CoVeriTest: Cooperative Verifier-Based Testing

Dirk Beyer and Marie-Christine Jakobs

LMU Munich, Munich, Germany

Abstract. Testing is a widely used method to assess software quality. Coverage criteria and coverage measurements are used to ensure that the constructed test suites adequately test the given software. Since manually developing such test suites is too expensive in practice, various automatic test-generation approaches were proposed. Since all approaches come with different strengths, combinations are necessary in order to achieve stronger tools. We study cooperative combinations of verification approaches for test generation, with high-level information exchange.
We present CoVeriTest, a hybrid approach for test-case generation, which iteratively applies different conditional model checkers. Thereby, it allows to adjust the level of cooperation and to assign individual time budgets per verifier. In our experiments, we combine explicit-state model checking and predicate abstraction (from CPAchecker) to systematically study different CoVeriTest configurations. Moreover, CoVeriTest achieves higher coverage than state-of-the-art test-generation tools for some programs.

Keywords: Test-case generation · Software testing · Test coverage · Conditional model checking · Cooperative verification · Model checking

1 Introduction

Testing is a commonly used technique to measure the quality of software. Since manually creating such test suites is laborious, automatic techniques are used: e.g., model-based techniques for black-box testing and techniques based on control-flow coverage for white-box testing. Many automatic techniques have been proposed, ranging from random testing [36,57] and fuzzing [26,52,53], over search-based testing [55] to symbolic execution [23,24,58] and reachability analyses [5,12,45,46]. The latter are well-suited to find bugs and derive test suites that achieve high coverage, and several verification tools support test generation (e.g., Blast [5], PathFinder [61], CPAchecker [12]). The reachability checks for all test goals seem too expensive, but in practice, those approaches can be made pretty efficient.

Encouraged by tremendous advances in software verification [3] and a recent case study that compared model checkers with test tools w.r.t. bug finding [17], we study a new kind of combination of reachability analyses for test generation. Combinations are necessary because different analysis techniques have different strength and weaknesses. For example, consider function `foo` in Listing 1. Explicit state model checking [18,33] tracks the values of variables i and s and easily

R. Hähnle and W. van der Aalst (Eds.): FASE 2019, LNCS 11424, pp. 389–408, 2019.
https://doi.org/10.1007/978-3-030-16722-6_23

detects the reachability of the statements in the outermost `if` branch (lines 3–6), while it has difficulties with the complex condition in the else-branch (line 8). In contrast, predicate abstraction [33,39] can easily derive test values for the complex condition in line 8, but to handle the `if` branch (lines 3–6) it must spent effort on the detection of the predicates $s = 0$, $s = 1$, and $i = 0$. Independently of each other, test approaches [1,34,47,54] and verification approaches [9,10,29,37] employ combinations to tackle such problems. However, there are no approaches yet that combine different reachability analyses for test generation.

```
0 void foo(int i, int n) {
1   int s=0;
2   if(i==0)
3     while (i==0)    {
4       if(s==0) init();
5       if(s==1) i = exec();
6       s=(s+1)%2;
7     }
8   else if(2*i<n && i>0) exec();
9 }
```

Fig. 1. Example program foo

Inspired by abstraction-driven concolic testing [32], which interleaves concolic execution and predicate abstraction, we propose CoVeriTest, which stands for cooperative verifier-based testing. CoVeriTest iteratively executes a given sequence of reachability analyses. In each iteration, the analyses are run in sequence and each analysis is limited by its individual, but configurable time limit. Furthermore, CoVeriTest allows the analysis to share various types of analysis information, e.g., which paths are infeasible, have already been explored, or which abstraction level to use. To get access to a large set of reachability analyses, we implemented CoVeriTest in the configurable software-analysis framework CPAchecker [15]. We used our implementation to evaluate different CoVeriTest configurations on a large set of well-established benchmark programs and to compare CoVeriTest with existing state-of-the-art test-generation techniques. Our experiments confirm that reachability analyses are valuable for test generation.

Contributions. In summary, we make the following contributions:

- We introduce CoVeriTest, a flexible approach for high-level interleaving of reachability analyses with information exchange for test generation.
- We perform an extensive evaluation of CoVeriTest studying 54 different configurations and two state-of-the-art test-generation tools[1].
- CoVeriTest and all our experimental data are publically available[2] [13].

2 Testing with Verifiers

The basic idea behind testing with verifiers is to derive test cases from counter-examples [5,61]. Thus, meeting a test goal during verification has to trigger a specification violation. First, we remind the reader of some basic notations.

[1] We choose the best two tools VeriFuzz and Klee from the international competition on software testing (Test-Comp 2019) [4]. https://test-comp.sosy-lab.org/2019/

[2] https://www.sosy-lab.org/research/coop-testgen/

Programs. Following literature [9], we represent programs by control-flow automata (CFAs). A CFA $P = (L, \ell_0, G)$ consists of a set L of program locations (the program-counter values), an initial program location $\ell_0 \in L$, and a set of control-flow edges $G \subseteq L \times Ops \times L$. The set Ops describes all possible operations, e.g., assume statements (resulting from conditions in `if` or `while` statements) and assignments. For the program semantics, we rely on an operational semantics, which we do not further specify.

Abstract Reachability Graph (ARG). ARGs record the work done by reachability analyses. An ARG is constructed for a program $P = (L, \ell_0, G)$ and stores (a) the abstract state space that has been explored so far, (b) which abstract states must still be explored, and (c) what abstraction level (tracked variables, considered predicates, etc.) is used. Technically, an ARG is a five-tuple $(N, succ, root, F, \pi)$ that consists of a set N of abstract states, a special node $root \in N$ that represents the initial states of program P, a relation $succ \subseteq N \times G \times N$ that records already explored successor relations, a set $F \subseteq N$ of frontier nodes, which remembers all nodes that have not been fully explored, and a precision π describing the abstraction level. Every ARG must ensure that a node n is either contained in F or completely explored, i.e., all abstract successors have been explored. We use ARGs for information exchange between reachability analyses.

Test Goals. In this paper, we are interested in structural coverage, e.g., branch coverage. Transferred to our notion of programs, this means that our test goals are a subset of the program's control-flow edges. For using a verifier to generate tests, we have to encode the test goals as a specification violation. Figure 2 shows a possible encoding, which uses a protocol automaton. Whenever a test goal is executed, the automaton transits from the initial, safe state q_0 to the accepting state q_e, which marks a property violation. Note that reachability analyses, which we consider for test generation, can easily monitor such specifications during exploration.

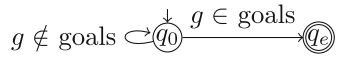


Fig. 2. Encoding test goals as specification violation

Now, we have everything at hand to describe how reachability analyses generate tests. Algorithm 1 shows the test-generation process. The algorithm gets as input a program, a set of test goals, and a time limit for test generation. For cooperative test generation, we need to guide state-space explorations. To this end, we also provide an initial ARG and a condition. A condition is a concept known from conditional model checking [10] and describes which parts of the state space have already been explored by other verifiers. A verifier, e.g., a reachability analysis, can use a condition to ignore the already explored parts of the state space. Verifiers that do not understand conditions can safely ignore them.

At the beginning, Alg. 1 sets up the data structures for the test suite and the set of covered goals. To set up the specification, it follows the idea of Fig. 2. As long as not all test goals are covered, there exist abstract states that must be explored, and the time limit has not elapsed, the algorithm tries to generate new tests. Therefore, it resumes the exploration of the current ARG [5] taking into

Algorithm 1. Generating tests with a (conditional) reachability analysis

Input: `prog` $= (L, \ell_0, G)$, `goals` $\subseteq G$, `limit` $\in \mathbb{N}$, `arg` =(N,succ, root, F, π), condition ψ
Output: generated `test_suite`, `covered` goals, updated `arg`

```
1: test_suite=∅;  covered=∅;
2: φ=generate_specification(goals);

3: while (goals ≠ ∅ and arg.F ≠ ∅ and elapsed_time<limit) do
4:     arg = explore(prog, φ, arg, ψ, limit − elapsed_time);

5:     if (arg.F ≠ ∅ and elapsed_time<limit) then
6:         τ = extract_counterexample_trace(arg);
7:         test_suite = test_suite ∪ generate_test_from_trace(τ);

8:         goals = goals\{last_edge(τ)};  covered = covered ∪ {last_edge(τ)}

9:     φ=generate_specification(goals);
10: return (test_suite, covered, arg);
```

account program `prog`, specification φ, and (if understood) the condition ψ. If the exploration stops, then it returns an updated ARG. Exploration stops due to one of three reasons: (1) the state space is explored completely ($F = \emptyset$), (2) the time limit is reached, or (3) a counterexample has been found.[3] In the latter case, a new test is generated. First, a counterexample trace is extracted from the ARG. The trace describes a path through the ARG that starts at the root and its last edge is a test goal (the reason for the specification violation). Next, a test is constructed from the path and added to the test suite. Basically, the path is converted into a formula and a satisfying assignment[4] is used as the test case. For the details, we refer the reader to the work that defined the method [5]. Additionally, the covered goal (last edge on the counterexample path) is removed from the set of open test goals and added to the set of covered goals. Finally, the specification is updated to no longer consider the covered goal. When the algorithm finishes, it returns the generated test suite, the set of covered goals and the last ARG considered. The ARG is returned to enable cooperation.

3 CoVeriTest

The previous section described how to use a single reachability analysis to produce tests for covering a set of test goals. Due to different strengths and weaknesses, some test goals are harder to cover for one analysis than for another. To

[3] We assume that an exploration is only complete if no counterexample exists.
[4] We assume that only feasible counterexamples are contained and infeasible counterexamples were eliminated by the reachability analysis during exploration.

Algorithm 2. CoVeriTest: alternating reachability analyses to generate tests

Input: `prog` $= (L, \ell_0, G)$, `goals` $\subseteq G$, `total_limit` $\in \mathbb{N}$, `configs` $\in (\text{analysis} \times \mathbb{N})^+$
Output: `test_suite`

```
 1: test_suite=∅;  args=⟨⟩;  current=0;
 2: while (goals ≠ ∅ and elapsed_time<total_limit) do
 3:     analysis = configs[current].first;  limit = configs[current].second;

 4:     (arg,ψ) = cooperateAndInit(prog, args, configs.length);
 5:     (tests, covered, arg) = analysis(prog, goals, limit, arg, ψ);

 6:     test_suite=test_suite ∪ tests; goals=goals\covered; args=args ∘⟨arg⟩;
 7:     if (arg.F=∅) then
 8:         return test_suite;
 9:     current = (current+1) % configs.length;
10: return test_suite;
```

maximize the number of covered goals, different analyses should be combined. In CoVeriTest, we rotate analyses for test generation. Thus, we avoid that analyses try to cover the same goal in parallel and we do not need to know in advance which analysis can cover which goals. Moreover, analyses that get stuck trying to cover goals that other analyses handle later, get a chance to recover. Additionally, CoVeriTest supports cooperation among analyses. More concrete: analyses may extract and use information from ARGs constructed by previous analysis runs.

Algorithm 2 describes the CoVeriTest workflow. It gets four inputs. Program, test goals, and time limit are already known from Alg. 1 (test generation with a single analysis). Additionally, CoVeriTest gets a sequence of configurations, namely pairs of reachability analysis and time limit. The time limit accompanied with the analysis restricts the runtime of the respective analysis per call (see line 5). In contrast to Alg. 1, CoVeriTest does not get an ARG or condition. To enable cooperation between analyses, CoVeriTest constructs these two elements individually for each analysis run. During construction, it may extract and use information from results of previous analysis runs.

After initializing the test suite and the data structure to store analysis results (`args`), CoVeriTest repeatedly iterates over the configurations. It starts with the first pair in the sequence and finishes iterating when its time limit exceeded or all goals are covered. In each iteration, CoVeriTest first extracts the analysis to execute and its accompanied time limit (line 3). Then, it constructs the remaining inputs of the analysis: ARG and condition. Details regarding the construction are explained later in Alg. 3. Next, CoVeriTest executes the current analysis with the given program, the remaining test goals, the accompanied time limit, and the constructed ARG and condition. When the analysis has finished, CoVeriTest adds the returned tests to its test suite, removes all test goals covered by the analysis run from the set of goals, and stores the analysis result for cooperation (concatenates `arg` to the sequence of ARGs). If the analysis finished its exploration (`arg`.F$=\emptyset$), any remaining test goal should be unreachable and

Algorithm 3. cooperateAndInit: set up start point for analysis exploration, possibly transferring knowledge from previous analysis runs

Input: `prog` $= (L, \ell_0, G)$, `args` $\in (arg)^+$, `numAnalyses` $\in \mathbb{N}$
Output: ARG for program `prog`, condition describing explored state space

```
1: ψ=false;   π = ∅;   root = (ℓ0, ⊤);
2: if (length(args)≥numAnalyses) then
3:     if (reuse-arg) then
4:         return (last_arg_of_analysis(numAnalyses, args), ψ);
5:     if (reuse-precision) then
6:         π = last_arg_of_analysis(numAnalyses, args).π;
7: if (use-condition ∧ length(args)>0) then
8:     ψ = extract_condition(args[length(args)-1]);
9: return (({root}, ∅, root, {root}, π), ψ);
```

CoVeriTest returns its test suite. Otherwise, CoVeriTest determines how to continue in the next iteration (i.e., which configuration to consider). At the end of all iterations, CoVeriTest returns its generated test suite.

Next, we explain how to construct the ARG and the condition input for an analysis. The ARG describes the level of abstraction and where to continue exploration while the condition describes which parts of the state space have already been explored. Both guide the exploration of an analysis, which makes them well-suited for cooperation. While there are plenty of possibilities for cooperation, we currently only support three basic options: continue exploration of the previous ARG of the analysis (`reuse-arg`), reuse the analysis' abstraction level (`reuse-precision`), and restrict the exploration to the state space left out by the previous analysis (`use-condition`). The first two options only ensure that an analysis does not loose too much information due to switching. The last option, which is inspired by abstraction-driven concolic execution [32], indeed realizes cooperation between different analyses. Note that the last two options can also be combined.[5] If all options are turned off, no information will be exchanged.

Algorithm 3 shows the cooperative initialization of ARG and condition discussed above. It gets three inputs: the program, a sequence of args needed to realize cooperation, and the number of analyses used. At the beginning, it initializes the ARG components and the condition assuming no cooperation should be done. The condition states that nothing has been explored, the abstraction level becomes the coarsest available, and the ARG root considers the start of all program executions (initial program location and arbitrary variable values). If no cooperation is configured or the ARG required for cooperation is not available (e.g., in the first round), the returned ARG and condition tell the analysis to explore the complete state space from scratch. In all other cases, the analysis will be guided by information obtained in previous iterations. Option `reuse-arg`

[5] In contrast, the options `reuse-arg` and `use-conditions` cannot be combined because they are incompatible. The existing ARG does not fit to the constructed condition. Since `reuse-arg` subsumes `reuse-precision`, a combination makes no sense.

looks up the last ARG of the analysis stored in `args`. `Reuse-precision` considers the same ARG as `reuse-arg`, but only provides the ARG's precision π. For `use-condition`, a condition is constructed from the last ARG in args. For the details of the condition construction, we refer to conditional model checking [10].

Next, we study the effectiveness of different CoVeriTest configurations and compare CoVeriTest with existing test-generation tools.

4 Evaluation

We systematically evaluate CoVeriTest along the following claims:

Claim 1. For analyses that discard their own results from previous iterations (i.e., `reuse-arg` and `reuse-precision` turned off), CoVeriTest achieves higher coverage if switches between analyses happen rarely. *Evaluation Plan:* We look at CoVeriTest configurations in which analyses discard their own, previous results and compare the number of covered test goals reported by configurations that only differ in the analyses' time limits.

Claim 2. For analyses that reuse knowledge from their own, previous execution (i.e., `reuse-arg` or `reuse-precision` turned on), CoVeriTest achieves higher coverage if favoring more powerful analyses. *Evaluation Plan:* We look at CoVeriTest configurations in which analyses reuse their own, previous knowledge and compare the number of covered test goals reported by configurations that only differ in the analyses' time limits.

Claim 3. CoVeriTest performs better if analyses reuse knowledge from their own, previous execution (i.e., `reuse-arg` or `reuse-precision` turned on). *Evaluation Plan:* From all sets of CoVeriTest configurations that only differ in the analyses' time limits, we select the best and compare these.

Claim 4. Interleaving multiple analyses with CoVeriTest often achieves better results than using only one of the analyses for test generation. *Evaluation Plan:* We compare the number of covered goals reported by the best CoVeriTest configuration with those numbers achieved when running only one analysis of the CoVeriTest configuration for the total time limit.

Claim 5. Interleaving verifiers for test generation is often better than running them in parallel. *Evaluation Plan:* We compare the number of covered goals reported by the best CoVeriTest configuration with the number achieved when running all analyses of the CoVeriTest configuration in parallel.

Claim 6. CoVeriTest complements existing test-generation tools. *Evaluation Plan:* We use the same infrastructure and resources as used by the International Competition on Software Testing (Test-Comp'19)[6] and let the best CoVeriTest configuration construct test suites. These test suites are executed by the Test-Comp'19 validator to measure the achieved branch coverage. Then, we compare the coverage achieved by CoVeriTest with the coverage of the best two test-generation tools from Test-Comp'19.

[6] https://test-comp.sosy-lab.org/2019/

4.1 Setup

CoVeriTest Configurations. We implemented CoVeriTest in the software analysis framework CPAchecker [15]. Basically, we implemented Algs. 1, 2 and integrated Alg. 3 into Alg. 2. For condition construction, we reuse the code from conditional model checking [10]. For our experiments, we combine value [18] and predicate analysis [16]. Both have been used in cooperative verification [10,11,21].

Value analysis. CPAchecker's value analysis [18] tracks the values of variables stored in its current precision explicitly while assuming that the remaining variables may have any possible value. It iteratively increases its precision, i.e., the variables to track, combining counterexample-guided abstraction [28] with path-prefix slicing [22], and refinement selection [21]. Value analysis is efficient if few variable values need to be tracked, but it may get stuck in loops or suffers from a large state space in case variables are assigned many different values.

Predicate analysis. CPAchecker's predicate analysis uses predicate abstraction with adjustable-block encoding (ABE) [16]. ABE is configured to abstract at loop heads and uses the strongest postcondition at all remaining locations. To compute the set of predicates—its precision—, it uses counterexample-guided abstraction refinement [28] combined with lazy refinement [43] and interpolation [41]. While the predicate analysis is powerful and often summarizes loops easily, successor computation may require expensive SMT solver calls.

For both analyses, a CoVeriTest configuration specifies how Alg. 3 reuses the ARGs returned by previous analysis runs to set up the initial ARG and condition. In our experiments, we consider the following types of reuses.

plain Ignores all ARGs returned by previous analysis runs, i.e., `reuse-arg`, `reuse-prec`, and `use-condition` are turned off.

cond$_{\text{v}}$ The value analysis does not obtain information from previous ARGs and the predicate analysis is only steered by the condition extracted from the ARG returned by the previous value analysis.

cond$_{\text{p}}$ The value analysis is steered by the condition extracted from the ARG returned by the previous run of the predicate analysis and the predicate analysis ignores all previous ARGs.

cond$_{\text{v,p}}$ Value and predicate analysis are steered by the condition extracted from the last ARG returned, i.e., only `use-condition` turned on.

reuse-prec In each round, each analysis resumes its precision from the previous round, but restarts exploration, i.e., only `reuse-prec` is turned on.

reuse-arg In each round, each analysis continues to explore the ARG it returned in the previous round, i.e., only `reuse-arg` is turned on.

cond$_{\text{v}}$**+r** Similar to cond$_{\text{v}}$, but additionally the value analysis continues to explore the ARG it returned in the previous round and the predicate analysis restarts exploration with its precision from the previous round.

cond$_{\text{p}}$**+r** Similar to cond$_{\text{p}}$, but additionally the value analysis restarts exploration with its precision from the previous round and the predicate analysis continues to explore the ARG it returned in the previous round.

cond$_{\text{v,p}}$**+r** Like cond$_{\text{v,p}}$, but additionally the value and predicate analysis reuse their previous precision, i.e., `reuse-prec` and `use-condition` are turned on.

Finally, we need to fix the time limit for each analysis. We want to find out whether switches between analyses are important to the CoVeriTest approach. Therefore, we chose four limits (10 s, 50 s, 100 s, 250 s) that are applied to both analyses and trigger switches often, sometimes, or rarely. Additionally, we want to study whether it is advantageous if the time CoVeriTest spends in a round is not equally spread among the analyses. Thus, we come up with two additional time limit pairs: (20 s, 80 s) and (80 s, 20 s).

We combine all nine reuse types with the six time limit pairs, which results in 54 CoVeriTest configurations. All 54 configurations aim at generating tests to cover the assume edges of a program.

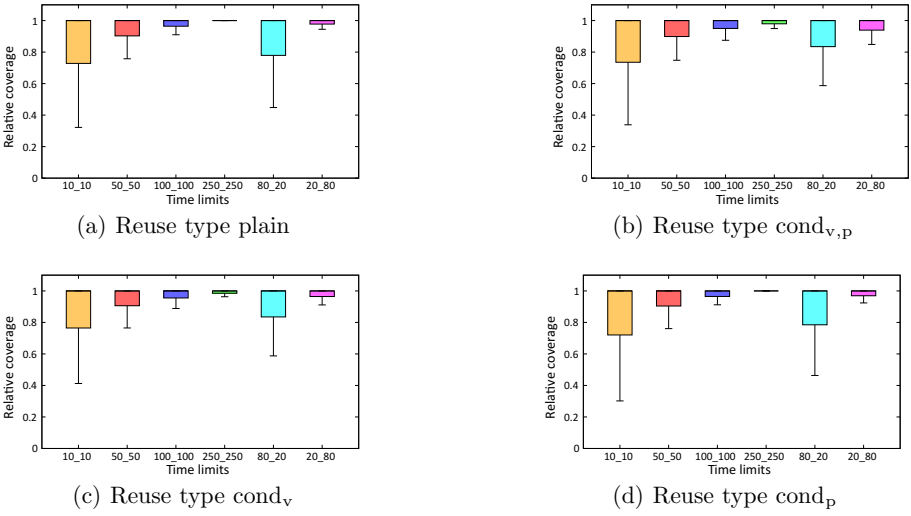
Tools. For CoVeriTest, we used the implementation in CPAchecker version 29 347. Moreover, we compare CoVeriTest against the two best tools VeriFuzz [26] and Klee [23] from Test-Comp'19 (in the versions submitted to Test-Comp'19[7]). The tool VeriFuzz is based on the evolutionary fuzzer AFL and uses verification techniques to compute initial input values and parameters for AFL. Klee applies symbolic execution. To compare CoVeriTest against Klee and VeriFuzz, we use the validator TBF Test-Suite Validator v1.2[8] to measure branch coverage. TBF Test-Suite Validator is based on gcov[9].

Programs. CoVeriTest, Klee, and VeriFuzz produce tests for C programs. All three tools participated in TestComp'19. Thus, for comparison of the three tools, we consider all 1 720 tasks of the TestComp'19 benchmark set[10] that support the branch-coverage property. Since we do not need to execute tests for the comparison of the different CoVeriTest configurations, we evaluated them on a larger benchmark set, which contains all 6 703 C programs from the well-established SV-benchmark set[11] in the version tagged svcomp18.

Computing Resources. We run our experiments on machines with 33 GB of memory and an Intel Xeon E3-1230 v5 CPU with 8 processing units and a frequency of 3.4 GHz. The underlying operating system is Ubuntu 18.04 with Linux kernel 4.15. As in TestComp'19, for test generation we grant each run a maximum of 8 processing units, 15 min of CPU time, and 15 GB of memory, and for test-suite execution (required to compare against Klee and VeriFuzz), the TBF Test-Suite Validator is granted 2 processing units, 3 h of CPU time, and 7 GB of memory per run. We use BenchExec [20] to enforce the limits of a run.

Availability. Our experimental data are available online[12] [13].

[7] https://gitlab.com/sosy-lab/test-comp/archives-2019/tree/testcomp19/2019
[8] https://gitlab.com/sosy-lab/test-comp/archives-2019/blob/testcomp19/2019/tbf-testsuite-validator.zip
[9] https://gcc.gnu.org/onlinedocs/gcc/Gcov.html
[10] https://github.com/sosy-lab/sv-benchmarks/tree/testcomp19
[11] https://github.com/sosy-lab/sv-benchmarks
[12] https://www.sosy-lab.org/research/coop-testgen/



(a) Reuse type plain

(b) Reuse type $\text{cond}_{\text{v,p}}$

(c) Reuse type cond_{v}

(d) Reuse type cond_{p}

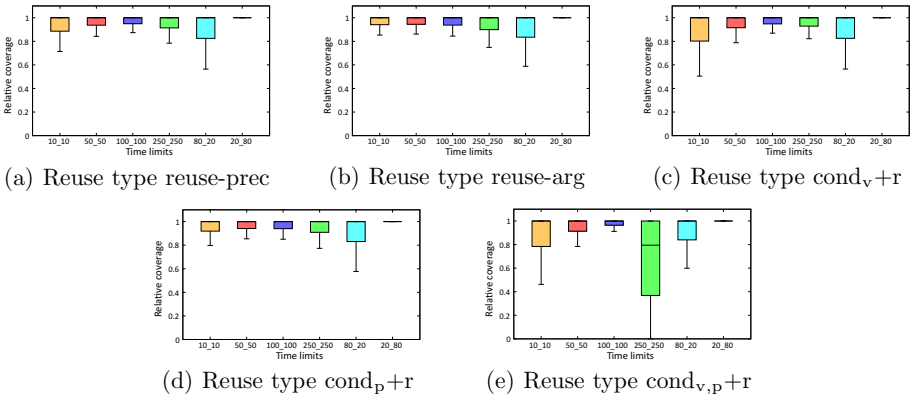
Fig. 3. Comparing relative coverage (number of covered goals divided by maximal number of covered goals) achieved by CoVeriTest configurations with different time limits. All configurations let analyses discard their own knowledge gained in previous executions.

4.2 Experiments

Claim 1 (Reduce switching when discarding own results). Four types of reuse (namely, plain, cond_{v}, cond_{p}, and $\text{cond}_{\text{v,p}}$) let the analyses discard their own knowledge from their previous executions. For each of these types, we compare the coverage achieved by all six CoVeriTest configurations that use this type[13]. More concrete, for all six CoVeriTest configurations applying the same reuse type, we first compute for each program the maximum over the number of covered goals achieved by each of these six configurations for that program. Then, for each of the six CoVeriTest configurations that use that reuse type, we divide the number of covered goals achieved for a program by the respective maximum computed. We call this measure *relative coverage* because the value is relative to the maximum and not the total number of goals. Figure 3 shows box plots per reuse type. The box plots show the distribution of the relative coverage. The closer the bottom border of a box is to value one, the higher coverage is achieved. For all four reuse types, the fourth box plot has the bottom border closest to value one. Since the fourth box plot is a configuration that grants each analysis 250 s per round (highest limit considered, only three switches), the claim holds.

Claim 2 (Favor powerful analysis when reusing own results). Five types of reuse (namely, reuse-prec, reuse-arg, cond_{v}+r, cond_{p}+r, and $\text{cond}_{\text{v,p}}$+r) let analyses reuse knowledge from their own, previous execution. Similar to the previous claim, we compute for each of these types the relative coverage of all six configurations using this particular type of reuse. For each reuse type,

[13] Note that those six configurations only differ in the analyses' time limits.



(a) Reuse type reuse-prec (b) Reuse type reuse-arg (c) Reuse type cond_v+r

(d) Reuse type cond_p+r (e) Reuse type $\text{cond}_{\text{v,p}}$+r

Fig. 4. Comparing relative coverage (number of covered goals divided by maximal number of covered goals) achieved by CoVeriTest configurations when using different time limits and a fixed reuse type. All considered configurations let analyses reuse knowledge from their own, previous execution.

Fig. 4 shows box plots of the distributions of the relative coverage. As before, a bottom border closer to value one reflects higher coverage. In all five cases, the last box plot has the bottom border closest to value one. The last box plots represent CoVeriTest configurations that grant the value analysis 20 s and the predicate analysis 80 s in each round. Since the predicate analysis, which gets more time per round, is more powerful than the value analysis, our claim is valid.[14]

Claim 3 (Better reuse own results). So far, we know how to configure time limits. Now, we want to find out how to reuse information from previous analysis runs. For each reuse type, we select from the six available configurations the configuration that performed best. Again, we use the relative coverage to compare the resulting nine configurations. Figure 5 shows box plots of the distributions of the relative coverage. The first four box plots show configurations in which analyses discard their own results, while the last five box plots refer to configurations in which analyses reuse knowledge from their own, previous executions. Since the last five boxes are smaller than the first four and their bottom borders are closer to one, the last five configurations achieve higher coverage. Hence, our claim holds. Moreover, from Fig. 5 we conclude that it is best to reuse the ARG (although cond_v+r and cond_p+r are close by).

Claim 4 (Interleave multiple analyses rather than use one of them). To evaluate whether CoVeriTest benefits from interleaving, we compare CoVeriTest against the analyses used by it. CoVeriTest interleaves value and predicate analysis. Figure 6(a) and 6(b) show scatter plots that compare for each program the coverage, i.e., number of covered goals divided by number of total goals, achieved by the best CoVeriTest configuration (x-axis) with the coverage achieved when only using either value or predicate analysis for test generation. Note that we excluded those programs from the scatter plots, for which we miss

[14] This insight is independently partially backed by a sequential combination of explicit-value analysis and predicate analysis that performed well in SV-COMP 2013 [62].

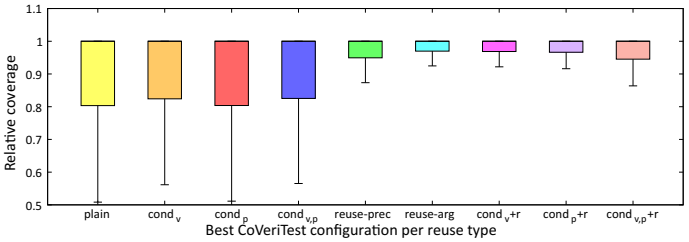
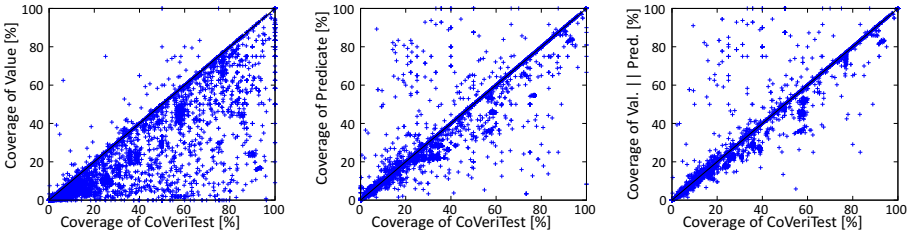


Fig. 5. Comparing relative coverage achieved by CoVeriTest configurations applying different strategies to reuse information gained by previous verifier runs.



(a) CoVeriTest (x-axis) vs. value analysis

(b) CoVeriTest (x-axis) vs. predicate analysis

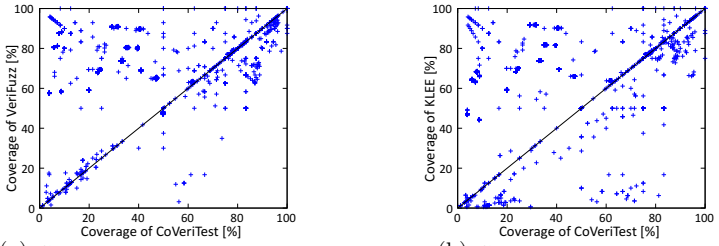
(c) CoVeriTest (x-axis) vs. value and predicate analysis in parallel

Fig. 6. Compares the coverage achieved by CoVeriTest (best configuration) with the coverage achieved when running CoVeriTest's analyses alone or in parallel

the number of covered goals for at least one test generator, e.g., due to timeout of the analysis. Figure 6(a) compares CoVeriTest and value analysis; we see that almost all points are in the lower right half. Thus, CoVeriTest typically achieves higher coverage than value analysis alone. Figure 6(b), comparing CoVeriTest with predicate analysis, is more diverse. About 54% of the points are on the diagonal, i.e., CoVeriTest and predicate analysis cover the same number of goals. The upper left half contains 19% of the points, i.e., predicate analysis alone achieves higher coverage. These points for example reflect float programs and ECA programs without arithmetic computations. In contrast, CoVeriTest achieves higher coverage in 27% of the programs. CoVeriTest is beneficial for programs that only need few variable values to trigger the branches, like ssh programs or programs from the product-lines subcategory. CoVeriTest also profits from the value analysis when considering ECA programs with arithmetic computations, since the variables have a fixed value in each loop iteration. All in all, CoVeriTest performs slightly better than predicate analysis alone.

Claim 5 (Interleave rather than parallelize). Figure 6(c) shows a scatter plot that compares for each program the coverage achieved by CoVeriTest (x-axis) and a test generator that runs the value analysis and the predicate analysis in parallel[15]. As before, we exclude programs for which

[15] The test generator uses CPAchecker's parallel algorithm and lets the two analyses share information about covered test goals.



(a) CoVeriTest vs. VeriFuzz (b) CoVeriTest vs. Klee

Fig. 7. Compares the branch coverage achieved by CoVeriTest (best configuration) with the branch coverage achieved by existing state-of-the-art test-generation tools

we could not get the number of covered goals for at least one of the analyses. Looking at Fig. 6(c), we observe that many points (60%) are on the diagonal, i.e., the achieved coverage is identical. Moreover, CoVeriTest performs better for 30% (lower right half), while approximately 10% of the points are in the upper left half. Since CoVeriTest achieves the same or better coverage results in about 90% of the cases, it should be preferred over parallelization. This is no surprise since we showed that a test generator should favor the more powerful analysis (which CoVeriTest does, but parallelization evenly distributes CPU time).

Claim 6 (CoVeriTest complementary). Our goal is to compare CoVeriTest and the two best tools of Test-Comp'19 [4]: VeriFuzz and Klee. All three tools aim at constructing test suites with high branch coverage. Thus, we use branch coverage as comparison criterion. We measure branch coverage with TBF Test-Suite Validator. Figure 7 shows two scatter plots. Each plot compares branch coverage achieved by CoVeriTest and by one of the other techniques.[16] Points in the lower right half indicate that CoVeriTest achieved higher coverage. Looking at the two scatter plots, we observe that there exist programs for which CoVeriTest performs better and vice versa. Generally, we observed that CoVeriTest has problems with array tasks and ECA tasks. We already know from verification that CPAchecker sometimes lacks refinement support for array tasks. Moreover, the problem with the ECA tasks is that CPAchecker splits conditions with conjunctions or disjunctions—which ECA tasks contain a lot—into multiple assume edges. Thus, the number of test goals is much larger than the actual branches to be covered. However, CoVeriTest seems to benefit from splitting for some of the float tasks. Additionally, CoVeriTest is often better on tasks of the sequentialized subcategory. We think that CoVeriTest benefits from the value analysis since the tasks of the sequentialized subcategory contain lots of branch conditions checking for a specific value or interpreting variable values as booleans. All in all, CoVeriTest is not always best, but is also not dominated. Thus, CoVeriTest complements the existing approaches.

[16] Note that the scatter plots only contain points that have a positive x and y value because there exist different reasons (timeout, out of memory, tool failure, etc.) why we might get no or a zero coverage value from the test validator. The plots contain points for about 98% of the 1 720 programs.

4.3 Threats to Validity

All our CoVeriTest configurations consider the same two analyses. Our results might not apply if using CoVeriTest with a different set of analyses. In our experiments, we used benchmark programs instead of real-world applications. Although the benchmark set is diverse and well-established, our results may not carry over into practice.

The validator TBF Test-Suite Validator might contain bugs that result in wrong coverage numbers. However, the validator was used in Test-Comp'19 already, and is based on the well-established coverage-measurement tool `gcov`.

For the comparison of the CoVeriTest configurations as well as the comparison of CoVeriTest with the single analyses and the parallel approach, we relied on the number of covered goals reported by CoVeriTest. Invalid counterexamples could be used to cover test goals. The analyses used by CoVeriTest apply CEGAR approaches and should detect spurious counterexamples. Moreover, these analyses run in the SV-COMP configuration of CPAchecker and are tuned to not report false results. Another problem is that whenever CPAchecker does not output statistics (due to timeout, out of memory, etc.), we use the last number of covered goals reported in the log. However, this might be an underapproximation of the number of covered goals. All these problems do not occur in the comparison of CoVeriTest with Klee and VeriFuzz, in which the coverage is measured by the validator. Thus, this comparison still supports the value of CoVeriTest.

5 Related Work

CoVeriTest interleaves reachability analyses to construct tests for C programs. To enable cooperation, CoVeriTest extracts information from ARGs constructed by previous analysis runs.

A few tools use reachability analyses for test generation. Blast [5] considers a target predicate p and generates a test for each program location that can be reached with a state fulfilling the predicate p. For test generation, Blast uses predicate abstraction. FShell [44–46] and CPA/Tiger [12] generate tests for a coverage criterion specified in the FShell query language (FQL) [46]. Both transform the FQL specification into a set of test-goal automata and check for each automaton whether its final state can be reached. FShell uses CBMC to answer those reachability queries and CPA/Tiger uses predicate abstraction.

Various combinations have been proposed for verification [2,10,11,14,25,27, 29–31,35,37,40,50,64] and test-suite generation [1,32,34,36,38,47,51,54,56,59, 60,63]. We focus on combinations that interleave approaches. SYNERGY [40] and DASH [2] alternate test generation and proof construction to (dis)prove a property. Similarly, SMASH [37] combines underapproximation with overapproximation. Interleaving is also used in test generation. Hybrid concolic testing [54] interleaves random testing with symbolic execution. When random testing gets stuck, symbolic execution is started from the current state. As soon as a new goal is covered, symbolic execution hands over to random testing providing the values used to cover the goal. Similarly, Driller [60] and Badger [56] combine fuzzing

with concolic execution. However, they only exchange inputs. Xu et al. [51,63] interleave different approaches to augment test suites. The approach closest to CoVeriTest is abstraction-driven concolic testing [32]. Abstraction-driven concolic testing interleaves concolic execution and predicate analysis. Furthermore, it uses conditions extracted from the ARGs generated by the predicate analysis to direct the concolic execution towards feasible paths. Abstraction-driven concolic testing can be seen as one particular configuration of CoVeriTest.

Also, ARG information has been reused in different contexts. Precision reuse [19] uses the precision determined in a previous analysis run to reverify a modified program. Similarly, extreme model checking [42] adapts an ARG constructed in a previous analysis to fit to the modified program. CPA/Tiger [12] transforms an ARG that was constructed for one test goal such that it fits to a new test goal. Lazy abstraction refinement [43] adapts an ARG to continue exploration after abstraction refinement. Configurable program certification [48,49] constructs a certificate from an ARG, which can be used to reverify a program. Similarly, reachability tools like CPAchecker construct witnesses [6,7] from ARGs. Conditional model checking [10,14] constructs a condition from an ARG when a verifier gives up. The condition describes the remaining verification task and is used by a subsequent verifier to restrict its exploration.

6 Conclusion

Testing is a standard technique for software quality assurance. But state-of-the-art techniques still miss many bugs that involve sophisticated branching conditions [17]. It turns out that techniques performing abstract reachability analyses are well-suited for this task. They simply need to check the reachability of every branch and generate a test for each positive check. However, in practice, for every such technique there exist reachability queries on which the technique is inefficient or fails [8]. We propose CoVeriTest to overcome these practical limitations. CoVeriTest interleaves different reachability analyses for test generation. We experimented with various configurations of CoVeriTest, which vary in the time limits of the analyses and the type of information exchanged between different analysis runs. CoVeriTest works best when each analysis resumes its exploration, different analyses only share test goals, and more powerful analyses get larger time budgets. Moreover, a comparison of CoVeriTest with (a) the reachability analyses used by CoVeriTest and (b) state-of-the-art test-generation tools witness the benefits of the new CoVeriTest approach.

CoVeriTest participated in Test-Comp 2019 [4] and achieved rank 3 (out of 9) in both categories, bug finding and branch coverage.[17]

In future, we plan to integrate further analyses, e.g., bounded model checking or symbolic execution, into CoVeriTest and to evaluate CoVeriTest on real-world applications.

[17] https://test-comp.sosy-lab.org/2019/results/

References

1. Baars, A.I., Harman, M., Hassoun, Y., Lakhotia, K., McMinn, P., Tonella, P., Vos, T.E.J.: Symbolic search-based testing. In: Proc. ASE, pp. 53–62. IEEE (2011). https://doi.org/10.1109/ASE.2011.6100119
2. Beckman, N., Nori, A.V., Rajamani, S.K., Simmons, R.J.: Proofs from tests. In: Proc. ISSTA, pp. 3–14. ACM (2008). https://doi.org/10.1145/1390630.1390634
3. Beyer, D.: Software verification with validation of results (Report on SV-COMP 2017). In: Proc. TACAS, LNCS, vol. 10206, pp. 331–349. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54580-5_20
4. Beyer, D.: International competition on software testing (Test-Comp). In: Proc. TACAS, Part 3, LNCS, vol. 11429, pp. 167–175. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17502-3_11
5. Beyer, D., Chlipala, A.J., Henzinger, T.A., Jhala, R., Majumdar, R.: Generating tests from counterexamples. In: Proc. ICSE, pp. 326–335. IEEE (2004). https://doi.org/10.1109/ICSE.2004.1317455
6. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M.: Correctness witnesses: Exchanging verification results between verifiers. In: Proc. FSE, pp. 326–337. ACM (2016). https://doi.org/10.1145/2950290.2950351
7. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Stahlbauer, A.: Witness validation and stepwise testification across software verifiers. In: Proc. FSE, pp. 721–733. ACM (2015). https://doi.org/10.1145/2786805.2786867
8. Beyer, D., Dangl, M., Wendler, P.: A unifying view on SMT-based software verification. J. Autom. Reasoning **60**(3), 299–335 (2018). https://doi.org/10.1007/s10817-017-9432-6
9. Beyer, D., Gulwani, S., Schmidt, D.: Combining model checking and data-flow analysis. In: Clarke, E.M., Henzinger, T.A., Veith, H. (eds.) Handbook on Model Checking, pp. 493–540. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-10575-8_16
10. Beyer, D., Henzinger, T.A., Keremoglu, M.E., Wendler, P.: Conditional model checking: A technique to pass information between verifiers. In: Proc. FSE, pp. 57:1–57:11. ACM (2012). https://doi.org/10.1145/2393596.2393664
11. Beyer, D., Henzinger, T.A., Théoduloz, G.: Program analysis with dynamic precision adjustment. In: Proc. ASE, pp. 29–38. IEEE (2008). http://dx.doi.org/10.1109/ASE.2008.13
12. Beyer, D., Holzer, A., Tautschnig, M., Veith, H.: Information reuse for multi-goal reachability analyses. In: Proc. ESOP, LNCS, vol. 7792, pp. 472–491. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_26
13. Beyer, D., Jakobs, M.C.: Replication package for article "CoVeriTest: Cooperative verifier-based testing" in Proc. FASE 2019. Zenodo (2019). https://doi.org/10.5281/zenodo.2566735
14. Beyer, D., Jakobs, M.C., Lemberger, T., Wehrheim, H.: Reducer-based construction of conditional verifiers. In: Proc. ICSE, pp. 1182–1193. ACM (2018). https://doi.org/10.1145/3180155.3180259
15. Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: Proc. CAV, LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_16

16. Beyer, D., Keremoglu, M.E., Wendler, P.: Predicate abstraction with adjustable-block encoding. In: Proc. FMCAD, pp. 189–197. FMCAD (2010). http://ieeexplore.ieee.org/document/5770949/
17. Beyer, D., Lemberger, T.: Software verification: Testing vs. model checking. In: Proc. HVC, LNCS, vol. 10629, pp. 99–114. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70389-3_7
18. Beyer, D., Löwe, S.: Explicit-state software model checking based on CEGAR and interpolation. In: Proc. FASE, LNCS, vol. 7793, pp. 146–162. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37057-1_11
19. Beyer, D., Löwe, S., Novikov, E., Stahlbauer, A., Wendler, P.: Precision reuse for efficient regression verification. In: Proc. FSE, pp. 389–399. ACM (2013). https://doi.org/10.1145/2491411.2491429
20. Beyer, D., Löwe, S., Wendler, P.: Benchmarking and resource measurement. In: Proc. SPIN, LNCS, vol. 9232, pp. 160–178. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23404-5_12
21. Beyer, D., Löwe, S., Wendler, P.: Refinement selection. In: Proc. SPIN, LNCS, vol. 9232, pp. 20–38. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23404-5_3
22. Beyer, D., Löwe, S., Wendler, P.: Sliced path prefixes: An effective method to enable refinement selection. In: Proc. FORTE, LNCS, vol. 9039, pp. 228–243. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19195-9_15
23. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proc. OSDI, pp. 209–224. USENIX Association (2008). http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf
24. Chalupa, M., Vitovská, M., Strejcek, J.: SYMBIOTIC 5: Boosted instrumentation (competition contribution). In: Proc. TACAS, LNCS, vol. 10806, pp. 442–446. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89963-3_29
25. Chebaro, O., Kosmatov, N., Giorgetti, A., Julliand, J.: Program slicing enhances a verification technique combining static and dynamic analysis. In: Proc. SAC, pp. 1284–1291. ACM (2012). http://doi.acm.org/10.1145/2245276.2231980
26. Chowdhury, A.B., Medicherla, R.K., Venkatesh, R.: VeriFuzz: Program aware fuzzing. In: Proc. TACAS, Part 3, LNCS, vol. 11429, pp. 244–249. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17502-3_22
27. Christakis, M., Müller, P., Wüstholz, V.: Guiding dynamic symbolic execution toward unverified program executions. In: Proc. ICSE, pp. 144–155. ACM (2016). http://doi.acm.org/10.1145/2884781.2884843
28. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM **50**(5), 752–794 (2003). http://doi.acm.org/10.1145/876638.876643
29. Cousot, P., Cousot, R.: Systematic design of program-analysis frameworks. In: Proc. POPL, pp. 269–282. ACM (1979). http://doi.acm.org/10.1145/567752.567778
30. Csallner, C., Smaragdakis, Y.: Check 'n' crash: Combining static checking and testing. In: Proc. ICSE, pp. 422–431. ACM (2005). http://doi.acm.org/10.1145/1062455.1062533
31. Czech, M., Jakobs, M.C., Wehrheim, H.: Just test what you cannot verify! In: Proc. FASE, LNCS, vol. 9033, pp. 100–114. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46675-9_7

32. Daca, P., Gupta, A., Henzinger, T.A.: Abstraction-driven concolic testing. In: Proc. VMCAI, LNCS, vol. 9583, pp. 328–347. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49122-5_16
33. D'Silva, V., Kröning, D., Weissenbacher, G.: A survey of automated techniques for formal software verification. IEEE Trans. CAD Integr. Circ. Syst. **27**(7), 1165–1178 (2008). https://doi.org/10.1109/TCAD.2008.923410
34. Galeotti, J.P., Fraser, G., Arcuri, A.: Improving search-based test suite generation with dynamic symbolic execution. In: Proc. ISSRE, pp. 360–369. IEEE (2013). https://doi.org/10.1109/ISSRE.2013.6698889
35. Ge, X., Taneja, K., Xie, T., Tillmann, N.: DyTa: Dynamic symbolic execution guided with static verification results. In: Proc. ICSE, pp. 992–994. ACM (2011). http://doi.acm.org/10.1145/1985793.1985971
36. Godefroid, P., Klarlund, N., Sen, K.: DART: Directed automated random testing. In: Proc. PLDI, pp. 213–223. ACM (2005). http://doi.acm.org/10.1145/1065010.1065036
37. Godefroid, P., Nori, A.V., Rajamani, S.K., Tetali, S.: Compositional may-must program analysis: Unleashing the power of alternation. In: Proc. POPL, pp. 43–56. ACM (2010). http://doi.acm.org/10.1145/1706299.1706307
38. Godefroid, P., Levin, M.Y., Molnar, D.A.: Automated whitebox fuzz testing. In: Proc. NDSS. The Internet Society (2008)
39. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: Proc. CAV, LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997). https://doi.org/10.1007/3-540-63166-6_10
40. Gulavani, B.S., Henzinger, T.A., Kannan, Y., Nori, A.V., Rajamani, S.K.: SYNERGY: A new algorithm for property checking. In: Proc. FSE, pp. 117–127. ACM (2006). https://doi.org/10.1145/1181775.1181790
41. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: Proc. POPL, pp. 232–244. ACM (2004). http://doi.acm.org/10.1145/964001.964021
42. Henzinger, T.A., Jhala, R., Majumdar, R., Sanvido, M.A.A.: Extreme model checking. In: Verification: Theory and Practice, pp. 332–358. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-39910-0_16
43. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Proc. POPL, pp. 58–70. ACM (2002). https://doi.org/10.1145/503272.503279
44. Holzer, A., Schallhart, C., Tautschnig, M., Veith, H.: FShell: Systematic test case generation for dynamic analysis and measurement. In: Gupta, A., Malik, S. (eds.) Proc. CAV, LNCS, vol. 5123, pp. 209–213. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-70545-1_20
45. Holzer, A., Schallhart, C., Tautschnig, M., Veith, H.: Query-driven program testing. In: Proc. VMCAI, LNCS, vol. 5403, pp. 151–166. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-540-93900-9_15
46. Holzer, A., Schallhart, C., Tautschnig, M., Veith, H.: How did you specify your test suite. In: Proc. ASE, pp. 407–416. ACM (2010). https://doi.org/10.1145/1858996.1859084
47. Inkumsah, K., Xie, T.: Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In: Proc. ASE, pp. 297–306. IEEE (2008). https://doi.org/10.1109/ASE.2008.40

48. Jakobs, M.C.: Speed up configurable certificate validation by certificate reduction and partitioning. In: Proc. SEFM, LNCS, vol. 9276, pp. 159–174. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-22969-0_12
49. Jakobs, M.C., Wehrheim, H.: Certification for configurable program analysis. In: Proc. SPIN, pp. 30–39. ACM (2014). https://doi.org/10.1145/2632362.2632372
50. Jalote, P., Vangala, V., Singh, T., Jain, P.: Program partitioning: A framework for combining static and dynamic analysis. In: Proc. WODA, pp. 11–16. ACM (2006). http://doi.acm.org/10.1145/1138912.1138916
51. Kim, Y., Xu, Z., Kim, M., Cohen, M.B., Rothermel, G.: Hybrid directed test suite augmentation: An interleaving framework. In: Proc. ICST, pp. 263–272. IEEE (2014). https://doi.org/10.1109/ICST.2014.39
52. Lemieux, C., Sen, K.: FairFuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In: Proc. ASE, pp. 475–485. ACM (2018). https://doi.org/10.1145/3238147.3238176
53. Li, J., Zhao, B., Zhang, C.: Fuzzing: A survey. Cybersecurity **1**(1), 6 (2018). https://doi.org/10.1186/s42400-018-0002-y
54. Majumdar, R., Sen, K.: Hybrid concolic testing. In: Proc. ICSE, pp. 416–426. IEEE (2007). https://doi.org/10.1109/ICSE.2007.41
55. McMinn, P.: Search-based software test data generation: A survey. Softw. Test. Verif. Reliab. **14**(2), 105–156 (2004). https://doi.org/10.1002/stvr.294
56. Noller, Y., Kersten, R., Pasareanu, C.S.: Badger: Complexity analysis with fuzzing and symbolic execution. In: Proc. ISSTA, pp. 322–332. ACM (2018). http://doi.acm.org/10.1145/3213846.3213868
57. Pacheco, C., Lahiri, S.K., Ernst, M.D., Ball, T.: Feedback-directed random test generation. In: Proc. ICSE, pp. 75–84. IEEE (2007). https://doi.org/10.1109/ICSE.2007.37
58. Pasareanu, C.S., Visser, W.: A survey of new trends in symbolic execution for software testing and analysis. STTT **11**(4), 339–353 (2009). https://doi.org/10.1007/s10009-009-0118-1
59. Sakti, A., Guéhéneuc, Y., Pesant, G.: Boosting search based testing by using constraint based testing. In: Proc. SSBSE, LNCS, vol. 7515, pp. 213–227. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33119-0_16
60. Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., Vigna, G.: Driller: Augmenting fuzzing through selective symbolic execution. In: Proc. NDSS. The Internet Society (2016). http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/driller-augmenting-fuzzing-through-selective-symbolic-execution.pdf
61. Visser, W., Păsăreanu, C.S., Khurshid, S.: Test input generation with Java PathFinder. In: Proc. ISSTA, pp. 97–107. ACM (2004). http://doi.acm.org/10.1145/1007512.1007526
62. Wendler, P.: CPAchecker with sequential combination of explicit-state analysis and predicate analysis (competition contribution). In: Proc. TACAS, LNCS, vol. 7795, pp. 613–615. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36742-7_45

63. Xu, Z., Kim, Y., Kim, M., Rothermel, G.: A hybrid directed test suite augmentation technique. In: Proc. ISSRE, pp. 150–159. IEEE (2011). https://doi.org/10.1109/ISSRE.2011.21
64. Yorsh, G., Ball, T., Sagiv, M.: Testing, abstraction, theorem proving: Better together! In: Proc. ISSTA, pp. 145–156. ACM (2006). http://doi.acm.org/10.1145/1146238.1146255