coarsen` in Figure 9. In a nutshell, ARC coarsens the model after either covering a node or identifying a node as unreachable, since in both these cases the refinements needed to decide the reachability of the node are not necessary anymore. To assist coarsening, ARC tracks the associations between the nodes and the refinements required to investigate their reachability in a map `split_for`. When function `refine` splits a pre-frontier node $\texttt{n}_{pre}$ according to a predicate RP, ARC updates the map `split_for` by adding the pair $\langle \texttt{n}_{pre}, \texttt{RP} \rangle$ to the set of pairs associated to the post-frontier node $\texttt{n}_{post}$. In a nutshell, the map `split_for` records the refinement step by tracking that node $\texttt{n}_{pre}$ has been split according to the predicate RP in order to investigate the reachability of $\texttt{n}_{post}$.

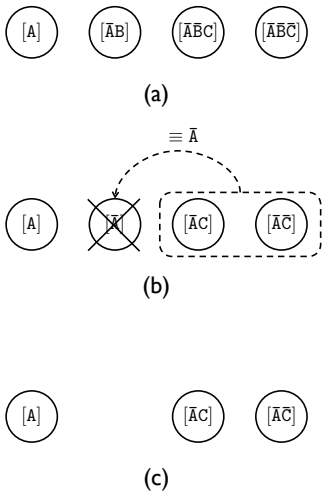
We present the coarsening step referring to the following notation:

- $\texttt{loc}(\texttt{n})$ is the program location, i.e., the value of the program counter, that corresponds to the node n,
- $\texttt{companions}(\texttt{M}, \texttt{n})$ is the set of nodes of the model M that correspond to the same program location of node n:

$$\texttt{companions}(\texttt{M}, \texttt{n}) \stackrel{\text{def}}{=} \{\texttt{n}' \in \texttt{nodes}(\texttt{M}) \mid \texttt{loc}(\texttt{n}') = \texttt{loc}(\texttt{n})\}.$$

We refer to this set as *companion set* of n.

The procedure `coarsen` (Figure 9) modifies an input model M and a corresponding map `split_for` to revert the refinements originated from a set of nodes N, as follows. For each node $\texttt{n}_{post}$ that belongs to N and is recorded as a frontier node in `split_for`,



LEGEND: $\bar{\mathtt{X}}$ stands for $\neg\mathtt{X}$, $\mathtt{XY}$ stands for $\mathtt{X} \wedge \mathtt{Y}$

Fig. 10 Example of coarsening: (a) four companion nodes, (b) after dropping the `B` refinement predicate, (c) after removing the redundant node

`coarsen` gets the originating refinements $\langle \mathtt{n}_{pre}, \mathtt{RP} \rangle$ from $\mathtt{split\_for}[\mathtt{n}_{\mathtt{post}}]$, and identifies the companion set of each node $\mathtt{n}_{pre}$ (line 4). The companion set collects all the nodes that are obtained by splitting a common ancestor node in the initial model, and thus correspond to the same program location. Then, the procedure simplifies the predicates of all the nodes in $\mathtt{companions}(\mathtt{M}, \mathtt{n}_{\mathtt{pre}})$ by removing the pairs of complementary predicates `RP` or $\neg\mathtt{RP}$ from the predicates associated to the nodes (line 5). Finally, it connects all these nodes to the node $\mathtt{n}_{post}$, thus restoring the edges removed by the previous refinement step (lines 6–7), and conservatively prunes the model by removing all redundant nodes in each companion set (lines 8–13).

Redundant nodes may arise during the coarsening process because eliminating parts of the predicates of the nodes may result in nodes that no longer represent different subsets of concrete states. ARC refines the model by partitioning the regions represented by the nodes, thus ensuring that the nodes regions do not overlap. Coarsening may result in a model where the region of a node $\mathtt{n}'_{pre}$ is completely covered by the regions of some of its companion nodes. When this happens (lines 9–10), $\mathtt{n}'_{pre}$ is redundant, and can be eliminated from the model, because its companions already represent that state space region. Figure 10 exemplifies this situation. Figure 10 (a) depicts four companion nodes, obtained by three consecutive refinements with predicates `A`, `B` and `C`, respectively. By removing the refinement predicate `B`, coarsening transforms the predicates of the nodes as shown in Figure 10 (b). The two rightmost companion nodes cover the same region $\neg\mathtt{A}$ that is also covered by the second node from the left. This node is thus redundant since the two rightmost companion nodes fully describe the region $\neg\mathtt{A}$ and with better precision. Coarsening eliminates the redundant node as shown in Figure 10 (c). More formally, ARC checks whether a node $\mathtt{n}'_{pre}$ is redundant by checking whether its predicate is logically implied by the predicate of at least one of its companions, $\mathtt{n}''_{pre}$ (lines 9–10). ARC checks the logical validity of $\mathtt{predicate}(\mathtt{n}''_{\mathtt{pre}}) \implies \mathtt{predicate}(\mathtt{n}'_{\mathtt{pre}})$ efficiently by syntactically comparing the clauses that compose the refinement predicates of the states.

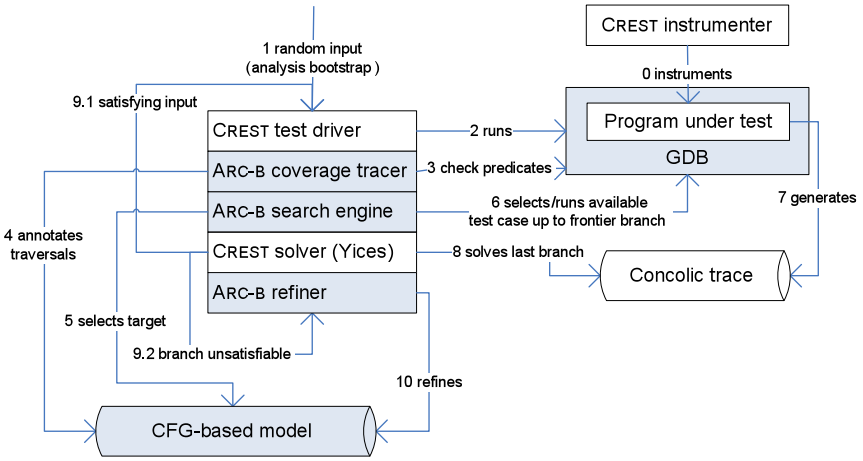


Fig. 11 Logical modules and workflow of the ARC-B tool

Coarsening eliminates useless predicates from nodes, and useless nodes from the model. In this way, it reduces both the amount of memory needed to store the model and the size and complexity of the predicates, thus increasing the scalability of the approach. Our hypothesis is that the additional computational effort introduced by the coarsening computation, and by the re-computation of some refinements that may be lost by coarsening, is counterbalanced by the reduced solver time because of shorter predicates.

3 An architecture for ARC

In this section we describe the architecture of the ARC-B tool (ARC for Branch coverage testing) that implements ARC for branch coverage, and works for programs written in C. ARC-B is built on top of CREST[2], an automatic test case generator for C, based on concolic execution. CREST relies on CIL[3] for the instrumentation and static analysis of C code, and on the YICES[4] SMT solver.

Figure 11 shows the logical modules and the basic workflow of ARC-B, and illustrates how it extends the functionality of CREST. White rectangles indicate the modules of CREST reused in ARC-B, while grey rectangles indicate the new modules of ARC-B. Arrows indicate both the computation steps, numbered from 0 to 10 according to the execution order, and the control and data dependencies between the modules. The distinction between control and data dependencies is clear from the context.

ARC-B first instruments the program under test with the CREST instrumenter to enable concolic execution (step 0 in Figure 11), and then generates test cases and analyzes the feasibility of code elements, by exploiting and refining a CFG-based model of the program as discussed in the previous section. ARC-B extracts the initial model

[2] `http://code.google.com/p/crest/`
[3] `http://sourceforge.net/projects/cil`
[4] `http://yices.csl.sri.com`

from the static control flow graph, and records both the executed test cases and the nodes of the model traversed by each test case.

Arc-B uses Crest to generate test cases and analyzes code elements for feasibility as follows. The Crest test driver executes the instrumented program with some random test inputs that represent the initial test suite (steps 1 and 2 in Figure 11). We adapted the original Crest test driver to run the program through the Gdb[5] debugger, in order to enable the Arc-B tool to dynamically intercept the execution of each statement. The Arc-B coverage tracer inspects the executed branches by querying Gdb on the validity of the model predicates at the executed branches (step 3), and annotates the branches traversed by each test run (step 4).

After each test run, the Arc-B search engine selects the next target as a not-yet-executed branch reachable on the model from the program entry point (step 5). Then, the Arc-B search engine chooses a test case that executes the program up to a frontier branch on a path towards the target, and executes this test case up to the frontier (step 6).

As in Crest, running the instrumented program generates a concolic trace up to the frontier branch (step 7). The Arc-B solver tries to satisfy the path condition (step 8) to generate a new test case that traverses the frontier towards the new target element. If the Arc-B solver succeeds in generating a test case that covers the new target element, it passes the satisfying input values to the Crest test driver (step 9.1), and the analysis iterates from step 2. Otherwise, Arc-B passes the unsatisfiable frontier to the Arc-B refiner (step 9.2) that refines the model as illustrated in Section 2 to prune the infeasible transitions from the model (step 10) before iterating step 5.

The Arc-B refiner does not produce false-positives upon recognizing infeasible branches, since the refinement process is based on precise weakest pre-conditions. However, Arc-B may not cover all target elements, if the input values that result from solving the path conditions at step 9.1 do not execute the corresponding frontiers. This problem occurs when the concolic execution at step 7 computes approximated path conditions, as Crest drops any non-linear symbolic sub-expression to keep the solver queries within theory of linear calculus. Where this problem occurs, Arc-B is unable to classify as either feasible or infeasible the frontier transitions, and consequently cannot decide the feasibility of any unreached branch that depends on these transitions, unless these branches are executed during the prosecution of the analysis.

4 Experimental results

We validated the technique proposed in this paper using the Arc-B prototype to estimate the ability of both generating test suites that cover feasible branches and identifying infeasible branches. Below, we report empirical data from two sets of experiments. First, we discuss the results obtained while experimenting with a sample of artificial programs, built as variants of the codes of Figures 1 and 2 that we have discussed earlier in this paper. The goal of these initial experiments was to evaluate the ability of the arc approach to solve problems that cannot be addressed well by concolic and symbolic tools, and to evaluate the performance of Arc-B for increasingly complex instances of these problems. Second, we report the results obtained while experimenting with a sample of third-party programs that contains feasible and infeasible

[5] `http://www.gnu.org/software/gdb`

		Arc-B					CrestDFS		CrestCFG		Klee	
subject	**br**	**it**	**tc**	**cbr**	**ibr**	**cov**	**cbr**	**cov**	**cbr**	**cov**	**cbr**	**cov**
scan1	8	11	3	6	2	100%	6	75%	6	75%	6	75%
scan2	16	23	5	12	4	100%	9	56%	12	75%	8	50%
scan5	40	59	11	30	10	100%	18	45%	30	75%	14	35%
scan10	80	119	21	60	20	100%	33	41%	60	75%	24	30%
scan20	160	239	41	120	40	100%	63	39%	120	75%	44	28%
scan50	400	599	101	300	100	100%	153	38%	300	75%	104	26%
scan100	800	1203	200	600	200	100%	303	37%	600	75%	204	26%
valves1	6	29	4	6	0	100%	5	83%	5	83%	5	83%
valves_rep2	12	65	7	12	0	100%	7	58%	10	83%	8	67%
valves_rep5	30	161	16	30	0	100%	13	43%	25	83%	17	57%
valves_rep10	60	316	31	60	0	100%	23	38%	50	83%	32	53%
valves_rep20	120	633	61	120	0	100%	43	36%	100	83%	62	52%
valves_rep50	300	1594	151	300	0	100%	103	34%	250	83%	152	51%
valves_rep100	600	10000	300	599	0	99%	203	34%	500	83%	302	50%
valves_nest2	12	59	7	12	0	100%	5	42%	5	42%	5	42%
valves_nest5	30	149	16	30	0	100%	5	17%	5	17%	5	17%
valves_nest10	60	299	31	60	0	100%	5	8%	5	8%	5	8%
valves_nest20	120	599	61	120	0	100%	5	4%	5	4%	5	4%
valves_nest50	300	1499	151	300	0	100%	5	2%	5	2%	5	2%
valves_nest100	600	10000	217	431	0	71%	5	1%	5	1%	5	1%

br: number of branches computed statically
it: number of iterations of Arc-B
tc: number of generated test cases
cbr: number of covered branches
ibr: number of identified infeasible branches
cov: reported coverage [as percentage], i.e.,
cbr / (br - ibr) , in the case of Arc-B
cbr / br , in the case of CrestDFS, CrestCFG and Klee

Table 1 Results of Arc-B, CrestDFS and CrestCFG on variants of programs `scan1` and `valves1`

branches. Throughout the experiments, we also compare the results of Arc-B with the ones obtained with a representative set of concolic, symbolic and random testing tools.

4.1 Experiments on artificial programs

Table 1 reports the results of analyzing increasingly complex instances of the programs discussed in the introduction of this paper. The labels `scan`$\langle i \rangle$ and `valves_rep`$\langle i \rangle$ denote programs that implement a sequence of i replicas of program `scan1` (Figure 1) and `valves1` (Figures 2), respectively. Programs `valves_nest`$\langle i \rangle$ are similar to `valves_rep`$\langle i \rangle$ with the difference that all code replicas are nested within the body of the last `if`, that is, within the hard-to-cover branch. Column *br* of Table 1 reports the number of static branches in each subject program.

We generated test cases for the subject programs with Arc-B, Klee and Crest. Klee implements the traditional approach based on static, depth-first symbolic execution (Cadar et al, 2008), while Crest exploits concolic execution and can be further configured to use either depth-first search (CrestDFS) or control-flow graph guided heuristic path exploration (CrestCFG) (Burnim and Sen, 2008). We ran the tools to generate test cases for each subject program with the goal of maximizing branch coverage, starting from a single input test case.

We report the output of each tool run as the number of covered branches (column *cbr*), the coverage measured based on the results of each tool (column *cov*), and, only for ARC-B, the number of branches identified as infeasible (column *ibr*). ARC-B generates test cases that cover all feasible branches in all the experiments except for `valves_rep100` and `valves_nest100` where it achieves a coverage of 99% and 71%, respectively. Since ARC-B calls the solver once per iteration, the number of iterations reported in column *it* gives a clear indication of the complexity of the analysis. When ARC-B did not terminate, we halted the analysis after 10,000 iterations. We report the total number of generated test cases in column *tc* for ARC-B. CREST and KLEE invoke the solver once per iteration, and never terminate when applied to the subject programs (recall from the introduction that unbounded depth-first path unrolling diverges for programs that contain loops). We limited all runs of CREST and KLEE to 10,000 iterations. In the case of CREST and KLEE, the number of generated test cases is almost equal to the number of iterations; for space reasons we do not report the precise number in the table.

The table shows that ARC-B steadily achieves high branch coverage while the other tools cannot achieve the same coverage and their results vary largely among the set of considered programs. ARC-B identifies all infeasible branches of programs `scan`$\langle i \rangle$, and covers all hard branches of both programs `valves_rep`$\langle i \rangle$ and `valves_nest`$\langle i \rangle$ up to 50 code replicas. It covers all but one branch of `valves_rep100` and about 30% of the brances of `valves_nest100`. The performance of CRESTdfs and KLEE always falls down to large extents for programs with increasing numbers of loops. CRESTcfg has stable performance on the programs `scan`$\langle i \rangle$, where it covers all feasible branches, and on programs `valves_rep`$\langle i \rangle$, where it only misses one feasible branch per code replica, but has disastrous performance for programs `valves_nest`$\langle i \rangle$, where most code is embedded in the hard-to-cover branches. All tools experience their worst performance for program `valves_nest100`, where ARC-B scores about 71% branch coverage, while both CRESTdfs, CRESTcfg and KLEE do not go beyond 1% coverage.

These preliminary experiments pinpoint the case in favor of combining test case generation and iterative refinement of abstract program models, as in ARC-B. ARC-B explicitly keeps the search focused on the uncovered branches and can precisely identify and isolate the infeasible branches. Doing so, it outperforms CRESTdfs and KLEE that engage themselves in the depth-first exploration of infinitely many program paths, and outperforms CRESTcfg that experiences bad results in presence of infeasible branches that deceive its heuristic search strategy. Furthermore, the results on programs `scan`$\langle i \rangle$ specifically indicate that identifying the infeasible branches allows ARC-B to increase the accuracy of the coverage scores.

4.2 Experiments on third-party programs

Table 2 lists the 12 subject programs that we used in our experiments. We selected these programs because they have both non-trivial branching structures and limited size. Thus, while on one hand they challenge the task of achieving high branch coverage rates, on the other hand they allow for the manual inspection of infeasible branches identified by ARC-B. The programs *linsearch* and *binsearch* implement the linear and binary search of an integer value in an array, respectively, *tcas* implements a component of an aircraft traffic control and collision avoidance system, as available from the Software-artifact Infrastructure Repository (Do et al, 2005), *week0..7* are programs that call

			ARC-B						
subject	**loc**	**br**	**tc**	**cbr**	**ibr**	**cov$_1$**	**it$_1$**	**cov$_2$**	**it$_2$**
linsearch	23	8	3	8	0	100%	3	100%	3
binsearch	39	12	4	12	0	100%	6	100%	6
tcas	180	106	21	99	7	93%	485	100%	1185
week0	154	58	15	53	3	91%	50	96%	50
week1	154	58	16	53	3	91%	65	96%	65
week2	154	56	17	52	4	93%	70	100%	121
week3	154	56	16	52	4	93%	50	100%	108
week4	154	58	16	43	13	74%	55	96%	125
week5	154	58	16	53	3	91%	85	96%	90
week6	154	56	18	52	4	93%	90	100%	128
week7	154	56	18	52	4	93%	110	100%	139
week_	154	84	27	81	3	96%	135	100%	135
TOTAL	1628	666	187	610	48	-	-	-	-

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$_1$: cbr / br [as percentage]
it$_1$: number of iterations to achieve cbr
cov$_2$: cbr / (br - ibr) [as percentage]
it$_2$: number of iterations to achieve both cbr and ibr

Table 2 Results of ARC-B

function `calc_week` (excerpted from the MySQL database management system) in different specialized ways,[6] *week_* is a program that call function `calc_week` with no specific customization. Column *loc* reports the size of the programs in lines of code as counted by the GNU utility `wc`. Column *br* reports the number of static branches of each program as counted by ARC-B.[7]

We used ARC-B to maximize the branch coverage of each subject program starting from a randomly generated input test case. Table 2 reports the numbers of test cases that ARC-B generated for each program (column *tc*), the numbers of covered branches (column *cbr*), the number of branches that ARC-B identified as infeasible (column *ibr*), the coverage computed with respect to the static branches both before (column cov_1) and after (column cov_2) pruning the ones identified as infeasible, and the number

[6] Function `calc_week` takes two parameters, `l_time` (a date) and `week_behavior` (the week counting options), and returns the corresponding week of the year (an integer value between 0 and 53). The parameter `week_behavior` is interpreted as a sequence of bits that indicate the day that starts the week (either Sunday or Monday), the baseline to count weeks (either 0 or 1), and the reference standard for the date representation (ISO standard 8601:1988 or not), respectively. In the context of a specific application, `calc_week` is typically used by passing a fixed constant value of `week_behavior` to all calls. This may cause some of the branches of `week_behavior` to be infeasible within a specific application, as it happens in general when reusable libraries are integrated in systems that use only subsets of their functionalities. For further details on this example, we refer the interested readers to Baluda et al (2010).

[7] ARC-B counts the static branches after the CIL pre-compilation that unrolls decisions with multiple conditions as a cascade of single condition decisions, and performs simple code optimizations based on constant propagation. For `calc_week`, this determines slightly different counts of static branches across the different specializations of the program.

of iterations (column it_1 and column it_2) to maximize the two coverage indicators, respectively.

Arc-B generated a total of 187 test cases that cover 610 out of 666 branches, and identified 48 infeasible branches, failing only on 8 branches that Arc-B could neither cover nor identify as infeasible. All runs completed within minutes on a common laptop. Arc-B produces test suites of manageable size that cover most feasible branches (100% in many cases and 96% in the worst cases). The table shows also that the value of the branch coverage that Arc-B computes after identifying and eliminating infeasible elements (column cov_2) is more accurate than the value that Arc-B computes referring to the statically identified branches (column cov_1), although computing the former value requires more iterations (column it_1) than computing the latter value (column it_2) in most cases. This can be observed for all programs that contain infeasible elements, that is, *tcas* and all the *week* programs. The improvement ranges from 4% for *week_* where the coverage grows from 96% to 100%, to 22% for *week4* where the coverage grows from 74% to 96%. We notice that identifying the infeasible branches requires a greater number of iterations (column it_2) than

We compared the effectiveness of Arc-B against plain random testing, directed random testing and automatic generation of test suites for branch coverage, as implemented by Crest and Klee. We used Crest to generate test cases according to three methods: depth-first concolic execution (CRESTdfs), concolic execution driven by control-flow search (CRESTcfg) and random test search strategy (CRESTrand).

Table 3 compares the values of branch coverage obtained with CRESTrand, CRESTdfs, CRESTcfg and Klee on the same subject programs of Table 2 with the value of coverage obtained with Arc-B. To make the results comparable, we ran all tools on exactly the same code, after the logical operators in the program conditionals were removed via the code transformation done by Cil. We terminated the executions after 60 minutes for each program.

	Reported coverage (%)				
subject	CREST**rand**	CREST**dfs**	CREST**cfg**	Klee	Arc-B (**cov$_2$**)
linsearch	100%$^=$	37%	100%$^=$	62%	100%
binsearch	100%$^=$	83%	100%$^=$	75%	100%
tcas	4%	93%	93%	93%	100%
week0	79%	79%	79%	91%	96%
week1	79%	79%	79%	91%	96%
week2	80%	82%	82%	93%	100%
week3	80%	82%	82%	93%	100%
week4	53%	53%	53%	74%	96%
week5	76%	76%	76%	91%	96%
week6	79%	77%	79%	93%	100%
week7	80%	80%	80%	93%	100%
week_	67%	67%	67%	96%	100%

$^=$: equals to cov_2 from Table 2

Table 3 Result of Crest and Klee

In Table 3, a = marks the few cases where a tool performs as well as Arc-B. Crest generates test suites that cover fewer branches than the tool suites generated with Arc-B in all cases, except for *linsearch* and *binsearch* where both Crest and Arc-B cover all branches. Conversely, Klee covers the same number of branches as Arc-B in most

cases, except for *linsearch* and *binsearch* where the tool suites generated with KLEE cover a small number of branches. In all cases, after pruning the branches identified as infeasible, ARC-B computes coverage scores greater than the ones computed by the other tools.

4.3 Threats to validity

ARC-B is an early research prototype that has not been extensively used and validated yet. Thus, the main threat to the *internal* validity of our experiments is the potential incorrectness of the data computed by ARC-B on the subject programs. We contrasted this threat in several ways. We systematically tested (and fixed) the prototype based on several reference programs with known branch reachability data, including the sample of artificial programs reported in this section. We confirmed all covered branches computed by ARC-B by re-running the test cases generated by the tool under the GCOV tool.[8] We confirmed a sample of the infeasible branches identified by ARC-B by manual inspection. We crosschecked that none of the other tools considered in the experiments covers branches that ARC-B identifies as infeasible.

KLEE and CREST/ARC-B use different constraint solvers, which may lead to question which branches were not covered by either tool because of the limitations of the constraint solver rather than the path exploration strategy. Unfortunately we could not extract information on the failures of the decision procedure from the reports provided by KLEE. We are currently developing the support for multiple constraint solvers in ARC-B to further study this issue.

One of the key contributions of KLEE is its ability to *close* a program by having a mock up of the system calls. For example KLEE can handle the set of GNU utility programs in this way. ARC-B does not currently include such a support. Admittedly, we suspect that our tool can experience problems with programs that are integrated in large systems due to this limitation, and we aim at extending the tool in this direction as future work.

The experimental results reported in this paper are encouraging but not conclusive yet. The main threat to their *external* validity relates to the limited size of the programs considered so far, due to the limits of our early prototype implementation of the technique. We are working on a second generation toolset that will allow us to experiment with industrial-size programs. In particular, we are extending the prototype to deal with functions, pointers and dynamic memory allocation, which are not currently supported, and we are optimizing the prototype for performance.

5 Related work

Research in the field of automated structural testing dates back to the seminal works by Boyer et al (1975), King (1976), and Clarke (1976). The last decade has seen an increasing industrial and academic interest on the topic.

A well-established line of research exploits symbolic execution, i.e., the simulation of program execution along a prescribed set of control flow paths to the code elements

[8] GCOV is a test coverage program included in the GNU development suite. GCOV can track the branches and statements covered when executing a C program, provided that this has been compiled with specific instrumentation options.

to be covered. Symbolic execution calculates path constraints over program inputs, and solves them to generate test cases. Approaches based on symbolic execution must select a set of feasible paths leading to the code elements to be covered. Static analysis of the control flow graph is perhaps the oldest, and best known, approach to identify such paths. Recent tools adopting this approach are Java PathFinder (Visser et al, 2004), Exe (Cadar et al, 2006), and Klee (Cadar et al, 2008).

An essential problem that this approach must face is the infeasibility of statically identifiable control flow paths: Paths found by static analysis may not correspond to any program computation, and thus tools must decide about path feasibility, usually by querying an external decision procedure. The problem of deciding about element feasibility makes an automated test case generator strongly dependent on the theories that the solver of choice is able to manage efficiently. Yates and Malevris (1989) provided experimental evidence that deciding about the feasibility of code elements depends on the complexity of the path constraint associated to the elements and thus on the depth of the elements in the control graph: The deeper an elements is in the control flow graph, the harder is to decide about its feasibility. Usually, invoking a decision procedure is the slowest step of test case generation.

To overcome these issues, several authors have studied *dynamic* test case generation techniques. Dynamic approaches learn about feasible control flow paths from the execution of previous test cases, and use such knowledge to simplify the construction of new test cases. Korel (1992) and Ferguson and Korel (1996) combined test case execution and dynamic dataflow analysis to identify the variables responsible for steering the program towards a given control flow path. Csallner and Smaragdakis (2005) introduced Check'n'Crash that generates test cases by trying to violate the verification conditions produced by a static symbolic program analysis. Csallner et al (2008) evolved Check'n'Crash into DSD-Crasher that strives at reducing the exploration scope by dynamically inferring likely preconditions from the execution of tests, and discarding all the executions that violates the inferred preconditions.

Concolic (concrete-symbolic) testing is a recent approach to test case generation, which is attracting considerable interest. Introduced by Godefroid et al (2005) with the Dart tool and by Sen et al (2005) with the Cute tool, concolic execution is a lightweight form of symbolic execution that is performed along the feasible control flow paths discovered by previous test cases. Concolic execution yields a set of path constraints that are mangled to obtain path predicates that characterize unexplored branches. Solving these path predicates yields test cases that both reach not-yet-covered branches and discover new paths throughout these branches. On average, concolic execution halves the number of solver invocations, and exploits input values from previous test cases to guess possible solutions to path conditions that do not belong to the theories of the solver in use. With Sage, Godefroid et al (2008) specializes concolic execution to target security faults.

Both static and dynamic test case generation techniques suffer from the state space explosion problem, as they explore the possible program behaviors in search of suitable test cases, and may diverge by getting stuck on the analysis of an infinite set of infeasible behaviors. Dynamic techniques also suffer from the fact that the initial set of program paths that seed the analysis can bias the explored region of the state space. Dart and Cute start from a set of random tests, and explore program paths in depth-first order. As a consequence, they may generate massive test suites that cover only a small subset of the code elements. For this reason, several studies investigated variants of the approach based on different exploration strategies. Burnim and Sen (2008) de-

fined heuristics that guide the exploration of the program execution space based on the program control flow to speed up branch coverage, and implemented these heuristics in CREST. Xie et al (2009) introduced heuristics based on a fitness function that measures the distance between paths and code elements not yet covered. The heuristics proposed so far yield fast convergence in some practical cases, but they may as well degrade convergence in other cases. Furthermore, heuristics do not solve the problem of infeasible elements that may cause the analysis to diverge when an infeasible element belongs to infinitely many static program paths.

Another strand of research aims at reducing the number of explored program paths at the expense of some precision. Majumdar and Sen (2007) propose hybrid concolic testing that alternates random and concolic testing: Random testing supports wide-range exploration of the program state space, and concolic testing guarantees exhaustive local search of the most promising regions. Chipounov et al (2011) introduce a framework for the testing and analysis of programs based on KLEE, which alternates symbolic and concrete execution to reduce the number of explored paths, and discuss how this framework allows to choose a performance/accuracy tradeoff suitable for a given analysis. Other approaches exploit the computational power of multi-core processors, and accelerate the exploration of the symbolic space by parallelizing it (Staats and Păsăreanu, 2010; Bucur et al, 2011).

A different line of research, pioneered by Callahan et al (1996), exploits model checking techniques to generate test cases. These approaches recast the testing problem as a model checking one by abstracting the program under test to a model, expressing the target coverage criterion in temporal logic formulas, and then returning the counterexamples produced by the model checker as test cases. Fraser et al (2009) provide a comprehensive survey on these approaches. These techniques suffer the problem of suitably reconciling tractability and precision when modeling the system under test, must deal with state space explosion, and may diverge while trying to cover infeasible elements.

Recently, some research groups investigated abstract interpretation techniques to generalize the code coverage problem beyond the classic control and data flow abstractions. Ball (2003) suggests to cover the feasible states of predicate abstractions of the systems under test as a new class of adequacy criteria, and proves that these criteria subsume many classic control flow criteria (Ball, 2004). Beyer et al (2004) integrate abstraction refinement and symbolic execution to better target coverage criteria. These approaches can prove the infeasibility of some targets by successive model refinements, and thus can converge even in presence of infeasible targets.

To the best of our knowledge, our approach is the first attempt to integrate test case generation and infeasibility analysis to improve structural code coverage. Our approach integrates test case generation and test case execution along the lines of SYNERGY (Gulavani et al, 2006), and adopts an inexpensive approach to the computation of refinement predicates based on weakest precondition as in DASH (Beckman et al, 2010). SYNERGY and DASH focus on formal verification, while we target structural testing. Furthermore, what clearly distinguishes ARC from the approaches available in literature is the introduction of coarsening, to control model size explosion in presence of multiple targets.

6 Conclusions

Sophisticated structural testing criteria are not yet widely adopted in industrial settings, mostly because of the difficulty of obtaining decent levels of coverage. This paper introduces abstraction refinement and coarsening (ARC), a technique that combines automatic test case generation and feasibility analysis to improve code coverage. ARC exploits static and dynamic techniques in an abstraction refinement framework, along the lines of previous work on software verification, and introduces the new concept of abstraction coarsening to adapt the approach for branch testing. Addressing multiple code targets challenges automatic test case generation with demanding scalability requirements. Coarsening increases the scalability of the analysis by dynamically balancing precision and memory requirements.

The experimental evaluation shows that ARC is effective in focusing test case generation on feasible targets, enhancing the precision of coverage measurements, and addressing non trivial programs. In most of the experiments, ARC achieves higher coverage with smaller test suites than popular state-of-the-research test case generation tools. In the next future, we aim to extend the approach to support sophisticated control- and data-flow coverage criteria, and investigate fallback strategies to handle models that do not completely fall in the theory of the theorem provers integrated in the prototype.

References

Ball T (2003) Abstraction-guided test generation: A case study. Tech. Rep. MSR-TR-2003-86, Microsoft Research

Ball T (2004) 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), Springer Berlin / Heidelberg, Lecture Notes in Computer Science, vol 3657, pp 1–22

Ball T, Cook B, Levin V, Rajamani SK (2004) 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, pp 1–20

Baluda M, Braione P, Denaro G, Pezzè M (2010) Structural coverage of feasible code. In: Proceedings of the Fifth International Workshop on Automation of Software Test (AST 2010)

Beckman NE, Nori AV, Rajamani SK, Simmons RJ, Tetali SD, Thakur AV (2010) Proofs from tests. IEEE Transactions on Software Engineering 36(4):495–508

Beyer D, Chlipala AJ, Henzinger TA, Jhala R, Majumdar R (2004) Generating tests from counterexamples. In: Proceedings of the 26th International Conference on Software Engineering (ICSE '04), IEEE Computer Society, pp 326–335

Beyer D, Henzinger T, Jhala R, Majumdar R (2007) The software model checker BLAST. International Journal on Software Tools for Technology Transfer (STTT) 9(5-6):505–525, DOI 10.1007/s10009-007-0044-z

Boyer RS, Elspas B, Levitt KN (1975) SELECT—a formal system for testing and debugging programs by symbolic execution. ACM SIGPLAN Notices 10:234–245

Bucur S, Ureche V, Zamfir C, Candea G (2011) Parallel symbolic execution for automated real-world software testing. In: Proceedings of EuroSys 2011, ACM

Burnim J, Sen K (2008) Heuristics for scalable dynamic test generation. In: Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), pp 443–446

Cadar C, Ganesh V, Pawlowski PM, Dill DL, Engler DR (2006) EXE: automatically generating inputs of death. In: Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS '06), ACM, New York, NY, USA, pp 322–335, DOI 10.1145/1180405.1180445

Cadar C, Dunbar D, Engler D (2008) 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)

Callahan J, Schneider F, Easterbrook S (1996) Automated software testing using model-checking. In: Proceedings of the 1996 SPIN Workshop (SPIN 1996). Also WVU Technical Report NASA-IVV-96-022., URL `citeseer.ist.psu.edu/article/callahan96automated.html`

Chipounov V, Kuznetsov V, Candea G (2011) 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, pp 265–278, DOI 10.1145/1950365.1950396

Clarke LA (1976) A system to generate test data and symbolically execute programs. IEEE Transactions on Software Engineering 2:215–222

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

Csallner C, Smaragdakis Y, Xie T (2008) DSD-crasher: a hybrid analysis tool for bug finding. ACM Transactions on Software Engineering and Methodology (TOSEM) 17(2):1–37

Do H, Elbaum SG, Rothermel G (2005) Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. Empirical Software Engineering: An International Journal 10(4):405–435

Ferguson R, Korel B (1996) The chaining approach for software test data generation. ACM Transactions on Software Engineering and Methodology 5:63–86

Frankl PG, Weyuker EJ (1988) An applicable family of data flow testing criteria. IEEE Transactions on Software Engineering 14(10):1483–1498, DOI 10.1109/32.6194

Fraser G, Wotawa F, Ammann PE (2009) Testing with model checkers: a survey. Software Testing, Verification and Reliability 19(3):215–261

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

Godefroid P, Levin MY, Molnar D (2008) Automated whitebox fuzz testing. In: Proceedings of the 16th Annual Network and Distributed System Security Symposium (NDSS 2008), pp 151–166, DOI 10.1.1.129.5914

Gulavani BS, Henzinger TA, Kannan Y, Nori AV, Rajamani SK (2006) Synergy: A new algorithm for property checking. In: Proceedings of the 14th ACM SIGSOFT symposium on Foundations of Software Engineering (FSE-14), pp 117–127

King JC (1976) Symbolic execution and program testing. Communications of the ACM 19(7):385–394

Korel B (1992) Dynamic method for software test data generation. Software Testing, Verification and Reliability 2(4):203–213

Majumdar R, Sen K (2007) Hybrid concolic testing. In: Proceedings of the 29th International Conference on Software Engineering, IEEE Computer Society, pp 416–426

Namin AS, Andrews JH (2009) The influence of size and coverage on test suite effectiveness. In: Proceedings of the eighteenth international symposium on Software testing and analysis (ISSTA '09), ACM, New York, NY, USA, pp 57–68

Pezzè M, Young M (2007) Software Testing and Analysis: Process, Principles and Techniques. Wiley, URL http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0471455938

RTCA, Inc (1993) Document RTCA/DO-178B. U.S. Department of Transportation, Federal Aviation Administration, Washington, D.C.

Sen K, Marinov D, Agha G (2005) 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), pp 263–272

Staats M, Păsăreanu CS (2010) Parallel symbolic execution for structural test generation. In: Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA 2010), ACM, pp 183–194, DOI 10.1145/1831708.1831732

Visser W, Păsăreanu CS, Khurshid S (2004) Test input generation with Java PathFinder. In: Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2004), ACM, pp 97–107

Weyuker EJ (1988) The evaluation of program-based software test data adequacy criteria. Communications of the ACM 31(6):668–675, DOI 10.1145/62959.62963

Xie T, Tillmann N, de Halleux P, Schulte W (2009) 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), pp 359–368, URL http://www.csc.ncsu.edu/faculty/xie/publications/dsn09-fitnex.pdf

Yates DF, Malevris N (1989) Reducing the effects of infeasible paths in branch testing. In: Proceedings of the ACM SIGSOFT '89 third symposium on Software testing, analysis, and verification, ACM, pp 48–54