International Journal on Software Tools for Technology Transfer (2021) 23:313–333
https://doi.org/10.1007/s10009-020-00587-8

GENERAL

Special Issue: FASE 2019

Cooperative verifier-based testing with CoVeriTest

Dirk Beyer[1] · Marie-Christine Jakobs[2]

Published online: 25 April 2021

Abstract
Testing is a widely applied technique to evaluate software quality, and coverage criteria are often used to assess the adequacy of a generated test suite. However, manually constructing an adequate test suite is typically too expensive, and numerous techniques for automatic test-suite generation were proposed. All of them come with different strengths. To build stronger test-generation tools, different techniques should be combined. In this paper, we study cooperative combinations of verification approaches for test generation, which exchange high-level information. We present CoVeriTest, a hybrid technique for test-suite generation. CoVeriTest iteratively applies different conditional model checkers and allows users to adjust the level of cooperation and to configure individual time limits for each conditional model checker. In our experiments, we systematically study different CoVeriTest cooperation setups, which either use combinations of explicit-state model checking and predicate abstraction, or bounded model checking and symbolic execution. A comparison with state-of-the-art test-generation tools reveals that CoVeriTest achieves higher coverage for many programs (about 15%).

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

1 Introduction

Verification is an integral part of software development processes [54,67]. Next to code reviews and static code analysis, testing is a widely adopted quality-assurance technique. Since manually constructing test suites is laborious, automatic test-case generation techniques are used, where possible. Black-box testing uses, for example, model-based techniques, and white-box testing might be based on control-flow coverage. In this paper, we are interested in white-box techniques for structural coverage. Existing, automatic test-generation techniques in this area range from random testing [68,104] and fuzzing [50,95,96], over search-based testing [99] to symbolic execution [43,47,93,105] and reachability analyses [13,25,80,81].

A preliminary version was published in Proc. FASE 2019 [26].
A replication package is available on Zenodo [27].

✉ Marie-Christine Jakobs
jakobs@cs.tu-darmstadt.de

1 LMU Munich, Oettingenstr. 67, 80538 Munich, Germany

2 Department of Computer Science, Technical University of Darmstadt, Hochschulstr. 10, 64289 Darmstadt, Germany

Reachability analyses perform well when they are applied to bug finding. This is supported by a recent case study that compared model checkers with test tools w.r.t. bug finding [32]. Furthermore, reachability analyses derive test suites that achieve high coverage, and several verification tools support test-case generation (e.g., Blast [13], PathFinder [123], CPAchecker [25]). At the first glance, performing a reachability check for each test goal seems too expensive. However, due to tremendous advances in software verification [11], in practice, those reachability analyses can be made pretty efficient. This motivated us to use reachability analyses for test-case generation.

It is well known, e.g., from the International Competition on Software Verification (SV-COMP) [11], that reachability analyses come with different strengths and weaknesses. For example, consider function `foo` in Listing 1. Let us assume that we want to generate test cases for all branches. Explicit state model checking [35,89] cannot detect the infeasibility of the if-branch in line 5 because it cannot capture a concrete value for variable `n`. In contrast, the different branches of the while loop can easily be reached when explicitly tracking the concrete values of variable `s`. An

```
1  void foo(int n) {
2    s=1; i=0;
3    if(n<0)
4      i = -(n+s);
5      if(i<n) pause();
6    else
7      while(i<n) {
8        if(s==1) compute();
9        if(s==2) publish();
10       if(s==3) s=0;
11       i++; s++;
12     }
13 }
```

Listing 1 Example program `foo`

analysis based on predicate abstraction [71] must learn the predicates $n < 0, i \geq 0, s = 1$ to detect the infeasibility of the if-branch in line 5. To reach the different branches in the while loop, predicates $s = 1$ and $s = 2$ are required. Learning these predicates might need expensive, counterexample-guided abstraction-refinement [53] steps. In CPAchecker, about half of the test-case generation time is spent on the refinement. For bounded model checking [41], it is easy to detect that the condition `i<n` is not feasible, but it needs to unroll the while loop at least three times. Thus, the loop bound k must be increased multiple times, and each time bounded model checking is restarted. Symbolic execution [93] can detect the infeasibility of the if-branch in line 5. Moreover, like explicit-state model checking, symbolic execution easily covers the branches occurring in the while loop, but it may fail to terminate the exploration of the while loop.

The combination of approaches, which is applied in a wide area of computer science [49,94,108,124], is a typical solution to overcome weaknesses of different approaches. Also, test and verification approaches already employ combinations. Existing approaches can be classified into parallel combinations [19,42,68,82,100,116,117,119], sequential combinations [28,33,51,57,66,83,88], selective combinations [4,15,61,65,91,121], nested combinations [6,7,64,113], and interleaved combinations [8,60,73,98,120]. In this paper, we are particularly interested in interleaved combinations because we think they provide a good trade-off between the effort for their implementation and the efficiency of the combined approaches. Existing interleaved test-case generation techniques typically alternate a fixed set of approaches and prescribe which information is exchanged. We propose a new cooperative, verifier-based testing approach called CoVeriTest, which interleaves different reachability analyses and exchanges various types of analysis information between analyses. In contrast to existing interleaving approaches, CoVeriTest allows us to configure the analyses that will be combined and the level of cooperation, i.e., which information is exchanged.

CoVeriTest is inspired by abstraction-driven concolic testing [60], which interleaves concolic execution and predicate abstraction and informs the concolic execution about infeasible paths detected by the predicate analysis. In detail, CoVeriTest iteratively executes a configurable sequence of reachability analyses. In each iteration, the analyses are run sequentially and each analysis in the sequence is limited to its individual, but configurable time limit. Moreover, we can configure CoVeriTest to exchange different types of information gained during a reachability analysis, e.g., which paths are infeasible, have already been explored, or which abstraction level to use. We implemented CoVeriTest in the configurable software-analysis framework CPAchecker [29], which offers a large variety of reachability analyses, and use a large, well-established benchmark set to evaluate 126 CoVeriTest configurations. Furthermore, we compare CoVeriTest with two existing state-of-the-art test-generation tools. Our experiments confirm that the CoVeriTest approach is valuable for test generation.

Summing up, we make the following contributions:

- We present the test-generation approach CoVeriTest that supports flexible, high-level interleavings of reachability analyses with information exchange.
- We perform an extensive evaluation of CoVeriTest studying 126 different CoVeriTest cooperation setups and comparing CoVeriTest against two state-of-the-art test-generation tools.[1]
- CoVeriTest's open-source tool implementation and our experimental data are publicly available for others to reproduce our results (see Sect. 6).

This paper is an extended version of a paper presented at FASE 2019 [26]. To become self-contained, we added material about test-case generation from counterexamples and about condition construction. We enriched our explanations with examples. Furthermore, we added a new cooperation type `transform-precision` to CoVeriTest, which transfers information about the abstraction level from one analysis to another. Our experiments are extended to study the new cooperation type and a second combination of verifiers. All our experiments are run on the same version of the benchmark set. CoVeriTest participated in Test-Comp 2019 [84] and 2020 [85].

[1] We choose the best two tools VeriFuzz and Klee from the 1st International Competition on Software Testing (Test-Comp 2019) [12]: https://test-comp.sosy-lab.org/2019/.

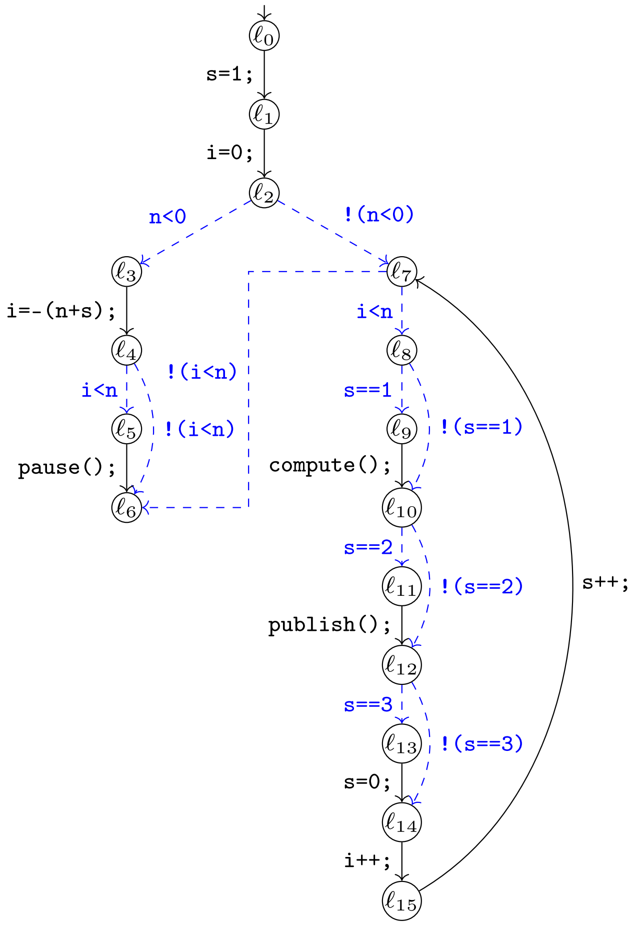


Fig. 1 CFA for example program `foo`

2 Foundations of test-case generation with reachability analyses

2.1 Programs

Following literature [22], we model a program by a control-flow automaton (CFA).

Definition 1 (*Control-Flow Automaton*) A *control-flow automaton* $P = (L, \ell_0, G)$ consists of

- a set L of program locations, which represent the program counter values,
- an initial program location $\ell_0 \in L$, and
- a set $G \subseteq L \times Ops \times L$ of control-flow edges, where Ops denotes the set of all possible operations.

Figure 1 shows the control-flow automaton for the function body of our example `foo` from Listing 1. The CFA contains an edge for every statement and two, dashed, blue edges for every condition in an if- or while-statement. The two edges represent the two possible evaluations of the respective condition.

For the semantics of a CFA, we assume a standard operational semantics, which we do not further detail.

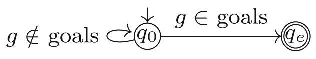


Fig. 2 Encoding test goals as specification violation

2.2 Test goals

Generally, our approach should work for all elementary coverage criteria [81], i.e., criteria that can be expressed by a set of independent predicates on program paths (the test goals). For simplicity, we demonstrate our CoVeriTest approach on simple, structural coverage properties like branch coverage. In our program representation, structural coverage properties map to coverage of control-flow edges. Thus, we formally represent a set of test goals by a subset of a program's control-flow edges.

Definition 2 (*Test Goals*) A set of *test goals* for a CFA $P = (L, \ell_0, G)$ is a subset $\texttt{goals} \subseteq G$ of control-flow edges.

Looking at program `foo` in Fig. 1, the test goals for branch coverage are the blue, dashed edges.

Our test-goal format is not a standard specification format for reachability analyses, which we want to apply for test generation. Specifications for reachability analyses typically encode when a property violation is reached, e.g., using an observer automaton [14]. Figure 2 shows how to turn a set of test goals into a common observer automaton specification. The observer automaton monitors the control-flow edges reached and traverses from the initial, safe state q_0 to the violation state q_e when an edge from the set of test goals is explored.

2.3 Generating tests from counterexamples

CoVeriTest employs reachability analyses to construct feasible counterexamples for test-goal specification violations. These counterexamples reflect executions that cover at least one test goal. Since counterexamples are not standard tests, they need to be transformed into tests. More concretely, we need to extract a test vector, i.e., a sequence of test inputs for parameters and external methods like `random`, `scanf`, `__VERIFIER_nondet_int`. We write our test vectors in the format[2] used by the International Competition on Software Testing [12].

[2] https://gitlab.com/sosy-lab/software/test-format/blob/master/doc/Format.md.

To compute a test vector from a feasible counterexample, we follow the approach of BLAST [13], which we briefly sketch now:

1. Compute the strongest postcondition [62] for the counterexample. We use an encoding that applies skolemization to replace existential quantifiers and that works similar to static single assignment [1].
2. Get a satisfying assignment for the strongest postcondition formula, e.g., using an SMT solver.
3. In the order of the occurrence of all inputs, find out the corresponding variable for the input in the formula, look up its value in the satisfying assignment, and write the value to the test vector.

For example, consider the following counterexample:

$$\ell_0 \xrightarrow{\texttt{s=1;}} \ell_1 \xrightarrow{\texttt{i=0;}} \ell_2 \xrightarrow{\texttt{!(n<0)}} \ell_7 \xrightarrow{\texttt{i<n}} \ell_8 \xrightarrow{\texttt{s==1}} \ell_9$$

The strongest postcondition for this counterexample is

$$s_1 = 1 \wedge i_1 = 0 \wedge \neg(n_0 < 0) \wedge i_1 < n_0 \wedge s_1 = 1$$

and a corresponding satisfying assignment is

$$n_0 \mapsto 1 \qquad i_1 \mapsto 0 \qquad s_1 \mapsto 1 \ .$$

The only input in our example is the parameter n, which is referenced by n_0 in the formula. The generated test looks as follows:

```
<testcase>
  <input>1</input>
</testcase>
```

2.4 Abstract reachability graphs

CoVeriTest uses abstract reachability graphs (ARGs) [13] for cooperation between analyses: ARGs are the primary artifacts for exchanging information inside CoVeriTest. We also extract counterexamples, which we need for test generation, from this data structure. An ARG is constructed for a program $P = (L, \ell_0, G)$ and tracks the work performed by a reachability analysis, that is, stores the abstract state space that has been explored so far and the frontier nodes (abstract states that still need to be explored). The representation of the explored abstract state space (abstract states and successor relation) depends on the respective reachability analysis. For example, a value analysis represents abstract states as value assignments, while a predicate-abstraction analysis represents abstract states as predicates. Additionally, the ARG keeps the information about the abstraction level of an analysis, e.g., tracked variables and considered predicates, respectively.

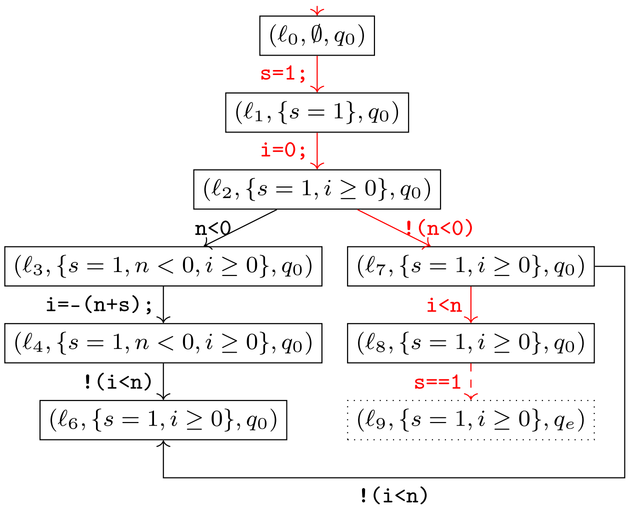

$(\ell_0, \emptyset, q_0)$
s=1;
$(\ell_1, \{s=1\}, q_0)$
i=0;
$(\ell_2, \{s=1, i \geq 0\}, q_0)$
n<0
!(n<0)
$(\ell_3, \{s=1, n<0, i \geq 0\}, q_0)$
$(\ell_7, \{s=1, i \geq 0\}, q_0)$
i=-(n+s);
i<n
$(\ell_4, \{s=1, n<0, i \geq 0\}, q_0)$
$(\ell_8, \{s=1, i \geq 0\}, q_0)$
!(i<n)
s==1
$(\ell_6, \{s=1, i \geq 0\}, q_0)$
$(\ell_9, \{s=1, i \geq 0\}, q_e)$
!(i<n)

Fig. 3 ARG for example program `foo`

Definition 3 (*Abstract Reachability Graph*) An *abstract reachability graph* $ARG = (N, succ, root, W, \pi)$ for a CFA $P = (L, \ell_0, G)$ consists of

- a set N of explored abstract states,
- a relation succ $\subseteq N \times G \times N$ that describes the already explored successor relations,
- the abstract state $root \in N$,
- a set $W \subseteq N$ of frontier nodes, and
- a precision π determining the abstraction level.

Next to the syntactical requirements mentioned in the above definition, an ARG must also fulfill the following semantic requirement: Each ARG node $n \in N$ must either be contained in W or all its abstract successors must have been explored. We do not formalize this requirement since this would require a complete formalization of abstract reachability analyses.

Figure 3 shows an ARG for our example `foo`. For simplicity, we labeled ARG edges by program operations instead of CFA edges. The reachability analysis constructing the ARG finished the exploration of the outer if-branch and stopped exploration when reaching a specification violation after executing statement `s==1`. The path in red shows the counterexample paths from Sect. 2.3. Furthermore, dotted states belong to the set of frontier nodes. All abstract states (nodes) contain information about the program counter (location information). Data values are abstracted by a set of predicates [71]. Here, we consider predicates $s = 1, i \geq 0$, and $n < 0$. Furthermore, the test-goal specification from Fig. 2 is monitored. For the example, we assume that the set of test goals only contains the conditions of the if-statements in the while-loop, e.g., all other test goals for branch coverage have already been covered.

2.5 Condition

Conditions were first proposed as an information exchange format in conditional model checking [23], a cooperative verification approach. A condition describes which program execution paths have already been explored by an analysis. Analyses can use conditions to focus their exploration on unexplored paths. However, verifiers that do not understand conditions can safely ignore them.

A condition is commonly modeled as an automaton that accepts the verified program paths. Verified program paths are described using syntactical program paths in combinations with data assumptions on the syntactical paths. For this paper, we ignore assumptions in our formal definition of conditions.

Definition 4 A *condition* $A = (Q, \delta, q_0, F)$ consists of

- a finite set of states Q,
- a transition relation $\delta \subseteq (Q \setminus F) \times 2^G \times Q$, where G denotes the set of control-flow edges of a program,
- an initial state $q_0 \in Q$, and
- a set $F \subseteq Q$ of accepting states.

The specific condition $A_{\text{none}} = (\{q_0\}, \emptyset, q_0, \emptyset)$ describes that no exploration has been done, i.e., the complete state space needs to be explored.

In CoVeriTest, we use conditions to focus test-case generation on paths that have not been explored by the previous analysis run. Conditions used in CoVeriTest do not impose any data restriction on syntactical paths.

Like in sequential conditional model checking [23], CoVeriTest will extract conditions from ARGs. Alg. 1 describes the extraction process. The idea is to copy incompletely explored ARG paths (i.e., paths ending in a frontier node $f \in W$) to the condition and to reduce the completely explored state-space parts to a single accepting state. In lines 2–6, the algorithm performs a backward search from the set W of incompletely explored nodes to detect all nodes on paths leading to frontier nodes. All these detected nodes become states in the condition. Line 7 builds the set of accepting states, which represent all non-extendable completely explored subgraphs of the ARG. Thus, we add one accepting state, whenever an ARG edge leads from a Q-state (state on an incompletely explored path) to a non-Q-state. Line 8 puts it all together and constructs a correct condition. The condition states are given by the union of the accepting states and the states detected in lines 2–6. The transition relation is a restriction of the ARG's successor relation to all edges on incompletely explored paths.

Figure 4 shows the condition constructed by Alg. 1 from our example ARG shown in Fig. 3. For the sake of representation, we labeled the edges only by operations instead of complete CFA edges. Since reachability analyses typically construct ARGs that are unrollings of CFAs, conditions extracted from ARGs are sparse and can be efficiently represented by an adjacency list.

Algorithm 1 Extracting condition from ARG

Input: arg = $(N, succ, root, W, \pi)$
Output: extracted condition

```
1: $Q$={$root$}∪ $W$;  waitlist=$W$;
2: while (waitlist ≠ ∅) do
3:     pop $q$ from waitlist
4:     for each $(p,(\ell,op,\ell'), q) \in succ$ do
5:         if ( $p \notin Q$) then
6:             $Q = Q \cup \{p\}$; waitlist = waitlist ∪ {$p$};

7: F = $\{q \mid \exists(p, g, q) \in succ \wedge p \in Q \wedge q \notin Q\}$;

8: return $(Q \cup F, root, succ \cap Q \times G \times (Q \cup F), F)$;
```

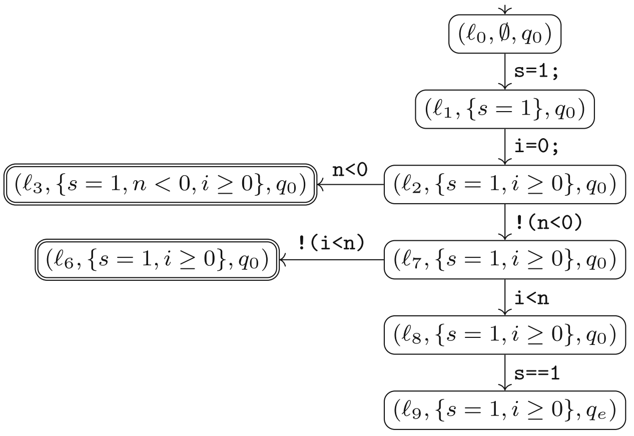

$(\ell_0, \emptyset, q_0)$
s=1;
$(\ell_1, \{s=1\}, q_0)$
i=0;
$(\ell_2, \{s=1, i \geq 0\}, q_0)$
n<0
$(\ell_3, \{s=1, n<0, i \geq 0\}, q_0)$
!(n<0)
$(\ell_7, \{s=1, i \geq 0\}, q_0)$
!(i<n)
$(\ell_6, \{s=1, i \geq 0\}, q_0)$
i<n
$(\ell_8, \{s=1, i \geq 0\}, q_0)$
s==1
$(\ell_9, \{s=1, i \geq 0\}, q_e)$

Fig. 4 Condition constructed from ARG shown in Fig. 3

2.6 Test generation with reachability analyses

Having introduced the basics, we finally describe how to generate tests with a single reachability analysis. Algorithm 2 depicts the test-generation workflow. For test generation, the algorithm of course needs the program and the set of test goals. We also provide a time limit for test generation. To apply the algorithm in CoVeriTest, we need two additional inputs, an initial ARG and a condition. These two inputs provide the information gained in cooperation to the component test-generation algorithm. Basically, they guide the state-space exploration. Without cooperation one would use condition A_{none} and an ARG consisting only of the root node (the beginning of the state-space exploration) and an empty precision.

First, Algorithm 2 initializes the data structures for the test suite and the set of covered goals. Additionally, it generates the initial specification as shown in Fig. 2. Then, it generates

Algorithm 2 Generating tests with a (conditional) reachability analysis

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

1: test_suite=$\emptyset$;  covered=$\emptyset$;
2: $\varphi$=generate_specification(goals);

3: while (goals $\neq \emptyset$ and arg.W $\neq \emptyset$ and elapsed<limit) do
4:     arg = explore(prog, $\varphi$, arg, $\psi$, limit − elapsed);

5:     if (arg.W $\neq \emptyset$ and elapsed<limit) then
6:         $\tau$ = extract_counterexample_path(arg);
7:         test_suite = test_suite $\cup$ generate_test($\tau$);

8:         covered = covered $\cup$ {last_edge($\tau$)};
9:         goals = goals\{last_edge($\tau$)};

10:    $\varphi$=generate_specification(goals);
11: return (test_suite, covered, arg);
```

tests until all test goals are covered, the state space is explored completely (arg.W $=\emptyset$), or the time limit is exceeded. Finally, it returns the generated test suite, the set of covered goals, and the last ARG built. The ARG is only returned to enable cooperation.

To generate tests, Algorithm 2 continues the exploration of the current ARG taking into account program prog, specification φ, current ARG arg, (if understood) condition ψ, and the remaining test-generation time. The exploration stops due to three reasons: (1) the state space is explored completely (arg.W $=\emptyset$), (2) the time limit exceeded, or (3) a counterexample has been found.[3] Reasons (1) and (2) indicate that the test-generation process should be stopped. Only when reason (3) applies, a test is generated. The test is generated from a counterexample. First, the counterexample path, which is a path from the root to a violating state, needs to be extracted from the ARG. Since only the traversal of a test goal leads to a specification violation (see Fig. 2), the last edge on the path is a test goal. After path extraction, Alg. 2 generates a test from the counterexample path following the procedure explained earlier and adds the test to the test suite. To finish test generation, the covered test goal (last edge of the counterexample path) is added to the set of covered test goals and removed from the set of (open) test goals. Since the set of test goals might change during loop-body execution, finally the specification φ is updated.

[3] For the presentation, we assume that the exploration does not stop if an *infeasible* counterexample is found. In practice, we add a counterexample check to imprecise analyses and skip lines 5–10 whenever the check does not confirm a counterexample.

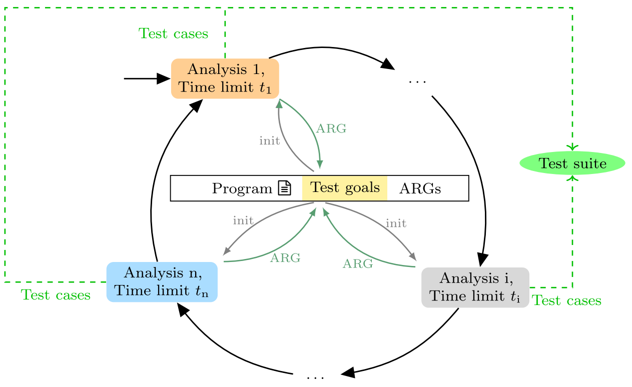


Fig. 5 Visualization of CoVeriTest's analysis cycle

3 CoVeriTest approach

The previous section introduced the basic concepts for test generation with a single component reachability analysis. In this section, we describe CoVeriTest.

3.1 CoVeriTest workflow

CoVeriTest combines different reachability analyses for test generation to accommodate for the different strengths and weaknesses of reachability analyses: certain test goals are harder to cover for one analysis than for another. To optimize the number of covered goals while keeping the instantiation effort of the combination simple, we decided to rotate analyses for test generation in cycles. In contrast to a sequential combination, analyses that get stuck trying to cover a particular goal may recover later. In advantage over parallel combination possibilities, we avoid to cover the same goals in parallel. In advantage over algorithm selection, we do not need to know in advance which analysis can cover which goal. Moreover, CoVeriTest supports cooperation among analyses, allowing them to exchange information about their exploration using ARGs.

Before we explain the CoVeriTest algorithm (Alg. 3), we provide an overall description of the workflow using the illustration in Fig. 5. While the standard test-generation in Alg. 2 gets an ARG and a condition, CoVeriTest gets a sequence of analysis configurations. An *analysis configuration* is a pair of an analysis and an individual time limit for that analysis. An *analysis run* is the execution of an analysis configuration. One *analysis cycle* is a sequence of n analysis runs, which is defined by a sequence of n analysis configurations. The analysis cycle is the core of CoVeriTest's alternating test-generation process, according to which ARG and condition are set up before running an analysis. As illustrated in Fig. 5, CoVeriTest cycles through a sequence of n analysis runs, each defined by an analysis configuration (analysis i, time limit i). Each analysis run gets initialized

Algorithm 3 CoVeriTest: alternating reachability analyses to generate tests

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

1:  test_suite = $\emptyset$;  args = $\langle\rangle$;  current = 0;
2:  while (goals $\neq \emptyset$ and elapsed_time<total_limit) do
3:      analysis = configs[current].first;
4:      limit = configs[current].second;

5:      (arg, $\psi$) = coopAndInit(prog, args, configs.length);
6:      (tests, covered, arg) =
            analysis(prog, goals, limit, arg, $\psi$);

7:      test_suite=test_suite $\cup$ tests;
8:      goals = goals\covered;  args = args $\circ$ $\langle$arg$\rangle$;

9:      if (arg.$W$ = $\emptyset$) then
10:         return test_suite;
11:     current = (current+1) % configs.length;
12: return test_suite;
```

Algorithm 4 coopAndInit: 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:  $\psi = A_{\text{none}}$;  $\pi = \emptyset$;  root = $(\ell_0, \top)$;
2:  if (length(args)$\geq$numAnalyses) then
3:      if (reuse-arg) then
4:          return
                (last_arg_of_analysis(numAnalyses, args), $\psi$);
5:      if (reuse-precision) then
6:          $\pi$ = last_arg_of_analysis(numAnalyses, args).$\pi$;
7:  if (length(args)>0) then
8:      if (use-condition) then
9:          $\psi$ = extract_condition(args[length(args)-1]);
10:     if (transform-precision) then
11:         $\pi = \pi \cup$ transform(args[length(args)-1].$\pi$);
12: return (({root}, $\emptyset$, root, {root}, $\pi$), $\psi$);
```

with information from the central data store, which contains the program, the remaining test goals, and a sequence of previously constructed ARGs. When an analysis terminates, it adds its constructed ARG in the central data store and adds the produced test cases to the central test suite.

Now consider Alg. 3, which more formally defines the overall CoVeriTest algorithm. The inputs `prog`, `goals`, and `total_limit` are similar to the inputs in Alg. 2. At the beginning (line 1), CoVeriTest initializes the test suite, the sequence of ARGs, and the index of the current analysis configuration. Then, it iterates over the analysis configurations. In each iteration, CoVeriTest extracts the current analysis configuration (pair of analysis and time limit[4]), sets up and runs the respective analysis, and afterward registers the results of the analysis run in its data structures. The newly generated tests are added to the test-suite. Covered test goals are removed from the set of (open) test goals, and the sequence of ARGs is extended by the returned ARG. The iteration stops if all test goals are covered, the global time limit is exceeded, or the waitlist W of the current analysis is empty. At the end, CoVeriTest returns the generated test suite.

Now, let us look closer at the specifics of the CoVeriTest workflow. In line 9, CoVeriTest checks whether the last analysis run finished its exploration. If the analysis finished its exploration, then all reachable goals are detected and all remaining, uncovered goals are unreachable. Thus, the generated test suite is returned. Otherwise, the index of the next analysis configuration is set for the next iteration. The method `coopAndInit` sets up the analysis, i.e., it prepares the ARG and condition. During preparation, it may reuse information from previous ARGs and, thus, supports the cooperation between different iterations in CoVeriTest. The cooperation options are explained next when describing the method `coopAndInit`.

Method `coopAndInit` (Alg. 4) is responsible for the setup of the cooperation between analysis runs. CoVeriTest provides the following 4 *cooperation options*: `reuse-arg`, `reuse-precision`, `use-condition`, and `transform- precision`. We distinguish between two types of cooperation, which mainly differ in the cooperation partner. First, we provide cooperation between different analysis runs of the same analysis (in different cycles): `reuse-arg` and `reuse-precision`. Currently, CoVeriTest allows to reuse the complete ARG or the precision[5] of the previous run of an analysis. Second, we offer cooperation between different analyses: `use-condition` and `transform- precision`. On the one hand, CoVeriTest can use conditions to exclude those paths from future exploration that have been explored by another analysis. On the other hand, CoVeriTest can refine the abstraction level of a given analysis by adding information from the precision of other analyses to the precision of the given analysis.

Except for the `reuse-arg` option, all cooperation options can be combined arbitrarily. The option `reuse-arg` cannot be combined with the other three options due to the following reasons: (1) a combination with the `reuse-`

[4] Generally, fixed time limits can become problematic if certain counterexample can only be found using more time than provided by the time limit. However, CoVeriTest provides a cooperation type in which analyses continue their previous exploration, i.e., the iteration time limit is transparent to the analysis.

[5] The *precision* specifies the level of abstraction of the abstract model that the analysis constructs.

`precision` option is always implied because the ARG already contains the precision. (2) A combination with the options for cooperation between different analyses is technically incompatible. To integrate information from other analyses in the exploration, currently the analysis needs to restart its exploration.

CoVeriTest uses ARGs to exchange information between analyses. Thus, callers of the method `coopAndInit` must transfer the sequence of already generated ARGs. To distinguish ARGs constructed by the same analysis from ARGs constructed by different analyses, Alg. 4 is also provided the number of different analysis configurations used by CoVeriTest.[6] Additionally, the program, which is needed for the setup, is passed to `coopAndInit`.

After having clarified the inputs, we now explain the behavior of Alg. 4. In line 1, it starts with the initialization of the ARG components and the condition. Thereby, it is pessimistic and assumes that no cooperation is enabled. The condition is set to the condition A_{none}, i.e., no path has been explored, and the precision is empty. Additionally, the root node describes all states pointing to the beginning of the program, i.e., the beginning of any new exploration. Lines 2–11 update precisions and conditions according to the configured cooperation options. Line 4 or line 12 returns the setup for the next analysis run. The ARG returned in line 12 only contains the root node, which needs to be explored. Hence, the next analysis run will restart the state-space exploration.

Lines 2–6 are responsible for cooperation of the same analysis. Such a cooperation is of course only possible when the analysis has been run before, i.e., CoVeriTest iterated over all analysis configurations at least once. For cooperation, we only consider the last ARG constructed by the analysis. Since the configuration is not changed during execution of CoVeriTest, the last ARG should contain all reuse information of the analysis' previous ARGs. Lines 3 and 4 handle the `reuse-arg` option, which is incompatible with the remaining options. Thus, the last ARG of the analysis is looked up and returned together with the non-restricting condition $\psi = A_{\text{none}}$. Lines 5 and 6 handle the `reuse-precision` option. Like in `reuse-arg`, the last ARG of the analysis is extracted. However, line 6 only updates the precision to the precision of that last ARG.

Lines 7–11 show the setup of the cooperation between different analyses. Cooperation between different analyses becomes possible after the first analysis run, i.e., the first loop iteration in Alg. 3 finished. Currently, CoVeriTest only considers the last generated ARG. Lines 8 and 9 handle the `use-condition` option, which extracts a condition that describes the paths explored by the previous ARG. The extraction process has been described in the previous section. Lines 10 and 11 handle the `transform-precision` option. This option refines the abstraction level (i.e., the precision) of the next analysis with information obtained from the precision used in the last analysis run. The next paragraph describes how precisions can be transformed for different analyses.

[6] Note that the index of the current analysis is not provided since it can be computed from the length of the sequence of ARGs and the number of analysis configurations.

3.2 Transformation of analysis precision

CoVeriTest transforms precisions to propagate knowledge about the abstraction level necessary for the reachability analyses from one analysis to another. The incentive of this information exchange is to avoid to rediscover the knowledge about the required abstraction level in expensive refinement steps. However, note that different analyses use different types of precisions. A precise transformation might not always be possible. Furthermore, there does not exist one transformation procedure for all types of precisions, but the procedure depends on the type of input and output precision.

If the format of the input and output precision is the same, the transform method will simply be the identity function. For all other pairs of input and target precision type, we need a dedicated transformation procedure. So far, CoVeriTest only supports one such dedicated transformation procedure that allows the transformation of a set of predicates into a set of tracked variables. The idea of this transformation procedure is that if a variable is used in a predicate, the variable will be somehow relevant for the reachability analysis. Thus, the procedure adds all variables occurring in a predicate to the returned target precision, formally $\pi_V = \bigcup_{p \in \pi_P} \text{vars}(p)$.

For example, consider the following set of predicates $\pi_P = \{i \geq 0, n < 0, s = 1\}$, a possible precision of the predicate analysis. Transforming the precision π_P into a set of tracked variables, e.g., a precision for the value analysis, results in $\pi_V = \{i, n, s\}$.

3.3 Implementation

We integrated our CoVeriTest approach into the software-analysis framework CPAchecker [29]. This framework is highly configurable and provides a large number of different reachability analyses. Due to its support for conditional model checking, CPAchecker also contains an implementation for condition extraction. Furthermore, CPAchecker supports various export formats for counterexamples. Thus, the generation of tests from counterexamples was already available in CPAchecker. To integrate CoVeriTest in CPAchecker, we basically implemented Alg. 2 and an algorithm integrating Alg. 3 and Alg. 4. While we could have used CPAchecker's specification format (observer automata) to provide the test-goal specification to the analyses, it is technically quite difficult to adapt CPAchecker's observer automaton whenever the set of test goals changes. Thus, we implemented our own updatable observer component to monitor uncovered test goals. Our observer component is a direct implementation of automata like the one shown in Fig. 2. It is integrated as a configurable program analysis [22], CPACHECKER's interface for an analysis (component). This way it can be easily composed with reachability analyses.

4 Experimental evaluation

We study COVERITEST cooperation setups using a combination of either explicit-state model checking—named value analysis in the following (Val)—and predicate abstraction (Pred), or bounded model checking (BMC) and symbolic execution (SymExec). The detailed cooperation setups are described later in this section. As test goals, we use branches. Branch coverage is a commonly used coverage criterion that is supported by many test-generation tools and is easy to express as a set of test goals. We evaluate our experiments along the following research questions.

4.1 Research questions to evaluate

In the following, we list the research questions together with brief mentioning of the results that we obtained, which we later describe in more detail.

RQ 1. Time Limits for Val+Pred. We study cooperation setups using Val+Pred, and compare the coverage achieved by cooperation setups that use the same reuse type (i.e., the same configuration of the cooperation options), and thus, only differ in the time limits. *Result:* The combination Val+Pred performs best if more runtime is assigned to the stronger predicate analysis.

RQ 2. Reuse in Val+Pred. From all sets of cooperation setups using Val+Pred that differ only in the time limits, we select the best and compare these. *Result:* The combination Val+Pred achieves higher coverage if it reuses own information and does not use conditions for the predicate analysis.

RQ 3. Time Limits for BMC+SymExec. We study cooperation setups using BMC+SymExec, and compare the coverage achieved by cooperation setups that use the same reuse type, and thus, only differ in the time limits. *Result:* The combination BMC+SymExec performs well if switches between analyses are avoided.

RQ 4. Reuse in BMC+SymExec. From all sets of cooperation setups using BMC+SymExec that differ only in the time limits, we select the best and compare these. *Result:* The combination BMC+SymExec performs best if the BMC analysis is restricted by a condition.

RQ 5. Best combination. We compare the coverage results of the best cooperation setup of each verifier combination. *Result:* COVERITEST performs best using combination Val+Pred.

RQ 6. Cooperation versus single analysis. For each of our analysis combinations, we compare the coverage achieved by the best cooperation setup using that analysis combination with the coverage achieved by one of the analyses of the combination alone. *Result:* Cooperative test-generation often performs better than a single analysis.

RQ 7. Cooperation versus parallel analyses. For each analysis combination, we compare the coverage achieved by the best cooperation setup for that analysis combination with the coverage achieved when running the analyses of the combination in parallel. *Result:* An interleaved combination often performs better than a parallel combination.

RQ 8. Cooperation versus other tools. We let the best cooperation setup construct test suites in the same environmental setup as in the International Competition on Software Testing (Test-Comp 2019).[7] Then, we compare the coverage of COVERITEST, which is measured by the Test-Comp 2019 validator, with the coverage of the best two test-generation tools from Test-Comp 2019. *Result:* Cooperative test-generation with COVERITEST complements existing test-generation tools.

4.2 Experimental setup

We now describe the setup of our experiments, i.e., the cooperation setups of COVERITEST, the tools and the evaluation tasks, as well as the computing resources.

4.2.1 COVERITEST cooperation setups

A COVERITEST *cooperation setup* consists of (1) a sequence of analysis configurations (each a pair of analysis and time limit, see Sect. 3.1) and (2) one of 10 reuse types (in order to configure Alg. 4 with the respective cooperation options). We restrict our evaluation to the 4 program analyses Val, Pred, BMC, and SymExec.

Value analysis (Val). CPACHECKER's value analysis [35] explicitly tracks the variable values of all variables in its current precision. Untracked variables are abstracted by any value. To determine which variables to track, the value analysis combines counterexample-guided abstraction refinement (CEGAR) [53] with path-prefix slicing [38] and refinement selection [37]. Value analysis can be efficient if only few variable values need to be tracked. If many different values are assigned to variables (e.g., for loop counters), then huge state spaces might be created.

Predicate analysis (Pred). CPACHECKER's predicate analysis applies predicate abstraction with adjustable block encoding (ABE) [30]. ABE abstracts at dedicated locations (in our case

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

loop heads) and computes the strongest postcondition at all remaining locations. The precision of the predicate analysis is a set of predicates that is determined with CEGAR [53], lazy refinement [77], and interpolation [74]. The predicate analysis is powerful and often handles loops easily. However, computing abstractions requires expensive SMT solver calls.
Bounded model checking (BMC). CPACHECKER's bounded model checking iteratively unrolls the CFA up to a given loop bound *k* while simultaneously encoding the unrolled CFA and the property specification into a formula. To find counterexamples, the satisfiability of the generated formula is checked. CPACHECKER's BMC formula encoding is based on the unified SMT-based approach for software verification [20]. Furthermore, BMC is enhanced with constant propagation applied to the unrolled CFA to rule out simple, infeasible paths. The loop bound *k*, which is the precision of BMC, is increased iteratively. BMC is very precise, but it may not terminate in case of unbounded loops and the satisfiability checks can become costly.
Symbolic execution (SymExec). We use SYMEX$^+$ [31], an approach that combines symbolic execution [93] and CEGAR [53]. During CEGAR we apply path-prefix slicing [38] and refinement selection [37]. The CEGAR approach determines which variables to track symbolically and which path condition constraints to consider. Tracked variable values and constraints form the precision. Symbolic execution can be efficient if it only tracks few symbolic variables and constraints, but may struggle with loops or many symbolic variables and constraints.

As in our previous work [26], we combine value and predicate analysis. Additionally, we combine BMC and symbolic execution. Note that we neither combine value analysis and symbolic execution nor BMC and predicate analysis because the latter analyses can subsume the first analyses. Furthermore, we do not consider a combination of value analysis and BMC because BMC uses constant propagation already, which is a special case of value analysis. We also excluded a combination of symbolic execution and predicate analysis since it is similar to the combination of value and predicate analysis (predicate analysis tracks the symbolic values already).

To complete the analysis configurations, we need to specify the time limit for each analysis run. We are interested in two questions: (1) Are switches between analyses beneficial for the test coverage and (2) does the coverage benefit from a non-uniform distribution of the time resources, i.e., different analyses get different individual time limits? To this end, we select four time limits (10 s, 50 s, 100 s, 250 s) that are applied to both analyses and trigger switches often, sometimes, or rarely. Furthermore, we apply the two non-uniform time-limit pairs (20 s, 80 s) and (80 s, 20 s). Combining all 6 time-limit pairs with the two analysis combinations, we get 12 analysis configurations.

plain: Ignores all ARGs from previous analysis runs.
$co_i = co_j = \{\}$

reuse-precision: In every analysis cycle, each analysis restarts the exploration with its precision from the previous analysis cycle.
$co_i = co_j = \{\texttt{reuse-precision}\}$

reuse-arg: Both analyses continue the exploration of their ARG from the last analysis cycle.
$co_i = co_j = \{\texttt{reuse-arg}\}$

cond$_{A_i}$: Analysis A_i does not reuse any ARG information and analysis A_j gets the condition extracted from the last ARG from analysis A_i.
$co_i = \{\}, co_j = \{\texttt{use-condition}\}$

cond: Both analyses get the condition that is extracted from the last ARG (from the other analysis).
$co_i = co_j = \{\texttt{use-condition}\}$

cond$_{A_i}$+r: Similar to cond$_{A_i}$, but additionally analysis A_i continues the exploration of its ARG from the previous analysis cycle and analysis A_j reuses its precision from the previous analysis cycle.
$co_i = \{\texttt{reuse-arg}\}$,
$co_j = \{\texttt{use-condition}, \texttt{reuse-precision}\}$

cond+r: Like cond, but additionally both analyses reuse their precision from the previous analysis cycle.
$co_i = co_j = \{\texttt{use-condition}, \texttt{reuse-precision}\}$

trans-prec: The first analysis A_1 in an analysis cycle uses the transformed precision from the last ARG (i.e., from the second analysis in the previous cycle). The second analysis A_2 does not reuse any ARG information.
$co_1 = \{\texttt{transform-precision}\}$, $co_2 = \{\}$

trans-prec+r: The first analysis A_1 in an analysis cycle uses the transformed precision from the last ARG (of the second analysis) and its own precision from the last analysis cycle. The second analysis A_2 continues exploration with its ARG from the previous analysis cycle.
$co_1 = \{\texttt{reuse-precision}, \texttt{transform-precision}\}$,
$co_2 = \{\texttt{reuse-arg}\}$

trans-prec+cond+r: Like cond+r, but the first analysis A_1 in an analysis cycle uses a combination of its own precision and the precision extracted from the last ARG.
$co_1 = \{\texttt{use-condition}, \texttt{reuse-precision}, \texttt{transform-precision}\}$
$co_2 = \{\texttt{use-condition}, \texttt{reuse-precision}\}$

Fig. 6 Reuse types specify the cooperation types for COVERITEST experiments. We abbreviate the set of cooperation options for analysis A_i as co_i. We assume that the cooperation setup is for two analyses A_i and A_j

In the following, we describe the reuse types that we use in our experiments to configure COVERITEST's cooperation setups. A *reuse type* specifies the cooperation options for Alg. 4, which prepares the initial ARG (including the precision) and the condition for the next analysis run. The algorithm has access to ARGs from previous analysis runs. Thus, the reuse type defines the cooperation between analysis runs. For the COVERITEST cooperation setup that we experiment with, we need to specify for both analyses how they use the information from previous ARGs. In our experiments, we use the 10 reuse types (2 of them are parametric) listed in Fig. 6.

The first seven reuse types are supported for all combinations of analyses; the last three types are only supported for Val+Pred, because the precisions of symbolic execution and BMC are not convertible. Additionally, reuse types cond$_{A_i}$ and cond$_{A_i}$+r are parametric. Analysis A_i can be instantiated with either analysis in the combination. Thus, we end up with 12 different reuse types for analysis combination Val+Pred and 9 for analysis combination BMC+SymExec. Combining the 12 analysis configurations (6 per analysis combination) with all compatible reuse-types, we obtain 126 cooperation setups (72 for Val+Pred, 54 for BMC+SymExec).

4.2.2 Tools

CoVeriTest is part of the software-analysis framework CPAchecker. For our experiments, we use version `cpachecker-1.8-coveritest-sttt` (revision 31 599) of CPAchecker. To compare CoVeriTest with state-of-the-art tools, we use the two best tools from Test-Comp 2019: VeriFuzz [50] and Klee [43]. We use their versions submitted to Test-Comp 2019. Klee uses symbolic execution. VeriFuzz is a test-generation tool based on the fuzzer AFL. To improve on AFL, VeriFuzz applies verification techniques to compute initial inputs and to set the parameters for AFL. For the comparison of CoVeriTest with VeriFuzz, and Klee, we used VeriFuzz's and Klee's results[8] from Test-Comp 2019 [12],[9] where the coverage of the test suites was measured using the test-suite validator TestCov [34] in version v1.2,[10] which is based on gcov[11] to measure branch coverage.

Note that we need to measure the branch coverage externally (using the original program) for this comparison because due to internal program transformations in CPAchecker, especially splitting of branch conditions, branches considered by CoVeriTest may differ from the actual program branches. Since all CoVeriTest cooperation setups are based on the same tool (CPAchecker), we do not need to measure branch coverage externally when comparing the CoVeriTest cooperation setups. Thus, we compare the number of generated tests when comparing CoVeriTest cooperation setups.

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

[9] Note that this is only possible because for the comparison we execute CoVeriTest in the environment (hardware, resource limits, etc.) used for Test-Comp 2019.

[10] https://gitlab.com/sosy-lab/software/test-format/tags/v1.2; more recent versions of TestCov are available from https://gitlab.com/sosy-lab/software/test-suite-validator.

[11] https://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

4.2.3 Programs and test goals

The test-generation tools CoVeriTest, Klee, and VeriFuzz generate tests for C programs, and they all participated in Test-Comp 2019. The structural coverage property considered in Test-Comp 2019 is branch coverage. Thus, we use the set of all branches as test goals. To compare these three test tools, we use all 1 720 programs of the Test-Comp 2019 benchmark set[12] that are used to generate tests for the branch-coverage property. For the comparison of the different CoVeriTest cooperation setups, we extend the benchmark set to all 7 644 programs considered in the software-verification competition SV-COMP 2019. Note that this is only possible because we do not need to execute tests to compare CoVeriTest cooperation setups.

4.2.4 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. We use the same resource limits as in Test-Comp 2019. Each test-generation run may use up to 8 processing units, 15 min of CPU time, and 15 GB of memory. Furthermore, the test-suite execution runs, which are required to compare against the other state-of-the-art test-generation tools Klee and VeriFuzz, are granted only 2 processing units and 7 GB of memory, but 3 h of CPU time. Measuring resources and enforcing limits is done by BenchExec [39].

4.2.5 Availability

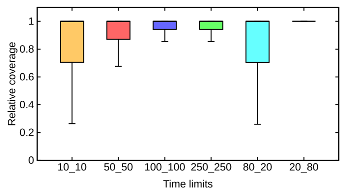
All our experimental data are available online[13] [27].

4.3 Experimental results

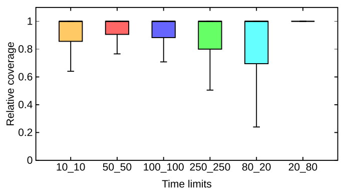
RQ 1. Time Limits for Val+Pred: Assign More Time to Pred. First, we study CoVeriTest cooperation setups that interleave analyses Val and Pred, looking at the configuration of time limits. Next to the already fixed analysis combination, we also fix the reuse type and compare for each of the 12 reuse type all six CoVeriTest cooperation setups that only differ in their time limits. For comparison, we use relative coverage, which is relative to the highest number of covered goals instead of the total number of test goals. We select this measure because it allows a better comparison of the approaches, especially when many test goals are either unreachable or not covered by any of the approaches. To compute the relative

[12] https://github.com/sosy-lab/sv-benchmarks/tree/testcomp19.

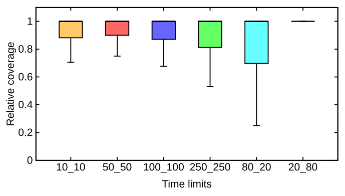
[13] https://www.sosy-lab.org/research/coop-testgen/STTT/.



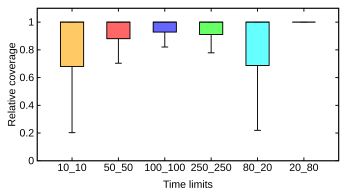
(a) Reuse type plain



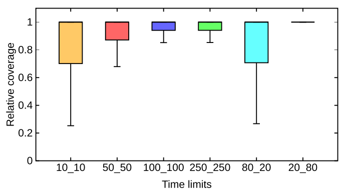
(b) Reuse type reuse-precision



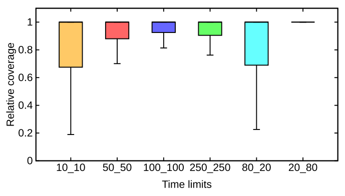
(c) Reuse type reuse-arg



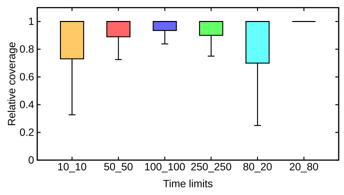
(d) Reuse type cond_v



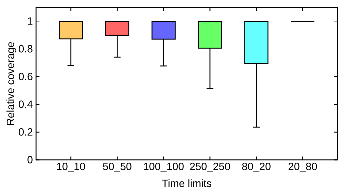
(e) Reuse type cond_p



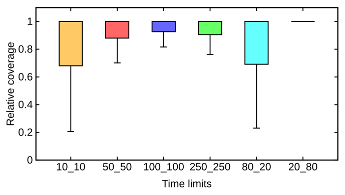
(f) Reuse type cond



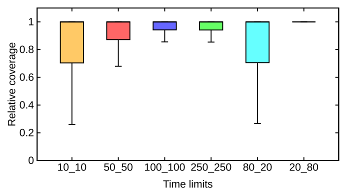
(g) Reuse type cond_v+r



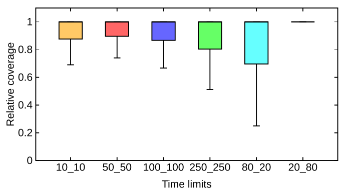
(h) Reuse type cond_p+r



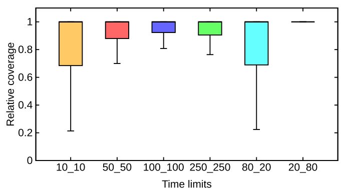
(i) Reuse type cond+r



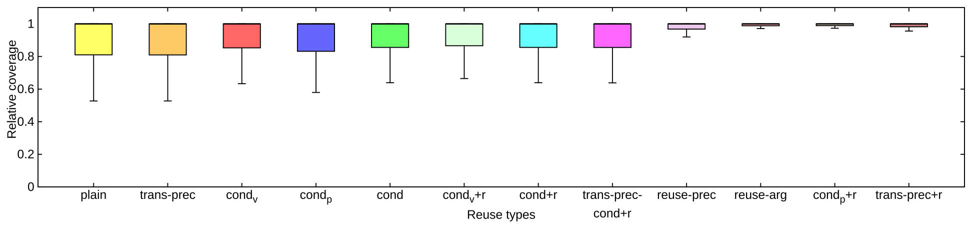
(j) Reuse type trans-prec



(k) Reuse type trans-prec+r



(l) Reuse type trans-prec+cond+r

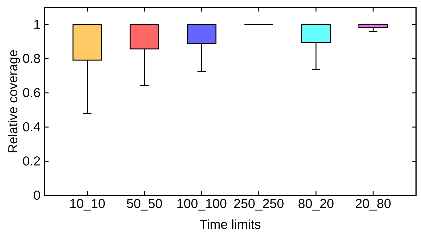


(m) Best limit (20 s, 80 s) comparison

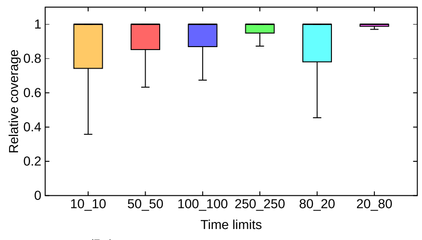
Fig. 7 Comparison of relative coverage (number of covered goals divided by maximal number of covered goals) achieved by 72 different CoVeriTest cooperation setups that use a combination Val+Pred. Figure 7a–l compares the relative coverage for cooperation setups using a fixed reuse type and different time limits. Figure 7m compares the best cooperation setup (which is always (20 s, 80 s)) of each reuse type

coverage for a set of cooperation setups, one extracts per task and cooperation setup the achieved number of covered goals and divides it by the maximum number of covered goals extracted for that task. Figure 7 shows box plots for each reuse type. The box plots show the distribution of the relative coverage. The closer the bottom border of a box is to one, the higher is the coverage achieved. For all 12 reuse types, the last box plot has the bottom border closest to one. These box plots represent cooperation setups that use a time limit of 20 s for Val and 80 s for Pred in each round.

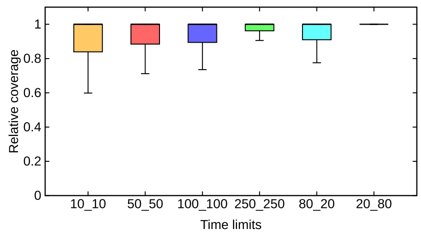
RQ 2. Reuse in Val+Pred: Use Own Information But No Condition for Pred. Up to now, we found out how to configure time limits for CoVeriTest with Val and Pred. Now, we look into the configuration of the reuse type. To this end, we fix the time limit to (20 s, 80 s), the time limit that performed best for each reuse type, and compare the relative coverage of the resulting 12 cooperation setups. Figure 7m shows box plots of the distribution of the relative coverage, which is relative to the highest number of covered goals. Since the highest number of covered goals depends on the compared cooperation setups, boxes in Fig. 7m can be significantly



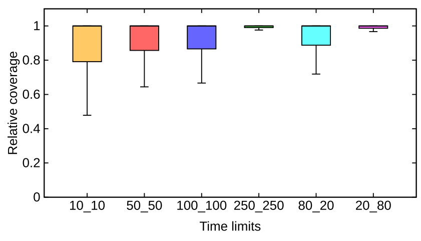
(a) Reuse type plain



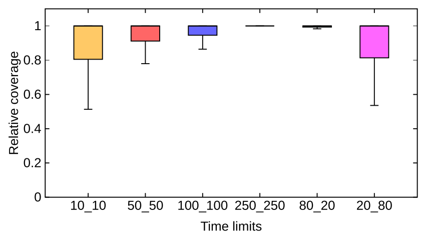
(b) Reuse type reuse-precision



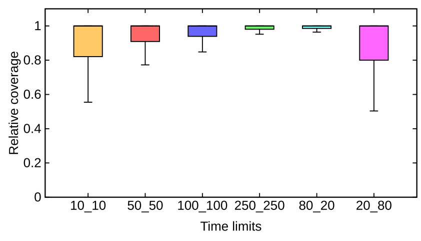
(c) Reuse type reuse-arg



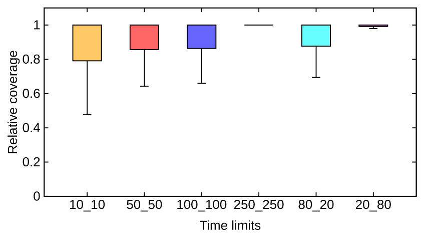
(d) Reuse type cond$_b$



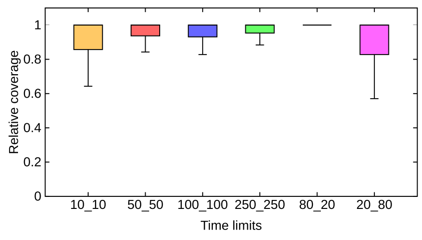
(e) Reuse type cond$_s$



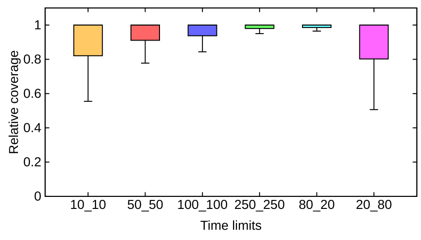
(f) Reuse type cond



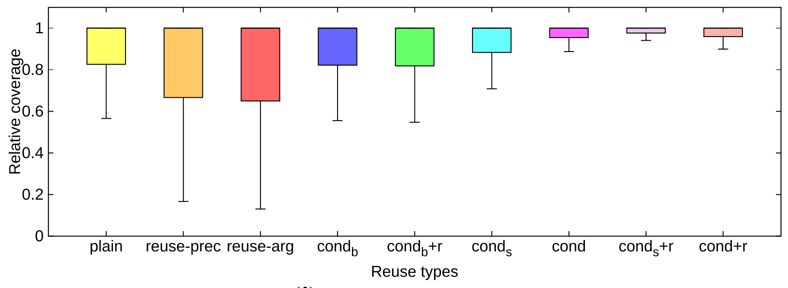
(g) Reuse type cond$_b$+r



(h) Reuse type cond$_s$+r



(i) Reuse type cond+r



(j) Best limit comparison

Fig. 8 Comparison of relative coverage (number of covered goals divided by maximal number of covered goals) achieved by 54 different CoVeriTest cooperation setups that use a combination BMC+SymExec. Figure 8a–i compares the relative coverage for cooperation setups using a fixed reuse type and different time limits. Figure 8j compares the best cooperation setup of each reuse type

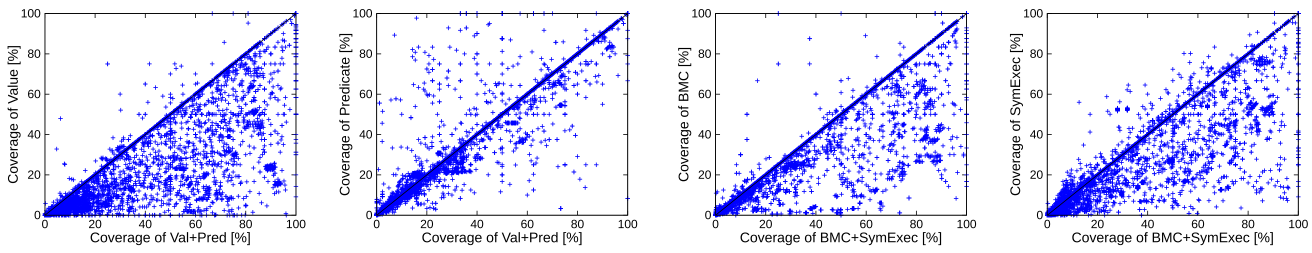


Fig. 9 For both analysis combinations, Val+Pred (first two scatter plotters) and BMC+SymExec (last two scatter plots), we compare the coverage achieved by the respective best CoVeriTest cooperation setup with the coverage achieved when running the single analyses alone

larger than in the respective figure of the reuse-type. The first five box plots in Fig. 7m show all cooperation setups that do not reuse own information. The sixth to eighth box plots show all cooperation setups that reuse own information, but in which Pred uses conditions. The ninth to twelfth box plots show those cooperation setups that reuse own information and do not use conditions for Pred. We observe that these last four box plots are smaller than the remaining box plots and their bottom borders are closer to one. Looking into our raw data, we found out that the best cooperation setup only reuses the ARG.

RQ 3. Time limits for BMC+SymExec: Switch Rarely. Next, we consider CoVeriTest cooperation setups that interleave BMC and SymExec. Again, we first look at the configuration of time limits. As before, we fix the reuse type and compare for each of the 9 reuse types all 6 CoVeriTest cooperation setups that only differ in their time limits. Figure 8 shows box plots for each reuse type. These box plots show the distribution of the relative coverage. For all 9 reuse types, the box plot that shows the time limit configuration (250 s, 250 s) has a bottom border close to one, but not necessarily the closest. Switching rarely is a good choice, but not necessarily the best. This is also supported when comparing BMC and SymExec for test generation in isolation (not in CoVeriTest). Both analyses achieve about the same coverage for one third of the tasks, for one third BMC performs better, and for another third SymExec is best. Nevertheless, when using the condition constructed from the ARG of the SymExec (reuse type $\texttt{cond}_s$, cond, $\texttt{cond}_s$+r, and cond+r) assigning more time to BMC than to SymExec is typically significantly better. A possible explanation is that the condition generated by SymExec might prevent BMC from merging at join points, which makes BMC inefficient. In contrast, if using cooperation option reuse-precision or reuse-arg, it is best to assign more time to SymExec than to BMC. The reason might be that BMC reuses only the loop bound k while SymExec reuses much more information, namely which variables and constraints to track.

RQ 4. Reuse in BMC+SymExec: Use Conditions from SymExec. So far, we learned how to configure time limits for different reuse types of BMC and SymExec. Next, we want to find out how to configure the reuse type. For each reuse type, we select from the six available cooperation setups the one that performed best. Again, we use the relative coverage for comparison, which depends on the compared cooperation setups. Therefore, the relative coverage of a cooperation setup varies when computed for different sets of cooperation setups. Figure 8j shows the box plots of the distributions of the relative coverage. Only the last four box plots show cooperation setups that use conditions constructed from the ARGs of SymExec. Since the last four, especially the last three, boxes are smaller than the first five boxes and their bottom borders are closer to one, we conclude that the

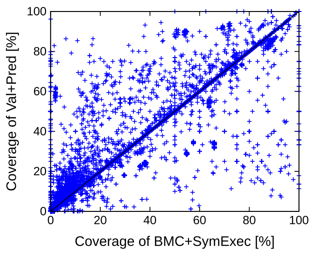


Fig. 10 Comparison of the coverage achieved by the best CoVeriTest cooperation setup for the two analysis combinations used by CoVeriTest

last four cooperation setups achieve higher coverage. The best cooperation setup ($\texttt{cond}_s$+r) reuses own information and restricts BMC to paths not yet explored by SymExec.

RQ 5. Best Combination: Val+Pred is Best. Our goal is to find out which of our analysis pairs performs best. To this end, we compare the coverage, i.e., number of covered goals divided by number of total goals, achieved by the best CoVeriTest cooperation setup of each analysis combination that we evaluated. Figure 10 shows a scatter plot that compares the coverage achieved by the best cooperation setup for BMC+SymExec (x-axis) with the coverage achieved by the best cooperation setup for Val+Pred. Note that we excluded those programs from the scatter plots, for which we miss the number of covered goals for at least one test generator, e.g., due to timeout of the analysis. Looking at the scatter plot, we observe that more data points are in the upper left half, i.e., the CoVeriTest cooperation setup interleaving Val+Pred often performs better. Indeed, the combination Val+Pred achieves a higher coverage for about 40% of the programs, and both combinations achieve the same coverage for another 40 % of the programs.

RQ 6. Cooperation Versus Single Analysis: Combination Better than Single Analysis. To find out whether CoVeriTest benefits from interleaving, we take the best CoVeriTest cooperation setup for each analysis combination (Val+Pred and BMC+SymExec) and compare it against the analyses of the combination. Each single analysis is granted the same time limit for test generation as the CoVeriTest cooperation setup. Figure 9 shows four scatter plots. Each scatter plot compares the coverage achieved by the respective best CoVeriTest cooperation setup (x-axis) with the coverage achieved by one of the CoVeriTest analyses alone. Again, we excluded those programs from the scatter plots, for which we miss the number of covered goals for at least one test generator. Looking at the scatter plots, we see that in the last three scatter plots most of the points are in the lower right half. Thus, the CoVeriTest cooperation setup often achieves a higher coverage than the respective single analysis. The second scatter plot, which compares CoVeriTest using Val+Pred with the predicate analysis, is more diverse. About 53 % of the points are on the diagonal, i.e., both

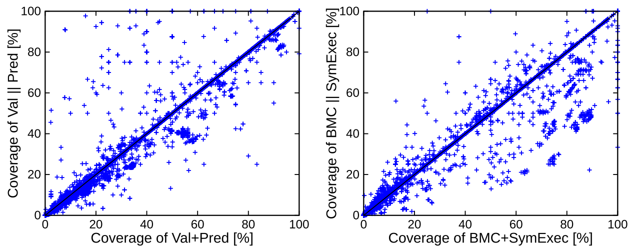


Fig. 11 Comparison of the coverage achieved by CoVeriTest's best cooperation setup using a combination of a combination Val+Pred (left) and BMC+SymExec (right) with the coverage achieved by the parallel combination of the respective analyses

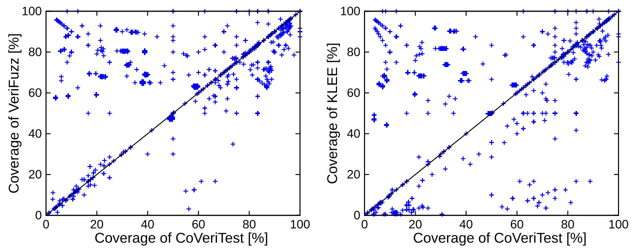


Fig. 12 Comparison of the branch coverage achieved by CoVeriTest (best cooperation setup) with the branch coverage achieved by the existing state-of-the-art test generation tools VeriFuzz (left) and Klee (right)

test generators achieved the same coverage. The predicate analysis achieves higher coverage for about 21 % of the programs (upper left half), while CoVeriTest performs better for 26 % of the programs. CoVeriTest is especially beneficial for programs that only need few variable values to trigger the branches, like `ssh` programs or programs from the `product-lines` subcategory.

RQ 7. Cooperation versus Parallel Analyses: Better Interleave than Parallelize. For both analysis combinations, Fig. 11 compares the coverage of the respective best CoVeriTest cooperation setup (x-axis) with the coverage of a test generator running a parallel combination of the same analyses.[14] As before, the scatter plots in Fig. 11 do not contain data points for which we miss the coverage value for one of the test generators. Looking at the scatter plots, we observe that many points (58 % and 64 %, respectively) are on the diagonal, i.e., the two test generators achieved the same coverage. Furthermore, CoVeriTest performs better in about 33 % (lower right half of the left scatter plot) and 20 % of the programs. In total, CoVeriTest achieves more often a better coverage than the parallel test generator and, thus, should be preferred over the parallel test generator. This is no surprise because parallelization evenly distributes the CPU time among the analyses, while we learned from previous experiments (e.g., RQ 1) that CoVeriTest cooperation setups often perform better with uneven runtime distribution.

RQ 8. Cooperation versus Other Tools: CoVeriTest*'s Cooperation Complements Well* We compare the best CoVeriTest cooperation setup with the two best tools of Test-Comp 2019 [12]: VeriFuzz and Klee. As already mentioned, we compare the branch coverage achieved by the respective tools, which was measured by the test-suite validator TestCov [34]. Figure 12 shows two scatter plots. Each scatter plot compares the branch coverage achieved by CoVeriTest with the branch coverage achieved by one of the other tools. Points in the lower right half indicate that CoVeriTest achieved higher coverage. Both scatter plots contain points in both halves.

In concrete numbers, CoVeriTest achieves higher coverage than VeriFuzz or Klee for about 15 % of the programs (257 and 246 programs, respectively). In contrast, VeriFuzz and Klee achieve higher coverage for about 77 % of the programs (1298 and 1022 programs, respectively). Thus, there exist programs for which CoVeriTest performs better and vice versa. For example, CoVeriTest is often better on tasks of the subcategory `sequentialized`. However, CoVeriTest has problems with tasks from categories `array` and `ECA`. We already know from verification that CPAchecker sometimes lacks refinement support for tasks with arrays. The problem with the `ECA` tasks is that these tasks contain a loop in which the feasibility of execution paths heavily depends on specific variable values, that these variables are initialized with concrete values, and that the values of these variables are changed in each loop iteration. Thus, it may take several loop iterations to reach certain branches. While we know from verification that the value analysis performs better on `ECA` tasks than the predicate analysis, which has to apply CEGAR to learn about the specific variable values, the CoVeriTest cooperation setup used for comparison grants the predicate analysis more time. Summing up, CoVeriTest is not always best, but it is also not dominated—CoVeriTest complements the existing approaches.

4.4 Threats to validity

Our results might not generalize for several reasons. We evaluated CoVeriTest on programs of a diverse and well-established benchmark set, which consists not only of verification tasks from real-world applications but also contains generated programs. CPAchecker's analyses are well trained on this benchmark set, and CoVeriTest may take an advantage of these benchmark programs. However, the training is performed w.r.t. a different property: reachability of a function call instead of reachability of branches (the test

[14] The parallel test generator uses CPAchecker's parallel algorithm and shares test goals between the analyses.

goals). Furthermore, our CoVeriTest cooperation setups use two combinations of verifiers. We already observed that some conclusions hold for one combination, but not for the other. Our results might not carry over if using CoVeriTest with a different set of verifiers.

The coverage results might be imprecise. The comparison of CoVeriTest with VeriFuzz and Klee relies on the coverage values reported by the test-suite validator TestCov [34]. Due to bugs, TestCov might report wrong coverage numbers. However, TestCov was used in Test-Comp 2019 and in other research projects, and thus, we trust it. Furthermore, it is based on the well-established coverage-measurement tool `gcov`. Therefore, severe bugs are unlikely. For the remaining comparisons, we relied on the number of covered goals reported by CoVeriTest. While in principle invalid counterexamples could be used to cover test goals, we assume this is unlikely. The analyses used by CoVeriTest are executed in the SV-COMP configuration of CPAchecker or use a CEGAR approach. In both cases, they try hard to avoid reporting 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 existing test-generation tools. Thus, this comparison still supports the value of CoVeriTest.

5 Related work

Different reachability analyses take turns in CoVeriTest to generate tests for C programs. To enable cooperation among analyses, CoVeriTest reuses different types of information from ARGs constructed by previous analysis runs.

5.1 Testing with verifiers

There is a survey on test-case generation with model checkers [63], but most approaches discussed in the survey rely on formal models instead of programs. However, also some model checkers for software support test generation. Blast [13] applies predicate abstraction to generate a test for each program location that can be reached with a state fulfilling a target predicate p. For test generation, Blast uses predicate abstraction. FShell [79–81] and CPA/Tiger [25] generate tests for a coverage criterion specified in the FShell query language (FQL) [81]. 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 [55] to answer those reachability queries, and CPA/Tiger uses CPAchecker's predicate abstraction. PathFinder [123] can generate tests with explicit-state model checking or symbolic execution. Generally, test-case generation with symbolic execution [93] has received lots of interest [44,105]. Moreover, conditional testing [33] proposes a template construction that builds an automatic test-case generator from an arbitrary verifier that can produce violation witnesses [17]—a standard exchange format for counterexamples, which is supported by many verifiers. Basically, the template combines the verifier with a transformer [18] that generates tests from witnesses.

5.2 Combined approaches for testing and verification

Combinations used for testing and verification can be classified into parallel, sequential, selective, nested, or interleaved combinations.

Parallel combinations. Portfolio combination approaches [72,78,82,102] run different, independent configurations in parallel. A second class of approaches [19,24,56,68,69] runs different approaches in parallel while letting them interoperate, e.g., exchange information. A third class splits the search space [21,23,52,100,116,117,119], e.g., program executions, among different workers. Workers often apply the same approach, but to different parts of the search space.

Sequential combinations. Also, sequential approaches may split the search space between different approaches [23,28, 33,51,59,88]. Typically, the subsequent approach is restricted to the search space not considered by the previous approaches. Like CoVeriTest, conditional model checking [23,28] uses a condition to restrict the search space. The condition is constructed from an ARG when a verifier gives up. Conditional testing [33] and CoVeriTest exchange information about covered test goals. Conditional testing uses reducers and extractors to exchange this information between arbitrary test tools. Evacon [83] combines symbolic execution and search-based testing and transfers the generated tests from one approach to the other. Further sequential combinations testify the result of the previous approach [48,57,66,97,107].

Selective combinations. Selective combination approaches [4,15,58,61,65,91,109,121] perform algorithm selection [108]. They use certain features of a verification or test task to choose the best approach for the particular task.

Nested combinations. Nested combinations use another approach as one component of the main approach. CBST [113] uses symbolic execution to compute the initial population for search-based testing. EvoSuite [64] uses concolic execution to compute some of the new individuals in search-based testing. EvoSE [7] uses concolic execution, and some others [6] apply symbolic execution during fitness computation of individuals in search-based testing.

VeriFuzz [50] applies verification techniques to compute initial inputs and to set the parameters for the fuzzer AFL.

Interleaved combinations. Interleaved combinations alternate different approaches. For example, the verifiers UFO [2] and SMASH [70] alternate underapproximation with overapproximation, while SYNERGY [73], DASH [8] and others [128] alternate test generation and proof construction to (dis)prove a property. Klee [43] alternates different exploration strategies. Hybrid concolic testing [98] interleaves random testing and symbolic execution. When random testing does not make progress, symbolic execution is started from the current state. Symbolic execution stops as soon as it covers a new goal and provides the input for covering the goal to random testing. Similarly, Driller [120] and Badger [103] alternate fuzzing with concolic execution. However, they exchange inputs when changing from one analysis to the other. Alternating different approaches [92,125] can also augment test suites. Abstraction-driven concolic testing [60] interleaves concolic execution and predicate analysis. It inspired us to work on CoVeriTest, which is designed as a generalization of abstraction-driven concolic testing, in order to explore more such combinations.

Conditional testing. The concept of conditional testing [33] explains how testing approaches can be combined such that approaches with different strengths can contribute to the test suite, and thus, increase the coverage. From a conceptual viewpoint, CoVeriTest is an instance of conditional testing: CoVeriTest and conditional testing maintain a *set of test goals* to book-keep what work is done already and what is still left to do (Fig. 3 [33] explains the passing of test-goal sets for *conditional testers*). CoVeriTest's sequence of analysis runs and analysis cycles (Fig. 5 and Alg. 3) can be expressed as *sequential tester* (Fig. 8 [33]) and *cyclic tester* (Fig. 9 [33]), respectively. The standard Alg. 2 for generating test cases using reachability analyses can be expressed as a combination of a *verifier-based tester* and a cyclic conditional tester, as described in Figs. 13 and 14 [33]. On top of the above features, CoVeriTest supports cooperation setups in which not only test-goal sets but also ARGs, condition automata, and abstraction precisions are exchanged between different analyses (ARGs can be huge in size).

5.3 Reusing information from state-space exploration

Information from state-space explorations has been reused in different context like, e.g., validation of verification results or incremental verification.

Validating results. Validation approaches use information provided by the verification to check the verification result. Many verifiers [11] construct verification witnesses [16,17] from the explored state space. To check correctness results, several proof-carrying code approaches provide (partial) state-space information [3,10,86,110], transform the state space into verification conditions [9,46,75,114], or transform the program into an easier verifiable program [87].

Incremental verification. The goal of incremental verification is to use information from a previous verification to reverify a program after the program or property changed. Some approaches [106,126,127] use the state-space information to skip the verification of unmodified program parts. Other approaches reuse the solutions of constraint or SAT proofs [5,90,101,122]. Precision reuse [36] and trace-abstraction reuse [111] reuse information on the abstraction level. Other types of approaches [25,40,45,76,112,115,118] adapt the explored state space to the change. Extreme model checking [76] and CPA/Tiger [25] adapt ARGs. Extreme model checking [76] reuses still valid ARG parts and reexplores invalid ARG subgraphs. CPA/Tiger [25] transforms an ARG that was constructed for one test goal such that it fits to a new test goal. Lazy abstraction refinement [77] adapts an ARG to continue exploration after abstraction refinement. CoVeriTest continues the exploration of the ARG, but does not need to adapt it. Furthermore, it integrates the idea of precision reuse and some of the analyses in CoVeriTest apply lazy abstraction refinement.

6 Conclusion

Software quality assurance is an important aspect in software development. Testing is a standard means for quality assurance, but state-of-the-art techniques have difficulties in covering sophisticated branching conditions [32]. Analyses that are designed to check reachability properties are well suited for this task because they only need to check the reachability of such a branching condition and generate a test if the branch condition is reachable. Nevertheless, for each technique there exist reachability queries (i.e., branch conditions) on which the technique is inefficient or fails in practice. To overcome this limitation, we propose CoVeriTest, which interleaves different reachability analyses to generate tests. In our experiments, we study various CoVeriTest cooperation setups that differ in the used analyses, the time limits of the analyses, and the information exchanged between analysis runs. CoVeriTest works best when interleaving value and predicate analysis, letting them resume their exploration, restricting the information exchange between them to covered test goals, and assigning the more mature predicate analysis a larger time limit. Furthermore, a comparison of CoVeriTest with (a) the analyses used by CoVeriTest and (b) state-of-the-art test-generation tools show the benefits of our CoVeriTest approach.

Future Work. Currently, not all reuse options are always available. Precision transformation is only available from predicate to value analysis. It is promising to develop such transformations for other combinations as well. Furthermore, not all options can be freely combined. It would be interesting to investigate how to automatically detect unavailable options. One question is, e.g., how to adapt the ARG to a new condition.

Another future direction focuses on a better understanding of CoVeriTest, i.e., when and in which cooperation setup to use CoVeriTest. Therefore, one could study the influence of program and analysis characteristics on the performance of CoVeriTest.

Funding Open Access funding provided by Projekt DEAL. This work was funded by the Deutsche Forschungsgemeinschaft (DFG)—https://gepris.dfg.de/gepris/projekt/418257054 and by the Hessian LOEWE initiative within the Software-Factory 4.0 project.

Data Availability Statement CoVeriTest's open-source tool implementation and all our experimental data are publicly available in an archive [27] and on a supplementary web site https://www.sosy-lab.org/research/coop-testgen/STTT/, in order to support reproducibility of our results.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley, Reading (1986)
2. Albarghouthi, A., Gurfinkel, A., Chechik, M.: From under-approximations to over-approximations and back. In: Proceedings of TACAS, LNCS, vol. 7214, pp. 157–172. Springer, Berlin (2012). https://doi.org/10.1007/978-3-642-28756-5_12
3. Albert, E., Puebla, G., Hermenegildo, M.V.: Abstraction-carrying code. In: Proceedings of LPAR, LNCS, vol. 3452, pp. 380–397. Springer, Berlin (2004). https://doi.org/10.1007/978-3-540-32275-7_25
4. Apel, S., Beyer, D., Friedberger, K., Raimondi, F., von Rhein, A.: Domain types: abstract-domain selection based on variable usage. In: Proceedings of HVC, LNCS 8244, pp. 262–278. Springer, Berlin (2013). https://doi.org/10.1007/978-3-319-03077-7
5. Aquino, A., Bianchi, F.A., Chen, M., Denaro, G., Pezzè, M.: Reusing constraint proofs in program analysis. In: Proceedings of ISSTA, pp. 305–315. ACM, New York (2015). https://doi.org/10.1145/2771783.2771802
6. Baars, A.I., Harman, M., Hassoun, Y., Lakhotia, K., McMinn, P., Tonella, P., Vos, T.E.J.: Symbolic search-based testing. In: Proceedings of ASE, pp. 53–62. IEEE (2011). https://doi.org/10.1109/ASE.2011.6100119
7. Baluda, M.: EvoSE: evolutionary symbolic execution. In: Proceedings of A-TEST, pp. 16–19. ACM, New York (2015). https://doi.org/10.1145/2804322.2804325
8. Beckman, N., Nori, A.V., Rajamani, S.K., Simmons, R.J.: Proofs from tests. In: Proceedings of ISSTA, pp. 3–14. ACM, New York (2008). https://doi.org/10.1145/1390630.1390634
9. Besson, F., Cornilleau, P., Jensen, T.P.: Result certification of static program analysers with automated theorem provers. In: Proceedings of VSTTE, LNCS, vol. 8164, pp. 304–325. Springer, Berlin (2013). https://doi.org/10.1007/978-3-642-54108-7_16
10. Besson, F., Jensen, T.P., Pichardie, D.: Proof-carrying code from certified abstract interpretation and fixpoint compression. TCS **364**(3), 273–291 (2006). https://doi.org/10.1016/j.tcs.2006.08.012
11. Beyer, D.: Automatic verification of C and Java programs: SV-COMP 2019. In: Proceedings of TACAS (3), LNCS, vol. 11429, pp. 133–155. Springer, Berlin (2019). https://doi.org/10.1007/978-3-030-17502-3_9
12. Beyer, D.: First international competition on software testing (Test-Comp 2019). Int. J. Softw. Tools Technol, Transf (2020)
13. Beyer, D., Chlipala, A.J., Henzinger, T.A., Jhala, R., Majumdar, R.: Generating tests from counterexamples. In: Proceedings of ICSE, pp. 326–335. IEEE (2004). https://doi.org/10.1109/ICSE.2004.1317455
14. Beyer, D., Chlipala, A.J., Henzinger, T.A., Jhala, R., Majumdar, R.: The Blast query language for software verification. In: Proceedings of SAS, LNCS, vol. 3148, pp. 2–18. Springer, Berlin (2004). https://doi.org/10.1007/978-3-540-27864-1_2
15. Beyer, D., Dangl, M.: Strategy selection for software verification based on Boolean features: a simple but effective approach. In: Proceedings of ISoLA, LNCS, vol. 11245, pp. 144–159. Springer, Berlin (2018). https://doi.org/10.1007/978-3-030-03421-4_11
16. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M.: Correctness witnesses: exchanging verification results between verifiers. In: Proceedings of FSE, pp. 326–337. ACM, New York (2016). https://doi.org/10.1145/2950290.2950351
17. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Stahlbauer, A.: Witness validation and stepwise testification across software verifiers. In: Proceedings of FSE, pp. 721–733. ACM, New York (2015). https://doi.org/10.1145/2786805.2786867
18. Beyer, D., Dangl, M., Lemberger, T., Tautschnig, M.: Tests from witnesses: execution-based validation of verification results. In: Proceedings of TAP, LNCS, vol. 10889, pp. 3–23. Springer, Berlin (2018). https://doi.org/10.1007/978-3-319-92994-1_1
19. Beyer, D., Dangl, M., Wendler, P.: Boosting k-induction with continuously-refined invariants. In: Proceedings of CAV, LNCS, vol. 9206, pp. 622–640. Springer, Berlin (2015). https://doi.org/10.1007/978-3-319-21690-4_42
20. Beyer, D., Dangl, M., Wendler, P.: A unifying view on SMT-based software verification. J. Autom. Reason. **60**(3), 299–335 (2018). https://doi.org/10.1007/s10817-017-9432-6
21. Beyer, D., Friedberger, K.: Domain-independent multi-threaded software model checking. In: Proceedings of ASE, pp. 634–644. ACM, New York (2018). https://doi.org/10.1145/3238147.3238195
22. 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, Berlin (2018). https://doi.org/10.1007/978-3-319-10575-8_16
23. Beyer, D., Henzinger, T.A., Keremoglu, M.E., Wendler, P.: Conditional model checking: a technique to pass information between verifiers. In: Proceedings of FSE. ACM, New York (2012). https://doi.org/10.1145/2393596.2393664
24. Beyer, D., Henzinger, T.A., Théoduloz, G.: Program analysis with dynamic precision adjustment. In: Proceedings of ASE, pp. 29–38. IEEE (2008). https://doi.org/10.1109/ASE.2008.13

25. Beyer, D., Holzer, A., Tautschnig, M., Veith, H.: Information reuse for multi-goal reachability analyses. In: Proceedings of ESOP, LNCS, vol. 7792, pp. 472–491. Springer, Berlin (2013). https://doi.org/10.1007/978-3-642-37036-6_26
26. Beyer, D., Jakobs, M.C.: CoVeriTest: cooperative verifier-based testing. In: Proceedings of FASE, LNCS, vol. 11424, pp. 389–408. Springer, Berlin (2019). https://doi.org/10.1007/978-3-030-16722-6_23
27. Beyer, D., Jakobs, M.C.: Replication package for article 'Cooperative, verifier-based testing with CoVeriTest' in STTT. Zenodo (2020). https://doi.org/10.5281/zenodo.3666060
28. Beyer, D., Jakobs, M.C., Lemberger, T., Wehrheim, H.: Reducer-based construction of conditional verifiers. In: Proceedings of ICSE, pp. 1182–1193. ACM, New York (2018). https://doi.org/10.1145/3180155.3180259
29. Beyer, D., Keremoglu, M.E.: CPAchecker: a tool for configurable software verification. In: Proceedings of CAV, LNCS, vol. 6806, pp. 184–190. Springer, Berlin (2011). https://doi.org/10.1007/978-3-642-22110-1_16
30. Beyer, D., Keremoglu, M.E., Wendler, P.: Predicate abstraction with adjustable-block encoding. In: Proceedings of FMCAD, pp. 189–197. FMCAD (2010)
31. Beyer, D., Lemberger, T.: Symbolic execution with CEGAR. In: Proceedings of ISoLA, LNCS, vol. 9952, pp. 195–211. Springer, Berlin (2016). https://doi.org/10.1007/978-3-319-47166-2_14
32. Beyer, D., Lemberger, T.: Software verification: testing vs. model checking. In: Proceedings of HVC, LNCS, vol. 10629, pp. 99–114. Springer, Berlin (2017). https://doi.org/10.1007/978-3-319-70389-3_7
33. Beyer, D., Lemberger, T.: Conditional testing: Off-the-shelf combination of test-case generators. In: Proceedings ATVA, LNCS, vol. 11781, pp. 189–208. Springer, Berlin (2019). https://doi.org/10.1007/978-3-030-31784-3_11
34. Beyer, D., Lemberger, T.: TestCov: Robust test-suite execution and coverage measurement. In: Proceedings of ASE, pp. 1074–1077. IEEE (2019). https://doi.org/10.1109/ASE.2019.00105
35. Beyer, D., Löwe, S.: Explicit-state software model checking based on CEGAR and interpolation. In: Proceedings of FASE, LNCS, vol. 7793, pp. 146–162. Springer, Berlin (2013). https://doi.org/10.1007/978-3-642-37057-1_11
36. Beyer, D., Löwe, S., Novikov, E., Stahlbauer, A., Wendler, P.: Precision reuse for efficient regression verification. In: Proceedings of FSE, pp. 389–399. ACM, New York (2013). https://doi.org/10.1145/2491411.2491429
37. Beyer, D., Löwe, S., Wendler, P.: Refinement selection. In: Proceedings of SPIN, LNCS, vol. 9232, pp. 20–38. Springer, Berlin (2015). https://doi.org/10.1007/978-3-319-23404-5_3
38. Beyer, D., Löwe, S., Wendler, P.: Sliced path prefixes: an effective method to enable refinement selection. In: Proceedings of FORTE, LNCS, vol. 9039, pp. 228–243. Springer, Berlin (2015). https://doi.org/10.1007/978-3-319-19195-9_15
39. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: requirements and solutions. Int. J. Softw. Tools Technol. Transfer **21**(1), 1–29 (2019). https://doi.org/10.1007/s10009-017-0469-y
40. Bianculli, D., Filieri, A., Ghezzi, C., Mandrioli, D.: Syntactic-semantic incrementality for agile verification. SCICO **97**, 47–54 (2015). https://doi.org/10.1016/j.scico.2013.11.026
41. Biere, A., Cimatti, A., Clarke, E.M., Zhu, Y.: Symbolic model checking without BDDs. In: Proceedings of TACAS, LNCS, vol. 1579, pp. 193–207. Springer, Berlin (1999). https://doi.org/10.1007/3-540-49059-0_14
42. Blicha, M., Hyvärinen, A.E.J., Marescotti, M., Sharygina, N.: A cooperative parallelization approach for property-directed k-induction. In: Proceedings of VMCAI, LNCS, vol. 11990, pp. 270–292. Springer (2020). https://doi.org/10.1007/978-3-030-39322-9_13
43. Cadar, C., Dunbar, D., Engler, D.R.: Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of OSDI, pp. 209–224. USENIX Association (2008)
44. Cadar, C., Sen, K.: Symbolic execution for software testing: three decades later. CACM **56**(2), 82–90 (2013). https://doi.org/10.1145/2408776.2408795
45. Carroll, M.D., Ryder, B.G.: Incremental data flow analysis via dominator and attribute updates. In: Proceedings of POPL, pp. 274–284. ACM, New York (1988). https://doi.org/10.1145/73560.73584
46. Chaieb, A.: Proof-producing program analysis. In: Proceedings of ICTAC, LNCS, vol. 4281, pp. 287–301. Springer, Berlin (2006). https://doi.org/10.1007/11921240_20
47. Chalupa, M., Vitovská, M., Strejcek, J.: Symbiotic 5: boosted instrumentation (competition contribution). In: Proceedings of TACAS, LNCS, vol. 10806, pp. 442–446. Springer, Berlin (2018). https://doi.org/10.1007/978-3-319-89963-3_29
48. Chebaro, O., Kosmatov, N., Giorgetti, A., Julliand, J.: Program slicing enhances a verification technique combining static and dynamic analysis. In: Proceedings of SAC, pp. 1284–1291. ACM, New York (2012). https://doi.org/10.1145/2245276.2231980
49. Cheng, W., Hüllermeier, E.: Combining instance-based learning and logistic regression for multilabel classification. Mach. Learn. **76**(2–3), 211–225 (2009). https://doi.org/10.1007/s10994-009-5127-5
50. Chowdhury, A.B., Medicherla, R.K., Venkatesh, R.: VeriFuzz: program aware fuzzing (competition contribution). In: Proceedings of TACAS, part 3, LNCS, vol. 11429, pp. 244–249. Springer, Berlin (2019). https://doi.org/10.1007/978-3-030-17502-3_22
51. Christakis, M., Müller, P., Wüstholz, V.: Guiding dynamic symbolic execution toward unverified program executions. In: Proceedings of ICSE, pp. 144–155. ACM, New York (2016). https://doi.org/10.1145/2884781.2884843
52. Ciortea, L., Zamfir, C., Bucur, S., Chipounov, V., Candea, G.: Cloud9: a software testing service. ACM SIGOPS Oper. Syst. Rev. **43**(4), 5–10 (2009). https://doi.org/10.1145/1713254.1713257
53. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Proceedings CAV, LNCS, vol. 1855, pp. 154–169. Springer, Berlin (2000). https://doi.org/10.1007/10722167_15
54. Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R.: Handbook of Model Checking. Springer, Berlin (2018). https://doi.org/10.1007/978-3-319-10575-8
55. Clarke, E.M., Kröning, D., Lerda, F.: A tool for checking ANSI-C programs. In: Proceedings of TACAS, LNCS 2988, pp. 168–176. Springer, Berlin (2004). https://doi.org/10.1007/978-3-540-24730-2_15
56. Cousot, P., Cousot, R.: Systematic design of program-analysis frameworks. In: Proceedings of POPL, pp. 269–282. ACM, New York (1979). https://doi.org/10.1145/567752.567778
57. Csallner, C., Smaragdakis, Y.: Check 'n' crash: combining static checking and testing. In: Proceedings of ICSE, pp. 422–431. ACM, New York (2005). https://doi.org/10.1145/1062455.1062533
58. Czech, M., Hüllermeier, E., Jakobs, M., Wehrheim, H.: Predicting rankings of software verification tools. In: Proceedings of SWAN, pp. 23–26. ACM, New York (2017). https://doi.org/10.1145/3121257.3121262
59. Czech, M., Jakobs, M., Wehrheim, H.: Just test what you cannot verify! In: Proceedings of FASE, LNCS, vol. 9033, pp. 100–114. Springer, Berlin (2015). https://doi.org/10.1007/978-3-662-46675-9_7
60. Daca, P., Gupta, A., Henzinger, T.A.: Abstraction-driven concolic testing. In: Proceedings of VMCAI, LNCS, vol. 9583, pp. 328–

347. Springer, Berlin (2016). https://doi.org/10.1007/978-3-662-49122-5_16
61. Demyanova, Y., Pani, T., Veith, H., Zuleger, F.: Empirical software metrics for benchmarking of verification tools. In: Proceedings of CAV, LNCS, vol. 9206, pp. 561–579. Springer, Berlin (2015). https://doi.org/10.1007/978-3-319-21690-4_39
62. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall, Englewood Cliffs (1976)
63. Fraser, G., Wotawa, F., Ammann, P.: Testing with model checkers: a survey. Softw. Test. Verif. Reliab. **19**(3), 215–261 (2009). https://doi.org/10.1002/stvr.402
64. Galeotti, J.P., Fraser, G., Arcuri, A.: Improving search-based test suite generation with dynamic symbolic execution. In: Proceedings of ISSRE, pp. 360–369. IEEE (2013). https://doi.org/10.1109/ISSRE.2013.6698889
65. Gargantini, A., Vavassori, P.: Using decision trees to aid algorithm selection in combinatorial interaction tests generation. In: Proceedings of ICST, pp. 1–10. IEEE (2015). https://doi.org/10.1109/ICSTW.2015.7107442
66. Ge, X., Taneja, K., Xie, T., Tillmann, N.: DyTa: dynamic symbolic execution guided with static verification results. In: Proceedings of ICSE, pp. 992–994. ACM, New York (2011). https://doi.org/10.1145/1985793.1985971
67. Ghezzi, C., Jazayeri, M., Mandrioli, D.: Fundamentals of Software Engineering, 2nd edn. Prentice Hall, Englewood Cliffs (2003)
68. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: Proceedings of PLDI, pp. 213–223. ACM, New York (2005). https://doi.org/10.1145/1065010.1065036
69. Godefroid, P., Levin, M.Y., Molnar, D.A.: Automated whitebox fuzz testing. In: Proceedings of NDSS. The Internet Society (2008)
70. Godefroid, P., Nori, A.V., Rajamani, S.K., Tetali, S.: Compositional may-must program analysis: unleashing the power of alternation. In: Proceedings of POPL, pp. 43–56. ACM, New York (2010). https://doi.org/10.1145/1706299.1706307
71. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: Proceedings of CAV, LNCS, vol. 1254, pp. 72–83. Springer, Berlin (1997). https://doi.org/10.1007/3-540-63166-6_10
72. Groce, A., Zhang, C., Eide, E., Chen, Y., Regehr, J.: Swarm testing. In: Proceedings of ISSTA, pp. 78–88. ACM, New York (2012). https://doi.org/10.1145/2338965.2336763
73. Gulavani, B.S., Henzinger, T.A., Kannan, Y., Nori, A.V., Rajamani, S.K.: SYNERGY: a new algorithm for property checking. In: Proceedings of FSE, pp. 117–127. ACM, New York (2006). https://doi.org/10.1145/1181775.1181790
74. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: Proceedings of POPL, pp. 232–244. ACM, New York (2004). https://doi.org/10.1145/964001.964021
75. Henzinger, T.A., Jhala, R., Majumdar, R., Necula, G.C., Sutre, G., Weimer, W.: Temporal-safety proofs for systems code. In: Proceedings of CAV, LNCS, vol. 2404, pp. 526–538. Springer, Berlin (2002). https://doi.org/10.1007/3-540-45657-0_45
76. Henzinger, T.A., Jhala, R., Majumdar, R., Sanvido, M.A.A.: Extreme model checking. In: Verification: Theory and Practice, pp. 332–358 (2003). https://doi.org/10.1007/978-3-540-39910-0_16
77. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Proceedings of POPL, pp. 58–70. ACM, New York (2002). https://doi.org/10.1145/503272.503279
78. Holík, L., Kotoun, M., Peringer, P., Soková, V., Trtík, M., Vojnar, T.: Predator shape analysis tool suite. In: Proceedings of HVC, LNCS, vol. 10028, pp. 202–209 (2016). https://doi.org/10.1007/978-3-319-49052-6_13
79. Holzer, A., Schallhart, C., Tautschnig, M., Veith, H.: FShell: Systematic test case generation for dynamic analysis and measurement. In: Proceedings of CAV, LNCS, vol. 5123, pp. 209–213. Springer, Berlin (2008). https://doi.org/10.1007/978-3-540-70545-1_20
80. Holzer, A., Schallhart, C., Tautschnig, M., Veith, H.: Query-driven program testing. In: Proceedings of VMCAI, LNCS, vol. 5403, pp. 151–166. Springer, Berlin (2009). https://doi.org/10.1007/978-3-540-93900-9_15
81. Holzer, A., Schallhart, C., Tautschnig, M., Veith, H.: How did you specify your test suite. In: Proceedings of ASE, pp. 407–416. ACM, New York (2010). https://doi.org/10.1145/1858996.1859084
82. Holzmann, G.J., Joshi, R., Groce, A.: Swarm verification. In: Proceedings of ASE, pp. 1–6. IEEE (2008). https://doi.org/10.1109/ASE.2008.9
83. Inkumsah, K., Xie, T.: Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In: Proceedings of ASE, pp. 297–306. IEEE (2008). https://doi.org/10.1109/ASE.2008.40
84. Jakobs, M.C.: CoVeriTest: Interleaving value and predicate analysis for test-case generation (competition contribution). Int. J. Softw. Tools Technol. Transf. (2020). https://doi.org/10.1007/s10009-020-00572-1
85. Jakobs, M.C.: COVERITEST with dynamic partitioning of the iteration time limit (competition contribution). In: Proceedings of FASE, LNCS, vol. 12076, pp. 540–544. Springer, Berlin (2020). https://doi.org/10.1007/978-3-030-45234-6_30
86. Jakobs, M.C., Wehrheim, H.: Certification for configurable program analysis. In: Proceedings of SPIN, pp. 30–39. ACM, New York (2014). https://doi.org/10.1145/2632362.2632372
87. Jakobs, M.C., Wehrheim, H.: Programs from proofs: a framework for the safe execution of untrusted software. ACM Trans. Program. Lang. Syst. **39**(2), 7:1–7:56 (2017). https://doi.org/10.1145/3014427
88. Jalote, P., Vangala, V., Singh, T., Jain, P.: Program partitioning: a framework for combining static and dynamic analysis. In: Proceedings of WODA, pp. 11–16. ACM, New York (2006). https://doi.org/10.1145/1138912.1138916
89. Jhala, R., Majumdar, R.: Software model checking. ACM Comput. Surv. **41**, 4 (2009). https://doi.org/10.1145/1592434.1592438
90. Jia, X., Ghezzi, C., Ying, S.: Enhancing reuse of constraint solutions to improve symbolic execution. In: Proceedings of ISSTA, pp. 177–187. ACM, New York (2015). https://doi.org/10.1145/2771783.2771806
91. Jia, Y., Cohen, M.B., Harman, M., Petke, J.: Learning combinatorial interaction test generation strategies using hyperheuristic search. In: Proceedings of ICSE, pp. 540–550. IEEE (2015). https://doi.org/10.1109/ICSE.2015.71
92. Kim, Y., Xu, Z., Kim, M., Cohen, M.B., Rothermel, G.: Hybrid directed test suite augmentation: an interleaving framework. In: Proceedings of ICST, pp. 263–272. IEEE (2014). https://doi.org/10.1109/ICST.2014.39
93. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976). https://doi.org/10.1145/360248.360252
94. Kotthoff, L.: Algorithm selection for combinatorial search problems: a survey. In: Data Mining and Constraint Programming–Foundations of a Cross-Disciplinary Approach, LNCS, vol. 10101, pp. 149–190. Springer, Berlin (2016). https://doi.org/10.1007/978-3-319-50137-6_7
95. Lemieux, C., Sen, K.: FairFuzz: a targeted mutation strategy for increasing greybox fuzz testing coverage. In: Proceedings of ASE, pp. 475–485. ACM, New York (2018). https://doi.org/10.1145/3238147.3238176
96. Li, J., Zhao, B., Zhang, C.: Fuzzing: a survey. Cybersecurity **1**(1), 6 (2018). https://doi.org/10.1186/s42400-018-0002-y

97. Li, K., Reichenbach, C., Csallner, C., Smaragdakis, Y.: Residual investigation: predictive and precise bug detection. In: Proceedings of ISSTA, pp. 298–308. ACM, New York (2012). https://doi.org/10.1145/2338965.2336789
98. Majumdar, R., Sen, K.: Hybrid concolic testing. In: Proceedings of ICSE, pp. 416–426. IEEE (2007). https://doi.org/10.1109/ICSE.2007.41
99. 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
100. Misailovic, S., Milicevic, A., Petrovic, N., Khurshid, S., Marinov, D.: Parallel test generation and execution with Korat. In: Proceedings of ESEC/FSE, pp. 135–144. ACM, New York (2007). https://doi.org/10.1145/1287624.1287645
101. Mudduluru, R., Ramanathan, M.K.: Efficient incremental static analysis using path abstraction. In: Proceedings of FASE, LNCS, vol. 8411, pp. 125–139. Springer, Berlin (2014). https://doi.org/10.1007/978-3-642-54804-8_9
102. Nguyen, T.L., Schrammel, P., Fischer, B., La Torre, S., Parlato, G.: Parallel bug-finding in concurrent programs via reduced interleaving instances. In: Proceedings of ASE, pp. 753–764. IEEE (2017). https://doi.org/10.1109/ASE.2017.8115686
103. Noller, Y., Kersten, R., Pasareanu, C.S.: Badger: Complexity analysis with fuzzing and symbolic execution. In: Proceedings of ISSTA, pp. 322–332. ACM, New York (2018). https://doi.org/10.1145/3213846.3213868
104. Pacheco, C., Lahiri, S.K., Ernst, M.D., Ball, T.: Feedback-directed random test generation. In: Proceedings of ICSE, pp. 75–84. IEEE (2007). https://doi.org/10.1109/ICSE.2007.37
105. Pasareanu, C.S., Visser, W.: A survey of new trends in symbolic execution for software testing and analysis. Int. J. Softw. Tools Technol. Transf. **11**(4), 339–353 (2009). https://doi.org/10.1007/s10009-009-0118-1
106. Person, S., Yang, G., Rungta, N., Khurshid, S.: Directed incremental symbolic execution. In: Proceedings of PLDI, pp. 504–515. ACM, New York (2011). https://doi.org/10.1145/1993498.1993558
107. Post, H., Sinz, C., Kaiser, A., Gorges, T.: Reducing false positives by combining abstract interpretation and bounded model checking. In: Proceedings of ASE, pp. 188–197. IEEE (2008). https://doi.org/10.1109/ASE.2008.29
108. Rice, J.R.: The algorithm selection problem. Adv. Comput. **15**, 65–118 (1976). https://doi.org/10.1016/S0065-2458(08)60520-3
109. Richter, C., Wehrheim, H.: PeSCo: predicting sequential combinations of verifiers (competition contribution). In: Proceedings of TACAS, LNCS, vol. 11429, pp. 229–233. Springer, Berlin (2019). https://doi.org/10.1007/978-3-030-17502-3_19
110. Rose, E.: Lightweight bytecode verification. J. Autom. Reason. **31**(3–4), 303–334 (2003). https://doi.org/10.1023/B:JARS.0000021015.15794.82
111. Rothenberg, B., Dietsch, D., Heizmann, M.: Incremental verification using trace abstraction. In: Proceedings of SAS, LNCS, vol. 11002, pp. 364–382. Springer, Berlin (2018). https://doi.org/10.1007/978-3-319-99725-4_22
112. Ryder, B.G.: Incremental data flow analysis. In: Proceedings of POPL, pp. 167–176. ACM Press, New York (1983). https://doi.org/10.1145/567067.567084
113. Sakti, A., Guéhéneuc, Y., Pesant, G.: Boosting search based testing by using constraint based testing. In: Proceedings of SSBSE, LNCS, vol. 7515, pp. 213–227. Springer, Berlin (2012). https://doi.org/10.1007/978-3-642-33119-0_16
114. Seo, S., Yang, H., Yi, K.: Automatic construction of Hoare proofs from abstract interpretation results. In: Proceedings of APLAS, LNCS, vol. 2895, pp. 230–245. Springer, Berlin (2003). https://doi.org/10.1007/978-3-540-40018-9_16
115. Sery, O., Fedyukovich, G., Sharygina, N.: Incremental upgrade checking by means of interpolation-based function summaries. In: Proceedings of FMCAD, pp. 114–121. FMCAD Inc., Palo Alto (2012)
116. Sherman, E., Dwyer, M.B.: Structurally defined conditional data-flow static analysis. In: Proceedings of TACAS (2), LNCS, vol. 10806, pp. 249–265. Springer, Berlin (2018). https://doi.org/10.1007/978-3-319-89963-3_15
117. Siddiqui, J.H., Khurshid, S.: Scaling symbolic execution using ranged analysis. In: Leavens, G.T., Dwyer, M.B. (eds.) Proceedings of SPLASH, pp. 523–536. ACM, New York (2012). https://doi.org/10.1145/2384616.2384654
118. Sokolsky, O., Smolka, S.A.: Incremental model checking in the modal mu-calculus. In: Proceedings of CAV, LNCS, vol. 818, pp. 351–363. Springer, Berlin (1994). https://doi.org/10.1007/3-540-58179-0_67
119. Staats, M., Pasareanu, C.S.: Parallel symbolic execution for structural test generation. In: Proceedings of ISSTA, pp. 183–194. ACM, New York (2010). https://doi.org/10.1145/1831708.1831732
120. 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: Proceedings of NDSS. Internet Society (2016). https://doi.org/10.14722/ndss.2016.23368
121. Tulsian, V., Kanade, A., Kumar, R., Lal, A., Nori, A.V.: MUX: algorithm selection for software model checkers. In: Proceedings of MSR. ACM, New York (2014). https://doi.org/10.1145/2597073.2597080
122. Visser, W., Geldenhuys, J., Dwyer, M.B.: GREEN: reducing, reusing, and recycling constraints in program analysis. In: Proceedings of FSE, pp. 58:1–58:11. ACM, New York (2012). https://doi.org/10.1145/2393596.2393665
123. Visser, W., Păsăreanu, C.S., Khurshid, S.: Test-input generation with Java PATHFINDER. In: Proceedings of ISSTA, pp. 97–107. ACM, New York (2004). https://doi.org/10.1145/1007512.1007526
124. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: SATzilla: portfolio-based algorithm selection for SAT. J. Artif. Intell. Res. **32**, 565–606 (2008). https://doi.org/10.1613/jair.2490
125. Xu, Z., Kim, Y., Kim, M., Rothermel, G.: A hybrid directed test-suite augmentation technique. In: Proceedings of ISSRE, pp. 150–159. IEEE (2011). https://doi.org/10.1109/ISSRE.2011.21
126. Yang, G., Dwyer, M.B., Rothermel, G.: Regression model checking. In: Proceedings of ICSM, pp. 115–124. IEEE (2009). https://doi.org/10.1109/ICSM.2009.5306334
127. Yang, G., Păsăreanu, C.S., Khurshid, S.: Memoized symbolic execution. In: Proceedings of ISSTA, pp. 144–154. ACM, New York (2012). https://doi.org/10.1145/2338965.2336771
128. Yorsh, G., Ball, T., Sagiv, M.: Testing, abstraction, theorem proving: Better together! In: Proceedings of ISSTA, pp. 145–156. ACM, New York (2006). https://doi.org/10.1145/1146238.1146255