Proc. ATVA 2019, © Springer

# Conditional Testing
## Off-the-Shelf Combination of Test-Case Generators

Dirk Beyer and Thomas Lemberger

LMU Munich, Germany

**Abstract.** There are several powerful automatic testers available, each with different strengths and weaknesses. To immediately benefit from different strengths of different tools, we need to investigate ways for quick and easy combination of techniques. Until now, research has mostly investigated integrated combinations, which require extra implementation effort. We propose the concept of *conditional testing* and a set of combination techniques that do not require implementation effort: Different testers can be taken 'off the shelf' and combined in a way that they *cooperatively* solve the problem of test-case generation for a given input program and coverage criterion. This way, the latest advances in test-case generation can be combined without delay. Conditional testing passes the test goals that a first tester has covered to the next tester, so that the next tester does not need to repeat work (as in combinations without information passing) but can focus on the remaining test goals. Our combinations do not require changes to the implementation of a tester, because we leverage a testability transformation (i.e., we reduce the input program to those parts that are relevant to the remaining test goals). To evaluate conditional testing and our proposed combination techniques, we (1) implemented the generic conditional tester CondTest, including the required transformations, and (2) ran experiments on a large amount of benchmark tasks; the obtained results are promising.

**Keywords:** Software testing, Test-case generation, Conditional model checking, Cooperative verification, Software verification, Program analysis, Test coverage

## 1 Introduction

Tool competitions in software verification and testing [1, 2, 27, 35] have shown that there is no tool that is superior, but that different tools and approaches have different strengths. Therefore, we need to combine different tools and approaches. Integrated combination approaches [8, 15, 19, 23] have shown their potential, but those combinations require additional implementation work.

The goal of this paper is to provide a generic framework that enables combinations of tools for test-case generation without the need to change the tools: We show how to take a set of testers 'off the shelf' and combine them on the

Supported in part by DFG grant BE 1761/7-1.

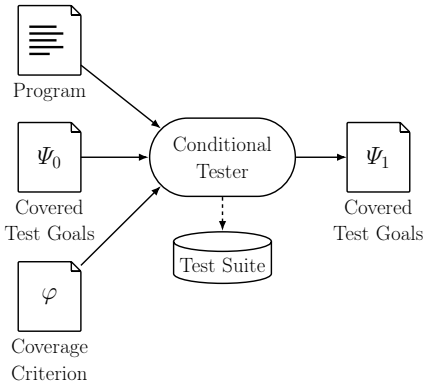


Fig. 1: Conditional testing

```
1 int main() {
2  int x = input();
3  if (x != 161) {
4   // ...
5  } else {
6   // ...
7  }
8 }
```

Fig. 2: Program under test

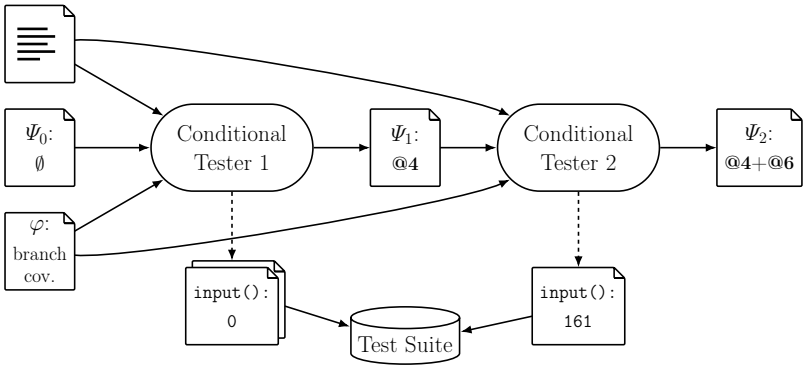


Fig. 3: Example usage of conditional testers

binary level. In other words, any tester can be taken as a black box, wrapped into a new meta tester (conditional tester) by a fully automated construction, and the new tester uses interfaces that make it possible to combine it with others (for an overview of combination techniques for software verification see [12]). There are several successful testers for C programs already; nine of them participated in the competition on software testing [2] and adhere to standard exchange formats for their input and output.

*Conditional testing* applies the idea of conditional model checking [7] to testing, as illustrated in Fig. 1. A *conditional tester* gets as input a program under test, a coverage criterion $\varphi$ (e.g., 'branch coverage'), and a condition $\Psi_0$ that describes a set of test goals that are already covered by existing tests. With this information, a conditional tester creates (1) a test suite that tries to cover as many test goals of $[\![\varphi]\!] \setminus \Psi_0$ as possible, and (2) a new condition $\Psi_1$ of test goals that have been covered. For a coverage criterion $\varphi$, we use $[\![\varphi]\!]$ to denote the test goals that are needed to fulfill $\varphi$. The condition $\Psi_1$ covers the test goals described by condition $\Psi_0$ and the test goals newly covered by the created test suite. With this interface for information passing, conditional testers can be combined to focus on different or remaining test goals.

Figure 2 shows a small program that we use to illustrate conditional testing. The program gets an arbitrary integer as input and stores it in program variable `x`. The program then checks whether `x` is un-equal to value 161. If it is, the if-branch is entered and some more code (`// ...`) is executed. Otherwise, the else-branch is entered. The coverage criterion of branch coverage defines two test goals for this program: (1) cover line 4 (denoted by @4 in FQL [26]), and (2) cover line 6 (@6). A test suite that covers both test goals would contain at least two test cases: One with input 161, and one with any other input. A randomized tester can quickly generate a test case with an input different from 161, because the number of possible values is very high and thus very probable to be fulfilled by a random test case. In contrast, it is difficult for a randomized tester to create a test case with input 161.

Let us consider the combination of a fast, but shallow randomized tester (Conditional Tester 1 in Fig. 3) with a tester that is slower, but uses an exhaustive reasoning technique (Conditional Tester 2), to obtain a test suite that covers all branches: Given the program under test, the coverage criterion $\varphi =$ `COVER EDGES(@DECISIONEDGE)` (branch coverage in FQL syntax), and the empty condition $\Psi_0 = \emptyset$ (i.e., no test goal covered yet), we run the conditional, random tester for a short amount of time. Assume it creates several different test cases, including one with input value 0. The created test cases cover line 4, but do not cover line 6. Thus, the conditional tester returns $\Psi_1 =$ **@4** for the now covered test goal. A second conditional tester, for example based on symbolic execution, then gets the same program and coverage criterion, but condition $\Psi_1$. The conditional tester focuses on the remaining test goal of covering line 6 and creates a test case with input 161. Now, all test goals are covered and $\Psi_2 =$ **@4**+**@6** describes both test goals.

Conditional testing does not prescribe a certain format or language to be used for specifying coverage criteria and conditions. The competition on software testing [2] uses FQL [25, 26] as test-specification language, and we use FQL in the example above to describe the condition, i.e., the already-covered test goals. FQL is a versatile language for defining various test criteria, allows to define explicit sets of test goals by enumerating single locations, but it also supports to specify full program paths as test goals, as well as value constraints on variables at certain program locations, and of course standard coverage criteria such as branch coverage are provided. For the first version of our tool implementation CondTest, we started with a simpler way to denote test goals.

Since existing testers do not accept conditions, we propose a testability transformation called `reducer` that uses the coverage criterion and the condition to transform the program under test into a *residual program* that is restricted to those parts of the program that are needed to generate test cases for the remaining test goals. This residual program is then given to an off-the-shelf tester (instead of the original program under test), such that the tester is forced to generate test cases for the remaining test goals. The resulting test suite is given to an `extractor` that extracts the test goals that are covered in the original program, and computes the new condition. This process of transforming off-the-shelf testers

into conditional testers can be split into three independent components: `reducer`, `tester`, and `extractor`. All three components are defined through their type and soundness-requirements, and many different implementations are possible.

To show the potential of our approach, we implemented examples for `reducer` and `extractor`. We use the common formats and infrastructure of the International Competition on Software Testing (Test-Comp) [2] to allow plug-and-play transformation of existing software testers (for example, CoVeriTest, CPA/Tiger, Klee) into conditional testers.

In addition, we contribute a construction based on conditional testing that turns an existing, formal software verifier into a conditional tester, such that existing verifiers and existing testers can be combined as well. Formal verifiers can be specialized for finding a counterexample to a certain specification, e.g., an assertion violation or a program location of interest. Verifiers have been able to create test cases from such counterexamples for over a decade [3, 40] and can thus be used for directed generation of test cases for hard-to-reach test goals. Our generic conditional tester can use all verifiers (31 tools in 2019) of the International Competition on Software Verification (SV-COMP) [1]. It uses the standard violation-witness exchange-format [4] and transforms created witnesses into executable tests [5]. To feed test goals to verifiers, we provide a tailored transformation that inserts function calls at test goals and defines the specification such that the verifier shall prove unreachability of such a function call. Since most verifiers stop their analysis after finding one counterexample (i.e., creating a single test case), we repeatedly apply conditional testing with the same verifier to obtain a full test suite.

**Related Work.** We base our work on conditional model checking [7], which is a general concept for information exchange between different model checkers through the use of *conditions*. The conditions are used to instruct the next conditional model checker which parts of the state space it does not need to verify because the previous model checker had successfully verified those parts of the state space already. To transform any off-the-shelf model checker into a conditional model checker, program reduction [9, 18] was proposed and successfully applied. We apply this general idea to testing and call it conditional testing. The conditions of conditional testing describe parts of the program that do not need to be tested, in terms of test goals. Similar to the reducer for conditional model checking [9] (which cuts off program paths that are already verified), we developed a reducer that cuts off program paths whose test goals are already covered. Further transformation techniques that reduce programs to only contain program paths that may be relevant for analysis include program slicing [18, 39] and program trimming [20].

Other works that allow combinations of different testing techniques exists; they are either limited to specific test-case-generation techniques [8, 29, 31, 32, 37, 41] or require changes of the existing implementations [8, 32]. In contrast, conditional testing is completely technique-agnostic and works with existing testers 'off the shelf', that is, without changing the existing testers. Some techniques of test-suite augmentation [28, 38] can be used to iteratively generate test suites with one

arbitrary tester, and one specific second technique that reuses the test suite generated by the first tester. These approaches are subsumed by conditional testing as special cases. Further combination approaches of tools for verification and testing include the Electronic Tools Integration platform (ETI) [30, 36], and the Evidential Tool Bus (ETB) [17, 33]. Conditional model checking was also applied to combine program analysis and testing [13, 16, 18].

**Contributions.** This article describes the following contributions:

1. We introduce the concept of *conditional testing* (Fig. 1), which enables quick and simple combinations of conditional testers with information passing. This provides the interface to combine existing testers.
2. We present a construction of conditional testers from *off-the-shelf test-case generators*, based on program reduction and test-goal extraction (Sect. 3).
3. We present several possible combinations for conditional testers (Sect. 4).
4. Using some of these combinations, we present a construction of testers from *off-the-shelf software verifiers*, based on conditional testing (Sect. 5).
5. We have implemented the generic conditional tester CondTest, which contains all components that are necessary for the above-mentioned constructions and combinations (`https://doi.org/10.5281/zenodo.3352401`).
6. We show the potential of conditional testing for software via a thorough experimental evaluation on the large Test-Comp benchmark set, consisting of 1 720 benchmark tasks (Sect. 6).

## 2 Background

In the following, we remind the reader of some notions that are necessary to instantiate the concept of conditional testing to software. A *test vector* $\bar{v} = \langle v_0, \ldots, v_n \rangle$ is a sequence of program inputs $v_i$ with $0 \leq i \leq n$. A test vector describes a test case over the program inputs, in the order that they are passed to the program under test. A *test suite* $\{\bar{v}_0, \ldots, \bar{v}_l\}$ is a set of test vectors $\bar{v}_i$ with $0 \leq i \leq l$. We store and exchange test suites in the Test-Comp test format [2]. All Test-Comp participants can write a generated test suite in this format, which stores a test suite in a test-suite directory with several files in XML format: (1) one metadata file that contains metadata about the created test suite, and (2) one additional file for each test vector. Each test-vector file lists the test values of that test case.

We represent programs as *control-flow automata* (CFA) [6]. A CFA is an automaton $P = (L, l_0, E)$ with a set $L$ of states, initial state $l_0$, and a set $E = L \times Ops \times L$ of edges, with set $Ops$ of all possible program operations. The set $L$ of states represents the program locations, the initial state $l_0$ represents the entry point of the program, and each control-flow edge $(l, op, l') \in E$ represents a program transfer where the control flows from program location $l$ to program location $l'$ and program operation $op$ is executed. An operation is either an *assignment*, an *assumption*, or a `nop`. An assignment $x := exp$ assigns the value of expression $exp$ to program variable $x$, where $exp$ is a either a constant or an

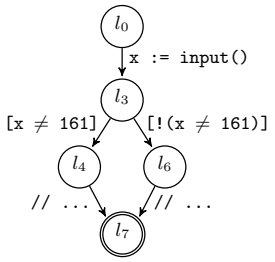


Fig. 4: CFA representation of the program in Fig. 2

arithmetic expression over constants and program variables. An assumption $[p]$ only transfers control from $l$ to $l'$ if $p$ is true, where $p$ is a boolean expression over constants and program variables. A `nop` is a program operation with no effect on the program's data state. A `nop` may have an arbitrary text label. Figure 4 shows a CFA representation of the program from our introductory example (Fig. 2). A *program path* $\pi = \langle l_0 \xrightarrow{op_0} l_1 \xrightarrow{op_1} \dots l_{n-1} \xrightarrow{op_{n-1}} l_n \rangle$ is a sequence of program locations that are sequentially connected through CFA edges $(l_i, op_i, l_{i+1}) \in E$. We write $\pi \in [\![P]\!]$ if $\pi$ is a program path of program $P$. The execution of a test vector $\bar{v}$ on a CFA $P$ results in a single, deterministic program path $\langle l_0 \xrightarrow{op_0} \dots \xrightarrow{op_{n-1}} l_n \rangle$, beginning at the program entry $l_0$. A test vector *covers* a test goal $g$ if its execution results in a program path that reaches $g$.

A *violation witness* [4] is a non-deterministic, finite-state automaton that describes a set of program paths from which at least one reaches a specification violation. From each violation witness, at least one test vector can be extracted [5] that follows a program path described by the witness.

A testability transformation [24] is a transformation $\mathcal{P} \times \mathcal{G} \to \mathcal{P} \times \mathcal{G}$ over the set $\mathcal{P}$ of programs and the set $\mathcal{G}$ of test-goal descriptions. A testability transformation $\tau$ transforms a given program $P$ and given test goals $G$ such that, for $\tau(P, G) = (P', G')$, the following holds: if a test-suite $S$ covers all test goals $G'$ on $P'$, test suite $S$ covers all test goals $G$ on $P$. The reducer presented in the following section will be based on a testability transformation that only transforms the program and keeps the test goals unchanged.

## 3 Construction of Conditional Testers from Existing Testers

Figure 5 shows how a conditional tester can be created from an off-the-shelf tester. A conditional software tester gets as input a program under test $P$, a coverage criterion $\varphi$, and a condition $\Psi_0$ (that describes already covered test goals). First, the set $G = [\![\varphi]\!] \setminus \Psi_0$ of remaining test goals that shall be covered is computed. Then, a program reducer `reducer` takes $G$ and $P$, and creates a residual program that contains the program behavior relevant for creating test cases that cover test goals in $G$ and that omits other program behavior. This residual program and coverage criterion $\varphi$ are then given to a (classic, existing)

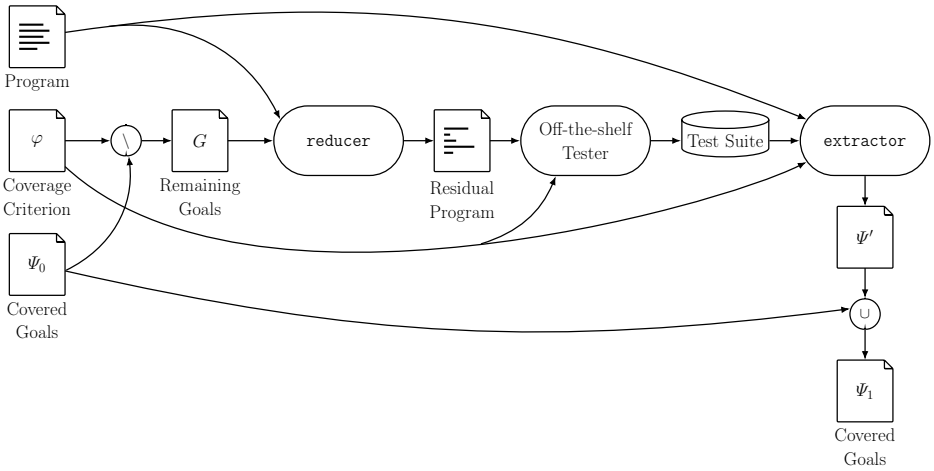


Fig. 5: Conditional tester $\texttt{tester}^{\texttt{cond}}$

tester, which creates new test cases based on them. Once the tester stops, the original program $P$, the coverage criterion $\varphi$, and the created test suite are given to a test-goal extractor `extractor`, which computes all test goals $\Psi'$ in $P$ that are described by $\varphi$ and that the test suite covers. Then, the newly covered goals $\Psi'$ are combined with $\Psi_0$ to get the full set $\Psi_1 = \Psi_0 \cup \Psi'$ of now covered test goals.

In the following, we will show requirements on the components `reducer` and `extractor`. We consider programs in their CFA representation. For ease of presentation, we assume that all program variables and constants are integers, and we only consider intra-procedural analysis here, i.e., programs with a single procedure. Our approach can be naturally extended to other data types and inter-procedural analysis. We represent test goals as CFA edges and describe conditions as sets of test goals.[1]

### 3.1 Program Reduction

A program reducer is a testability transformation $\texttt{reducer}_G : P \to P'$ that transforms, for a given set $G$ of test goals, a program $P$ to a program $P'$ that is G-coverage-equivalent to $P$. Two programs $P$ and $P'$ are *G-coverage-equivalent* if the two executions of $P$ and $P'$ on a test vector $\bar{v}$ cover the same subset $G_{\bar{v}} \subseteq G$ of test goals. Compared to traditional testability transformation [24], the set $G$ of test goals is not changed by `reducer` (we write $\texttt{reducer}_G : P \to P'$ as abbreviation for $\texttt{reducer} : P \times G \to P' \times G$). This allows us to run testers and generated test cases on the same coverage criterion, and no mapping between test goals is necessary. We require a program reducer to be *sound* and *complete*. Soundness is the basic requirement for testability transformations [24]. We also require completeness to ensure that test-case generation on the reduced program does not miss any test goal that is reachable in the original program.

[1] All coverage criteria that are based on code reachability can be reduced to reachability of CFA edges through testability transformations [24, 34].

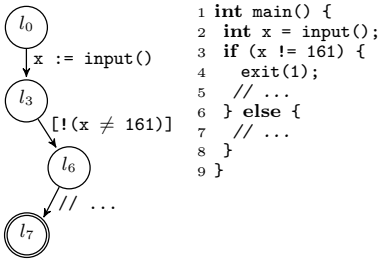


Fig. 6: Residual program for test goal $(l_6, \texttt{//}\ \ldots, l_7)$

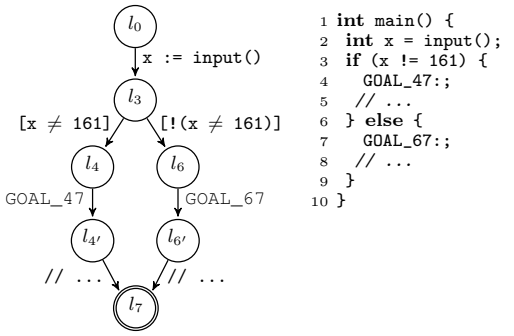


Fig. 7: Program instrumented by Alg. 1 for test-goal extraction

*Soundness.* Given a program $P$ and a set $G$ of test goals, the reducer $\texttt{reducer}_G$ is *sound* if the following holds: if a test vector $\bar{v}$ on program $P' = \texttt{reducer}_G(P)$ covers a test goal $g \in G$, then $\bar{v}$ on program $P$ covers $g$.

*Completeness.* Given a program $P$ and a set $G$ of test goals, the reducer $\texttt{reducer}_G$ is *complete* if the following holds: if a test vector $\bar{v}$ on program $P$ covers a test goal $g \in G$, then $\bar{v}$ on program $P' = \texttt{reducer}_G(P)$ covers $g$.

**Identity Reducer.** The program reducer $\texttt{reducer}^{\texttt{id}}$ is the identity, i.e., it returns a given program without any modification.

**Pruning Reducer.** The program reducer $\texttt{reducer}^{\texttt{prune}}$ is based on syntactic reachability. Given a CFA $P = (L, l_0, E)$ and a set $G \subseteq E$ of test goals, $\texttt{reducer}^{\texttt{prune}}$ computes a new CFA $P' = (L', l_0, E')$ that only contains program locations and their corresponding edges from which a test goal is reachable. Formally, $L' = \{l \in L \mid \exists (l_g, op_g, l'_g) \in G : \langle \ldots \xrightarrow{op} l \xrightarrow{op'} \ldots l_g \xrightarrow{op_g} l'_g \rangle \in [\![P]\!]\}$ and $E' = \{(l, op, l') \in E \mid l, l' \in L'\}$.

Figure 6 shows the result of $\texttt{reducer}^{\texttt{prune}}_{\{(l_6, \texttt{// } \ldots, l_7)\}}(P)$ for our example program (Fig. 4), as CFA and translated to C code. Because the left branch with condition $x \neq 161$ can never reach test goal $(l_6, \texttt{//}\ \ldots, l_7)$, it is removed from the CFA. C code can not express single assumption edges, so we translate the CFA by placing an `exit`-call after the first assumption that is not part of the CFA (line 4).

**Proposition 1.** *Program reducer* $\texttt{reducer}^{\texttt{prune}}$ *is sound.*

*Proof.* Given a program $P = (L, l_0, E)$ and a set $G \subseteq E$ of test goals, if a program path $\langle l_0 \xrightarrow{op} \ldots l_g \xrightarrow{op_g} l'_g \rangle$ with $(l_g, op_g, l'_g) \in G$ exists in program $P' = \texttt{reducer}^{\texttt{prune}}_G(P)$, then the same program path must exist in the original program $P$, by construction. So if the execution of a test vector $\bar{v}$ on $P'$ results in program path $\langle l_0 \xrightarrow{op} \ldots l_g \xrightarrow{op_g} l'_g \ldots \rangle$, then its execution on $P$ will result in the same program path, and thus also reach test goal $(l_g, op_g, l'_g)$.

**Proposition 2.** *Program reducer* $\texttt{reducer}^{\texttt{prune}}$ *is complete.*

*Proof.* Given a program $P = (L, l_0, E)$ and a set $G \subseteq E$ of test goals, if a program path $\langle l_0 \xrightarrow{op} \ldots l_g \xrightarrow{op_g} l'_g \rangle$ with $(l_g, op_g, l'_g) \in G$ exists in program $P$, then the same program path must exist in the reduced program $P' = \texttt{reducer}_G^{\texttt{prune}}(P)$, by construction. So if the execution of a test vector $\bar{v}$ on $P$ results in program path $\langle l_0 \xrightarrow{op} \ldots l_g \xrightarrow{op_g} l'_g \ldots \rangle$, then its execution on $P'$ will result in the same program path, and thus also reach test goal $(l_g, op_g, l'_g)$.

**Annotating Reducer.** Program reducer $\texttt{reducer}^{\texttt{annot}}$ is based on program annotations. Given a CFA $P = (L, l_0, E)$ and a set $G \subseteq E$ of test goals, $\texttt{reducer}^{\texttt{annot}}$ computes (analogous to adding labels, Alg. 1) a new CFA $P' = (L', l_0, E')$ that contains a call to custom method `VERIFIER_error` before each test goal. Method `VERIFIER_error` is defined as an empty method, i.e., it has no effect on the program state, but it can be used to guide supporting testers. Since $\texttt{reducer}^{\texttt{annot}}$ does not change program behavior, it is a sound and complete program reducer.

### 3.2 Test-Goal Extraction

A test-goal extractor `extractor` takes as input a program $P$, a coverage criterion $\varphi$, and a test suite, and returns as output a set $\Psi$ of test goals that are covered by the test suite. We require a test-goal extractor to be *sound* and *complete*.

*Soundness.* Given a program $P$, a coverage criterion $\varphi$, and a test suite $S$ that covers a set $G \subseteq [\![\varphi]\!]$ of test goals, then a test-goal extractor `extractor` is *sound*, if the set $\Psi = \texttt{extractor}(P, \varphi, S)$ only contains test goals that are covered by $S$, i.e., $\Psi \subseteq G$.

*Completeness.* Given a program $P$, a coverage criterion $\varphi$, and a test suite $S$ that covers a set $G \subseteq [\![\varphi]\!]$ of test goals, then a test-goal extractor `extractor` is *complete*, if the set $\Psi = \texttt{extractor}(P, \varphi, S)$ contains all test goals that are covered by $S$, i.e., $\Psi \supseteq G$.

**Test-Goal Extraction Based on Test Execution.** Test-goal extractor $\texttt{extractor}^{\texttt{exec}}$ computes covered test goals through execution. For a program $P$, a coverage criterion $\varphi$, and a test suite $S$, it executes each test vector $\bar{v}_i \in S$ on program $P$ and records the CFA edges of the resulting program path $\pi_i = \langle l_0 \xrightarrow{op} \ldots \xrightarrow{op_{n-1}} l_n \rangle$. From these, it computes the set of test goals covered by $S$, i.e., $\Psi = \bigcup_{\pi_i} \{(l, op, l') \in \pi_i\}$.

To be able to easily identify test goals in real C code, we perform a testability transformation that adds, for each test goal $g \in [\![\varphi]\!]$, a `nop` with label `GOAL_i_j`. Test-goal extraction for branch coverage consists of four steps: (1) Computing the set of test goals (*test-goal computation*), (2) adding, for each test goal, a label to the original program that identifies that test goal in the code (*testability transformation*), (3) executing the test suite on that transformed program (*test execution*), and (4) checking which labels are covered by the test suite (*coverage measurement*).

**Algorithm 1** Testability Transformation: $addLabels(P, G)$

**Input:** CFA $P = (L, l_0, E)$, test goals $G \subseteq E$
**Output:** CFA $(L', l_0, E')$ with test-goal labels
**Variables:** Sets $waitlist, visited \subseteq L$

```
L', E' = {}
waitlist, visited = {l_0}
while waitlist ≠ ∅ do
  choose l_i from waitlist; remove l_i from waitlist
  for (l_i, op, l_j) ∈ E do
    L' = L' ∪ {l_i, l_j}
    if (l_i, op, l_j) ∈ G then
      L' = L' ∪ {l'_i}
      E' = E' ∪ {(l_i, GOAL_i_j, l'_i), (l'_i, op, l_j)}
    else
      E' = E' ∪ {(l_i, op, l_j) ∈ E}
    if l_j ∉ visited then
      waitlist = waitlist ∪ {l_j}
      visited = visited ∪ {l_j}
return (L', l_0, E')
```

*(1) Test-Goal Computation.* As an example, we use the coverage criterion of branch coverage. For branch coverage and a CFA $(L, l_0, E)$, we use as test goals the set of all edges that are preceded by assume edges, i.e., $[\![\varphi]\!] = \{(l, \cdot, \cdot) \in E \mid \exists (\cdot, op, l) \in E : op \text{ is assume operation}\}$.

*(2) Testability Transformation.* We first translate a given program in real C code to a CFA $P$. Algorithm 1 takes this CFA $P$ and creates a semantically equivalent CFA with additional edges for program labels. For $P = (L, l_0, E)$, the new CFA $P' = (L', l_0, E')$ is computed as follows: Initially, the sets $L'$ and $E'$ are empty. A waitlist is initialized with the initial program location $l_0$. As long as the waitlist is not empty, a program location $l_i$ is selected and removed from the waitlist and each outgoing edge $(l_i, op, l_j) \in E$ is considered. First, $l_i$ and $l_j$ are added to $L'$.

Then, if $(l_i, op, l_j)$ is a test goal, a new program label `GOAL_i_j` is introduced just before $op$ as follows: A new program location $l'_i$ is added to $L'$, and the two edges $(l_i, \texttt{GOAL\_i\_j}, l'_i)$ and $(l'_i, op, l_j)$ are added to $E'$.

If the edge $(l_i, op, l_j)$ is not a test goal, it is added to $E'$ without modifications. After this, if $l_j$ was not encountered before, it is added to the waitlist and the set of visited nodes. As soon as the waitlist is empty, all locations of the original CFA have been traversed and the new CFA $(L', l_0, E')$ is returned. This transformation traverses each program location only once and thus scales well. At the end, we translate the transformed CFA back into C code.

Figure 7 shows the result of $addLabels(P, G)$ for our example program $P$ (Fig. 4) and branch coverage, i.e., $G = \{(l_4, \texttt{//} \ \ldots, l_7), (l_6, \texttt{//} \ \ldots, l_7)\}$. The figure shows the resulting CFA and the translation to C code.

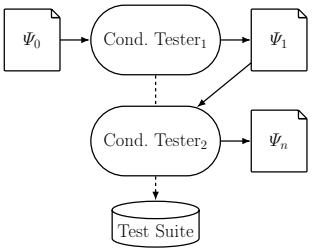


Fig. 8: `tester`$^{\texttt{seq}}$

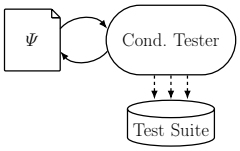


Fig. 9: `tester`$^{\texttt{cyc}}$

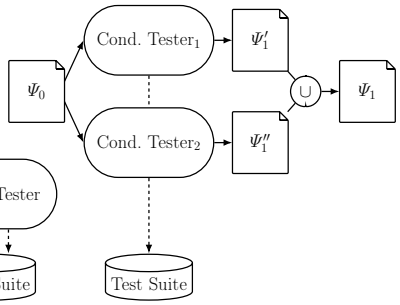


Fig. 10: `tester`$^{\texttt{par}}$

*(3) Test Execution.* We execute all test cases of the given test suite on the transformed program as follows: We generate a test harness that reads test values from the standard input and provides the test values to the C program. We compile this test harness with the transformed program and feed each test vector to the harness in individual executions.

*(4) Coverage Measurement.* We use GCov to obtain a coverage report that lists for each line [2] of the transformed C program whether it was covered by the test suite. From this report, `extractor`$^{\texttt{exec}}$ extracts the program labels of test goals that are covered, and returns the corresponding test goals.

Since `extractor`$^{\texttt{exec}}$ is based on concrete execution of the test suite on a semantically equivalent program, the method is assumed to be both sound and complete.

## 4 Combinations of Conditional Testers

Conditional testing enables versatile combinations of testers. We have already seen a sequential combination in the introduction (Fig. 3), but it is also possible to combine conditional testers in other ways, such as in cycles, in general portfolios (i.e., also parallel), with strategy selection, or for compositional reasoning. In the following, we will present different possible combinations of conditional testers to show the potential of conditional software testing. Note that all of these combinations are themselves conditional testers, so they can be combined with each other in any way. From now on, we will omit the program under test and the coverage criterion in figures, to have simpler diagrams.

**Sequential Tester.** A sequential tester `tester`$^{\texttt{seq}}(T_1, T_2)$ (Fig. 8) consists of two component testers $T_1$ and $T_2$ that are executed sequentially to generate test cases. Several sequential testers can be used to sequentially combine an arbitrary number of testers. For simplicity, we write `tester`$^{\texttt{seq}}(T_1, T_2, T_3)$ for `tester`$^{\texttt{seq}}(T_1, $`tester`$^{\texttt{seq}}(T_2, T_3))$. Each tester provides the covered test goals

[2] Since the transformed program is generated such that each operation is written on an own line in the output code, a line uniquely identifies a test-goal label.

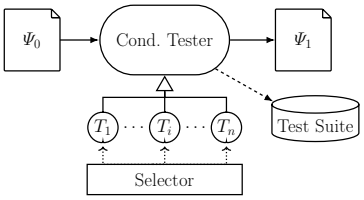


Fig. 11: $\texttt{tester}^{\texttt{select}}$

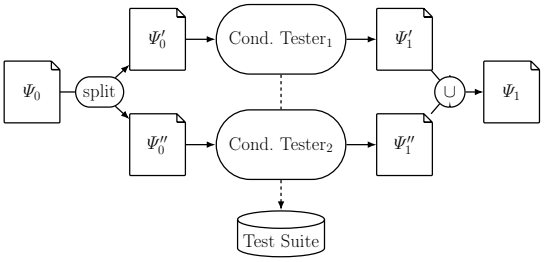


Fig. 12: $\texttt{tester}^{\texttt{comp}}$

after its run, and the set of remaining test goals will decrease. This can be used to combine strengths of different testers without further knowledge about them; testers can either get a certain time limit each, or stop early if they encounter a program feature they don't support.

**Cyclic Tester.** A cyclic tester $\texttt{tester}^{\texttt{cyc}}(T)$ (Fig. 9) iteratively calls a conditional tester $T$ with the increasing set $\Psi$ of covered test goals. This can be used, for example, to restart a tester after a certain limit is reached (e.g., memory consumption or size of path constraints in symbolic execution). In combination with $\texttt{tester}^{\texttt{seq}}$, this can also be used to cycle through a sequence of testers (round-robin principle).

**Parallel Tester.** A parallel tester $\texttt{tester}^{\texttt{par}}(T_1, T_2)$ (Fig. 10) runs testers $T_1$ and $T_2$ in parallel on the same inputs. Each tester produces its own set $\Psi'_1, \Psi''_1$ of covered test goals, and their union $\Psi'_1 \cup \Psi''_1 = \Psi_1$ is the final set of covered test goals. Several parallel testers can be used to combine an arbitrary number of testers, similar to $\texttt{tester}^{\texttt{seq}}$. In contrast to $\texttt{tester}^{\texttt{seq}}$, there is no information exchange between testers $T_1$ and $T_2$, so they may do redundant work.

**Strategy-Selection Tester.** A strategy-selection tester $\texttt{tester}^{\texttt{select}}(T_1, \ldots, T_n)$ (Fig. 11) uses a selector function to select to which of testers $T_1, \ldots, T_n$ the task of test-case generation is delegated. The selector function can be an arbitrary function that returns one of $T_1$ to $T_n$, e.g., a random selection, or based on a selection model that selects the most suited tester based on features of the program under test.

**Compositional Tester.** A compositional tester $\texttt{tester}^{\texttt{comp}}(T_1, T_2)$ (Fig. 12) first splits the condition $\Psi_0$ into two sets $\Psi'_0$ and $\Psi''_0$, so that $\Psi_0 = \Psi'_0 \cup \Psi''_0$. Then, tester $T_1$ gets as input the first set $\Psi'_0$, and tester $T_2$ gets as input the second set $\Psi''_0$. Both testers work on the original program $P$ and original coverage criterion $\varphi$, but due to $\Psi'_0$ and $\Psi''_0$, the first tester only works on test goals $[\![\varphi]\!] \setminus \Psi'_0$, and the second tester only works on $[\![\varphi]\!] \setminus \Psi''_0$. They produce individual sets $\Psi'_1$ and $\Psi''_1$ of covered test goals. These are then merged into the final set $\Psi_1 = \Psi'_1 \cup \Psi''_1$ of now covered test goals. More than two testers can be combined compositionally through nested combinations. With $\texttt{tester}^{\texttt{comp}}$, work can be split (decomposition principle), for example for parallelization or to let each tester solve the test goals it is most suited for.

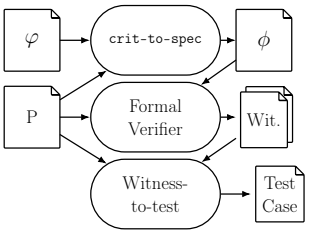


Fig. 13: $\texttt{tester}^{\texttt{veri}}$

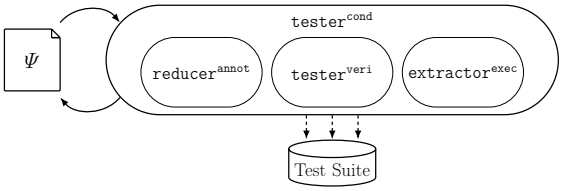


Fig. 14: $\texttt{tester}^{\texttt{cyc}}_{\texttt{veri}}$

## 5 Construction of Conditional Testers from Existing Verifiers

It has long been possible to use formal verification of reachability properties to generate tests [3]. Compared to testers, many formal verification techniques specialize on finding single program paths to specific program states or program locations of interest; this makes them suitable for hard-to-reach test goals [10]. Figure 13 shows the (non-conditional) tester $\texttt{tester}^{\texttt{veri}}(V)$ that is based on a formal verifier $V$: First, function `crit-to-spec` transforms the coverage criterion $\varphi$, based on program $P$, to a safety specification $\phi$ which is constructed such that $P$ violates $\phi$ if $P$ covers a test goal from $[\![\varphi]\!]$. Then, $\phi$ and $P$ are given to formal verifier $V$, which checks whether $P$ satisfies $\phi$. The verifier outputs one or more violation witnesses if test goals are reachable. From these violation witnesses, test cases are created by witness-to-test [5].

We use the established formats for input programs and specifications for the reachability category of SV-COMP [3] to get access to a large catalog of tools for formal verification. There are two adaptations necessary for using SV-COMP verifiers: (1) they are only required to support the property "no call to method `__VERIFIER_error` is reachable", and may not support more general reachability properties, and (2) they are only required to output a single violation witness, and thus will always lead to a test suite that only consists of one test case.

We solve both issues in the following way: We let `crit-to-spec` always return the specification that no call to `__VERIFIER_error` is reachable. We then take $\texttt{tester}^{\texttt{veri}}$ and construct from it a conditional tester based on $\texttt{tester}^{\texttt{cond}}$ with program reducer $\texttt{reducer}^{\texttt{annot}}$ and test-goal extractor $\texttt{extractor}^{\texttt{exec}}$. At this point, we have a conditional tester that uses a formal verifier to always produce a test suite with a single test case, and that returns the set of test goals covered by that test case. To produce a full test suite for all test goals, we use a cyclic tester $\texttt{tester}^{\texttt{cyc}}(\texttt{tester}^{\texttt{cond}}(\texttt{tester}^{\texttt{veri}}(V)))$ (Fig. 14). After each test-case generation run, the newly created test case is used by $\texttt{extractor}^{\texttt{exec}}$ to update the covered test goals. Then, $\texttt{reducer}^{\texttt{annot}}$ will insert calls to `__VERIFIER_error` for the remaining test goals, and $\texttt{tester}^{\texttt{veri}}(V)$ will create a new test case that covers at least one of the remaining test goals. We use $\texttt{tester}^{\texttt{cyc}}_{\texttt{veri}}(V)$ to denote a verifier-based tester that is constructed from formal verifier $V$.

[3] https://sv-comp.sosy-lab.org/2019/rules.php

Through the use of any of the previously mentioned combinations, a tester $\texttt{tester}^{\texttt{cyc}}_{\texttt{veri}}$ can be combined with other conditional testers.

## 6 Evaluation

We evaluate our tool implementation CondTest and some combinations of testers using conditional testing along the following claims:

**C1** Conditional software testing with $\texttt{extractor}^{\texttt{exec}}$ and $\texttt{reducer}^{\texttt{prune}}$ does not significantly impact the performance of individual testers.
**C2** Sequential combinations of different testers without information exchange can improve the coverage of generated test suites, compared to single testers.
**C3** Sequential combinations of different testers with conditional software testing can improve the coverage of generated test suites, compared to sequential combinations without information exchange.
**C4** Sequential combinations of traditional testers and verifier-based testers can improve the coverage of generated test suites.

### 6.1 Setup

**Implementation.** We implemented a generic conditional software tester (CondTest) according to Fig. 5, including the operators $\texttt{reducer}^{\texttt{id}}$, $\texttt{reducer}^{\texttt{prune}}$, $\texttt{reducer}^{\texttt{annot}}$, and $\texttt{extractor}^{\texttt{exec}}$. CondTest can be instantiated as $\texttt{tester}^{\texttt{cond}}$, $\texttt{tester}^{\texttt{seq}}$, and $\texttt{tester}^{\texttt{cyc}}_{\texttt{veri}}$, is able to create test suites for C programs that adhere to the Test-Comp rules [2], and is available under the open-source license Apache 2.0. We use CondTest in version `2.0` [4]. CondTest uses the BenchExec tool-info modules [5] and benchmark definitions of Test-Comp [6] and SV-COMP [7] for plug-and-play integration of testers and formal verifiers. Formal verifiers are turned into testers by wrapping them each in their own instance of CondTest (configuration $\texttt{tester}^{\texttt{cyc}}_{\texttt{veri}}$).

**Tools.** We consider the best three testers of Test-Comp '19 whose licenses allow evaluation and publication of results: Klee [14], CoVeriTest [8], and CPA/Tiger[8]. We use all three tools in their respective versions of Test-Comp '19. In addition, we select the best formal verifier for reaching program locations of interest in testable programs according to a previous study [10], i.e., Esbmc-kind [21]. We use Esbmc-kind in its SV-COMP '19 version. To measure the coverage of test suites, we use GCov `7.3.0`. To ensure reproducible results, we use the benchmarking toolkit BenchExec `1.20` [11].

[4] https://gitlab.com/sosy-lab/software/conditional-testing/tree/v2.0
[5] https://github.com/sosy-lab/benchexec/tree/2.0/benchexec/tools
[6] https://gitlab.com/sosy-lab/test-comp/bench-defs/tree/testcomp19/benchmark-defs
[7] https://github.com/sosy-lab/sv-comp/tree/svcomp19/benchmark-defs
[8] https://www.es.tu-darmstadt.de/testcomp19/

**Environment.** We perform our experiments on a cluster of 168 machines, each with 33 GB of memory and an Intel Xeon E3-1230 v5 CPU, with 3.4 GHz and 8 processing units (with hyper-threading). We use Ubuntu 18.04 with Linux kernel 4.4 as operating system. We limit each benchmark run to 4 processing units and a time limit of 900 s. Each run of CondTest is limited to 15.5 GB. Each individual test-case generation run (e.g., execution of CPA/Tiger) is limited to 15 GB, both for native execution and as part of CondTest. This way, each test-case generation run has the same amount of memory for both native execution and execution within CondTest. Extractor $\texttt{extractor}^{\texttt{exec}}$ uses a time limit of 3 s for each test execution, to prevent hangups in case of incomplete or non-terminating tests. To measure the achieved coverage of the complete final test suites, we execute test cases with a memory limit of 7 GB, 2 processing units, and a time limit of 3 hours for each generated test suite. At this time limit, no timeouts occurred during coverage measurement.

**Reproducibility and Benchmark Tasks.** We use all 1 720 test tasks of the *Cover-Branches* category of the Test-Comp '19 benchmark. All of our experimental data are available online. [9] and through a replication package. [10]

### 6.2 Results

**C1: No Significant Overhead in CondTest.** Figure 15 shows the branch coverage per task achieved by the test suites created by CoVeriTest, CPA/Tiger, and Klee, respectively, in their original Test-Comp configurations (x-axis), and as conditional testers $\texttt{tester}^{\texttt{cond}}$ (y-axis) inside CondTest with $\texttt{reducer}^{\texttt{prune}}$ and $\texttt{extractor}^{\texttt{exec}}$ ($\texttt{reducer}^{\texttt{prune}}$ does not really prune anything because of the full set of test goals, but parses the program, runs the pruning algorithm, and writes out the transformed C program; the idea is to find out whether this process is efficient and does not negatively impact the overall process). The CPU-time limit for each test-case generation was set to 900 s (the CPU time consumed by $\texttt{reducer}^{\texttt{prune}}$ is included in the measured CPU time, and thus, implicitly subtracted from the CPU-time available for the tester). Since $\texttt{extractor}^{\texttt{exec}}$ only runs after the tester, it has no influence on the time limit in a configuration with a single tester. For points on the diagonal, the same coverage was achieved by the original tester and its integration in $\texttt{tester}^{\texttt{cond}}$; points above the diagonal represent tasks for which $\texttt{tester}^{\texttt{cond}}$ achieved a higher coverage, and the points below the diagonal represent tasks for which $\texttt{tester}^{\texttt{cond}}$ achieved a lower coverage. For CoVeriTest, the coverage for a few tasks are a bit worse when run with CondTest (just below the diagonal). This is because CondTest uses a different, more strict technique to enforce the memory limit than the benchmarking tool BenchExec, due to technical reasons. For CPA/Tiger, outliers on the left (vertical stack of points) are due to crashes from memory exhaustion. CPA/Tiger operates close to the memory limit for many tasks. Because of this, small variations in

[9] https://www.sosy-lab.org/research/conditional-testing/
[10] https://doi.org/10.5281/zenodo.3352401

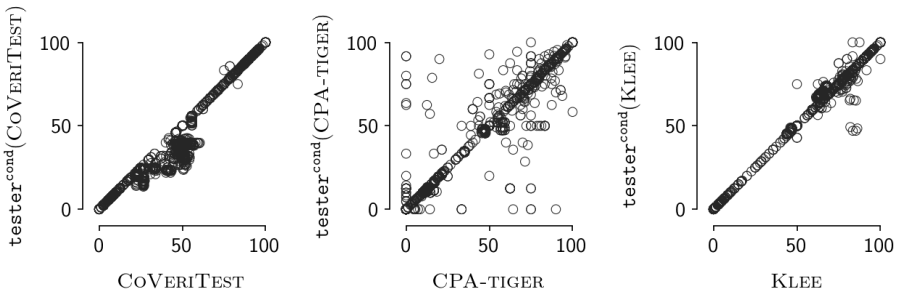


Fig. 15: Branch coverage of test suites created by original tools vs. their integration in `tester`$^{\texttt{cond}}$ (in percent)

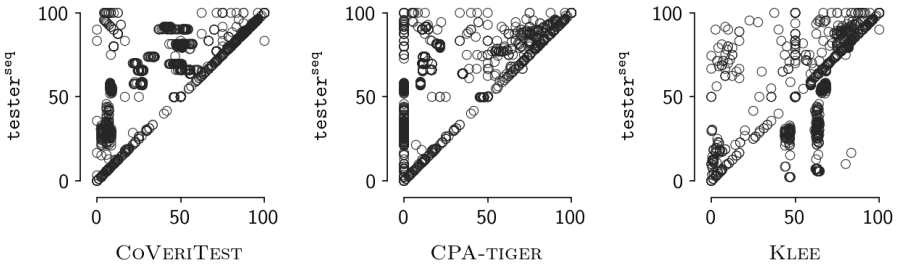


Fig. 16: Branch coverage of test suites created by original tools vs. their sequential combinations with `reducer`$^{\texttt{id}}$, i.e., without information exchange (in percent)

memory usage can lead to crashes. In our experiments, for most of these tasks it was random whether CPA/Tiger stayed closely below the memory limit, or exceeded it and crashed. Thus, the issue is not related to CondTest, but results from memory exhaustion of the native tester. Besides these issues, it is visible that for all three testers, no significant differences in branch coverage exist. This suggests that using `tester`$^{\texttt{cond}}$ with the proposed operators `reducer`$^{\texttt{prune}}$ and `extractor`$^{\texttt{exec}}$ does not lead to a significant negative impact on the performance just by using the conditional-testing construction.

**C2: Combinations Can Improve Coverage.** Figure 16 shows the branch coverage per task achieved by the test suites created by CoVeriTest, CPA/Tiger, and Klee, respectively, in their original Test-Comp configurations with 900 s CPU-time limit (x-axis), and the coverage per task achieved by the test suites created by CondTest (y-axis) with the sequential combination `tester`$^{\texttt{seq}}$(`tester`$^{\texttt{cond}}$(CPA/Tiger)$_{300}$, `tester`$^{\texttt{cond}}$(CoVeriTest)$_{300}$, `tester`$^{\texttt{cond}}$(Klee)$_{300}$) and the reducer `reducer`$^{\texttt{id}}$, i.e., without information exchange between the three testers. Each single conditional tester (i.e., `tester`$^{\texttt{cond}}$ based on CPA/Tiger, CoVeriTest, and Klee) was stopped after 300 s each, and each full test-case generation `tester`$^{\texttt{seq}}$ run was stopped after a total of 900 s (the CPU time consumed by CondTest for, e.g., process management, is included in the measured CPU time, and thus, implicitly subtracted from the CPU-time available for the last tester, `tester`$^{\texttt{cond}}$(Klee)).

Table 1: Coverage of test suites generated without information reuse ($\texttt{reducer}^{\texttt{id}}$) and with information reuse through $\texttt{reducer}^{\texttt{prune}}$

| **Task** | branch coverage | | |
|---|---|---|---|
| | `id` | → | `prune` |
| mod3.c.v+sep-reducer | 75.0 | +5.00 | 80.0 |
| Problem07_label35 | 52.0 | +2.00 | 54.0 |
| Problem07_label37 | 54.2 | +1.97 | 56.2 |
| Problem04_label35 | 79.5 | +1.79 | 81.3 |
| Problem06_label02 | 57.0 | +1.70 | 58.7 |
| Problem06_label27 | 57.5 | +1.09 | 58.6 |
| Problem04_label02 | 80.2 | +1.06 | 81.3 |
| Problem06_label18 | 57.5 | +1.05 | 58.6 |
| Problem04_label16 | 79.1 | +1.01 | 80.1 |
| Problem04_label34 | 80.2 | +0.99 | 81.2 |

Table 2: Coverage of test suites generated without (`prune`) and with (`vb`) support of ESBMC-KIND

| **Task** | branch coverage | | |
|---|---|---|---|
| | `prune` | → | `vb` |
| Problem08_label30 | 5.72 | +56.2 | 62.0 |
| Problem08_label32 | 5.72 | +56.1 | 61.9 |
| Problem08_label06 | 5.72 | +56.1 | 61.8 |
| Problem08_label35 | 5.72 | +56.0 | 61.7 |
| Problem08_label00 | 5.72 | +55.9 | 61.6 |
| Problem08_label11 | 5.72 | +55.8 | 61.5 |
| Problem08_label19 | 5.72 | +55.7 | 61.5 |
| Problem08_label29 | 5.67 | +55.7 | 61.4 |
| Problem08_label22 | 5.72 | +55.7 | 61.5 |
| Problem08_label56 | 5.72 | +55.7 | 61.5 |

The scatter plots in Fig. 16 show that the branch coverage of the test suites created by the sequential combination is significantly higher for a significant amount of benchmark tasks. This shows that the used testers (with CPU time limit of 300 s) can complement each other well, and that combinations can perform better than a single tester running for a longer time on its own (900 s CPU time limit).

**C3: Condition Passing Can Further Improve Coverage.** To show that conditional software testing can lead to generated test suites with improved coverage, we compare the branch coverage of the test suites generated by CONDTEST with $\texttt{tester}^{\texttt{seq}}(\texttt{tester}^{\texttt{cond}}(\text{CPA/TIGER})_{300}$, $\texttt{tester}^{\texttt{cond}}(\text{COVERITEST})_{300}$, $\texttt{tester}^{\texttt{cond}}(\text{KLEE})_{300}$), and the two reducers $\texttt{reducer}^{\texttt{id}}$, i.e., without information exchange, and $\texttt{reducer}^{\texttt{prune}}$, i.e., with program reduction based on syntactic reachability. Table 1 shows a comparison of the branch coverage of test suites generated by both techniques on a selection of benchmark tasks (programs with complicated branching), rounded to three digits. It shows that information exchange can lead to generated test suites with improved branch coverage, adding up to 5 % branch coverage.

**C4: Verifiers as Test-Generators Can Improve Coverage.** To show that verifier-based testers can generate test suites with improved coverage compared to combinations of traditional testers, we compare the branch coverage of the test suites generated by CONDTEST with $\texttt{tester}^{\texttt{seq}}(\texttt{tester}^{\texttt{cond}}(\text{CPA/TIGER})_{300}$, $\texttt{tester}^{\texttt{cond}}(\text{COVERITEST})_{300}$, $\texttt{tester}^{\texttt{cond}}(\text{KLEE})_{300}$) (called `prune`) and the test suites generated by CONDTEST with $\texttt{tester}^{\texttt{seq}}(\texttt{tester}^{\texttt{cond}}(\text{CPA/TIGER})_{200}$, $\texttt{tester}^{\texttt{cond}}(\text{COVERITEST})_{200}$, $\texttt{tester}^{\texttt{cond}}(\text{KLEE})_{200}$, $\texttt{tester}^{\texttt{cyc}}_{\texttt{veri}}(\text{ESBMC})_{300}$) (called `vb`). Both `prune` and `vb` use $\texttt{reducer}^{\texttt{prune}}$ and $\texttt{extractor}^{\texttt{exec}}$. For `prune`, each individual tester is stopped after 300 s. For `vb`, CPA/TIGER, COVERITEST, and KLEE are each stopped after 200 s, and ESBMC runs for 300 s. The total time of each run of CONDTEST is 900 s (i.e., the CPU time required by $\texttt{reducer}^{\texttt{prune}}$ and $\texttt{extractor}^{\texttt{exec}}$ is implicitly subtracted from the CPU time available for the last tester, i.e., KLEE in `prune` and ESBMC in `vb`).

Table 2 shows a comparison of the branch coverage of test suites generated by `prune` and `vb`, respectively, on a selection of benchmark tasks (programs with complicated branching). It shows that for some tasks, the use of Esbmc as directed tester can greatly improve branch coverage compared to combinations of only traditional testers, creating test suites that achieve up to 56 % additional branch coverage.

## 7 Conclusion

We have presented the concept of *conditional testing* and the tool implementation CondTest, a versatile and modular framework for constructing cooperative combinations of testers based on conditional testing. First, we defined a construction of a conditional tester from a given *existing tester*, based on the components `reducer` and `extractor`. Second, we defined a set of *generic combinations* that are now all possible using conditional testing. Third, we defined a construction of a conditional tester from a given *existing verifier*, based on the outlined combination opportunities. All our concepts are implemented in an adjustable framework, and we showed the potential of some new combinations through an experimental evaluation.

There are many powerful techniques for automatic test-case generation. Our goal is to construct even more powerful combinations by leveraging *cooperation*, and we hope that our construction techniques based on *conditional testing* help also other researchers and engineers to construct powerful tool combinations, without changing the implementation of the existing tools. This contributes to optimally use the techniques that we have to further improve the quality of software.

## References


1. Beyer, D.: Automatic verification of C and Java programs: SV-COMP 2019. In: Proc. TACAS (3). pp. 133–155. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_9
2. Beyer, D.: Competition on software testing (Test-Comp). In: Proc. TACAS (3). pp. 167–175. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_11
3. Beyer, D., Chlipala, A.J., Henzinger, T.A., Jhala, R., Majumdar, R.: Generating tests from counterexamples. In: Proc. ICSE. pp. 326–335. IEEE (2004). https://doi.org/10.1109/ICSE.2004.1317455
4. Beyer, D., Dangl, M., Dietsch, D., Heizmann, M., Stahlbauer, A.: Witness validation and stepwise testification across software verifiers. In: Proc. FSE. pp. 721–733. ACM (2015). https://doi.org/10.1145/2786805.2786867
5. Beyer, D., Dangl, M., Lemberger, T., Tautschnig, M.: Tests from witnesses: Execution-based validation of verification results. In: Proc. TAP. pp. 3–23. LNCS 10889, Springer (2018). https://doi.org/10.1007/978-3-319-92994-1_1
6. Beyer, D., Gulwani, S., Schmidt, D.: Combining model checking and data-flow analysis. In: Handbook on Model Checking, pp. 493–540. Springer (2018). https://doi.org/10.1007/978-3-319-10575-8_16
7. Beyer, D., Henzinger, T.A., Keremoglu, M.E., Wendler, P.: Conditional model checking: A technique to pass information between verifiers. In: Proc. FSE. ACM (2012). https://doi.org/10.1145/2393596.2393664

8. Beyer, D., Jakobs, M.C.: CoVeriTest: Cooperative verifier-based testing. In: Proc. FASE. pp. 389–408. LNCS 11424, Springer (2019). https://doi.org/10.1007/978-3-030-16722-6_23
9. Beyer, D., Jakobs, M.C., Lemberger, T., Wehrheim, H.: Reducer-based construction of conditional verifiers. In: Proc. ICSE. pp. 1182–1193. ACM (2018). https://doi.org/10.1145/3180155.3180259
10. Beyer, D., Lemberger, T.: Software verification: Testing vs. model checking. In: Proc. HVC. pp. 99–114. LNCS 10629, Springer (2017). https://doi.org/10.1007/978-3-319-70389-3_7
11. 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
12. Beyer, D., Wehrheim, H.: Verification artifacts in cooperative verification: Survey and unifying component framework. arXiv/CoRR **1905**(08505) (May 2019), https://arxiv.org/abs/1905.08505
13. Böhme, M., d. S. Oliveira, B.C., Roychoudhury, A.: Partition-based regression verification. In: Proc. ICSE. pp. 302–311. IEEE (2013). https://doi.org/10.1109/ICSE.2013.6606576
14. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proc. OSDI. pp. 209–224. USENIX Association (2008)
15. Chowdhury, A.B., Medicherla, R.K., Venkatesh, R.: VeriFuzz: Program-aware fuzzing (competition contribution). In: Proc. TACAS (3). pp. 244–249. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_22
16. Christakis, M., Müller, P., Wüstholz, V.: Collaborative verification and testing with explicit assumptions. In: Proc. FM. pp. 132–146. LNCS 7436, Springer (2012). https://doi.org/10.1007/978-3-642-32759-9_13
17. Cruanes, S., Hamon, G., Owre, S., Shankar, N.: Tool integration with the Evidential Tool Bus. In: Proc. VMCAI. pp. 275–294. LNCS 7737, Springer (2013). https://doi.org/10.1007/978-3-642-35873-9_18
18. Czech, M., Jakobs, M., Wehrheim, H.: Just test what you cannot verify! In: Proc. FASE. pp. 100–114. LNCS 9033, Springer (2015). https://doi.org/10.1007/978-3-662-46675-9_7
19. Daca, P., Gupta, A., Henzinger, T.A.: Abstraction-driven concolic testing. In: Proc. VMCAI. pp. 328–347. LNCS 9583, Springer (2016). https://doi.org/10.1007/978-3-662-49122-5_16
20. Ferles, K., Wüstholz, V., Christakis, M., Dillig, I.: Failure-directed program trimming. In: Proc. ESEC/FSE. pp. 174–185. ACM (2017). https://doi.org/10.1145/3106237.3106249
21. Gadelha, M.Y.R., Monteiro, F.R., Cordeiro, L.C., Nicole, D.A.: ESBMC v6.0: Verifying C programs using k-induction and invariant inference (competition contribution). In: Proc. TACAS (3). pp. 209–213. LNCS 11429, Springer (2019). https://doi.org/10.1007/978-3-030-17502-3_15
22. Gennari, J., Gurfinkel, A., Kahsai, T., Navas, J.A., Schwartz, E.J.: Executable counterexamples in software model checking. In: Proc. VSTTE. pp. 17–37. LNCS 11294, Springer (2018). https://doi.org/10.1007/978-3-030-03592-1_2
23. Gulavani, B.S., Henzinger, T.A., Kannan, Y., Nori, A.V., Rajamani, S.K.: Synergy: A new algorithm for property checking. In: Proc. FSE. pp. 117–127. ACM (2006). https://doi.org/10.1145/1181775.1181790

24. Harman, M., Hu, L., Hierons, R.M., Wegener, J., Sthamer, H., Baresel, A., Roper, M.: Testability transformation. IEEE Trans. Software Eng. **30**(1), 3–16 (2004). https://doi.org/10.1109/TSE.2004.1265732
25. Holzer, A., Schallhart, C., Tautschnig, M., Veith, H.: Query-driven program testing. In: Proc. VMCAI. pp. 151–166. LNCS 5403, Springer (2009). https://doi.org/10.1007/978-3-540-93900-9_15
26. Holzer, A., Schallhart, C., Tautschnig, M., Veith, H.: How did you specify your test suite. In: Proc. ASE. pp. 407–416. ACM (2010). https://doi.org/10.1145/1858996.1859084
27. Howar, F., Isberner, M., Merten, M., Steffen, B., Beyer, D., Păsăreanu, C.S.: Rigorous examination of reactive systems. The RERS challenges 2012 and 2013. Int. J. Softw. Tools Technol. Transfer **16**(5), 457–464 (2014). https://doi.org/10.1007/s10009-014-0337-y
28. Kim, Y., Xu, Z., Kim, M., Cohen, M.B., Rothermel, G.: Hybrid directed test suite augmentation: An interleaving framework. In: Proc. ICST. pp. 263–272. IEEE (2014). https://doi.org/10.1109/ICST.2014.39
29. Majumdar, R., Sen, K.: Hybrid concolic testing. In: Proc. ICSE. pp. 416–426. IEEE (2007). https://doi.org/10.1109/ICSE.2007.41
30. Margaria, T., Nagel, R., Steffen, B.: jETI: A tool for remote tool integration. In: Proc. TACAS. pp. 557–562. LNCS 3440, Springer (2005). https://doi.org/10.1007/978-3-540-31980-1_38
31. Noller, Y., Kersten, R., Pasareanu, C.S.: Badger: complexity analysis with fuzzing and symbolic execution. In: Proc. ISSTA. pp. 322–332. ACM (2018). https://doi.org/10.1145/3213846.3213868
32. Qiu, R., Khurshid, S., Pasareanu, C.S., Wen, J., Yang, G.: Using test ranges to improve symbolic execution. In: Proc. NFM. pp. 416–434. LNCS 10811, Springer (2018). https://doi.org/10.1007/978-3-319-77935-5_28
33. Rushby, J.M.: An Evidential Tool Bus. In: Proc. ICFEM. pp. 36–36. LNCS 3785, Springer (2005). https://doi.org/10.1007/11576280_3
34. Schneider, F.B.: Enforceable security policies. ACM Trans. Inf. Syst. Secur. **3**(1), 30–50 (2000). https://doi.org/10.1145/353323.353382
35. Song, J., Alves-Foss, J.: The DARPA cyber grand challenge: A competitor's perspective, part 2. IEEE Security and Privacy **14**(1), 76–81 (2016). https://doi.org/10.1109/MSP.2016.14
36. Steffen, B., Margaria, T., Braun, V.: The Electronic Tool Integration platform: Concepts and design. STTT **1**(1-2), 9–30 (1997). https://doi.org/10.1007/s100090050003
37. Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., Vigna, G.: Driller: Augmenting fuzzing through selective symbolic execution. In: Proc. NDSS. Internet Society (2016). https://doi.org/10.14722/ndss.2016.23368
38. Taneja, K., Xie, T., Tillmann, N., de Halleux, J.: eXpress: Guided path exploration for efficient regression test generation. In: Proc. ISSTA. pp. 1–11. ACM (2011). https://doi.org/10.1145/2001420.2001422
39. Tip, F.: A survey of program slicing techniques. J. Programming Languages **3**, 121–189 (1995)
40. Visser, W., Păsăreanu, C.S., Khurshid, S.: Test input generation with Java PathFinder. In: Proc. ISSTA. pp. 97–107. ACM (2004). https://doi.org/10.1145/1007512.1007526
41. Zhu, Z., Jiao, L., Xu, X.: Combining search-based testing and dynamic symbolic execution by evolvability metric. In: Proc. ICSME. pp. 59–68. IEEE (2018). https://doi.org/10.1109/ICSME.2018.00015