Abstraction-driven Concolic Testing[*]

Przemysław Daca[1], Ashutosh Gupta[2], and Thomas A. Henzinger[1]

[1] IST Austria, Austria
[2] Tata Institute for Fundamental Research, India

Abstract. Concolic testing is a promising method for generating test suites for large programs. However, it suffers from the path-explosion problem and often fails to find tests that cover difficult-to-reach parts of programs. In contrast, model checkers based on counterexample-guided abstraction refinement explore programs exhaustively, while failing to scale on large programs with precision. In this paper, we present a novel method that iteratively combines concolic testing and model checking to find a test suite for a given coverage criterion. If concolic testing fails to cover some test goals, then the model checker refines its program abstraction to prove more paths infeasible, which reduces the search space for concolic testing. We have implemented our method on top of the concolic-testing tool CREST and the model checker CPACHECKER. We evaluated our tool on a collection of programs and a category of SV-COMP benchmarks. In our experiments, we observed an improvement in branch coverage compared to CREST from 48% to 63% in the best case, and from 66% to 71% on average.

1 Introduction

Testing has been a corner stone of ensuring software reliability in the industry, and despite the increasing scalability of software verification tools, it still remains the preferred method for debugging large software. A test suite that achieves high code coverage is often required for certification of safety-critical systems, for instance by the DO-178C standard in avionics [2].

Many methods for automated test generation have been proposed [9,32,13,36,37,28,10,18]. In the recent years, concolic testing has gained popularity as an easy-to-apply method that scales to large programs. Concolic testing [33,35] explores program paths by a combination of concrete and symbolic execution. This method, however, suffers from the path-explosion problem and fails to produce test cases that cover parts of programs that are difficult to reach.

Concolic testing explores program paths using heuristic methods that select the next path depending on the paths explored so far. Several heuristics for path exploration have been proposed that try to maximize coverage of concolic

---

testing [11,20,19], e.g., randomly picking program branches to explore, driving exploration toward uncovered branches that are closest to the last explored branch, etc. These heuristics, however, are limited by their "local view" of the program semantics, i.e., they are only aware of the (in)feasibility of the paths seen so far. In contrast to testing, abstraction-based model checkers compute abstract reachability graph of a program [3,26]. The abstract reachability graph represents a "global view" of the program, i.e., the graph contains all feasible paths. Due to abstraction, not all paths contained in the abstract reachability graph are guaranteed to be feasible, therefore abstract model checking is not directly useful for generating test suites.

In this paper, we present a novel method to guide concolic testing by an abstract reachability graph generated by a model checker. The inputs to our method are a program and set of test goals, e.g. program branches or locations to be covered by testing. Our method iterativly runs concolic testing and a counterexample-guided abstraction refinement (CEGAR) based model checker [14]. The concolic tester aims to produce test cases covering as many goals as possible within the given time budget. In case the tester has not covered all the goals, the model checker is called with the original program and the remaining uncovered goals marked as error locations. When the model checker reaches a goal, it either finds a test that covers the goal or it refines the abstraction. We have modified the CEGAR loop in the model checker such that it does not terminate as soon as it finds a test, but instead it removes the goal from the set of error locations and continues building the abstraction. As a consequence, the model checker refines the abstraction with respect to the remaining goals. After the model checker has exhausted its time budget, it returns tests that cover some of the goals, and an abstraction. The abstraction may prove that some of the remaining goals are unreachable, thus they can be omitted by the testing process.

We further use the abstraction computed by the model checker to construct a *monitor*, which encodes the proofs of infeasibility of some paths in the control-flow graph. To this end, we construct a program that is an intersection of the monitor and the program. In the following iterations we run concolic testing on the intersected program. The monitor drives concolic testing away from the infeasible paths and towards paths that still may reach the remaining goals. Due to this new "global-view" information concolic testing has fewer paths to explore and is more likely to find test cases for the remaining uncovered goals. If we are still left with uncovered goals, the model checker is called again to refine the abstraction, which further reduces the search space for concolic testing. Our method iterates until the user-defined time limit is reached.

The proposed method is configured by the ratio of time spent on model checking to the time spent on testing. As we demonstrate in Section 2, this ratio has a strong impact on the test coverage achieved by our method.

We implemented our method in a tool called CRABS, which is built on top of a concolic-testing tool CREST [11] and a CEGAR-based model checker CPACHECKER [8]. We applied our tool on three hand-crafted exam-

ples, three selected published examples, and on 13 examples from an Sv-Comp category. We compared our implementation with two tools: a concolic tool CREST [11], and a test-case generator FSHELL based on bounded model checking [27]. The test objective was to cover program branches, and we calculate test coverage as the ratio of branches covered by the generated test suite to the number of branches that have not been proved unreachable. For a time limit of one hour, our tool achieved coverage of 63% compared to 48% by other tools in the best case, and average coverage of 71% compared to 66% on the category examples. In absolute numbers, our experiments may not appear very exciting. However, experience suggests that in automated test generation increasing test coverage by every 1% becomes harder. The experiments demonstrate that our method can cover branches that are difficult to reach by other tools and, unlike most testing tools, can prove that some testing goals are unreachable.

To summarize, the main contributions of the paper are:

– We present a novel configurable algorithm that iterativly combines concolic testing and model checking, such that concolic testing is guided by a program abstraction and the abstraction is refined for the remaining test goals.
– We also present a modified CEGAR procedure that refines the abstraction with respect to the uncovered goals.
– We provide an open-source tool [1] that implements the presented algorithm.
– An experimental evaluation of our algorithm and comparison with other methods.

The paper is organized as follows. In Section 2 we motivate our approach on examples. Section 3 presents background notation and concolic testing. In Section 4 we present our modified CEGAR procedure, and in Section 5 we describe our main algorithm. Finally, Section 6 describes the experimental evaluation.

2 Motivating Example

In this section, we illustrate effectiveness of our method on two examples: a hand-crafted program, and a benchmark for worst-case execution time analysis adapted from [4].

Simple loop In Figure 1 we present a simple program with a single while loop. The program iterates 30 times through the while loop, and in every iteration it reads an input. The test objective is to cover all locations of the program, in particular to cover location 8, where the library function `foo()` is called. To cover the call site to `foo()` the inputs in all iterations must equal 10, so only one out of 2^{30} ways to traverse the loop covers `foo()`. The standard concolic testing easily covers all locations, except for `foo()` since it blindly explores exponentially many possible ways to traverse the loop. As a consequence, a concolic-testing tool is not able to generate a complete test suite that executes `foo()` within one hour.

```
int i=0; bool b = false;

while (i<30){
  int x = input();
  if (x != 10)
    b=true;
  i++;
}

if (b == false)
  foo();
```
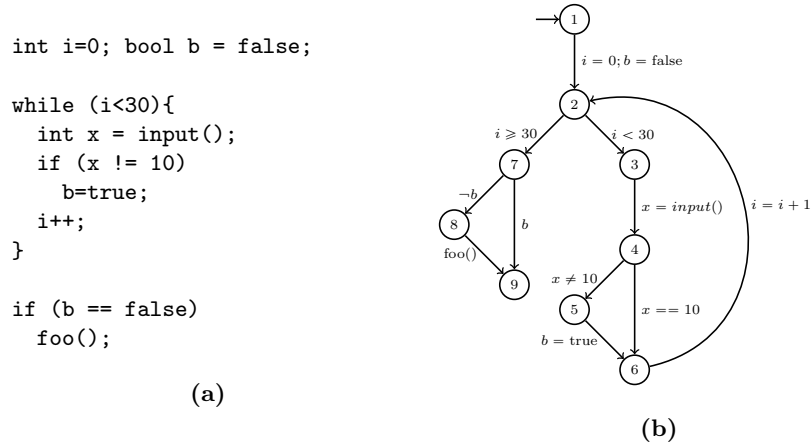
(a)

(b)

Fig. 1: (a) A simple while program. (b) The control-flow graph of the program.

Our algorithm uses a concolic tester and model checker based on predicate abstraction, and runs them in alternation. First, we run concolic tester on the example with a time budget of 1s. As we have observed earlier, the concolic tester covers all locations of the program except for `foo()`. Then, we declare the call site to `foo()` as an error location and call the model checker on the program for 5s. This time budget is sufficient for the model checker to perform only a few refinements of the abstraction, without finding a feasible path that covers `foo()`. In particular, it finds an abstract counterexample that goes through locations $1, 2, 3, 4, 5, 6, 2, 7, 8, 9$. This counterexample is spurious, so the refinement procedure finds the predicate "b holds." The abstraction refined with this predicate is showed in Figure 2(a).

In the second iteration of the algorithm, we convert the refined abstraction into a monitor $\mathcal{M}$ shown in Figure 2(b). A monitor is a control-flow graph that represents all the paths that are allowed by the abstraction. A monitor is constructed by removing subsumed states from the abstraction. We say that an abstract state s_a is *subsumed* by a state s_a', if $s_a = s_a'$, or s_a' is more general than s_a. To this end, the monitor includes all the abstract states that are not subsumed and the edges between them. The edges to the subsumed states are redirected to the states that subsume them.

The monitor contains all the feasible paths of the program and is a refinement of the control-flow graph of the original program. Therefore, we may perform our subsequent concolic testing on the monitor interpreted as a program. In our example, the structure of the monitor in Figure 2(b) encodes the information that `foo()` can be reached only if b is never set to true. The refined control flow graph makes it easy for concolic testing to cover the call to `foo()` — it can simply backtrack whenever the search goes to the part of the refined program where `foo()` is unreachable. Now, if we run CREST on the monitor $\mathcal{M}$ then it finds the test case in less than 1s.
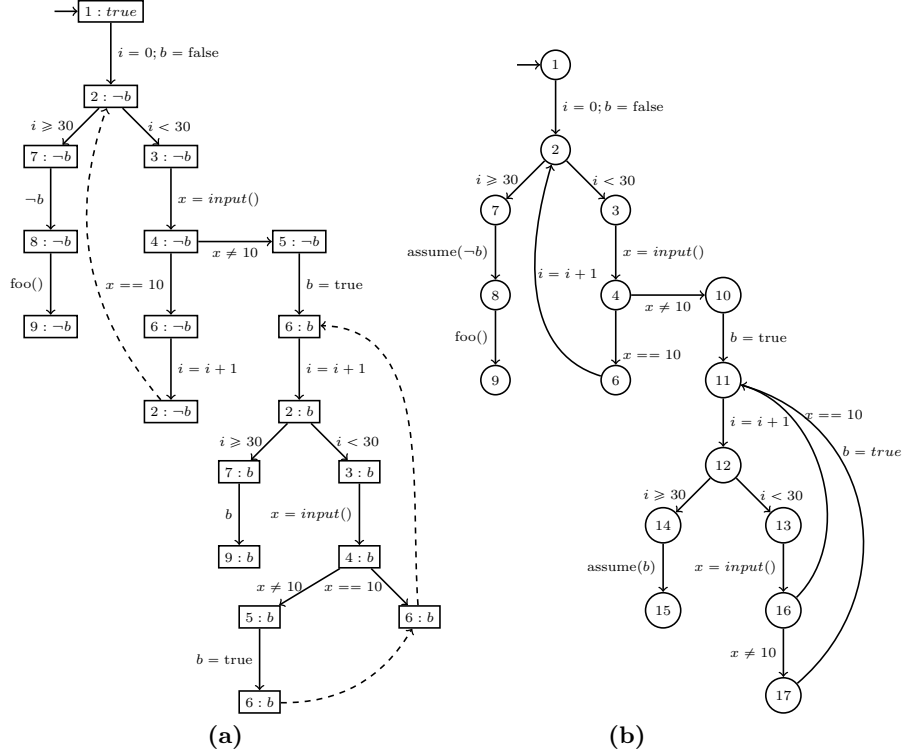
4

Fig. 2: (a) Abstraction refined with the predicate b. Dashed arrows show subsumption between abstract state. (b) The monitor obtained from the abstraction.

Nsichneu The "nsichneu" example is a benchmark for worst-case execution time analysis [24] and it simulates a Petri net. This program consists of a large number of if-then-else statements closed in a deterministic loop. The program maintains several integer variables and fixed-sized arrays of integers. These data objects are marked as `volatile` meaning that their value can change at any time. We made their initial values the input to the program.

The structure of this benchmark makes it challenging for many testing techniques. Testing based on bounded model checking (such as FSHELL[27]) unwinds the program up to a given bound and encodes the reachability problem as a constraint-solving problem. However, this method may not find goals that are deep in the program, as the number of constraints grows quickly with the bound. Test generation based on model checking [7] also fails to deliver high coverage on this example. The model checker needs many predicates to find a feasible counterexample, and the abstraction quickly becomes expensive to maintain. In contrast, pure concolic testing quickly covers easy-to-reach parts of the program. However, later it struggles to cover goals that are reachable by fewer paths.

5

In our method, we run concolic testing and model checking alternatively, each time with a time budget of 100s. Every iteration of model checking gives us a more refined monitor to guide the testing process. Initially, our approach covers goals at similar rate as pure concolic testing. When the easy goals have been reached, our tool covers new goals faster than concolic testing, due to the reachability information encoded in the monitor, which allows the testing process to skip many long paths that would fail to cover new goals. After one hour, our tool covers 63% of the test goals compared to 48% by concolic testing.

Furthermore, our method is configurable by the ratio of time spent on model checking and concolic testing. In Figure 3 we present the effect of changing this ratio on the example. If we run only concolic testing then we obtain only 48% coverage. As we decrease the time spent on concolic testing, the coverage increases up to 64% and then starts decreasing. On the other side of the spectrum, we generate tests by model checking (as in [7]) and obtain only 13.9% coverage. This observation allows one to configure our method for most effective testing depending on the class of examples.
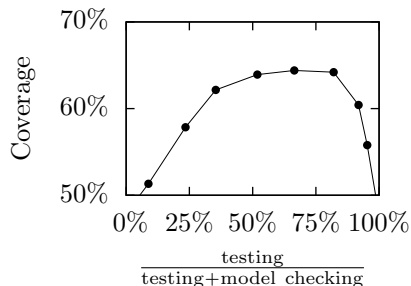

Fig. 3: Test coverage vs. ratio of testing to total time in our method.

3 Preliminaries

In this paper, we consider only sequential programs and, for ease of presentation, we consider programs without procedures. Our method, however, is easily applicable on programs with procedures and our implementation supports them.

Let V be a vector of variables names and V' be the vector of variables obtained by placing prime after each variable in V. Let $F(V)$ be the set of first-order-logic formulas that only contain free variables from V.

Definition 1 (Program) *A program P is a tuple (V, Loc, ℓ^I, E), where V is a vector of variables, Loc is a finite set of locations, $\ell^I \in Loc$ is the initial location, and $E \subseteq Loc \times F(V, V') \times Loc$ is a set of program transitions.*

A *control-flow graph* (CFG) is a graph representation of a program. We define the *product* of two programs $P_{i=1..2} = (V, Loc_i, \ell_i^I, E_i)$ as the program $P_1 \times P_2 = (V, Loc_1 \times Loc_2, (\ell_1^I, \ell_2^I), E)$, where

$$E = \{((\ell_1, \ell_2), e, (\ell_1', \ell_2')) \mid (\ell_1, e, \ell_1') \in E_1 \wedge (\ell_2, e, \ell_2') \in E_2\}.$$

A *guarded command* is a pair of a formula in $F(V)$ and a list of updates to variables in V. For ease of notation, we may write the formula in a program transition as a guarded command over variables in V. For example, let

us consider $V = [x, y]$. The formula represented by the guarded command $(x > y, [x := x + 1])$ is $x > y \wedge x' = x + 1 \wedge y' = y$. In our notation if a variable is not updated in the command then the variable remains unchanged. We use a special command $variable := input()$ to model inputs to the program, which logically means unconstrained update of the variable. For example, the formula represented by the guarded command $x := input()$ is $y' = y$. For an expression or formula F we write $F[/i]$ to denote a formula that is obtained after adding subscript $i + 1$ to every primed variable and i to every unprimed variable.

A *valuation* is a mapping from the program variables V to values in the data domain. A *state* $s = (l, v)$ consists of a program location l and a valuation v. For a state $s = (l, v)$ and a variable x, let $s(x)$ denote the valuation of x in v and let $loc(s) = l$. A *path* is a sequence $e_0, \ldots, e_{n-1}$ of program transitions such that $e_0 = (\ell^I, \_, \_)$, and for $0 \leqslant i < n$, $e_i = (\ell_i, \_, \ell_{i+1}) \in E$. An *execution* corresponding to the path $e_0, \ldots, e_{n-1}$ is a sequence of states $s_0 = (\ell_0, v_0), \ldots s_n = (\ell_n, v_n)$, such that 1) $\ell_0 = \ell^I$, and 2) for all $0 \leqslant i < n$, if $e_i = (\_, c_i(V, V'), \ell')$ then $\ell_{i+1} = \ell'$ and $c_i(v_i, v_{i+1})$ holds true. We assume that for each execution of the program there exist exactly one corresponding path, i.e., there is no non-determinism in the program except inputs.

A path is represented symbolically by a set of path constraints, which we define as follows. Let $frame(x)$ be the formula $\bigwedge_{y \in V : y \neq x} y' = y$. Let r_k be a variable that symbolically represent the kth input on some path. We assume the program does not contain any variable named r. Let $e_0, \ldots, e_{n-1}$ be a path. If $e_i = (\_, [F, x := exp], \_)$ then let $C_i = (F \wedge x' = exp \wedge frame(x))[/i]$ and if $e_i = (\_, [F, x := input()], \_)$ then let $C_i = (F \wedge frame(x))[/i] \wedge x_{i+1} = r_k$, where r_0 up to r_{k-1} have been used in $C_0, \ldots, C_{i-1}$. The *path constraints* for the path is $C_0, \ldots, C_{n-1}$.

A *test* of the program is a sequence of values. A test $u_1, \ldots, u_k$ *realizes* an execution $s_0, \ldots, s_n$ and its corresponding path $e_0, \ldots, e_{n-1}$ if the following conditions hold true:

- if $n = 0$, then $k = 0$.
- If $n > 0$ and $e_{n-1} = (\_, x := input(), \_)$, $s_n(x) = u_k$ and $u_1, \ldots, u_{k-1}$ realizes $s_0, \ldots, s_{n-1}$.
- Otherwise, $u_1, \ldots, u_k$ realizes $s_0, \ldots, s_{n-1}$.

A path is said to be *feasible* if there exists a test that realizes it. In the above, we assume that the program does not read a variable until its value is initialized within the program or explicitly taken as input earlier. Thus, the initial values are not part of tests.

In the context of test suit generation, we may refer to a transition as a *branch* if the source location of the transition has multiple outgoing transitions. A test t *covers* branch e if the test realizes a path that contains e. Branch e is *reachable* if there exists a test t that covers e. The *test generation problem* is to find a set of tests that covers every reachable branch in the program.

Algorithm 1 Concolic$(P = (V, L, \ell^I, E), G, t_b)$

Require: program $P = (V, L, \ell^I, E)$, uncovered branches G, time budget t_b
Ensure: tests suite, uncovered branches
1: $tst \leftarrow ()$;
2: $\ell \leftarrow \ell^I$; arbitrary v; $\mathcal{S} \leftarrow \lambda x \in V.\bot$ $\triangleright$ initial values
3: $pathC \leftarrow ()$; $suite \leftarrow \varnothing$; $k = 0$;
4: **while** $ct < t_b$ and $G \neq \varnothing$ **do** $\triangleright$ ct always has the current time
5:     **if** $\exists e = (\ell, [F, x := exp], \ell') \in E$ such that $v \models F$ **then** $\triangleright$ expand
6:         $G \leftarrow G - \{e\}$; $\ell \leftarrow \ell'$;
7:         $pathC.push(F(\mathcal{S}))$ $\triangleright$ $F(\mathcal{S})$ is substitution
8:         **if** $exp = input()$ **then**
9:             **if** $|tst| = k$ **then** $w \leftarrow randVal()$; $tst.push(w)$; **else** $w \leftarrow tst(k)$;
10:             $v \leftarrow v[x \mapsto w]$; $\mathcal{S} \leftarrow \mathcal{S}[x \mapsto r_k]$; $k = k + 1$
11:         **else**
12:             $v \leftarrow v[x \mapsto exp(v)]$
13:             $\mathcal{S} \leftarrow \mathcal{S}[x \to UpdateSymMem(\mathcal{S}, exp, v)]$
14:     **else** $\triangleright$ backtrack
15:         $suite \leftarrow suite \cup \{tst\}$
16:         **if** $\exists i < |pathC|$ such that $\phi = \bigwedge_{j<i} pathC(j) \wedge \neg pathC(i)$ is sat **then**
17:             $m = getModel(\phi)$
18:             $l \leftarrow$ number of distinct r_is that occur in ϕ
19:             $tst \leftarrow (m(r_0), \ldots, m(r_{l-1}))$
20:             **goto** 2
21:         **else break;**
22: **return** $(suite, G)$

3.1 Concolic Testing

In concolic testing, a test suite is generated using both symbolic and concrete execution. In Algorithm 1 we reproduce the procedure; the presentation is modified such that we may use the procedure in our main algorithm. For simplicity of the presentation, we assume that there are at most two outgoing transitions at any program location and their guards are complementary to each other. This assumption does not restrict the applicability of the method.

The procedure takes a program $P = (V, Loc, \ell^I, E)$, a set of goal branches G, and a time budget t_b as input, and returns a test suite that covers a subset of G within the time budget t_b. The procedure maintains a symbolic memory $\mathcal{S}$, which is a partial function from the program variables V to symbolic expressions. We use the symbol $\bot$ to denote an undefined value in a partial function. In addition, the procedure uses the following data structures: the current location ℓ, current valuation v of variables, list $pathC$ that contains constraints along the current path, test tst that produces the current path, counter k of inputs that have been read on the current path, and a set $suite$ of tests seen so far. We initialize all the collecting data structures to be empty, ℓ is initialized to be the initial location ℓ^I, and the symbolic memory to be empty.

The algorithm proceeds by extending the current path by a transition in each iteration of the while loop at line 4. The loop runs until there are no goals to be covered or the procedure runs out of its time budget. In the loop body, the condition checks if it is possible to extend the current path by a transition $e = (\ell, [F, x := exp], \ell')$. If the guard of e satisfies the current valuation v then e is removed from the set of goals and the current location is updated to ℓ'. In case e has an input command $x := input()$, then 1) the algorithm updates $v(x)$ to the kth value from tst if it is available, 2) otherwise $v(x)$ is assigned a random value w, and w is appended to tst. In either case, $\mathcal{S}$ is updated by a fresh symbol r_k, assuming r_0 to r_{k-1} have been used so far. If e is not an input command, then both concrete and symbolic values of x are updated in v and $\mathcal{S}$ at line 10.

The symbolic memory is updated by the procedure $UpdateSymMem$. $UpdateSymMem$ first computes $exp(\mathcal{S})$, and if the resulting formula is beyond the capacity of available satisfiability checkers, then it simplifies the formula by substituting the concrete values from v for some symbolic variables to make the formula decidable in the chosen theory. $UpdateSymMem$ is the key heuristics in concolic testing that brings elements of concrete testing and symbolic execution together. For details of this operation see [33,35].

At line 7, $pathC$ is extended by $F(\mathcal{S})$, which is the formula obtained after substituting every variable x occurring in F by $\mathcal{S}(x)$. We assume that variables are always initialized before usage, so $\mathcal{S}$ is always defined for free variables in F.

In case the current path cannot be further extended, at lines 16–19 the procedure tries to find a branch on the path to backtrack. For a chosen branch with index i, a formula is built that contains the path constraints up to $i - 1$ and the negation of the ith constraint. If this formula is satisfiable, then its model is converted to a new test and path exploration restarts. Note that the branch can be chosen non-deterministically, which allows us to choose a wide range of heuristics for choosing the next path. For example, the branch can be chosen at random or in the depth-first manner by picking the largest unexplored branch i. Another important heuristic that is implemented in CREST is to follow a branch that leads to the closest uncovered branch.

4 Coverage-driven Abstraction Refinement

In this section, we present a modified version of CEGAR-based model checking that we use in our main algorithm. Our modifications are: 1) the procedure continues until all goal branches are covered by tests, proved unreachable or until the procedure reaches the time limit, 2) the procedure always returns an abstract reachability graph that is closed under the abstract post operator.

The classical CEGAR-based model checking executes a program using an abstract semantics, which is defined by an abstraction. Typically, the abstraction is chosen such that the reachability graph generated due to the abstract execution is finite. If the computed reachability graph satisfies the correctness specification, then the input program is correct. Otherwise, the model checker finds an abstract counterexample, i.e., a path in the reachability graph that reaches an

---

Algorithm 2 ABSTRACTMC($P = (V, L, \ell^I, E), \pi, G, t_b$)

Require: program $P = (V, L, \ell^I, E)$, predicates π, uncovered branches G,
  time budget t_b
Ensure: tests, remaining branches, branches proved unreachable, new predicates,
  abstract reachability graph
1: $worklist \leftarrow \{(\ell^I, \varnothing)\}$; $reach \leftarrow \varnothing$; $subsume \leftarrow \lambda s_a.\bot$; $parent((\ell_0, \varnothing)) \leftarrow \bot$
2: **while** $worklist \neq \varnothing$ **do**
3:     choose $(\ell, A) \in worklist$
4:     $worklist \leftarrow worklist \backslash \{(\ell, A)\}$
5:     **if** false $\in A$ or $\exists s_a \in parent^*((\ell, A)).\ s_a \in sub$ **then continue**
6:     $reach \leftarrow reach \cup \{(\ell, A)\}$
7:     **if** $\exists (\ell, A') \in reach - sub.\ A \subseteq A'$ **then** $subsume \leftarrow subsume[(\ell, A) \mapsto (\ell, A')]$
8:     **else**
9:         **if** $\exists (\ell, A') \in reach - sub.\ A' \subseteq A$ **then** $subsume \leftarrow subsume[(\ell, A') \mapsto (\ell, A)]$
10:         **for each** $e = (\ell, \rho, \ell') \in E$ **do**
11:             $A' \leftarrow sp_a(A, \rho)$; $worklist \leftarrow worklist \cup \{(\ell', A')\}$
12:             $parent((\ell', A')) = (\ell, A)$; $trans((\ell', A')) = e$
13:             **if** $e \in G$ **then**
14:                 **if** $\exists m \models pathCons$(path to (ℓ', A')) **then**
15:                     $G \leftarrow G - \{e\}$
16:                     $suite \leftarrow suite \cup \{$the sequence of values of r_ks in $m\}$
17:                 **else**
18:                   **if** $ct < t_b$ **then** $\triangleright$ ct has current time
19:                     $\pi \leftarrow \pi \cup$ REFINE$((\ell', A'))$; **goto 1**
20: $U = G - \{e \mid \exists s_a \in reach.\ trans(s_a) = e\}$ $\triangleright$ Unreachable goals
21: **return** ($suite$,$G - U$,U,π,($reach, parent, subsume, trans$))

---

error state. The abstract counterexample is spurious if there is no concrete execution that corresponds to the abstract counterexample. If the counterexample is not spurious then a bug has been found and the model checker terminates. In case of a spurious counterexample, the refinement procedure refines the abstract model. This is done by refining the abstraction to remove the spurious counterexample, and the process restarts with the newly refined abstraction. After a number of iterations, the abstract model may have no more counterexamples, which proves the correctness of the input program.

In this paper, we use predicate abstraction for model checking. Let π be a set of predicates, which are formulas over variables V. We assume that π always contains the predicate "false." We define abstraction and concretization functions α and γ between the concrete domain of all formulas over V, and the abstract domain of 2^π:

$$\alpha(\rho) = \{\varphi \in \pi \mid \rho \implies \varphi\} \qquad \gamma(A) = \bigwedge A,$$

where $A \subseteq \pi$, and ρ is a formula over V. An abstract state s_a of our program is an element of $Loc \times 2^\pi$. Given an abstract state (ℓ, A) and a program transition

(ℓ, ϕ, ℓ'), the abstract strongest post is defined as:

$$sp_a(A, \phi) = \alpha((\exists V. \; \gamma(A) \wedge \phi(V, V'))[V'/V]).$$

The abstraction is refined by adding predicates to π.

In Algorithm 2, we present the coverage-driven version of the CEGAR procedure. We do not declare error locations or transitions, instead the procedure takes goal transitions G as input along with a program $P = (V, Loc, \ell^I, E)$, predicates π, and a time budget t_b. Reachable states are collected in $reach$, while $worklist$ contains the frontier abstract states whose children are yet to be computed. The procedure maintains functions $parent$ and $trans$, such that if an abstract state s'_a is a child of a state s_a by a transition e, then $parent(s'_a) = s_a$ and $trans(s'_a) = e$. To guarantee termination, one needs to ensure that abstract states are not discovered repeatedly. Therefore, the procedure also maintains the $subsume$ function, such that $subsume((\ell, A)) = (\ell', A')$ only if $\ell = \ell'$ and $A \subseteq A'$. We write $sub = \{s \mid subsume(s) \neq \bot\}$ for the set of subsumed states. We denote the reflexive transitive closure of $parent$ and $subsume$, by $parent^*$ and $subsume^*$, respectively.

The algorithm proceeds as follows. Initially, all collecting data structures are empty, except $worklist$ containing the initial abstract state $(\ell^I, \varnothing)$. The loop at line 2 expands the reachability graph in every iteration. At lines 3–4, it chooses an abstract state (l, A) from $worklist$. If any ancestor of the state is already subsumed or the state is false, the state is discarded and the next state is chosen. Otherwise, (l, A) is added to $reach$. At lines 7–9, the $subsume$ function is updated. Afterwords, if (l, A) became subsumed then we proceed to choose another state from $worklist$. Otherwise, we create the children of (l, A) in the loop at line 10 by the abstract post sp_a. At line 12, $parent$ and $trans$ relations are updated. At line 13, the procedure checks if the abstract reachability has reached any of the goal transitions. If yes, then it checks the feasibility of the reaching path. If the path is found to be feasible, we add the feasible solution as a test to the suite at line 16. Otherwise, we refine and restart the reachability computation to remove the spurious path from the abstract reachability at lines 18–19. In case the algorithm has used its time budget, the refinement is not performed, but the algorithm continues processing the states remaining in $worklist$. As a consequence, the algorithm always returns a complete abstract reachability graph.

We do not discuss details of the REFINE procedure. The interested reader may read a more detailed exposition of CEGAR in [25].

Abstract reachability graph (ARG) The relations $parent$, $subsume$, and $trans$ together define an *abstract reachability graph (ARG)*, which is produced by ABSTRACTMC. A sequence of transitions $e_0, \ldots, e_{n-1}$ is a *path in an ARG* if there is a sequence of abstract state $s_0, \ldots, s_n \in reach$, such that

1. $s_0 = (\ell^I, \varnothing)$,
2. for $1 < i \leqslant n$ we have $parent(s_i) \in subsume^*(s_{i-1})$ and $e_{i-1} = trans(s_i)$.

Theorem 1 *Every feasible path of the program P is a path of an ARG. Moreover, every path in the ARG is a path of P.*

ABSTRACTMC returns a set *suite* of tests, set G of uncovered goals, proven unreachable goals U, set π of predicates, and the abstract reachability graph.

Lazy abstraction Model checkers often implement various optimizations in the computation of ARGs. One of the key optimization is lazy abstraction [26]. CEGAR may learn many predicates that lead to ARGs that are expensive to compute. In lazy abstraction, one observes that not all applications of sp_a require the same predicates. Let us suppose that the refinement procedure finds a new predicate that *must* be added in specific place along a spurious counterexample to remove this counterexample from future iterations. In other paths, however, this predicate may be omitted. This can be achieved by localizing predicates to parts of an ARG. Support for lazy abstraction can easily be added by additional data structures that record the importance of a predicate in different parts of programs.

5 Abstraction-driven Concolic Testing

In this section, we present our algorithm that combines concolic testing and model checking. The key idea is to use the ARG generated by a model checker to guide concolic testing to explore more likely feasible parts of programs.

We start by presenting the function MONITORFROMARG that converts an ARG into a monitor program. Let $\mathcal{A} = (reach, parent, subsume, trans)$ be an ARG. The *monitor of* $\mathcal{A}$ is defined as a program $\mathcal{M} = (V, reach - sub, (\ell^I, \varnothing), E_1 \cup E_2)$, where

- $E_1 = \{(s_a, e, s'_a) \mid s_a = parent(s'_a) \wedge e = trans(s'_a) \wedge s'_a \notin sub\}$,
- $E_2 = \{(s_a, e, s''_a) \mid \exists s'_a.\ s_a = parent(s'_a) \wedge e = trans(s'_a) \wedge \wedge s''_a \in subsume^+(s'_a) \wedge s''_a \notin sub\}$.

The transitions in E_1 are due to the child-parent relation, when the child abstract state is not subsumed. In case the child state s'_a is subsumed, then E_2 contains a transition from the parent of s'_a to the non-subsumed state s''_a in $subsume^+(s'_a)$, where $subsume^+$ denotes the transitive closure of $subsume$. From the way we built an ARG, it follows that the state s''_a is uniquely defined and the monitor is always deterministic.

In Algorithm 3 we present our method CRABS. CRABS takes as input a program P, a set G of goal branches to be covered, and time constraints: the total time limit t_b, and time budgets t_c, t_m for a single iteration of concolic testing and model checking, respectively. The algorithm returns a test suite for the covered goals, and a set of goals that are provably unreachable. The algorithm records in G the set of remaining goals. Similarly, U collects the goal branches that are proved unreachable by the model checker. The algorithm maintains a set π of predicates for abstraction, a program $\overline{P}$ for concolic testing, and a set $\overline{G}$ of goals

Algorithm 3 CRABS($P = (V, Loc, \ell^i, E)$, G, t_b, t_c, t_m)

Require: program $P = (V, Loc, \ell^i, E)$, branches $G \subseteq E$ to cover, time budget for concolic testing t_c, time budget for model checking t_m, total time budget t_b,
Ensure: a test suite, set of provably unreachable branches
1: $\pi \leftarrow \{false\}$; $U \leftarrow \varnothing$;   ▷ U is a set of provably unreachable goals
2: $suite \leftarrow \varnothing$ ▷ $suite$ is a set of test
3: $\overline{P} \leftarrow P$; $\overline{G} \leftarrow G$ ▷ program and goals for testing
4:
5: **while** $G \neq \varnothing$ and $ct < t_b$ **do**   ▷ ct always has current time.
6:    $(suite', \_) \leftarrow$ CONCOLICTEST$(\overline{P}, \overline{G}, ct + t_c)$
7:    $G \leftarrow G - \{g \in E \mid \exists tst \in suite'.tst$ covers $g\}$
8:    $suite \leftarrow suite \cup suite'$;
9:    **if** $G \neq \varnothing$ **then**
10:       $(suite', G, U', \pi, \mathcal{A}) \leftarrow$ ABSTRACTMC$(P, \pi, G, ct + t_m)$
11:       $suite \leftarrow suite \cup suite'$; $U \leftarrow U \cup U'$
12:       $\overline{P} \leftarrow \overline{P} \times$ MONITORFROMARG$(\mathcal{A})$ ▷ see sec. 5 for MonitorFromARG
13:       $\overline{G} = \{((\ell, \_), e, (\ell', \_)) \in E_{\overline{P}} \mid (\ell, e, \ell') \in G\}$
14: **return** $(suite, U)$

---

for concolic testing. The program $\overline{P}$ is initialized to the original program P, and in the following iterations becomes refined by the monitors. The algorithm collects in *suite* the tests generated by concolic testing and model checking.

The program $\overline{P}$ is a refinement of the original program P, so a single goal branch in P can map to many branches in the program $\overline{P}$. For this reason, we perform testing for the set $\overline{G}$ of all possible extensions of G to the branches in $\overline{P}$. For simplicity, in our algorithm concolic testing tries to reach all goals in $\overline{G}$, even if they map to the same goal branch in G. In the implementation, however, once concolic testing reaches a branch in $\overline{G}$, it removes all branches from $\overline{G}$ that have the same projection.

CRABS proceeds in iterations. At line 6, it first runs concolic testing on the program $\overline{P}$ and the goal branches $\overline{G}$ with the time budget t_c. The testing process returns a tests *suite'* and the set of remaining branches. Afterwords, if some branches remain to be tested, a model checker is called on the program P with predicates π, and a time budget t_m at line 10. As we discussed in the previous section, the model checker builds an abstract reachability graph (ARG), and produces tests if it finds concrete paths to the goal branches. Since the model checker runs for a limited amount of time, it returns an abstract reachability graph that may have abstract paths to the goal branches, but no concrete paths were discovered. Moreover, if the ARG does not reach some goal branch then it is certain that the branch is unreachable. The model checker returns a new set *suite'* of tests, remaining goals G, and a set U' of newly proved unreachable goals. Furthermore, it also returns a new set π of predicates for the next call to the model checker, and an abstract reachability graph $\mathcal{A}$. At line 12, we construct a monitor from $\mathcal{A}$ by calling MONITORFROMARG. We construct the next program $\overline{P}$ by taking a product of the current $\overline{P}$ with the monitor. We also

update $\overline{G}$ to the set of all extensions of the branches in G to the branches in $\overline{P}$. In the next iteration concolic testing is called on $\overline{P}$, which essentially explores the paths of P that are allowed by the monitors generated from the ARG. The algorithm continues until it runs out of time budget t_b or no more goals remain.

The program $\overline{P}$ for testing is refined in every iteration by taking a product with a new monitor. This ensures that $\overline{P}$ always becomes more precise, even if the consecutive abstractions do not strictly refine each other, i.e. the ARG from iteration i allows the set $\mathcal{L}$ of paths, while the ARG from iteration $i + 1$ allows the set $\mathcal{L}'$ such that $\mathcal{L}' \nsubseteq \mathcal{L}$. This phenomenon occurs when the model checker follows the lazy abstraction paradigm, described in Section 4. In lazy abstraction, predicates are applied locally and some may be lost due to refinement. As a consequence, program parts that were pruned from an ARG may appear again in some following ARG. Another reason for this phenomenon may be a deliberate decision to remove some predicates when the abstraction becomes too expensive to maintain.

6 Experiments

We implemented our approach in a tool Crabs, built on top of the concolic tester Crest [11] and the model checker CpaChecker [8]. In our experiments, we observed an improvement in branch coverage compared to Crest from 48% to 63% in the best case, and from 66% to 71% on average.

Benchmarks We evaluated our approach on a collection of programs: 1) a set of hand-crafted examples (listed in Appendix), 2) example "nsichneu" [24] described in Section 2 with varying number of loop iterations, 3) benchmarks "parport" and "cdaudio1" from various categories of SvComp [6], 4) all 13 benchmarks from the "ddv-machzwd" SvComp category.

Optimizations Constructing an explicit product of an program and a monitor would be cumbersome, due to complex semantics of the C language, e.g. the type system and scoping rules. To avoid this problem, our tool explores the product on-the-fly, by keeping track of the program and monitor state. We have done minor preprocessing of the examples, such that they can be parsed by both Crest and CpaChecker. Furthermore, CpaChecker does not deal well with arrays, so in the "nsichneu" example we replaced arrays of fixed size (at most 6) by a collection of variables.

Comparison of heuristics and tools We compare our tool with four other heuristics for guiding concolic search that are implemented in Crest : the depth-first search (DFS), random branch search (RndBr), uniform random search (UnfRnd), and CFG-guided search; for details see [11]. The depth-first search is a classical way of traversing a tree of program paths. In the random branch search, the branch to be flipped is chosen from all the branches on the current execution with equal probability. Similarly, in the uniform random search the branch to be

flipped is also picked at random, but the probability decreases with the position of the branch on the execution. In the CFG-guided heuristic the test process is guided by a distance measure between program branches, which is computed statically on the control-flow graph of the program. This heuristic tries to drive exploration in into branches that are closer to the remaining test goals. The concolic component of our tool uses the CFG-guided heuristic to explore the product of a program and a monitor; this way branches closer in the monitor are explored first. In Appendix we show experimental results for our tool with the other concolic heuristics, and we demonstrate that our approach improves coverage for each of them.

We compared our approach with the tool FSHELL [27], which is based on the bounded model checker CBMC. FSHELL unwinds the control-flow graph until it fully explores all loop iterations and checks satisfiability of paths that hit the testing goals. This tool does not return a test suite, unless all loops are fully explored.

Experimental setup All the tools were run with branch coverage as the test objective. The coverage of a test suite is measured by the ratio $\frac{c}{r}$, where c is the number of branches covered by a test suite, and r is the number of branches that have not been proved unreachable. For CREST, we set r to be the number of branches that are reachable in the control-flow graph by graph search, which excludes code that is trivially dead. Our tool and CREST have the same number of test goals, while FSHELL counts more test goals on some examples. We run our tool in a configuration, where testing takes approximately 80% of the time budget. All experiments were performed on a machine with an AMD Opteron 6134 CPU and a memory limit of 12GB, and were averaged over three runs.

Results The experimental evaluation for a time budget of one hour is presented in Table 1; for more detailed results see Appendix.

After one hour, our tool achieved the highest coverage on most examples. The best case is "nsichneu(17)," where our tool achieved 63% coverage compared to 48% by the best other tool. We show in Appendix, that if we run our tool with the DFS heuristic, we obtain even higher coverage of 69%. The hand-crafted examples demonstrate that our method, as well as FSHELL, can reach program parts that are difficult to cover for concolic testing. In the benchmark category, our tool obtained average coverage of 71% compared to 66% by CREST. In many examples, we obtain higher coverage by both reaching more goals and proving that certain goals are unreachable. FSHELL generated test suites only for three examples, since on other examples it was not able to fully unwind program loops.

Example		Crabs-CFG (this paper)	Crest-DFS[33,35]	Crest-CFG[11]	Crest-UnfRnd[11]	Crest-RndBr[11]	Fshell[27]
name	branches	coverage	coverage	coverage	coverage	coverage	coverage
simple-while	12	12/12 (100%)	11/12 (91.2%)	11/12 (91.2%)	11/12 (91.2%)	11/12 (91.2%)	12/12 (100%)
branches	12	12/12 (100%)	9/12 (75%)	7/12 (58.3%)	12/12 (91.2%)	12/12 (91.2%)	12/12 (100%)
unreach	10	9/9 (100%)	9/10 (90%)	9/10 (90%)	9/10 (90%)	9/10 (90%)	9/9 (100%)
nsichneu(2)	5786	3843/5753 (66.8%)	**5365/5786 (92.7%)**	3098/5786 (53.5%)	2559/5786 (44.2%)	2196/5786 (38.0%)	4520/5786 (78.1%)
nsichneu(9)	5786	3720/5756 (64.6%)	**4224/5786 (73.0%)**	2843/5786 (49.1%)	2493/5786 (43.1%)	2187/5786 (37.8%)	1261/5786 (21.8%)
nsichneu(17)	5786	**3619/5746 (63.0%)**	2086/5786 (36.1%)	2758/5786 (47.7%)	2476/5786 (42.8%)	2161/5786 (37.3%)	TO
parport	920	**215/598 (35.9%)**	215/920 (23.4%)	215/920 (23.4%)	215/920 (23.4%)	215/920 (23.4%)	TO
cdaudio1	340	248/249 (99.6%)	250/340 (73.5%)	250/340 (73.5%)	246/340 (72.3%)	250/340 (73.5%)	**266/266 (100%)**
ddv_outb	206	**137/194 (70.8%)**	78/206 (37.9%)	136/206 (66.2%)	111/206 (54.2%)	135/206 (65.7%)	TO
ddv_pthread	200	**134/189 (71.3%)**	73/200 (36.7%)	131/200 (65.5%)	109/200 (54.8%)	130/200 (65.2%)	TO
ddv_outwp	200	**134/189 (70.8%)**	73/200 (36.7%)	131/200 (65.5%)	107/200 (53.8%)	129/200 (64.7%)	TO
ddv_allfalse	214	**143/199 (72.0%)**	83/214 (38.9%)	141/214 (66.0%)	123/214 (57.6%)	140/214 (65.6%)	TO
ddv_inwp	200	**134/189 (70.7%)**	76/200 (38.2%)	131/200 (65.5%)	108/200 (54.2%)	129/200 (64.8%)	TO
ddv_inbp	200	**133/189 (70.3%)**	73/200 (36.5%)	130/200 (65.3%)	109/200 (54.5%)	130/200 (65.2%)	TO
ddv_outlp	200	**133/189 (70.1%)**	73/200 (36.5%)	130/200 (65.2%)	109/200 (54.7%)	130/200 (65.2%)	TO
ddv_outbp	200	**134/188 (71.2%)**	73/200 (36.5%)	130/200 (65.0%)	106/200 (53.3%)	130/200 (65.0%)	TO
ddv_inl	200	**134/190 (70.7%)**	89/200 (44.8%)	131/200 (65.8%)	109/200 (54.7%)	129/200 (64.8%)	TO
ddv_inlp	200	**134/189 (71.1%)**	75/200 (37.8%)	131/200 (65.5%)	108/200 (54.3%)	129/200 (64.8%)	TO
ddv_inw	206	**139/194 (72.0%)**	80/206 (39.2%)	136/206 (66.3%)	112/206 (54.5%)	135/206 (65.7%)	TO
ddv_inb	200	**133/189 (70.5%)**	73/200 (36.5%)	130/200 (65.3%)	114/200 (57.3%)	130/200 (65.2%)	TO
ddv_outl	200	**133/189 (70.5%)**	73/200 (36.5%)	131/200 (65.5%)	109/200 (54.8%)	131/200 (65.7%)	TO

Table 1: Experimental results for one hour. RndBr stands for "random branch search" and UnfRnd for "uniform random search." TO means that no suite was generated before the time limit.

7 Related Work

Testing literature is rich, so we only highlight the most prominent approaches. Random testing [32,13,9] can cheaply cover shallow parts of the program, but it may quickly reach a plateau where coverage does not increase. Another testing method is to construct symbolic objects that represent complex input to a program [36,37]. In [10] objects for testing program are systematically constructed up to a given bound. The approach of [18] tests a concurrent program by exploring schedules using partial-order reduction techniques.

Concolic testing suffers from the path-explosion problem, so various search orders testing have been proposed, several of them are discussed in Section 6. In [20] multiple input vectors are generated from a single symbolic path by negating constraints on the path one-by-one, which allows the algorithm to exercise paths at different depths of the program. Hybrid concolic testing [30] uses random testing to quickly reach deep program statements and then concolic testing to explore the close neighborhood of that point.

Our work is closest related to SYNERGY [22,5,21]. SYNERGY is an approach for verification of safety properties that maintains a program abstraction and a forest of tested paths. Abstract error traces are ordered such that they follow some tested execution until the last intersection with the forest. If an ordered abstract trace is feasible, then a longer concrete path is added to the forest; otherwise, the abstraction is refined. Compared to SYNERGY our method has several key differences. First, in SYNERGY model checking and test generation work as a single process, while in our approach these components are independent and communicate only by a monitor. Second, unlike us, SYNERGY does not pass the complete abstract model of the program to concolic testing, where the testing heuristics guides the search. Finally, in our approach we can configure the ratio of model checking to testing, while in SYNERGY every unsuccessful execution leads to refinement.

Another related work is [12], where concolic testing is guided towards program parts that a static analyzer was not able to verify. In contrast to our approach, the abstraction is not refined. In [17] conditional model checking is used to generate a residual that represents the program part that has been left unverified; the residual is then tested.

The work of [34] applies program analysis to identify control locations in a concurrent program that are relevant for reaching the target state. These locations guide symbolic search toward the target and predicates in failed symbolic executions are analyzed to find new relevant locations. The CHECK'N'CRASH [15] tool uses a constraint solver to reproduce and check errors found by static analysis of a program. In [16] the precision of static analysis was improved by adding a dynamic invariant detection .

The algorithm of [31] presents a testing method, where a program is simplified by replacing function calls by unconstrained input. Spurious counterexample are removed in a CEGAR loop by lazily inserting function bodies. In contrast, our method performs testing on a concrete program and counterexamples are always sound.

A number of papers consider testing program abstraction with bounded model checking (BMC). If the abstraction is sufficiently small, then a program invariant can be established by exhaustively testing the abstraction with BMC. In [29] a Boolean circuit is abstracted, such that it decreases the bound that needs to be explored in an exhaustive BMC search. In [23] BMC is run on an abstract model up to some bound. If the invariant is not violated, then the model is replaced by an unsat core and the bound is incremented. If a spurious counterexample is found, then clauses that appear in the unsat core are added to the abstraction.

8 Conclusion

We presented an algorithm that combines model checking and concolic testing synergistically. Our method iterativly runs concolic testing and model checking, such that concolic testing is guided by a program abstraction, and the abstraction is refined for the remaining test goals. Our experiments demonstrated that the presented method can increase branch coverage compared to both concolic testing, and test generation based on model checking.

We also observed that our method is highly sensitive to optimizations and heuristics available in the model checker. For instance, lazy abstraction allows the model checker to get pass bottlenecks created due to over-precision in some parts of ARGs. However, lazy abstraction may lead to a monitor that is less precise than the monitors of the past iterations, which may lead to stalled progress in covering new goals by our algorithm. In the future work, we will study such complimentary effects of various heuristics in model checkers to find the optimal design of model checkers to assist a concolic-testing tool. We believe that adding this feature will further improve the coverage of our tool.

References

1. CRABS tool. `http://pub.ist.ac.at/~przemek/crabs_tool.html`.
2. Radio Technical Commission for Aeronautics. `www.rtca.org`.
3. T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL*, 2002.
4. A. Banerjee, S. Chattopadhyay, and A. Roychoudhury. Static analysis driven cache performance testing. In *RTSS*, pages 319–329, 2013.
5. N. E. Beckman, A. V. Nori, S. K. Rajamani, and R. J. Simmons. Proofs from tests. In *ISSTA*, pages 3–14, 2008.
6. D. Beyer. Software verification and verifiable witnesses - (report on SV-COMP 2015). In *TACAS*, pages 401–416, 2015.
7. D. Beyer, A. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. Generating tests from counterexamples. In A. Finkelstein, J. Estublier, and D. S. Rosenblum, editors, *ICSE*, pages 326–335. IEEE Computer Society, 2004.
8. D. Beyer and M. E. Keremoglu. CPAchecker: A tool for configurable software verification. In *CAV*, pages 184–190, 2011.
9. D. L. Bird and C. U. Munoz. Automatic generation of random self-checking test cases. *IBM Systems Journal*, 22(3):229–245, 1983.
10. C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on java predicates. In *ISSTA*, pages 123–133, 2002.
11. J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *ASE*, pages 443–446, 2008.
12. M. Christakis, P. Müller, and V. Wüstholz. Guiding dynamic symbolic execution toward unverified program executions. Technical report, ETH Zurich, 2015.
13. I. Ciupa, A. Leitner, M. Oriol, and B. Meyer. Artoo: adaptive random testing for object-oriented software. In W. Schäfer, M. B. Dwyer, and V. Gruhn, editors, *ICSE*, pages 71–80. ACM, 2008.
14. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV*, 2000.
15. C. Csallner and Y. Smaragdakis. Check 'n' crash: combining static checking and testing. In *ICSE*, pages 422–431, 2005.
16. C. Csallner, Y. Smaragdakis, and T. Xie. DSD-Crasher: A hybrid analysis tool for bug finding. *ACM Trans. Softw. Eng. Methodol.*, 17(2), 2008.
17. M. Czech, M. C. Jakobs, and H. Wehrheim. Just test what you cannot verify! In A. Egyed and I. Schaefer, editors, *Fundamental Approaches to Software Engineering*, volume 9033, pages 100–114. Springer, 2015.
18. P. Godefroid. Model checking for programming languages using verisoft. In *POPL*, pages 174–186, 1997.
19. P. Godefroid. Compositional dynamic test generation. In *POPL*, pages 47–54, 2007.
20. P. Godefroid, M. Y. Levin, and D. A. Molnar. Automated whitebox fuzz testing. In *NDSS*. The Internet Society, 2008.
21. P. Godefroid, A. V. Nori, S. K. Rajamani, and S. Tetali. Compositional may-must program analysis: unleashing the power of alternation. In *POPL*, pages 43–56, 2010.
22. B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. Synergy: a new algorithm for property checking. In *SIGSOFT FSE*, pages 117–127, 2006.

23. A. Gupta and O. Strichman. Abstraction refinement for bounded model checking. In *CAV*, pages 112–124, 2005.

24. J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. The Mälardalen WCET benchmarks – past, present and future. In B. Lisper, editor, *WCET*, pages 137–147, Brussels, Belgium, July 2010. OCG.

25. T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *POPL*, 2004.

26. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, 2002.

27. A. Holzer, C. Schallhart, M. Tautschnig, and H. Veith. Fshell: Systematic test case generation for dynamic analysis and measurement. In *CAV*, pages 209–213, 2008.

28. A. Holzer, C. Schallhart, M. Tautschnig, and H. Veith. How did you specify your test suite. In C. Pecheur, J. Andrews, and E. D. Nitto, editors, *ASE*, pages 407–416. ACM, 2010.

29. D. Kroening. Computing over-approximations with bounded model checking. *Electr. Notes Theor. Comput. Sci.*, 144(1):79–92, 2006.

30. R. Majumdar and K. Sen. Hybrid concolic testing. In *ICSE*, ICSE '07, pages 416–426, Washington, DC, USA, 2007. IEEE Computer Society.

31. R. Majumdar and K. Sen. Latest : Lazy dynamic test input generation. Technical Report UCB/EECS-2007-36, EECS Department, University of California, Berkeley, 2007.

32. C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *ICSE*, pages 75–84. IEEE Computer Society, 2007.

33. N. K. Patrice Godefroid and K. Sen. Dart: directed automated random testing. In *PLDI*, pages 213–223. ACM, 2005.

34. N. Rungta, E. G. Mercer, and W. Visser. Efficient testing of concurrent programs with abstraction-guided symbolic execution. In *Model Checking Software, 16th International SPIN Workshop, Grenoble, France, June 26-28, 2009. Proceedings*, LNCS, pages 174–191, 2009.

35. K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *ESEC/SIGSOFT FSE*, pages 263–272, 2005.

36. W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with Java PathFinder. In G. S. Avrunin and G. Rothermel, editors, *ISSTA*, pages 97–107. ACM, 2004.

37. T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *TACAS*, pages 365–381, 2005.

Appendix

Experiments for concolic search orders

Our approach is independent of the search order used in concolic testing. We modified three search orders in CREST to work with a monitor: DFS, CFG and RndBr. The experimental evaluation is shown in Tables 2 and 3. The results suggest that using the information from a monitor increases test coverage for all three heuristics.

Example		Crabs-DFS (this paper)	Crest-DFS[33,35]	Crabs-CFG (this paper)	Crest-CFG[11]	Crabs-RndBr (this paper)	Crest-RndBr[11]
name	branches	coverage	coverage	coverage	coverage	coverage	coverage
nsichneu(2)	5786	**5329/5765 (92.4%)**	5264/5786 (91.0%)	3351/5763 (58.1%)	2807/5786 (48.5%)	2344/5756 (40.7%)	2143/5786 (37.0%)
nsichneu(9)	5786	**3332/5771 (57.7%)**	2086/5786 (36.1%)	2978/5761 (51.7%)	2656/5786 (45.9%)	2269/5764 (39.4%)	2128/5786 (36.8%)
nsichneu(17)	5786	**2949/5776 (51.1%)**	1453/5786 (25.1%)	2842/5764 (49.3%)	2546/5786 (44.0%)	2305/5758 (40.0%)	2117/5786 (36.6%)
ddv_allfalse	214	87/208 (42.0%)	83/214 (38.9%)	**142/205 (69.3%)**	134/214 (62.9%)	139/203 (68.8%)	131/214 (61.5%)
ddv_outlp	200	81/195 (41.4%)	73/200 (36.5%)	**133/189 (70.1%)**	123/200 (61.7%)	125/191 (65.7%)	121/200 (60.7%)
ddv_inl	200	76/194 (39.4%)	89/200 (44.8%)	**132/189 (69.9%)**	126/200 (63.3%)	130/192 (67.5%)	119/200 (59.8%)
ddv_pthread	200	76/195 (39.1%)	73/200 (36.7%)	**132/189 (69.9%)**	122/200 (61.0%)	130/190 (68.7%)	122/200 (61.0%)
ddv_outl	200	84/194 (43.6%)	73/200 (36.5%)	**132/190 (69.7%)**	117/200 (58.7%)	130/189 (68.8%)	125/200 (62.5%)
ddv_inlp	200	77/194 (39.9%)	73/200 (36.7%)	**132/190 (69.7%)**	122/200 (61.0%)	129/190 (67.9%)	121/200 (60.5%)
ddv_outwp	200	84/194 (43.6%)	73/200 (36.7%)	**133/190 (69.9%)**	122/200 (61.2%)	129/188 (68.7%)	124/200 (62.0%)
ddv_inbp	200	88/194 (45.4%)	73/200 (36.5%)	**132/190 (69.7%)**	120/200 (60.0%)	129/189 (68.4%)	120/200 (60.2%)
ddv_outbp	200	76/194 (39.2%)	73/200 (36.5%)	117/189 (62.0%)	119/200 (59.8%)	**131/190 (69.2%)**	119/200 (59.5%)
ddv_inw	206	83/200 (41.5%)	80/206 (39.2%)	**137/195 (70.3%)**	124/206 (60.5%)	136/196 (69.6%)	125/206 (60.7%)
ddv_outb	206	87/199 (43.7%)	78/206 (37.9%)	**136/195 (69.8%)**	130/206 (63.4%)	131/196 (67.1%)	129/206 (62.8%)
ddv_inwp	200	83/195 (42.5%)	76/200 (38.2%)	**132/189 (70.1%)**	122/200 (61.2%)	129/191 (67.5%)	118/200 (59.2%)
ddv_inb	200	85/194 (44.2%)	73/200 (36.5%)	**131/189 (69.7%)**	122/200 (60.5%)	128/189 (67.7%)	121/200 (60.8%)

Table 2: Detailed experimental results for 15m. RndBr stands for "random branch search."

Example		Crabs-DFS (this paper)	Crest-DFS[33,35]	Crabs-CFG (this paper)	Crest-CFG[11]	Crabs-RndBr (this paper)	Crest-RndBr[11]
name	branches	coverage	coverage	coverage	coverage	coverage	coverage
nsichneu(2)	5786	**5406/5758 (93.9%)**	5365/5786 (92.7%)	3843/5753 (66.8%)	3098/5786 (53.5%)	2565/5738 (44.7%)	2196/5786 (38.0%)
nsichneu(9)	5786	4042/5755 (70.2%)	**4224/5786 (73.0%)**	3720/5756 (64.6%)	2843/5786 (49.1%)	2474/5742 (43.1%)	2187/5786 (37.8%)
nsichneu(17)	5786	**3951/5761 (68.6%)**	2086/5786 (36.1%)	3619/5746 (63.0%)	2758/5786 (47.7%)	2460/5746 (42.8%)	2161/5786 (37.3%)
ddv_allfalse	214	87/207 (42.2%)	83/214 (38.9%)	**143/199 (72.0%)**	141/214 (66.0%)	142/199 (71.4%)	140/214 (65.6%)
ddv_outlp	200	78/193 (40.7%)	73/200 (36.5%)	**133/189 (70.1%)**	130/200 (65.2%)	132/188 (70.0%)	130/200 (65.2%)
ddv_inl	200	80/193 (41.7%)	89/200 (44.8%)	**134/190 (70.7%)**	131/200 (65.8%)	132/188 (70.0%)	129/200 (64.8%)
ddv_pthread	200	78/191 (41.1%)	73/200 (36.7%)	**134/189 (71.3%)**	131/200 (65.5%)	132/190 (69.5%)	130/200 (65.2%)
ddv_outl	200	102/192 (53.1%)	73/200 (36.5%)	**133/189 (70.5%)**	131/200 (65.5%)	131/189 (69.6%)	131/200 (65.7%)
ddv_inlp	200	79/193 (40.9%)	75/200 (37.8%)	**134/189 (71.1%)**	131/200 (65.5%)	133/188 (70.6%)	129/200 (64.8%)
ddv_outwp	200	82/191 (42.9%)	73/200 (36.7%)	**134/189 (70.8%)**	131/200 (65.5%)	131/188 (69.9%)	129/200 (64.7%)
ddv_inbp	200	90/191 (47.5%)	73/200 (36.5%)	**133/189 (70.3%)**	130/200 (65.3%)	132/189 (69.8%)	130/200 (65.2%)
ddv_outbp	200	77/192 (40.1%)	73/200 (36.5%)	**134/188 (71.2%)**	130/200 (65.0%)	132/188 (70.3%)	130/200 (65.0%)
ddv_inw	206	91/198 (46.0%)	80/206 (39.2%)	**139/194 (72.0%)**	136/206 (66.3%)	137/194 (70.4%)	135/206 (65.7%)
ddv_outb	206	94/195 (48.2%)	78/206 (37.9%)	137/194 (70.8%)	136/206 (66.2%)	**137/193 (71.2%)**	135/206 (65.7%)
ddv_inwp	200	87/191 (45.5%)	76/200 (38.2%)	**134/189 (70.7%)**	131/200 (65.5%)	132/188 (70.0%)	129/200 (64.8%)
ddv_inb	200	82/192 (42.8%)	73/200 (36.5%)	**133/189 (70.5%)**	130/200 (65.3%)	132/189 (69.8%)	130/200 (65.2%)

Table 3: Detailed experimental results for 1h.

Hand-craft program

We present hand-craft example that were used for experimental evaluation.

```
const int B=30;

int main(void) {
  int i=0;
  int x;
  int b = 0;
  while (i<B){
    x = input();
    // we distinguish two cases
    // so that corner cases of DFS
    //don't give full coverage
    if (i==6 && x== 4){b=1;}

    if (i!=6 && x!= 4){b=1;}

    i++;
  }
  if (b==0) {printf("Goal!\n");}
}
```

Fig. 4: The "simple-while" example.

```
const int N=30;

// function that takes long time to analyze
int foo(int y){
  int i,c=0;
  for (int j=0;j<N;j++){
    i = input();
    if (i == c)
      c++;
  }
  return c;
}

void bar(int y){
  if (y<20){
    printf("Goal one\n");
  } else {
    printf("Goal two\n");
  }
}

int main(void) {
  int y,x;
  y = input();
  if (y>0){
    x = foo(y);
  } else {
    x = foo(y+1);
  }
  if (x==5 && y>10){
    // discover that can be reached only if y>0
    bar(y);
  }
}
```

Fig. 5: The "branches" example.

```
const int N=30;

// function that takes long time to analyze
int h(int y){
  int i,c=0;
  for (int j=0;j<N;j++){
    i = input();
    if (i == c)
      c++;
  }

  return c;
}

// some library call
void call(int i){
  if (i<0){
    printf("Goal unreach\n");
  } else {
    printf("Goal two\n");
  }
}

int main(void) {
  int i,j,r=0;
  i = 5;
  while(i>0){
    int tmp = h(i);
    if (tmp > r){
      r = tmp;
    }
    i--;
  }
  call(r);
}
```

Fig. 6: The "unreach" example.